Hollywood 10.0

The Cross-Platform Multimedia Application Layer

Andreas Falkenhahn

Table of Contents

1 General information

1.1 Introduction

Hollywood is a multimedia-oriented programming language that can be used to create applications and games very easily. Designed with the paradigm to make software creation as easy as possible in mind, Hollywood is suited for beginners and advanced users alike. Hollywood comes with an extensive function library (encompassing over 1000 commands) that simplifies the creation of applications and games to a great extent. Having been in development since 2002 it is a very mature and stable software package today.

One of the highlights of Hollywood is its inbuilt cross-compiler which can be used to deploy software on many different platforms without having to change a single line of the code. The cross-compiler can compile executables for all platforms from any platform Hollywood is running on. For instance, you can compile macOS application bundles using the Windows version of Hollywood and vice versa. On top of that, there are also extensions that allow you to compile your Hollywood projects into native applications for iOS and Android.

Hollywood is a light-weight, but still powerful programming language whose core is just about two megabytes in size and does not require any external components. Hence, it is ideal for creating programs which run right out of the box. In fact, Hollywood programs will run perfectly from a USB flash drive without any prior installation whatsoever. Furthermore, Hollywood's core functionality can be greatly enhanced through lots of freely available plugins allowing you to access OpenGL and SDL through Hollywood, for instance.

The following platform architectures are currently supported by Hollywood:

- AmigaOS 3 (m68k)
- AmigaOS 4 (ppc)
- Android (arm)
- AROS (x86)
- iOS (arm)
- Linux (x86)
- Linux (x64)
- Linux (ppc)
- Linux (arm)
- macOS (arm)
- macOS (x86)
- macOS (x64)
- macOS (ppc)
- MorphOS (ppc)
- WarpOS (m68k/ppc)
- Windows (x86)
- Windows (x64)

Hollywood features

Graphics:

- Very flexible layer-based graphics engine
- Support for alpha channel graphics
- Sprites of any size can be used
- Extensive text support incl. on-the-fly formatting, wordwrapping, and rotation
- Platform independent TrueType text rendering
- Video playback fully supported
- Loads real vector image formats like SVG
- PDF import and export supported
- Many graphics primitives supported (ellipses, arcs, lines, rectangles, polygons...)
- Support for vector-based drawing (Bézier splines...)
- Optional antialiasing for text and graphics primitives
- Over 150 transition effects for graphics and text
- Tons of image processing functions
- Powerful off-screen rendering functions incl. rendering to masks and alpha channels
- Extensive clipping support (rectangular and custom shaped clipping regions)
- Support for hardware accelerated double-buffered displays
- Animation support
- Graphics can be exported as PNG, or even as AVI video streams
- Windows can use alpha channel transparency
- OpenGL 3D programming supported via plugin
- Video streaming supported via a dedicated plugin

Sound:

- Multichannel sound interface
- Support for samples and sound streams
- Protracker modules can be played
- Volume and pitch of samples can be changed during playback
- Multichannel mixer for manipulating samples
- Dynamically generated sound can be sent to the audio device
- Audio streaming supported via a dedicated plugin

GUI:

- Native GUI development supported via the RapaGUI plugin
- Create native GUIs for Windows, Linux (GTK), macOS and AmigaOS (MUI)
- GUIs are conveniently layouted using XML files
- Support for over 40 different widgets
- Full flexibility because Hollywood displays can be embedded in native GUIs

- Completely platform-independent GUI development - use the same code on every platform!

Network:

- Full Internet and network support
- Create server and client connections
- Transfer data using a wide variety of protocols like HTTP, FTP and SCP
- Serial I/O support through RS/232 or USB adapters
- IPC support for communicating with other programs
- IPv4 and IPv6 interfaces supported
- Full SSL/TLS support

System:

- Powerful yet extremely easy to use programming language
- Cross-platform compiler that supports Amiga compatibles, Windows, macOS and Linux
- Android and iOS support via the freely available Hollywood Player
- APKs can be generated via the optionally available APK Compiler
- Executables compiled by Hollywood do not require any additional libraries/DLLs (they also run from USB sticks!)
- All data files (including fonts) can be embedded in a single executable
- Unicode fully supported
- Windowed and full screen mode supported
- Multi-monitor support
- Sandbox container: programs can never crash
- Creation of OS menus is supported
- Event-based programming model guarantees moderate CPU usage
- Low latency interval and timer functions
- Extensive DOS library to work with the file system
- Support for ZIP and other archivers
- Convenient access to clipboard
- Access to system dialogs (file chooser, string prompt, etc.)
- Drag'n'drop support
- Database management via SQL supported
- Comprehensive string and math library
- Mouse cursor can be easily replaced with a custom cursor
- Convenient data serialization to and from JSON and XML
- Date and time functions
- Easy internationalization of programs via catalog system
- Joystick support

Plugin:

- Extremely powerful, cross-platform plugin system
- Publicly available SDK with over 500 pages of documentation and example code
- Default display driver is completely retargetable through entirely different subsystems (e.g. OpenGL)
- Default audio driver is completely retargetable through entirely different subsystems
- Plugins can add loaders and savers for additional image, animation, sound, font, and video formats
- All file I/O can be rerouted through customized handlers
- Hollywood's language command set can be extended via plugins

1.2 Philosophy

The philosophy behind Hollywood is to offer an easy yet powerful platform which can be used to write stunning programs in a very short time. The language used to program Hollywood is very easy for the beginner but also has enough features that the experienced programmer will love. You can program Hollywood on a very simple BASIC-like level but it is also possible to dive into a fully object oriented world with Hollywood. The extensive command set with over 1000 inbuilt functions provides the programmer with all the tools needed to create amazing software with Hollywood. There is almost nothing you can't do with Hollywood. On top of that, Hollywood is a cross-platform multimedia application layer, which means that you can run your programs on many different platform without changing a single line of code, and it is even possible to cross-compile executables for these platforms. For example, you can cross-compile macOS applications from your AmigaOS 3.1 installation. All this makes Hollywood the ultimate tool for the multimedia programmer and an experience in its own right!

1.3 Terms and conditions

Trial version:

You are allowed to test the trial version of Hollywood for 30 days. If you want to continue using Hollywood after your trial period of 30 days has expired, you need to buy the full version. Otherwise you have to uninstall Hollywood after 30 days. It is not allowed to continue using Hollywood after the trial period of 30 days has expired.

The trial version of Hollywood may be freely distributed as long as the following conditions are met:

1. No modifications must be made to the trial version.
2. It is not allowed to sell the trial version.

3. Putting the trial version on a cover disc is only allowed with a written permission by the author.

Full version:

The full version of Hollywood may be installed on multiple computers as long as it is only used by the person who bought the license. All Hollywood licenses are single user licenses so everybody who wants to use Hollywood needs to purchase an individual, personalized license. Licensees may then install and use Hollywood on multiple computers but it must only be used by the person who bought and owns the license. It is not allowed to share a Hollywood license with other people. The name of the person who purchased the license, and thus is the only one who is allowed to use the program, is shown in both the Hollywood GUI and interpreter.

It is forbidden to spread the full version of Hollywood without a written permission by the author.

What you may create with Hollywood:

There are no restrictions on what you may create with Hollywood except that it is forbidden to create programs with Hollywood which could be considered competing products to Hollywood itself, specifically it is not allowed to create any kind of wrapper programs that make Hollywood libraries, plugins, and/or commands available to other programming languages or the end-user. It is also not allowed to create any sort of middleware that enables the user to access Hollywood libraries, plugins, and/or commands through the middleware. It is also not allowed to create a programming language which builds upon Hollywood's libraries, plugins, and/or commands.

Disclaimer:

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDER AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Acknowledgements:

This software uses Lua by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo. See Section A.1 [Lua license], page 1219, for details.

This software uses libjpeg by the Independent JPEG Group.

This software uses libpng by the PNG Development Group and zlib by Jean-loup Gailly and Mark Adler.

This software uses PTPlay © Copyright 2001, 2003, 2004 by Ronald Hof, Timm S. Mueller, Per Johansson.

This software uses the OpenCV library by Intel Corporation. See Section A.2 [OpenCV library license], page 1219, for details.

This software uses ImageMagick by ImageMagick Studio LLC. See Section A.3 [ImageMagick license], page 1219, for details.

This software uses the GD Graphics Library by Thomas Boutell. See Section A.4 [GD Graphics Library license], page 1223, for details.

This software uses the pixman library. See Section A.6 [Pixman license], page 1224, for details.

Portions of this software are copyright © 2010 The FreeType Project (`http://www.freetype.org`). All rights reserved.

Hollywood uses the Bitstream Vera font family. See Section A.5 [Bitstream Vera fonts license], page 1223, for details.

The Linux version of Hollywood uses gtk, glibc, and the Advanced Linux Sound Architecture (ALSA) all of which are licensed under the LGPL license. See Section A.11 [LGPL license], page 1227, for details.

The Android version of Hollywood uses the Simple DirectMedia Layer (SDL) by Sam Lantinga. See Section A.10 [SDL license], page 1226, for details.

This software uses codesets.library by Alfonso Ranieri and the codesets.library Open Source Team. See Section A.11 [LGPL license], page 1227, for details.

This software uses LuaSocket by Diego Nehab. See Section A.7 [LuaSocket license], page 1225, for details.

This software uses librs232 by Petr Stetiar. See Section A.8 [librs232 license], page 1225, for details.

This software uses UsbSerial by Felipe Herranz. See Section A.9 [UsbSerial license], page 1226, for details.

All trademarks are the property of their respective owners.

1.4 Requirements

Windows version:

- requires at least Windows 2000

macOS version:

- arm64 version: requires at least macOS 11.0 (Big Sur)
- Intel version: requires at least macOS 10.6 (Snow Leopard)
- PowerPC version: requires at least macOS 10.4 (Tiger)

Linux version:

- requires an X11 server and glibc
- optional: ALSA library for sound output
- optional: GTK for dialog boxes support
- optional: XFree86 video mode extension library, Xfixes, Xrender, Xcursor, Xrandr, Xss for some advanced functionality

AmigaOS/MorphOS/AROS versions:

- Kickstart 3.0 (V39)
- 68020+ or PowerPC
- CyberGraphX or Picasso96
- `codesets.library` on AmigaOS 3 and 4, WarpOS, and AROS
- `charsets.library` on MorphOS
- optional: AHI by Martin Blom for sound output
- optional: `reqtools.library` for the `StringRequest()` function (except on OS4) and for the `ColorRequest()` function (also on OS4)
- optional: `popupmenu.library` for the `PopupMenu()` function (except on MorphOS)

iOS version:

- iOS 8 or better

Android version:

- Android 4.0 or better
- ARM CPU (32-bit or 64-bit)

1.5 Credits

Hollywood was written by Andreas Falkenhahn. But I would not have gotten so far without the help of many persons whom I would like to thank here.

First, special thanks go to Timm S. Müller for his essential hints in the early phase of conceptualizing Hollywood.

Secondly, big thanks to the team of Lua 5.0.2 for making this powerful light-weight language: Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo.

Thanks go to Frank Wille for constantly improving the wonderful vbcc compiler which is necessary for the 68k and WarpOS builds of Hollywood.

Special thanks must go to Dominic Widmer, Helmut Haake and Alexander Pfau for translating the huge documentation to German. Alexander Pfau was the first one to work on a German translation. He maintained the German Hollywood manual until version 1.9. His work has later been continued in a Swiss-German joint effort by Dominic Widmer and Helmut Haake who maintain the German Hollywood translation until today.

Further thanks have to go out to Grzegorz Kraszewski, Martin Blom, Tomasz Wiszkowski, Kimmo Pekkola, Olaf Barthel, Thomas Richter, Christoph Gutjahr, Jean-Yves Auger, Ralph Schmidt, Detlef Würkner, Stephan Rupprecht, Frank Mariak, Jacek Piszczek, Torgeir Vee, Christoph Poelzl, Fabio Falcucci, Michael Jurisch, Petteri Valli and to all beta testers and to everyone that should be here but has been forgotten.

The Amiga version of Hollywood was developed under SAS/C 6.58 (68k version), VBCC (WarpOS version), GCC 4.4.4 (MorphOS version) and GCC 4.0.2 (AmigaOS4 version). Additionally the following programs were used: GoldED Studio AIX, Directory Opus 4, PPaint 7, CyberGuard, CyberGraphX 4, MUI. Main development was done on a Pegasos 2 with a 1Ghz G4 CPU and MorphOS 1.4.5. Further development was done on an Amiga 1200 equipped with a Phase5 Blizzard PPC 603e 240mhz with SCSI, a 68060 CPU, a Phase5 BVision PPC graphics board and 82 Megabyte RAM. Hollywood was widely tested on CyberGraphX 3 and 4, Picasso96, MorphOS, AmigaOS4, AROS, DraCo, Amithlon and WinUAE. Hollywood will in no way access the hardware directly. It respects all system style guides and uses system-friendly functions only.

The macOS version was developed on a 1.5 Ghz Mac Mini using macOS 10.4 (Tiger) and on an Intel iMac 2.4 Ghz using macOS 10.5 (Leopard). The code was written using Allan Odgaard's flexible text editor TextMate. Hollywood was compiled using the gcc that comes with Apple's macOS SDK.

The Windows version was developed on a 2.6 Ghz Pentium IV using Win XP Home Edition with the latest service packs. The code was written using the famous UltraEdit by IDM Comp. Hollywood was compiled using Microsoft's Visual C.

The Linux version was developed using openSUSE 11.2 on a 2.6 Ghz Pentium IV.

1.6 Forum

The official Hollywood forums are online at http://www.hollywood-mal.com. Feel free to stop by and talk to other Hollywood users. This is the perfect place to ask for help from other users.

Besides the forum, we also have a newsletter that is used for announcements. If you want to be notified about new releases, Hollywood plugins, updates, and everything else about the Hollywood ecosystem don't hesitate to sign up for the newsletter at the official Hollywood portal at http://www.hollywood-mal.com.

Users from Germany might want to take a look at the "Hollywood User Page" maintained by Helmut Haake. It offers lots of source codes and workshops and also has a forum where you can ask your questions. Here is the link: https://forum.amiga-resistance.info/viewforum.php?f=38

1.7 Contact

If you need to contact me, you can either send an e-mail to andreas@airsoftsoftwair.de or use the contact form on http://www.hollywood-mal.com.

2 Getting started

2.1 Overview

Here's an overview of the most important components you have to know when programming with Hollywood:

Interpreter:

The Hollywood main executable is what we call "the interpreter". It can read your source code files (the scripts) and translate them into custom Hollywood bytecode. It can also compile your source codes into stand-alone executables or applets and link data files into those executables or applets. Once you have compiled your source codes into executables or applets, you can distribute them. If you distribute your projects as applets, your users first have to install the freely available Hollywood Player before they can run your project. If you choose to distribute your projects as stand-alone executables, no further components are needed. The interpreter is a console program and doesn't have any GUI. Hollywood's GUI and the Windows IDE will run simply start the interpreter when you choose to run a script.

Player:

The Hollywood Player can only run applets (see below). It cannot run source codes. The Hollywood Player isn't part of Hollywood's commercial distribution but is available for free download from the official Hollywood portal at `http://www.hollywood-mal.com`. In contrast to the interpreter, the Hollywood Player is freely distributable. If you choose to distribute your projects as Hollywood applets, your users first have to download and install the Hollywood Player before they can run your projects. If you choose to distribute your projects as stand-alone executables, the Hollywood Player isn't necessary because it has already been linked into your executable. Note that the Hollywood Player for Android isn't available from the Hollywood portal but from Google Play: `http://play.google.com/store/apps/details?id=com.airsoftsoftwair.hollywood`

GUI:

Since the Hollywood interpreter is just a console program, it is accompanied by a separate GUI program which allows you to conveniently use the interpreter. On Windows, the Hollywood GUI is a full-blown IDE whereas on all other systems (Amiga, Linux, macOS) it is just a front-end which you can use to start and compile scripts but it doesn't allow you to edit scripts. Thus, on Amiga, Linux, and macOS you have to use your favourite text editor for editing Hollywood scripts. But since the interpreter is a console program, it is quite easy to integrate it with your favourite IDE.

Scripts:

Source codes in the Hollywood language are called Hollywood scripts. Hollywood scripts are simply text files that contain a number of statements Hollywood can understand. Thus, they have to follow certain syntactical rules which are explained in this documentation. Hollywood scripts use the extension `*.hws`. It is recommended to use UTF-8 encoding for all your scripts.

Applets:

Hollywood applets are binary files that contain the bytecode of a script as well as additional data like images, sounds, fonts, etc. Applets can be run by the Hollywood Player or the Hollywood interpreter. Hollywood applets use the extension `*.hwa`. Choosing to distribute your project as an applet can save lots of disk space since applets are usually very small because they do not contain the Hollywood Player. See Section 4.2 [Applets], page 58, for details.

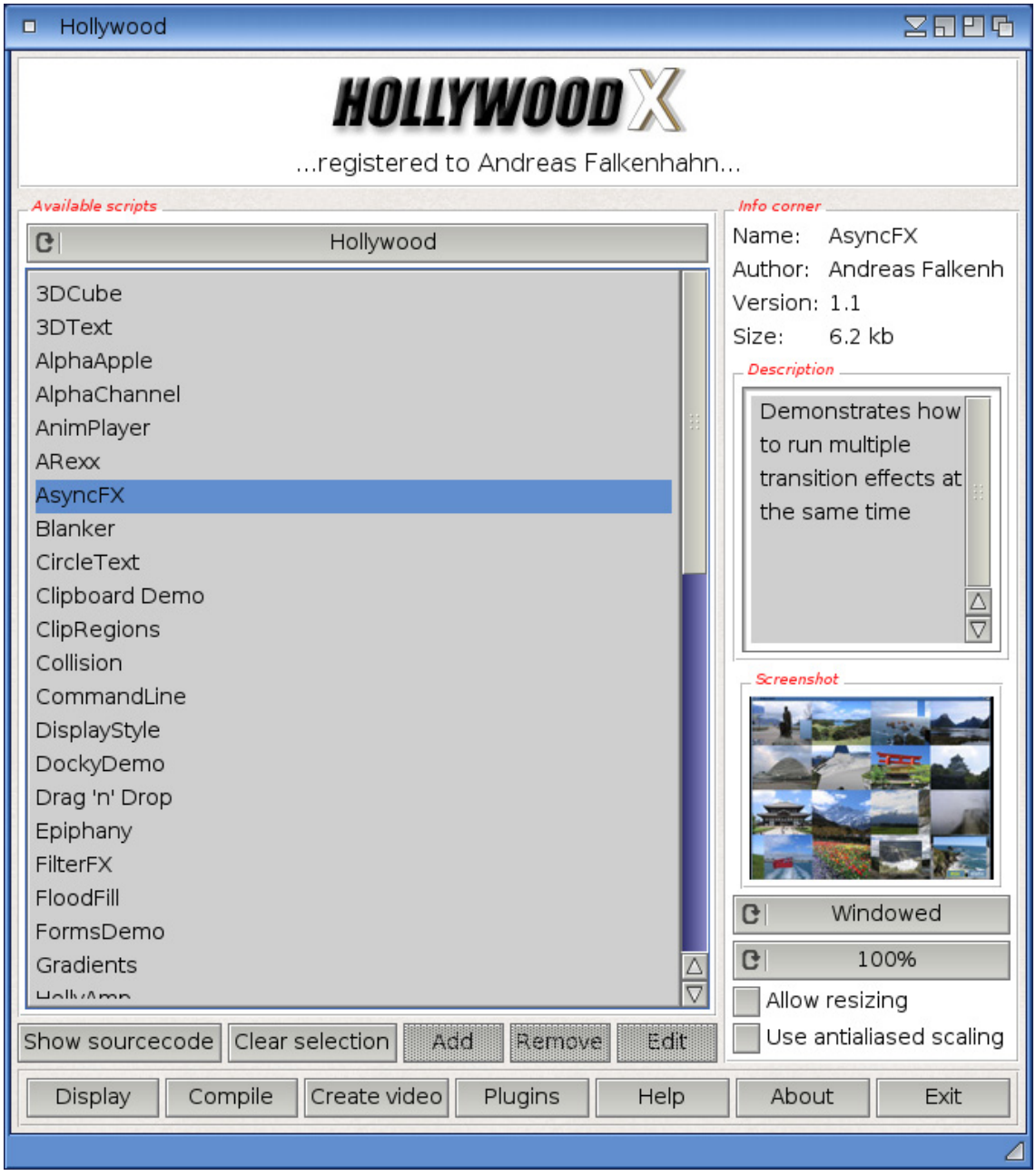
Plugins:

On all systems except on AmigaOS and compatibles, plugins must be stored in a directory named "Plugins" that is in the same directory as the main Hollywood program. On AmigaOS and compatible systems, plugins must be installed to `LIBS:Hollywood` instead. On macOS, the "Plugins" directory must be inside the "Resources" directory of the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle itself, namely in `Hollywood.app/Contents/Resources`. When distributing a compiled Hollywood program, plugins required by your program must simply be put into the same directory as your program. When compiling application bundles for macOS, plugins need to be put in the "Resources" directory of the application bundle, i.e. in `MyProject.app/Contents/Resources`. Alternatively, you can also choose to link plugins into your executables. Hollywood plugins use the extension `*.hwp`. See Section 5.1 [Plugins], page 65, for details.

2.2 The GUI

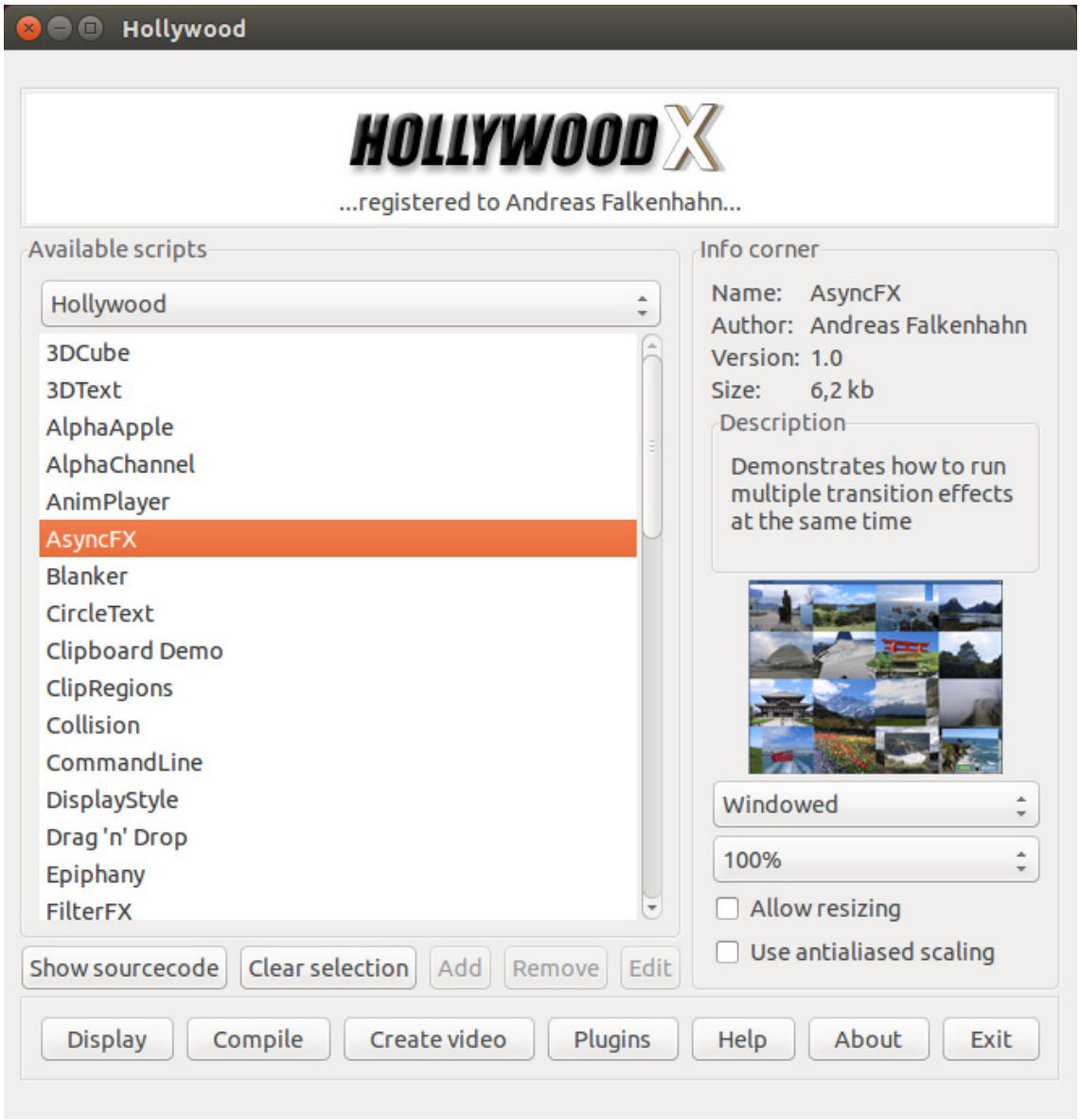
Hollywood itself is a console program but for reasons of convenience there is also a front-end that can be used to control Hollywood without having to resort to the console.

Here is a screenshot of Hollywood's GUI for Amiga systems:



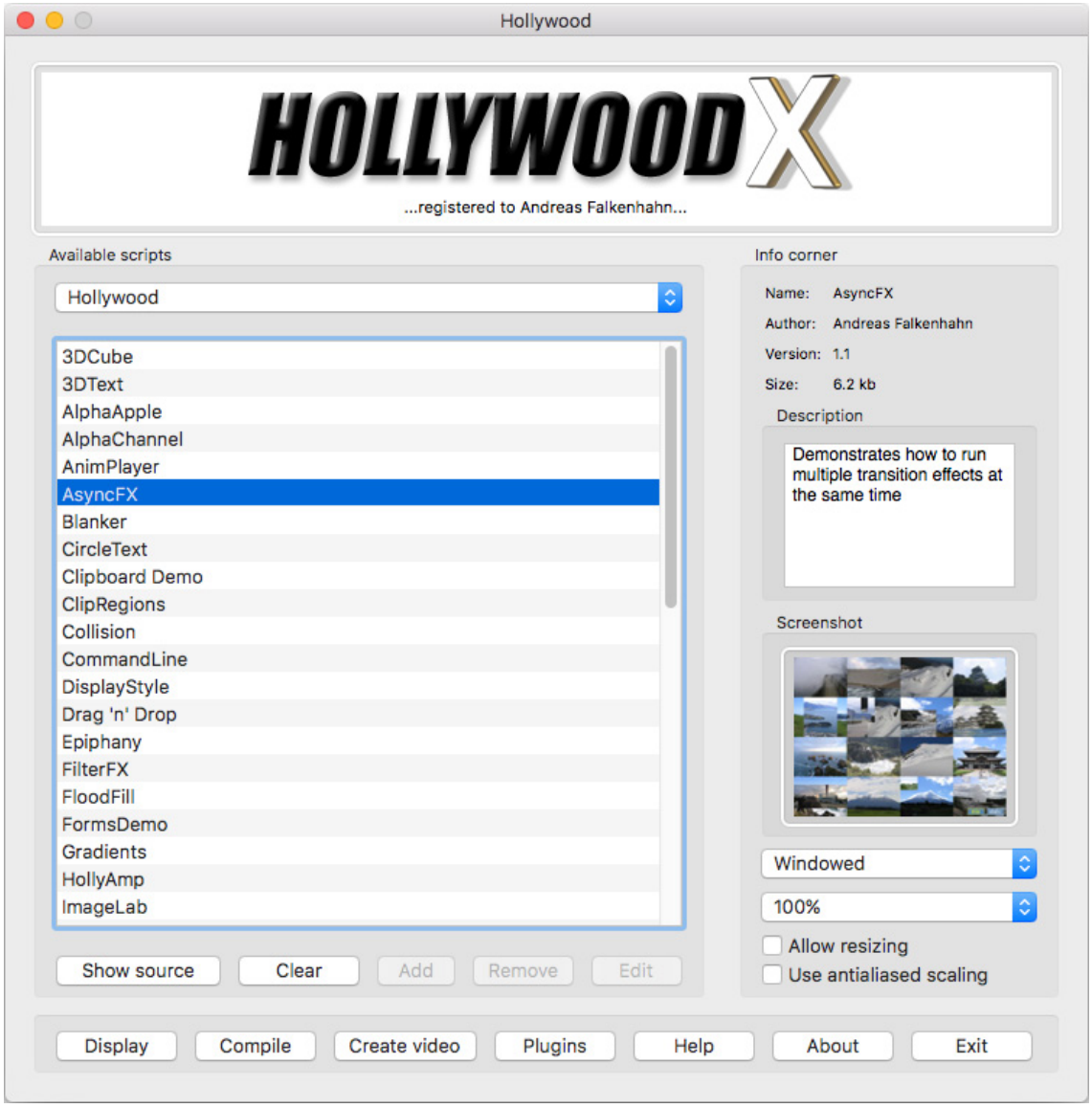
The Linux and macOS GUIs are completely identical to the Amiga one. As you can see the GUI is divided into three parts: A listview containing all available Hollywood projects, an info corner which displays information about the currently selected Hollywood project and a screenshot of it (if available), and lastly a range of buttons that can be used to execute most of Hollywood's standard actions.

This is what the Hollywood GUI looks like on Linux:



Upon startup, Hollywood will automatically scan the `Examples` folder stored inside Hollywood's installation directory and will add all scripts it can find into the different categories. You can then use the choice widget to switch between the different categories. The last category is always called "My projects." The scripts listed here must not be stored inside the `Examples` folder but should be kept in a different place so that they do not get lost when updating to a newer version.

Here is a screenshot of the Hollywood GUI on macOS:



The following functionality is available in the GUI:

Show sourcecode:

Press this button to show the sourcecode of the currently selected Hollywood project. You can configure the viewer to use here in the preferences menu.

Clear selection:

Clear the currently selected project. This is necessary if you would like to run a project that doesn't appear in one of the two project listviews because in case no project is selected, the **Display** button will pop up a file requester prompting you to select a Hollywood script to run. So if you want this file requester to appear, you first have to clear the selection using this button.

Add: This button will only be enabled if the **My projects** group has been selected in the choice widget. In that case you can use this button to add a new project to the list of projects that appear in the **My projects** section of the GUI.

Remove: This button will only be enabled if the **My projects** group has been selected in the choice widget. In that case you can use this button to remove a project from the list of projects that appear in the **My projects** section of the GUI.

Edit: This button will only be enabled if the **My projects** group has been selected in the choice widget. In that case you can use this button to open the currently selected project in your favourite text editor. You can configure the text editor to use in the preferences menu. If you want to edit the properties of the currently selected project, go to the project menu and select the properties menu.

Display: Click this button to run the currently selected project. If no project is selected this button will pop up a file requester prompting you to select a project.

Compile: You can use this button to compile a Hollywood script into an executable or Hollywood applet. You will be prompted to select a Hollywood script as well as an output file and a target platform. The front-end will then invoke the Hollywood compiler to build your executable or applet. See Section 4.1 [Using the compiler & linker], page 57, for details.

Create video:
This button allows you to record a video of a Hollywood project. You will be prompted to a select a Hollywood script as well as an output file and a video format. The front-end will then invoke the Hollywood video recorder to create a video of your script. See Section 4.6 [Using the video recorder], page 62, for details.

Plugins: Press this button to open a dialog that shows all currently available Hollywood plugins and some information about them.

Help: Click this button to open this documentation.

About: Displays copyright and licensing information as well as an overview of all available Hollywood commands.

Exit: Closes the GUI.

There are some more options in the right-hand side part of the GUI:

Display mode:
This choice widget allows you to choose the desired display mode for the project. This can be either **Windowed**, **Full screen** or **Full screen (scaled)**. If you choose **Full screen**, the monitor will change its physical resolution to the script's resolution whereas **Full screen (scaled)** preserves the monitor's physical resolution and just scales the script to fit to your monitor's current resolution.

Scale factor:
You can use this choice widget to enable Hollywood's auto scaling engine. If you choose a setting different than 100% here your script will automatically be scaled by the specified factor. See Section 25.18 [Scaling engines], page 401, for details.

Allow resizing:

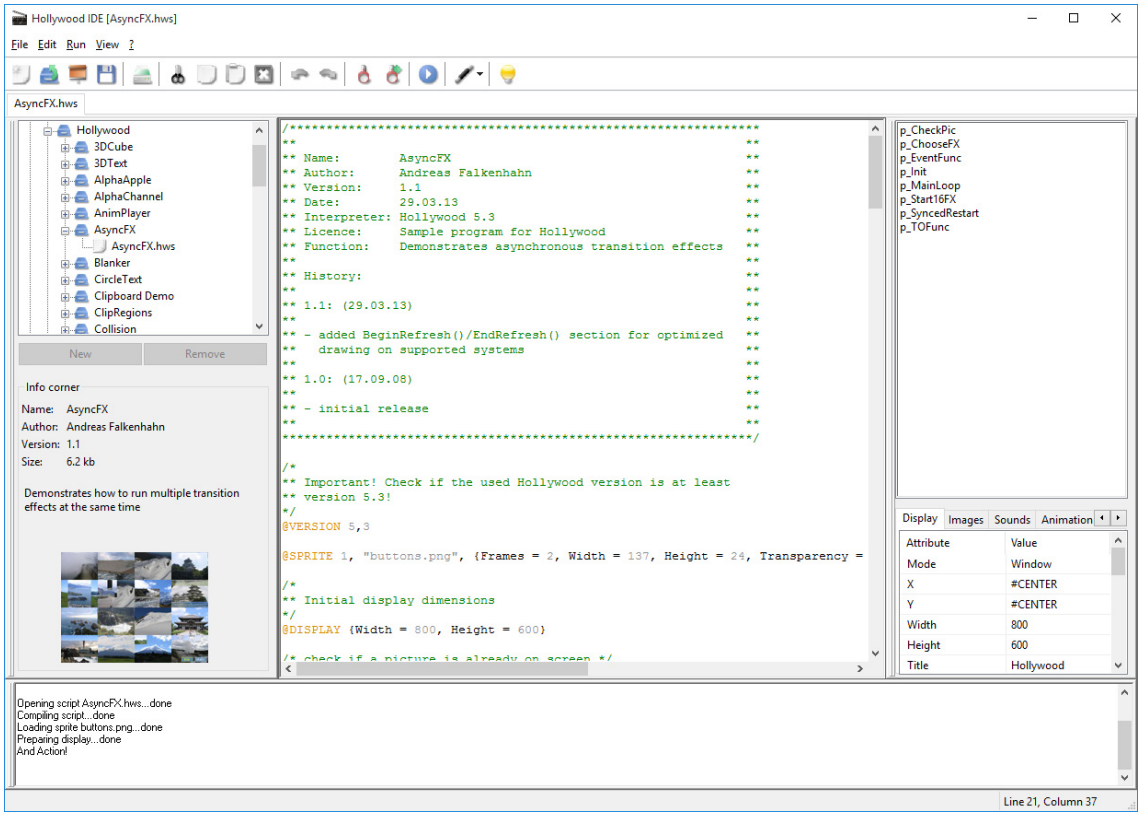
If you tick this box Hollywood will open in resizeable mode. The user will then be able to use the size box of Hollywood's window to change the dimensions of your script. Leave this box unticked if you do not want this.

Use antialiased scaling:

If scaling is enabled, you can use this checkbox to define whether or not antialiased scaling should be used. Antialiased scaling looks better but is slower than hard scaling.

2.3 Windows IDE

On the Windows platform Hollywood comes with a fully-featured integrated development environment (IDE) which can be used to create Hollywood projects very easily. The IDE features a multi-tabbed text editor with syntax highlighting, live help while typing, a function browser as well as a convenient overview of external data referenced by the preprocessor commands used in your script. Here's a screenshot of Hollywood's Windows IDE:



As you can see the IDE is made up of six different parts:

1. On the upper left-hand side of the IDE there is the **Project browser** treeview. It is divided into the trees "Examples" and "My projects". All "Examples" that are delivered with Hollywood are listed in the first tree whereas your own projects appear in the second tree. To add items to the "My projects" tree, simply press the "Add"

button that is located below the treeview. To remove items from the "My projects", use the "Remove" button.

2. On the lower left-hand side of the IDE you can find the **Info corner**. This section displays some information about the currently active Hollywood project. This is currently only implemented for the example projects that come with Hollywood. If you select one of your own projects the info corner will display nothing.
3. The **Editor** constitutes the heart of the IDE and is located in the center of the IDE. The editor will automatically display live help in the IDE's status bar if it detects you typing in a known Hollywood command. It will also highlight all keywords and comments, and it will automatically add function names to the function browser (see below). If you right-click your mouse on the editor area, a context menu will pop up allowing you to jump to the definition of the function that is currently selected by the cursor or opening the file that is currently selected by the cursor. You can also get context sensitive help this way.
4. On the upper right-hand side of the IDE you can find the **Function browser**. This listview contains the names of all functions that have been declared by the currently active project. The list is updated on-the-fly as you edit your script. You can double-click on a name in this list to jump to the function declaration automatically.
5. On the lower right-hand side of the IDE there is the **Preprocessor command tool**. In the first tab you can configure several attributes of the `@DISPLAY` preprocessor command which controls the appearance of your display. You can for example specify the initial position of the display on the desktop screen, whether or not it should open in full screen mode, or if the display should have a border and a size widget. You can also set the title of the display (defaults to "Hollywood"). Of course, you can also configure all these attributes directly in the editor by passing them to `@DISPLAY` preprocessor command directly. In fact, when you edit these attributes using the preprocessor command tool, it will immediately update the corresponding line that contains the `@DISPLAY` specification in your script. If there is none, the preprocessor command tool will add it your script. The other tabs in the preprocessor command tool contain listviews that show all external files referenced by the currently active script. If you click on one of the files, the IDE will jump to the point where this file has been declared in the script. If you double-click on one of the files, the IDE will open Hollywood to show the file.
6. On the bottom of the IDE you can find the **Hollywood output window**. Whenever the IDE starts Hollywood, its output will be redirected to this window. You can also print to this window from your Hollywood script by using the `DebugPrint()` command. You can clear the contents of this window by opening the context menu with the right mouse button and selecting "Clear" then.

Most of the IDE constituents presented above are implemented using dock windows. This means that you can rearrange them according to your personal tastes: You can drag them to a different place in the IDE window or you can even move them out of the IDE window. In that case they will appear as toolbox windows. Finally, you can also hide them if you do not need their functionality. To hide one of these windows, either use the "View" menu or drag the docks out of the window and then close their toolbox window.

The IDE can be controlled either through the toolbar that appears in the top of the window or using the menu. There are also keyboard shortcuts that you can use. Live help is displayed in the status bar when the mouse is over a toolbar or menu item. You can use the toolbar and the menu to run the current project, compile it, record a video of it, or send it to your printer. Furthermore, several standard functions are accessible in toolbar and menu like "Save", "Save as", "Find", "Find and replace", "Copy", "Cut", "Paste", "Undo", "Redo", and so on.

The IDE can also be controlled using keyboard shortcuts. Here are some common shortcuts:

`F1:` Opens context sensitive help for the function/keyword that is at the current cursor position.

`F2:` Jumps to the declaration of the function at the current cursor position.

`F4:` Opens the file whose name is at the current cursor position.

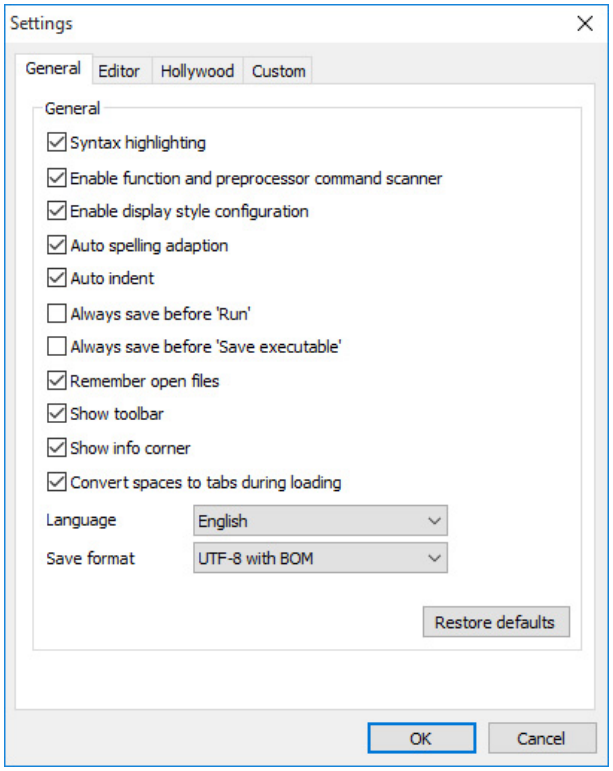
`F5:` Runs the current project in Hollywood.

`Ctrl-F:` Opens the find dialog to search for a string inside the current project.

`Ctrl-G:` Jumps to a specific line in the project.

`Ctrl-S:` Saves the current project.

Several IDE settings are user-configurable. Select the entry "IDE settings..." from the "File" menu to open a dialog that allows you to adjust several settings to your personal taste. The settings dialog consists of four pages: General, Editor, Hollywood, and Custom. Here is a screenshot of the first page:



The following things can be configured on this page:

`Syntax highlighting`
: Tick this box to enable syntax highlighting in the IDE's script editor. Normally this setting should be enabled because it makes your code much more readable. On very slow systems or if you work with extremely large scripts it might be necessary to turn it off for performance reasons.

`Enable function and preprocessor command scanner`
: If you enable this option, the IDE will automatically add all function names to the function browser and it will add all files referenced by preprocessor commands to the preprocessor command tool. This is only worth disabling on very slow systems or with very large scripts.

`Enable display style configuration`
: If this option is enabled, you will be able to configure the parameters of the `@DISPLAY` preprocessor command using the preprocessor command tool in the lower right-hand side of the IDE. If you do not want that, leave this box unchecked.

`Auto spelling adaptation`
: This box allows you to configure whether or not the IDE should automatically adapt the spelling of commands and keywords it knows. For instance, if you type `waitleftmouse` and this option is enabled, the spelling will automatically be corrected into `WaitLeftMouse`.

`Auto indent`
: Check this box if you want the IDE to automatically indent the code after statements that open a new code scope (for example `If`, `While`, `Function` etc.). If you enable this option, the IDE will indent the next line by inserting a tab character. This is very useful for code readability and should not be turned off.

`Always save before Run`
: If you tick this box, the IDE will always save your current project automatically when you run it. Be careful with this option because you might lose some important changes.

`Always save before Compile`
: If you tick this box, the IDE will always save your current project automatically when you compile it into an executable.

`Remember open files`
: Tick this box to tell the IDE that it should remember all tabs that were open in the previous session for the next session.

`Show toolbar`
: You can use this option to configure whether or not you want the toolbar to be shown.

`Show info corner`
: You can use this option to configure whether or not you want the info corner to be shown.

`Convert spaces to tabs during loading`
: This option allows you to configure whether or not you want spaces to be converted into tabs before loading. This is a useful option because it is much easier to structure your code using tabs instead of spaces but be careful because it might destroy the layout of your code if you use a differing tab settings between the Hollywood IDE and other text editors. You can configure the tab space on the "Editor" page.

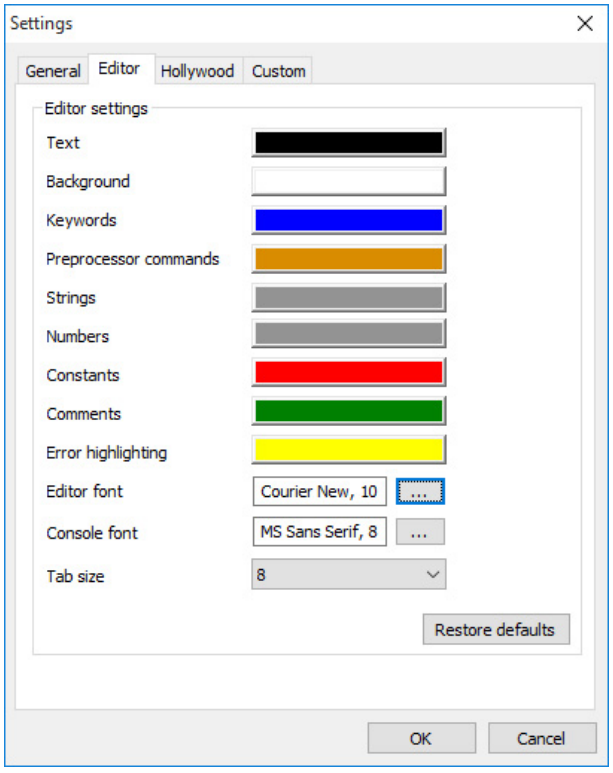
`Keep help window on top`
: If you activate this option, Hollywood's help window will always appear on top of all other windows that belong to the IDE.

`Language`
: This widget allows you to change the language used by the IDE. You will have to restart the IDE for this change to take effect.

`Save format`
: You can set the script's output format here. Normally, this should always be UTF-8, either with or without BOM. ISO 8859-1 shouldn't be used any longer since it can lead to compatibility problems on systems with a different locale.

`Restore defaults`
: Use this button to reset all settings on this page to their default values.

The second page allows you to configure the appearance of the editor:



`Colors`
: You can use these buttons to adjust the colors used by the editor for syntax highlighting to your personal tastes.

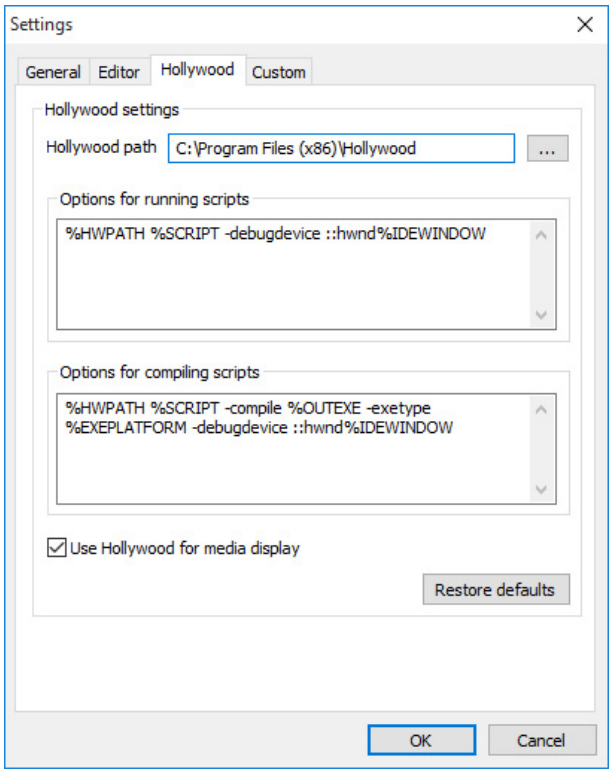
`Font` The font you specify here is used by the editor. You must use a fixed-width font here. Otherwise the layout will get messed up.

`Console font`
The font you specify here is used by the console window. You can use a monospace or proportional font here.

`Tab size` Here you can specify the tab width that should be used by the editor. You can choose between a tab width of 2, 4, 6, and 8 space characters.

`Restore defaults`
Use this button to reset all settings on this page to their default values.

The third page allows you to configure the Hollywood interface:



`Hollywood path`
This field must be set to the path where Hollywood is located. Whenever the IDE invokes Hollywood, it will look for it in the path specified here. Normally, Hollywood resides in the same path as the IDE executable.

`Options for running scripts`
The template specified in this text box is used by the IDE to run a Hollywood script. Normally, you do not have to change anything here. See below for tokens that can be used here.

`Options for compiling scripts`
The template specified in this text box is used by the IDE to compile a Hollywood script. Normally, you do not have to change anything here. See below for tokens that can be used here.

`Use Hollywood for media display`
: Check this box if you want the IDE to use Hollywood to show any external media files like images and animations. It is recommended to enable this option because Hollywood supports several exotic formats like IFF ILBM and Protracker modules which are not supported by the standard media viewers that come with Windows.

`Restore defaults`
: Use this button to reset all settings on this page to their default values.

The following tokens can be specified in the template that is used to start Hollywood to run or compile a script:

`%HWPATH:` This token will be replaced by the path to the Hollywood executable.

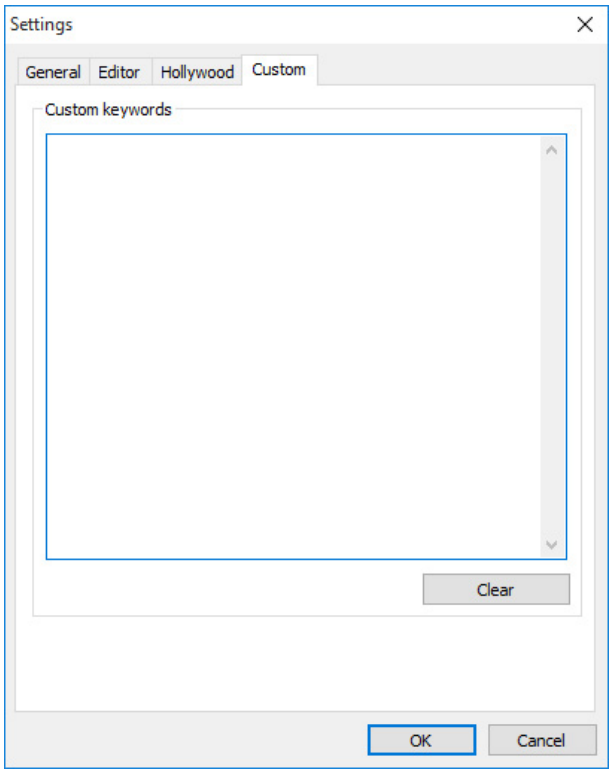
`%SCRIPT:` This token will be set to the Hollywood script that should be compiled.

`%IDEWINDOW:`
: This token inserts the handle of the IDE window.

`%OUTEXE:` Points to a destination file that should be created by the compiler.

`%EXEPLATFORM:`
: Points to one or more platforms that the compiler should target.

The fourth page allows you to add custom keywords that should be taken into account by the editor:



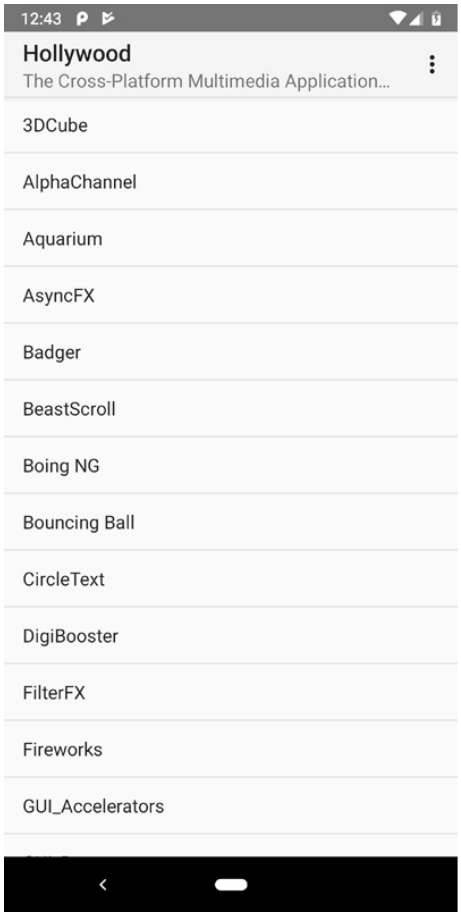
If you want the editor to highlight other keywords than the predefined ones, you can add those keywords here. Please note that keywords must always start with an alphabetic letter

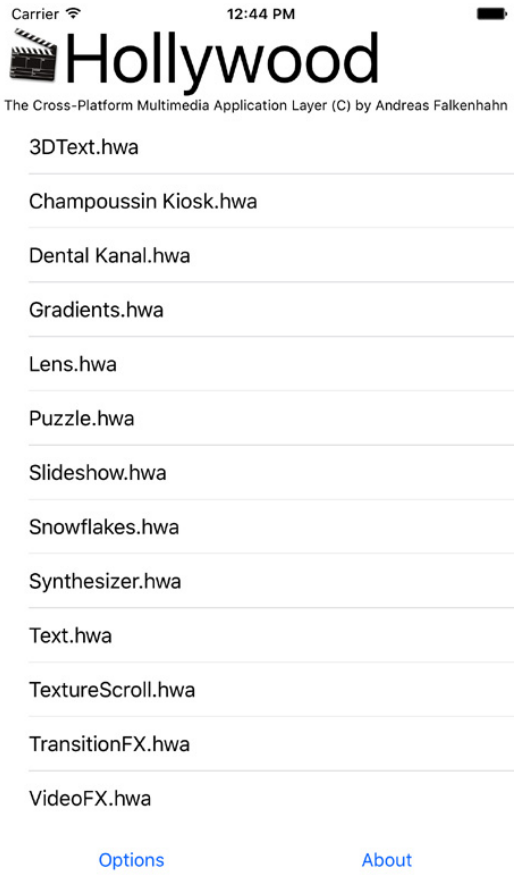
or an underscore. They can contain numbers but not in initial position. The only special characters allowed are the underscore, the dollar sign and the exclamation sign. Other characters are not allowed. There must only be one keyword per line.

2.4 Mobile platforms

Hollywood is also available as a player-only version for Android and iOS which allows you to run your Hollywood applets on smartphones and tablets. The Hollywood Player for Android requires at least Android 4.0 and is freely available on Google Play at this URL: http://play.google.com/store/apps/details?id=com.airsoftsoftwair.hollywood. Unfortunately, the Hollywood Player for iOS is currently not available on the App Store because its capabilities seem to conflict with Apple's App Store rules.

Here are screenshots of the Hollywood Player for Android (left) and iOS (right):





If you want to run your Hollywood projects on your mobile device, you first need to compile them as Hollywood applets, copy them to your device and then run them using the Hollywood Player for your mobile platform.

Alternatively, there is also an add-on named Hollywood APK Compiler which can be used to compile your Hollywood projects into stand-alone APK files for Android. Please visit

the Hollywood portal for more information on the Hollywood APK Compiler: http://www.hollywood-mal.com

If you want to use the freely available Hollywood Player, read through this step-by-step guide to get your Hollywood projects onto your mobile device:

1. Compile your Hollywood project as a Hollywood applet (`*.hwa`) in the Hollywood GUI or in Designer or from the command line.

2. On Android systems: Copy the `*.hwa` file to the directory `Hollywood` on your Android's device SD card. The directory `Hollywood` is automatically created when you first launch Hollywood on your Android device. If it is not there, create it on your own. It has to be in the root directory of your SD card.

3. On iOS systems: Use iTunes to copy the `*.hwa` file to the documents directory of the Hollywood Player app.

4. If your applet does not load any external files and has all files linked in, skip to item 7.

5. If your applet loads external files, create a directory for your applet inside the `Hollywood` directory (on Android) or inside the Hollywood Player app's documents directory (on iOS). The directory name must be identical to the name of your applet minus the file extension (`*.hwa`). For example: If your applet is called `My cool game.hwa`, you have to create the directory `My cool game` for it inside the Hollywood or documents directory.

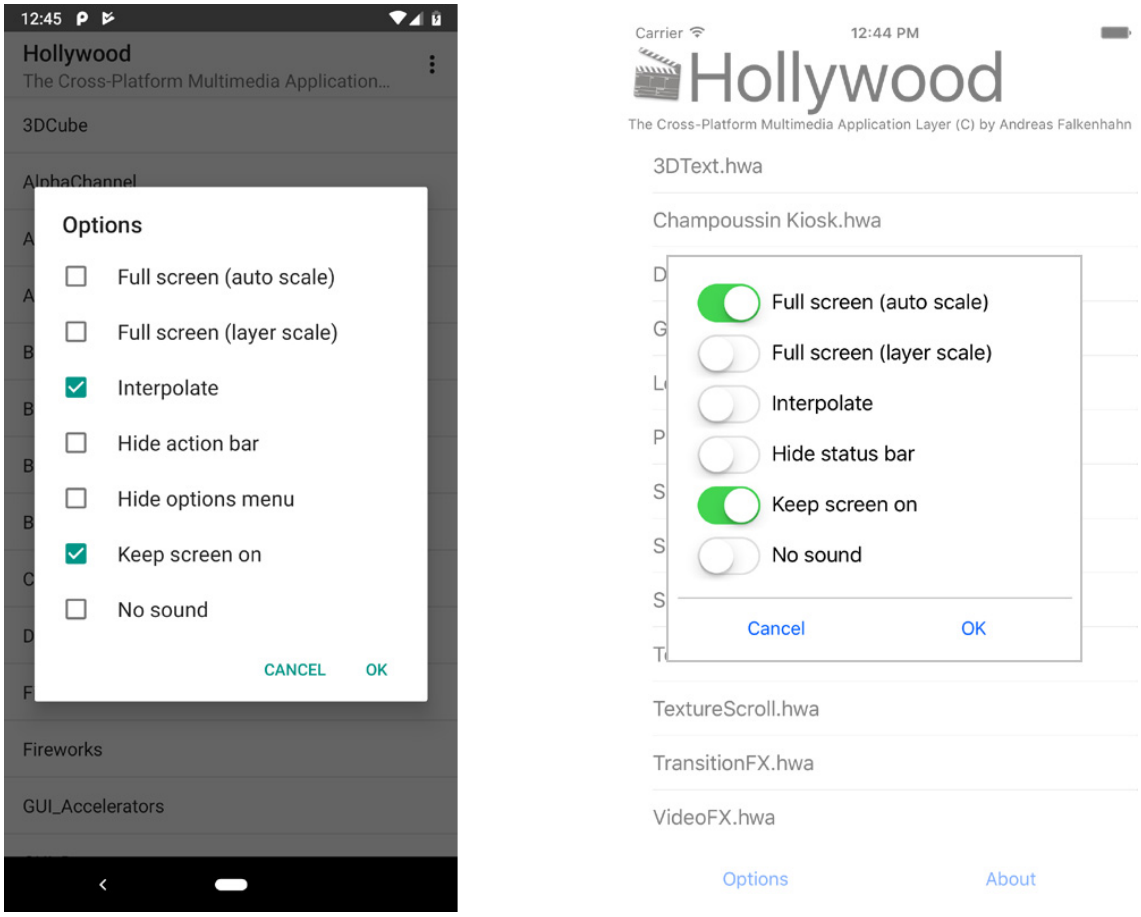
6. Copy all data files required by your applet into the newly created directory.

7. Run the Hollywood Player on your device. A list of all applets that could be found will be shown. Select your applet and then touch the play button in the control bar at the bottom of the screen.

8. Now your applet should start!

On Android you can alternatively also copy the Hollywood applet to an arbitrary location on your SD card and start it by simply running the `*.hwa` file directly from your favourite file manager. In that case all external files referenced by the applet will be loaded from the location relative to the folder where the `*.hwa` file is located. Hollywood will not change the directory to a subdirectory named after the applet in the case that the applet has been placed outside of the `/sdcard/Hollywood` folder.

You can configure some options by touching the "Options" button. The options screen looks like this on Android (left) and iOS (right):



The following options are currently available:

`Full screen (auto scale):`
: Runs the script in full screen mode using the auto scaling engine. This is the recommended way to make sure that scripts designed for a different resolution cover the whole screen of your mobile device. If your script uses layers and vector graphics, however, you should use the layer scaling engine instead to get perfectly crisp graphics in any resolution (see below).

`Full screen (layer scale):`
: Runs the script in full screen mode using the layer scaling engine. This is only supported for scripts which use layers. It is slower than the auto scaling engine but can lead to perfectly crisp graphics in any resolution if your script only uses vector graphics. For scripts that don't use layers, you should use the auto scaling engine (see above). Note that you can also use the auto scaling engine for scripts which use layers but then the quality won't be as good as with the layer scaling engine if your script uses vector graphics. That's why it is recommended to use the layer scaling engine only for scripts which use layers

and vector graphics. For scripts which use layers and pixel graphics, you should use the auto scaling engine because it is faster.

`Interpolate:`
: Use anti-aliased interpolation when scaling. This is recommended to get smoother graphics with less scaling artifacts.

`Hide action bar`
: Hide the action bar on Android. This is recommended for scripts that don't need any user interaction through options in the Android action bar.

`Hide status bar:`
: Hide the status bar on iOS if there is one. Enabling this option will result in a full screen display.

`Hide options menu:`
: Do not allow changes to the display configuration through the options menu in the action bar. If you activate this, no options menu will be visible in the action bar. This is only supported on Android. On iOS, Hollywood doesn't have support for changing options while it is running a script.

`Keep screen on:`
: Force the mobile device to keep its screen on even if there is no user activity. Normally, mobile devices will automatically switch into battery saver mode after a certain amount of user inactivity time. Ticking this option will disable this behaviour.

`No sound:` Activates this option to start Hollywood in mute mode.

See Section 3.2 [console arguments], page 33, for more detailed information on the options listed above.

The mobile versions of Hollywood have some special features in comparison to the desktop versions. Here is a list of special features in Hollywood for Android and iOS:

1. On Android and iOS Hollywood comes with native support for Ogg Vorbis and Ogg Theora. You do not need any plugins to open these file formats. They will run out of the box on Android and iOS.
2. You can show a virtual keyboard by using the `ShowKeyboard()` and `HideKeyboard()` functions. You can also listen to the `HideKeyboard` event handler to find out when the user closes the virtual keyboard. See Section 29.13 [InstallEventHandler], page 553, for details.
3. You can listen to the `OrientationChange` event handler to find out when the user rotates the device. See Section 29.13 [InstallEventHandler], page 553, for details.
4. You can also hard-code an orientation that your script shall use by specifying the `"Orientation"` tag that is supported by the `@DISPLAY` preprocessor command. In that case, Hollywood will not react to orientation changes when the user rotates the device. Instead, it will keep the orientation mode that you specified in the `"Orientation"` tag. See Section 25.8 [DISPLAY], page 380, for details.
5. On Android, you can also toggle between fullscreen and non-fullscreen mode by using the options menu in the action bar.

Naturally, there are some limitations in the mobile versions of Hollywood. Most prominently, in contrast to the desktop versions of Hollywood, the iOS version of Hollywood does not support multiple displays but only a single one. Also on iOS, displays cannot use menus. Furthermore, it is not possible to change the mouse pointer because there is typically none on Android and iOS. It is also not possible to use transparent BGPics and the `Desktop` tag in `@DISPLAY` is also unsupported.

Here is a list of commands which are currently not supported by the mobile version of Hollywood:

- `ActivateDisplay() (only unsupported on iOS)`
- `ChangeDisplayMode() (only unsupported on iOS)`
- `CloseDisplay() (only unsupported on iOS)`
- `CreatePointer()`
- `CreatePort()`
- `DebugPrompt()`
- `FontRequest()`
- `FreePointer()`
- `GetEnv()`
- `GetFileArgument()`
- `GetRawArguments()`
- `HideDisplay() (only unsupported on iOS)`
- `HidePointer()`
- `MovePointer()`
- `OpenDisplay() (only unsupported on iOS)`
- `SelectDisplay() (only unsupported on iOS)`
- `SendMessage()`
- `SetPointer()`
- `SetEnv()`
- `ShowDisplay() (only unsupported on iOS)`
- `ShowPointer()`
- `UnsetEnv()`

There are also some functions that are exclusive to the mobile versions of Hollywood. Here is a list of functions which are only available in the mobile versions of Hollywood:

- `CallJavaMethod() (Android only)`
- `GetAsset() (Android only)`
- `HideKeyboard()`
- `ImageRequest() (Android only)`
- `PerformSelector() (iOS only)`
- `PermissionRequest() (Android only)`
- `ShowKeyboard()`
- `ShowToast()`
- `Vibrate() (Android only)`

The Android version of Hollywood also supports Hollywood plugins. You have to copy them to the directory `Hollywood/Plugins` on your SD card. Hollywood will scan this location on every startup and load all plugins from there. The iOS version of Hollywood currently doesn't support plugins.

If your script runs extremely slow on a mobile platform, then you need to change your drawing technique. Refreshing the screen can be quite expensive on a mobile device which is why you should try to minimize the frequency of screen refreshes. The best way to do this is to use double buffered drawing using `BeginDoubleBuffer()` and `Flip()`. Things that will slow down the graphics engine on mobile devices are many drawing operations that affect only small areas. Thus you should try to combine as many drawing operations in just a single call. A double buffer is the best solution for this problem. If you cannot use a double buffer for some reason (for example, because you are using sprites or layers), encapsulate all drawing commands needed for drawing a frame within a `BeginRefresh()` and `EndRefresh()` section. Then the performance should be much better on mobile devices. See Section 30.4 [BeginRefresh], page 591, for details.

3 Console usage

3.1 Console mode

Hollywood can also be used from the console. Actually, the GUIs that ship with the Amiga, Linux, and macOS versions as well as the IDE that ships with the Windows version of Hollywood are just front-ends for the Hollywood interpreter which is a console program. Thus, you can also use Hollywood from the console and of course you can also develop console programs with Hollywood since Hollywood has an extensive console library. Here's an overview of where you can find the Hollywood interpreter that can be started from the console:

- AmigaOS: On AmigaOS and compatibles the Hollywood interpreter is installed in `Hollywood:System/Hollywood`. Since the Hollywood installer adds `Hollywood:System` to your path, you can start the Hollywood interpreter from the console by just entering `Hollywood`.
- Linux: On Linux the Hollywood interpreter is simply called `Interpreter` and can be found in the root directory of your Hollywood installation. There's also a file named `Hollywood` in that directory but that is just the GUI front-end for the Hollywood interpreter. The Hollywood interpreter is simply known as `Interpreter`.
- macOS: On macOS you can find the Hollywood interpreter inside the `Contents/Resources` directory of `Hollywood.app`. The Hollywood interpreter is stored as its own app bundle which is named `HollywoodInterpreter.app`. To start the Hollywood interpreter on macOS from the console, you'd have to do something like this:

  ```
  cd Hollywood.app/Contents/Resources
  ./HollywoodInterpreter.app/Contents/MacOS/Hollywood test.hws
  ```

- Windows: On Windows things are a little more complicated because Windows distinguishes between console programs and non-console programs. Thus, Hollywood is actually available in two flavours on Windows: In a version that is meant to be used from the console and in a version that is meant to be used without a console. The console version of Hollywood is named `Hollywood_Console.exe` and you can find it in the directory where you have installed Hollywood, typically `C:/Program Files/Hollywood`. Additionally, there's also a non-console version of Hollywood available for Windows. This version is named `Hollywood.exe` and it can be found in the very same directory as `Hollywood_Console.exe`.

  The Hollywood IDE will always use the non-console version of Hollywood, i.e. `Hollywood.exe`. If you want to have console output from Hollywood, however, you need to use the console version of Hollywood, i.e. `Hollywood_Interpreter.exe`. You can start that manually from a Windows console like this:

  ```
  cd "C:/Program Files/Hollywood"
  Hollywood_Console.exe test.hws
  ```

  Knowing the difference between console and non-console programs on Windows is also important when it comes to distributing your app: Since Windows distinguishes between console and non-console programs, Hollywood can also compile two different kinds of Windows executables: Windows executables that are console programs and

Windows executables that are non-console programs. If you want to compile a console program for Windows, you need to pass the `-consolemode` argument to the compiler. See Section 3.2 [console arguments], page 33, for details. Note that this option currently isn't available from the Hollywood IDE or the Hollywood GUI front-ends. If you want to compile a Windows console program, you need to run Hollywood from the console or manually add the `-consolemode` argument to the IDE or Hollywood GUI configuration.

Also note that even non-console programs can open a console on Windows. This can be done using the `OpenConsole()` function. See Section 23.44 [OpenConsole], page 349, for details.

Once you know how to start Hollywood from the console, you could then ask it to print a list of all available options by passing the `-help` argument to it. On Windows, this could be done like this:

```
cd "C:/Program Files/Hollywood"
Hollywood_Console.exe -help
```

On Linux like this:

```
cd <Hollywood-installation-directory>
./Interpreter -help
```

On macOS like this:

```
cd /Applications/Hollywood.app/Contents/Resources
./HollywoodInterpreter.app/Contents/MacOS/Hollywood -help
```

And on AmigaOS like this:

```
Hollywood -help
```

Passing `-help` to Hollywood will print a comprehensive list of all available console arguments. See Section 3.2 [Console arguments], page 33, for details. If you omit the `-help` argument, Hollywood will open a file requester prompting you to select a Hollywood script or applet to run.

If you want to start a Hollywood script from the console, you could use the following commands on Windows:

```
cd "C:/Program Files/Hollywood"
Hollywood_Console.exe script.hws
```

The same is possible on the other platforms, see above.

It is important to know that all of Hollywood's features are available from the console as well. After all, the Windows, Amiga, Linux and macOS GUIs for Hollywood are just front-ends for the console-based main program too. Thus, you can do everything from the command line as well. For example, here is how you would ask Hollywood to compile `test.hws` into an AmigaOS3 executable on Linux:

```
cd <Hollywood-installation-directory>
./Interpreter test.hws -compile ~/MyTest_AmigaOS3 -exetype classic
```

See Section 3.2 [Console arguments], page 33, for a detailed description of all command-line parameters.

Note that on AmigaOS and compatible systems Hollywood is automatically added to your path upon installation. Thus, you can simply type `Hollywood` in the console and Hollywood will be started - no matter where you installed the program.

3.2 Console arguments

If you do not want to use the GUI for some reason, you can also start Hollywood from the console. When you start Hollywood from the console without any arguments, a file requester will be opened prompting you to select a script or applet to run. For a list of supported arguments, start Hollywood using the `-help` argument. The template for using Hollywood from the console is as follows:

```
Hollywood file.hws [-alldisplays] [-askdisplaymode] [-audiodevice name]
[-autofullscreen] [-autoscale] [-backfill type] [-borderless]
[-brush brush] [-brushfile file] [-compile file] [-compress]
[-consolemode] [-cxkey hotkey] [-debugoutput] [-depth depth]
[-disableblanker] [-dpiaware] [-encoding enc] [-endcolor color]
[-exetype type] [-exportcommands file] [-exportconstants file]
[-exporthelpstrings] [-exportplugins file] [-exportpreprocs file]
[-fakefullscreen] [-fixed] [-fitscale] [-forceflush] [-forcesound]
[-formaterror] [-fullscreen] [-fullscreenscale] [-globalplugins] [-help]
[-hideoptionsmenu] [-hidepointer] [-hidetitlebar] [-icon16x16 file]
[-icon24x24 file] [-icon32x32 file] [-icon48x48 file] [-icon64x64 file]
[-icon96x96 file] [-icon128x128 file] [-icon256x256 file]
[-icon512x512 file] [-icon1024x1024 file] [-keepproportions]
[-keepscreenon] [-layerfullscreen] [-layerscale] [-legacyaudio]
[-linkfiles file] [-linkfonts file] [-linkplugins list] [-locksettings]
[-mastervolume vol] [-maximized] [-moderequester] [-monitor num]
[-nativeunits] [-nobackfill] [-nochdir] [-nocommodity] [-nodebug]
[-nodocky] [-nohardwarescale] [-nohide] [-nokeepproportions]
[-nolegacyaudio] [-noliveresize] [-nomodeswitch] [-nomousehook]
[-noscaleengine] [-noscaleswitch] [-nosmoothscale] [-nosound]
[-nostyleoverride] [-numchannels chans] [-overrideplacement]
[-overwrite] [-pictrans transparency] [-picxpos x] [-picypos y]
[-printerror] [-pubscreen name] [-quiet] [-requireplugins list]
[-requiretags tags] [-resourcemonitor] [-scalefactor s] [-scalepicture]
[-scaleswitch] [-scalewidth width] [-scaleheight height]
[-scrwidth width] [-scrheight height] [-setconstants list] [-sizeable]
[-skipplugins mask] [-smoothscale] [-softtimer] [-softwarerenderer]
[-startcolor color] [-stayactive] [-systemscale] [-tempdir path]
[-usequartz] [-usewpa] [-videofps fps] [-videoout file] [-videopointer]
[-videoquality quality] [-videostrategy strategy] [-vsync] [-window]
[-winwidth width] [-winheight height] [-wpamode mode] [-xserver name]
```

Important information: The majority of these arguments are also supported by every program compiled by Hollywood. If you do not want your compiled programs to support command line arguments, you have to compile your scripts using the `-locksettings` argument (see below for a description).

Important information #2: Most of the arguments listed above can also be passed to Hollywood programs without using a console. See Section 3.3 [Arguments without console], page 54, for details.

Here are detailed descriptions for every command:

`-alldisplays:`
: By default, command line arguments like `-borderless` or `-sizeable` will only affect the first display. If you pass the parameter `-alldisplays`, all command line arguments controlling the display style will be applied to all Hollywood displays.

`-askdisplaymode:`
: If you specify this command line argument, Hollywood will pop up a requester asking the user to select whether the script should be shown in windowed or full screen mode.

`-audiodevice name:`
: This argument can be used to specify the ALSA sound device that Hollywood should use for audio output. You only need to pass this argument if you want to use a sound device that is different from the default ALSA sound device. So normally, you do not have to pass this argument at all. [Linux only]

`-autofullscreen:`
: This will put the display into full screen mode using the auto scaling engine instead of changing the monitor's resolution but only when running Hollywood on systems that support GPU-accelerated scaling. On all other platforms a normal full screen mode will be used, i.e. Hollywood will change the monitor's resolution to fit the current display dimensions. Currently, GPU-accelerated scaling is supported on Windows, macOS, Android, and iOS which means that on those platforms no monitor resolution change will occur because Hollywood can simply scale the graphics to fit to the current monitor dimensions. On AmigaOS compatibles and Linux, however, there will still be a monitor resolution change with this mode because Hollywood doesn't support GPU-accelerated scaling on those platforms.

`-autoscale:`
: If you specify this argument, the auto scaling engine will be activated. This means that your script can be displayed in any resolution that you define and it works completely automatically - you do not have to make any changes to your code. If auto scaling is enabled, Hollywood will pretend to your script that it is still running in its usual resolution but in reality it will get upscaled or downscaled (depending on the chosen scaling resolution). You can specify the initial auto scaling resolution by using the `-scalewidth` and `-scaleheight` arguments, or the `-scalefactor` or the `-systemscale` argument. The scaling resolution can be changed by the user at any time by resizing the Hollywood window (don't forget to make your window resizeable by using the `@DISPLAY preprocessor command` or the `-sizeable` argument). If you do not specify `-scalewidth` and `-scaleheight` or `-scalefactor` or `-systemscale` at startup, the script will be started without auto scaling, but auto scaling will be activated as soon as the user resizes the window. If you want anti-aliased auto scaling (slower), specify the `-smoothscale` argument. Hollywood supports another scaling engine which can be activated by specifying the `-layerscale` argument (see below). See Section 25.18 [Scaling engines], page 401, for details.

`-backfill type:`
: This argument allows you to specify a backfill type for Hollywood's display. If you specify this argument, Hollywood will fill the whole screen. `type` can be one of the following keywords:

  `color` Fill the background with the color specified in the `-startcolor` argument

  `picture` Display the brush specified in the `-brush`/`-brushfile` argument as background picture (centered); if you specify the `-startcolor` argument also, the background will be cleared with that color; if you specify `-endcolor` too, the background will be cleared with a gradient between `-startcolor` and `-endcolor`.

  `gradient` Display a gradient as the background (with a fade from the color specified in the `-startcolor` argument to the color specified in the `-endcolor` argument).

  `texture` Display the brush specified in the `-brush`/`-brushfile` argument as a texture.

  As you can see, all the backfill types require an additional argument as a parameter. You have to use the arguments `-brush`, `-brushfile`, `-startcolor`, and `-endcolor` for this (as documented above).

`-borderless:`
: If you specify this argument, Hollywood will open its window without borders. This is especially useful for transparent windows.

`-brush id:`
: Only required in connection with `-backfill` set to `texture` or `picture`. Specifies the identifier of the brush to use with the backfill mode. You can also use `-brushfile` instead of this argument (if you want to use a brush for backfilling that hasn't been declared in the script).

`-brushfile file:`
: Only required in connection with `-backfill` set to `texture` or `picture`. Specifies the file name of the brush to use with the backfill mode. You can also use the `-brush` argument instead of this one.

`-compile file:`
: If you specify this argument, Hollywood will compile your script to a standalone executable. You will have to use this option if you want to publish your script. Your script will not be executed. It will be compiled and saved to `file`. Use the `-exetype` argument to specify the platform for which you want to save your script.

`-compress:`
: You can use this switch to enable compression of Hollywood projects. If this argument is specified, Hollywood will compress applets and executables. This argument can only be used in connection with `-compile`.

`-consolemode:`
: If you specify this argument, Hollywood will compile an executable that runs in console mode on Windows. On Windows, there is a distinction between console and non-console programs so if you want to compile a program for the console, you will explicitly have to tell Hollywood to do so. You can do that by passing this argument. Note that this argument is obviously only handled when `-compile` is specified as well. Note that `-consolemode` is also available in the `@OPTIONS` preprocessor command. See Section 52.25 [OPTIONS], page 1088, for details. See Section 3.1 [Console mode], page 31, for details.

`-cxkey hotkey:`
: This argument can be used to install the specified key combination as a system-wide hotkey for your application. Whenever the user presses the specified key combination, your application will get a `Hotkey` event which you can listen to through the `InstallEventHandler()` function. [Amiga OS only]

`-debugoutput:`
: Specifying this argument enables debug output for this script. This console argument has the same effect as calling `DebugOutput()` at the beginning of your script.

`-disableblanker:`
: This argument can be used to disable the screen blanker while Hollywood is running.

`-dpiaware:`
: This argument is only supported on Windows. If you pass this argument, Hollywood will start in DPI-aware mode. This means that it will not ask the OS to automatically scale Hollywood to fit to the monitor's DPI. If `-dpiaware` is not specified, Hollywood will automatically apply scaling on high-DPI monitors so that its display doesn't appear too small on them. For example, a display of 640x480 pixels will appear really tiny on a high-DPI monitor. By automatically adapting displays to the monitor's DPI, Hollywood will try to avoid this. However, that scaling can make displays appear blurry on high-DPI monitors. So if you don't want that, pass the `-dpiaware` argument. Note, however, that you'll need to take care of making sure that your display appears correctly on high-DPI monitors then. You can do this by setting the `SystemScale` tag in the `@DISPLAY` preprocessor command, for example. Note that `-dpiaware` is also available in the `@OPTIONS` preprocessor command. See Section 52.25 [OPTIONS], page 1088, for details.

`-encoding enc:`
: This argument can be used to set the script's character encoding. `Enc` can be one of the following:

  `utf8:` Script's character encoding is UTF-8 (with or without BOM). This is also the default and should be used whenever and whereever possible.

`iso8859_1`:
: Script's character encoding is ISO 8859-1. Note that due to historical reasons Hollywood will not use ISO 8859-1 character encoding on AmigaOS and compatibles but whatever is the system's default character encoding. `iso8859_1` will put Hollywood in legacy mode and should make your script fully compatible with Hollywood versions older than 7.0. However, since ISO 8859-1 mode has several drawbacks, it isn't recommended to use this legacy mode permanently. Instead, you should adapt your scripts to work correctly in Unicode mode.

Note that it isn't recommended to use `iso8859_1` because Hollywood will only run correctly on locales compatible with Western European languages then. You should always use `utf8` because this will put Hollywood in Unicode mode and make sure that Hollywood runs correctly on all locales.

The encoding you specify here is automatically set as the default encoding for both the text and string library using `SetDefaultEncoding()`. This means that all functions of the string and text libraries will default to this encoding.

`-endcolor color:`
: Only required in connection with `-backfill` set to `gradient`. `color` is a color specified in RGB format (e.g. $FF0000 for red). Can also be specified with `-backfill` set to `picture`. This will create a gradient behind the picture then.

`-exetype type:`
: Only required in connection with `-compile`. This argument specifies the output format of the executable that the Hollywood compiler shall create. `Type` can be one of the following:

`amigaos4` AmigaOS 4 executable (PowerPC)

`android` Hollywood applet which has the platform-specific constants for Android set (see below).

`aros` AROS executable (x86)

`classic` AmigaOS 3.x executable (68020+)

`classic881`
: AmigaOS 3.x executable (68020+) with math co-processor (68881/2 or 68040/68060)

`ios` Hollywood applet which has the platform-specific constants for iOS set (see below).

`linux` Linux executable (x86)

`linux64` Linux executable (x64)

`linuxarm` Linux executable (arm)

`linuxppc` Linux executable (PowerPC)

`macos` macOS application bundle (PowerPC)

`macosarm64`
: macOS application bundle (arm64)

`macos86`
: macOS application bundle (x86)

`macos64`
: macOS application bundle (x64)

`morphos`
: MorphOS executable (PowerPC)

`warpos`
: WarpOS mixed-binary executable (68040/PowerPC)

`win32`
: Windows executable (x86)

`win64`
: Windows executable (x64)

`applet`
: Universal Hollywood applet which can be started on any system with a Hollywood Player

This argument defaults to `classic` in the 68k version of Hollywood. In the 32-bit Windows version it defaults to `win32` and so on.

Note that the targets `applet`, `android`, and `ios` will all compile platform-independent applets that can be run with the Hollywood Player on any platform. The difference between `applet` and `android` and `ios` is that when you compile for `android` or `ios`, Hollywood will set the respective platform-specific constants, i.e. `#HW_ANDROID` for Android and `#HW_IOS` for iOS. If you specify `applet` as the target, however, none of the platform-specific constants will be set. See Section 52.17 [IF], page 1080, for details. Of course, applets compiled using `applet` will work on Android and iOS as well. The scripts just won't know that they are being compiled for Android or iOS. This can only be detected if you specifically pass `ios` or `android` as the build target. Conversely, applets compiled using `android` or `ios` will also run on non-Android and non-iOS devices. The only difference between `applet`, `android`, and `ios` really is just related to the platform-specific constants. See Section 52.17 [IF], page 1080, for details.

You can also compile for multiple platforms at once. In that case, you have to pass several platform names separated by a vertical bar character (`|`). For example, to compile `test.hws` for AmigaOS 3 and MorphOS, use the following call:

```
Hollywood test.hws -compile test -exetype classic|morphos
```

If you specify multiple target platforms, the output file name specified to `-compile` is regarded as a template and will get platform specific extensions. (i.e. the call above will generate a series of executables named `test_OS3` and `test_MOS`)

`-exportcommands file:`
: This argument can be used to export a list of available commands into the specified file. The list of available commands will be sorted by libraries. Inside the library sections the lists will be unsorted. Only native Hollywood commands are exported. Commands installed by plugins will not be listed here. You can get these by using the `-exportplugins` argument. This option is probably not of much use for normal users but it can be helpful for authors of IDEs who would like to integrate Hollywood into their programming environment.

`-exportconstants file:`
: This argument can be used to export a list of available constants into the specified file. The list of available constants will be entirely unsorted. Only native Hollywood constants are exported. Constants installed by plugins will not be listed here. You can get these by using the `-exportplugins` argument. This option is probably not of much use for normal users but it can be helpful for authors of IDEs who would like to integrate Hollywood into their programming environment.

`-exporthelpstrings:`
: If this argument is used together with the `-exportplugins` argument, Hollywood will write three lines instead of one line for every plugin command to the file specified in the `-exportplugins` argument. The first line will be the command's name, the second line will be its help text and the third line will be the command's help node in the accompanying documentation for the plugin. This information is useful for IDEs which would like to provide help for plugin commands. Note that both the second and the third line can be empty if the plugin doesn't export a help string or a help node for the command.

`-exportplugins file:`
: This argument can be used to export a list of available plugins into the specified file. If a plugin exports commands and/or constants, these will also be appended to the export file. This option is probably of not much use for normal users but it can be helpful for authors of IDEs who would like to integrate Hollywood into their programming environment. If you also specify the `-exporthelpstrings` argument (see above), Hollywood will export the help texts and nodes for all plugin commands as well. See above for details.

`-exportpreprocs file:`
: This argument can be used to export a list of all preprocessor commands supported by Hollywood into the specified file. The list will be entirely unsorted and the individual preprocessor commands won't contain the at prefix. This option is probably not of much use for normal users but it can be helpful for authors of IDEs who would like to integrate Hollywood into their programming environment.

`-fakefullscreen:`
: This argument allows you to put Hollywood into fake full screen mode. This means that Hollywood will open on the desktop but the backfill window will be configured to shield the desktop completely. Thus, the user gets the impression as if Hollywood was running full screen, although it is running on the desktop.

`-fitscale:`
: This argument is only handled when either `-layerscale` or `-autoscale` is active. In that case, `-fitscale` will set the scaling resolution to the current screen's resolution so that the script will always fill out the whole screen. Using `-fitscale` is basically the same as passing the current screen's dimensions in `-scalewidth`/`-scaleheight`. But you cannot know the screen resolution on your user's computers and that is why `-fitscale` is here to do this job. Note that using `-fitscale` might distort the appearance of your script in case

the current screen resolution uses a different aspect-ratio than your script. To prevent distortion, you have to use `-keepproportions` (see below) alongside `-fitscale`.

`-fixed:` If you specify this argument, Hollywood's display will be fixed on the screen which means that you cannot move it. This is useful when Hollywood opens in full screen mode.

`-forceflush:`
Specify this argument to force a buffer flush after every single line that Hollywood writes to the debug device. This is only useful when the debug device is a file or a pipe because consoles always flush buffers after every line anyway.

`-forcesound:`
Normally, when a script tries to play a sound and the audio hardware cannot be allocated, Hollywood will continue running normally, just without sound. If you don't want that, i.e. if you want Hollywood to fail in case the audio hardware cannot be allocated, pass this argument. In that case Hollywood will throw an error in case the audio hardware cannot be allocated.

`-formaterror:`
This argument tells Hollywood to format its error messages in a certain way so that they can be easily distinguished from other console output. Note that this argument is only handled when the `-printerror` option is active as well. In that case, errors are logged to the console like this:

```
@_hwerror<line>:<file>*<message>
```

Note that `<file>` may be enclosed by double quotes and it may be empty if the error is not related to a script file. In that case, `<line>` will be 0.

`-fullscreen:`
This argument will run Hollywood in full screen mode. It will scan your monitor's display modes to determine the best resolution for your script and will then switch the monitor's display mode into that resolution and run the script. If you would like to run your script in full screen mode without switching the monitor resolution, use the `-fullscreenscale` mode instead (see below).

`-fullscreenscale:`
This is a special full screen mode which won't change your monitor's resolution. Instead, Hollywood's display will be resized to fit your monitor's dimensions. Additionally, this full screen mode will activate the auto scaling engine so that your display is automatically scaled to fit your monitor's dimensions. `-fullscreenscale` will use auto scaling by default. If you would like it to use layer scaling, you have to pass the `-layerscale` option as well. `-fullscreenscale` is especially useful on mobile devices whose display hardware has a hard-coded resolution and doesn't support resolution changes in the same way as an external monitor connected to a desktop computer does. The downside of `-fullscreenscale` is that it is slower because Hollywood has to scale all rendering operations to the monitor's dimensions.

`-globalplugins:`
: On AmigaOS and compatibles, plugins can also be globally installed in `LIBS:Hollywood`. Executables compiled by Hollywood, however, will only load the plugins that are stored next to the executable in its directory. If you want your executable to load all plugins in `LIBS:Hollywood` as well, you can specify this argument. Alternatively, you can also set `GlobalPlugins` to `True` in the `@OPTIONS` preprocessor command. [AmigaOS only]

`-help:`
: Print a list of supported console arguments.

`-hideoptionsmenu:`
: When the user opens the options menu on Android devices, Hollywood will allow the user to configure several display parameters like enabling or disabling autoscaling or layerscaling. If you do not want to give the user this possibility to change the display parameters via the app's options menu, pass this argument to Hollywood. [Android only]

`-hidepointer:`
: If you specify this argument, the mouse pointer will automatically be hidden as soon as Hollywood enters full screen or fake full screen mode. This argument has the advantage over the `HidePointer()` command in that it only hides the mouse pointer in full screen mode. If Hollywood opens in windowed mode, the mouse pointer will remain visible because hiding the mouse pointer in windowed mode usually causes confusion with the user.

`-hidetitlebar:`
: This argument hides the title bar of the host screen. On desktop systems this argument is only effective when Hollywood opens on its own screen or when you use the `-backfill` option. On mobile devices this option will hide the status bar (iOS) or action bar (Android). [Amiga OS, macOS, iOS and Android only]

`-icon16x16 file:`
: This and all the other `-iconXXX` console arguments can be used to specify the icons for your application. On Windows, macOS, and Linux these icons will appear in the window's border and they will also be used by certain elements of the window manager like the task bar on Windows. The icons will also be linked into any applets or executables you compile with Hollywood. By default, Hollywood will always use the standard Hollywood icon (the clapperboard). If you prefer to use your own icon instead, you can do so by specifying one or more of these arguments. For the best results, you should use multiple icons handcrafted for all individual sizes. Hollywood currently supports these icon sizes: 16x16, 24x24, 32x32, 48x48, 64x64, 96x96, 128x128, 256x256, 512x512, and 1024x1024. Not all sizes are currently supported on all platforms but you should make sure to provide icons for all these sizes. If you leave a size out, Hollywood might fall back to its default icon (clapperboard) for the size. So if you intend to use your own icons, make sure that you always provide it in all sizes. The image file that is required as a parameter by these arguments should be a PNG image with alpha channel. Images without alpha channel are supported as well, but this is not recommended because it doesn't look too

good. Alternatively, you can also use the `@APPICON` preprocessor command to specify custom icons for your project.

`-icon24x24 file:`
: Same as `-icon16x16` but embeds an icon of size 24x24.

`-icon32x32 file:`
: Same as `-icon16x16` but embeds an icon of size 32x32.

`-icon48x48 file:`
: Same as `-icon16x16` but embeds an icon of size 48x48.

`-icon64x64 file:`
: Same as `-icon16x16` but embeds an icon of size 64x64.

`-icon96x96 file:`
: Same as `-icon16x16` but embeds an icon of size 96x96.

`-icon128x128 file:`
: Same as `-icon16x16` but embeds an icon of size 128x128.

`-icon256x256 file:`
: Same as `-icon16x16` but embeds an icon of size 256x256.

`-icon512x512 file:`
: Same as `-icon16x16` but embeds an icon of size 512x512.

`-icon1024x1024 file:`
: Same as `-icon16x16` but embeds an icon of size 1024x1024.

`-keepproportions:`
: This argument is only handled when either `-layerscale` or `-autoscale` is active. In that case, `-keepproportions` will not distort the resolution of the current script when the user resizes the window. Instead, black borders will be used to pad the non-proportional window regions. The display itself will always keep its aspect-ratio. This is very useful for scripts that should not be distorted.

`-keepscreenon:`
: If you specify this argument, battery saving mode will be disabled on mobile devices. This means that the device's screen will never be dimmed or turned off to save energy. Useful for scripts that do not require user input. [Android and iOS only]

`-layerfullscreen:`
: This will put the display into full screen mode using the layer scaling engine instead of changing the monitor's resolution but only when running Hollywood on systems that support GPU-accelerated scaling. On all other platforms a normal full screen mode will be used, i.e. Hollywood will change the monitor's resolution to fit the current display dimensions. Currently, GPU-accelerated scaling is supported on Windows, macOS, Android, and iOS which means that on those platforms no monitor resolution change will occur because Hollywood can simply scale the graphics to fit to the current monitor dimensions. On AmigaOS compatibles and Linux, however, there will still be a monitor resolution

change with this mode because Hollywood doesn't support GPU-accelerated scaling on those platforms.

`-layerscale:`
: If you specify this argument, the layer scaling engine will be activated. This means that your script can be displayed in any resolution that you define and everything is done completely automatically - you do not have to make any changes to your code. However, as the very name implies, the layer scaling engine will only work if layers are enabled. In layer scaling mode, all layers will automatically be adapted to the new resolution and Hollywood will pretend to your script that it is still running in its original resolution to make sure that your script is executed in exactly the same way as without layer scaling. The advantage of layer scaling is that vector layers (i.e. graphics primitives, true type text, vector brushes, vector anims) will be scaled in vector mode so that there won't be any loss of quality even if you change the resolution of your script from 320x240 to 1280x1024. You can specify the initial layer scaling resolution by using the `-scalewidth` and `-scaleheight` arguments or the `-scalefactor` or the `-systemscale` arguments (see below). The scaling resolution can be changed by the user at any time by resizing the Hollywood window (don't forget to make your window resizeable by using the `@DISPLAY preprocessor command` or the `-sizeable` argument). If you do not specify `-scalewidth` and `-scaleheight` or `-scalefactor` or `-systemscale` at startup, the script will be started without layer scaling, but layer scaling will be activated as soon as the user resizes the window. If you want anti-aliased layer scaling (slower), specify the `-smoothscale` argument. Hollywood supports another scaling engine which can be activated by specifying the `-autoscale` argument (see above). See Section 25.18 [Scaling engines], page 401, for details.

`-legacyaudio:`
: Starting with Hollywood 6.0, the AmigaOS versions of Hollywood come with a new audio driver. The old audio driver is still supported and can be enabled by specifying this command line option. Please note that on Amiga OS 3.x the old audio driver is enabled by default due to performance reasons. If you would like to use the new audio driver on AmigaOS 3.x too, you have to pass the `-nolegacyaudio` console argument. [AmigaOS only]

`-linkfiles file:`
: This argument is only handled when `-compile` is also specified. You can use this argument to link files into your applet or executable. Your script will then automatically load these files from your applet or executable. This is an alternative for using the preprocessor commands to link files into your applet or executable. If you do not want to use preprocessor commands to link files into your applet or executable, use `-linkfiles` for that. Alternatively, you can also use the `@LINKER` preprocessor command to specifiy a list of files to be linked into your applet or executable. See Section 4.3 [Linking data files], page 58, for details.

`-linkfonts file:`
: This argument is only handled when `-compile` is also specified. You can use this argument to link fonts into your applet or executable. Your script will then automatically load these fonts from your applet or executable when you call `SetFont()`. Using `-linkfonts` is an alternative to using the `@FONT` preprocessor command to link fonts into your applet or executable. Normally, using `@FONT` should be much easier than using `-linkfonts` so you should use the latter only with good reasons. Alternatively, you can also use the `@LINKER` preprocessor command to specifiy a list of fonts to be linked into your applet or executable. See Section 4.4 [Linking fonts], page 60, for details.

`-linkplugins list:`
: This argument is only handled when `-compile` is also specified. You can use this argument to link plugins into your executable. Your executable will then automatically load these plugins on startup and you have everything in a single file. It is no longer necessary to store or install plugins externally if you link them to your executable. Note that plugins can only be linked to executables, not to applets, since applets are platform-independent and plugins are not. You have to pass a list of plugins that should be linked to your executable to this argument. If you want to link more than one plugin, separate the individual plugins by using a vertical bar character (`|`). For example, to link your script against `"plugin1.hwp"` and `"plugin2.hwp"` you would have to specify `"plugin1|plugin2"` here. Make sure to carefully read the licenses of all plugins you link to your executable because licenses like LGPL affect your project if you statically link against LGPL software. Note that before you can use the `-linkplugins` argument, you first have to set up the infrastructure for the plugin linker. See Section 4.5 [Linking plugins], page 61, for details.

`-locksettings`
: This argument is only handled when `-compile` is also specified. You can use this argument to fix your script's display settings. Normally, when you compile an executable with Hollywood, the user will be able to change the appearance of it by passing arguments like `-borderless` or `-fullscreen` to the executable. The user could also change the backfill settings of the executable by specifying `-backfill` etc. By default, the user is given all flexibility to adjust your program to his wishes. He could also enable a scaling engine or make your program sizeable. If you do not want the user to be able to change your display settings, you will have to compile your executables using `-locksettings`. If `-locksettings` is used, Hollywood will always use the settings specified in your script's `@DISPLAY` preprocessor command. For example, if you specify `Mode=FullScreen` in your script's `@DISPLAY` command, and you compile using `-locksettings`, then the user will not be able to run your program in windowed mode. Your program will always open in full screen mode. Think twice before using `-locksettings` because it impedes the flexibility of your programs.

`-mastervolume vol:`
: This argument allows you to specify the master volume Hollywood shall use. Use this only if you experience distortion when Hollywood plays sounds. Nor-

mally, you do not have to use an other value here. Master volume can range from 0 to 64. [AmigaOS only]

`-maximized:`
: Open the display in maximized mode. This display has to be sizeable for this parameter to take effect. [Windows only]

`-moderequester:`
: If you specify this command line argument together with the `-fullscreen` argument, Hollywood will pop up a requester prompting the user to select a monitor resolution for the full screen mode. Hollywood will then show your script in full screen mode using the display mode just chosen by the user.

`-monitor num:`
: This argument allows you to specify the monitor your script's display should be opened on. Monitors are counted from 1 which is the primary monitor.

`-nativeunits:`
: If this console argument is specified, Hollywood will use the host system's native coordinate space and units instead of pixels. This currently only has an effect on macOS and iOS because both operating systems use custom units instead of pixels when running on a Retina device. By default, Hollywood will enforce the use of pixels on Retina Macs and iOS devices for cross-platform compatibility reasons but you may want to override this setting by using this console argument.

`-nobackfill:`
: By default, Hollywood will always install a backfill for your display if it is opened in full screen mode. If you don't want this, specify this option. Hollywood will open in full screen mode then but it won't shield the areas that are not covered by the display itself. This is useful on Amiga systems if you'd like Hollywood to open on a new screen without hiding the screen's visuals like its title bar and its background decoration from the user. If you use this argument, you might also want to use the `-nostyleoverride` argument. [AmigaOS only]

`-nochdir:`
: By default, Hollywood will always change the current directory to the directory of the script or applet it is currently running. Pass this argument if you don't want this behaviour. In that case, Hollywood won't change the current directory when running a script.

`-nocommodity:`
: On AmigaOS systems, if this argument is specified Hollywood will not add itself to the system's list of commodities. [AmigaOS only]

`-nodebug:`
: If this argument is specified, the commands `DebugPrint()`, `DebugPrintNR()`, `Assert()`, `DebugOutput()` and `@WARNING` will be skipped when running the script or applet. This allows you to globally disable debugging functions with just a single call.

`-nodocky:`
: On AmigaOS 4 systems, if this argument is specified, Hollywood will not show the application in AmiDock. This tag is useful if you would like to have an invisible application that can use all the application functionality like the message mechanism and Ringhio but doesn't appear in AmiDock. This tag is only recognized if `RegisterApplication` has been set to `True` in `@OPTIONS`. [AmigaOS 4 only]

`-nohardwarescale:`
: For performance reasons Hollywood will try to use hardware accelerated scaling when autoscaling is enabled on Android devices by default. Some devices, however, do not implement hardware accelerated scaling properly so if you experience strange behaviour when using autoscale mode, try to disable hardware accelerated scaling using this argument and see if it helps. This is obsolete since Hollywood 8.0. Hollywood will always use hardware-accelerated scaling now. [Android only]

`-nohide:` If you specify this argument, the user will not able to hide the Hollywood display, i.e. the Hollywood window will not have any minimize button. This argument does not affect the `ShowDisplay()` and `HideDisplay()` commands. You can still hide and show the display using these commands.

`-nokeepproportions:`
: When the user switches between windowed and full screen mode using the `CMD+RETURN` hotkey (on Windows it is `LALT+RETURN`) and Hollywood chooses to scale the display to the current monitor's resolution, it will add padding borders if necessary to keep the display's proportions. If you don't want that, pass this argument.

`-nolegacyaudio:`
: Starting with Hollywood 6.0, the AmigaOS versions of Hollywood come with a new audio driver but this new driver is not enabled on AmigaOS 3.x by default due to performance reasons. If you would like to use the new audio driver on AmigaOS 3.x too, pass this console argument. [AmigaOS only]

`-noliveresize:`
: On many platforms Hollywood will use live resizing when the user is resizing a display. This means that the display's contents will be scaled while the user is resizing the display. If you don't want this, you can set this console argument.

`-nomodeswitch:`
: If you specify this argument it will not be possible to switch between windowed and full screen mode by pressing the `CMD+RETURN` hotkey (on Windows it is `LALT+RETURN`). If `-nomodeswitch` is specified, Hollywood will always remain in its initial display mode and no switches between windowed and full screen will be made.

`-nomousehook:`
: If you specify this argument, Hollywood won't install a hook that constantly polls the mouse position. This is only useful on Linux if the connection to the X Server is quite slow. If that is the case, using `-nomousehook` might give

you a performance boost. The downside of using this option is that you will no longer be notified about `OnMouseMove` events if they occur outside the window's boundaries because this notification only works with a mouse hook. [Linux only]

`-noscaleengine:`
This console argument is only handled if you pass the `-fullscreenscale` argument as well. In that case Hollywood will not use any scaling engine but will simply open your display in the same dimensions as the monitor's resolution. Your script then needs to manually adapt to the monitor's resolution. This allows you to write scripts which can dynamically adapt to different resolutions without simply scaling their graphics.

`-noscaleswitch:`
When the user switches between windowed and full screen mode using the `CMD+RETURN` hotkey (on Windows it is `LALT+RETURN`), Hollywood might choose to scale the display to the monitor's current resolution instead of switching the monitor's physical resolution. If you don't want Hollywood to simulate full screen mode by just scaling the display to the monitor's current resolution, pass this console argument. In that case, pressing the mode switch hotkey will always change the monitor's physical resolution.

`-nosmoothscale:`
When the user switches between windowed and full screen mode using the `CMD+RETURN` hotkey (on Windows it is `LALT+RETURN`) and Hollywood chooses to scale the display to the current monitor's resolution, it will use anti-aliased interpolation for smoother scale results by default. If you don't want that, pass this argument.

`-nosound:`
This argument disables all sound functions of Hollywood. Hollywood will start in mute mode.

`-nostyleoverride:`
If Hollywood runs your script in full screen mode it will automatically modify your display's window decoration style and make the window fixed and borderless. If you don't want this, you can use this argument to force Hollywood to leave window styles untouched. This argument is mostly used together with the `-nobackfill` argument. [AmigaOS only]

`-numchannels chans:`
By default, Hollywood allocates 8 audio channels for sound playback. This means that Hollywood will run out of channels in case your script tries to play more than 8 different samples, music objects, or video streams at a time. If your script needs more than 8 channels for some particular reasons, you can use this argument to tell Hollywood how many channels it should allocate.

`-overrideplacement:`
If this argument is specified, Hollywood will ignore any saved position or size information for displays that have the `RememberPosition` tag set to `True`. Instead, those displays will always use their default position and size.

`-overwrite:`
: If you specify this argument, Hollywood will automatically overwrite existing files when `-compile` is used. Normally, Hollywood will ask you to confirm overwriting existing files in `-compile` mode. You can suppress the compulsory confirmation by specifying this argument.

`-pictrans transparency:`
: Only possible with `-backfill` set to `picture`. This argument allows you to assign a transparency color to your picture. Defaults to `#NOTRANSPARENCY`.

`-picxpos x:`
: Only possible with `-backfill` set to `picture`. You can use this argument to specify the position where the backfill picture shall be displayed. Defaults to `#CENTER`.

`-picypos y:`
: Only possible with `-backfill` set to `picture`. You can use this argument to specify the position where the backfill picture shall be displayed. Defaults to `#CENTER`.

`-printerror:`
: If this argument is specified, Hollywood won't show script errors in a dialog box, but will simply print them to the console. This can be useful when integrating Hollywood into IDEs. If you need to parse Hollywood's error messages, you should also specify the `-formaterror` argument to force Hollywood to output errors in a format that can be parsed, i.e. split into its individual constituents like script file, line number, error message.

`-pubscreen name:`
: If specified, Hollywood will open on the public screen specified by name instead of the desktop screen. [AmigaOS only]

`-quiet:` If you specify this argument, Hollywood will not display any information during its startup.

`-requireplugins list:`
: This argument allows you to specify a list of plugins that your script explicitly requires. If you need to specify more than one plugin, separate the individual plugins by using a vertical bar character (`|`). For example, to make your script require `"plugin1.hwp"` and `"plugin2.hwp"` you would have to specify `"plugin1|plugin2"` here. Using this preprocessor command has the same effect as using the `@REQUIRE` preprocessor command. See Section 45.6 [REQUIRE], page 930, for details. If you need to pass additional arguments to the plugin's initialization routine, use the `-requiretags` console argument.

`-requiretags tags:`
: This console argument allows you to pass additional arguments to the initialization routine of plugins. Additional arguments for plugin initialization are normally passed to the plugin by using the `@REQUIRE` preprocessor command but you can also pass them from the command line using this argument. This is especially useful for testing purposes because you won't have to modify your script

all the time if you pass additional initialization arguments via `-requiretags`. You can pass additional arguments to more than one plugin. The format of the string you pass to this argument must be like this:

```
name1[tag1=value1,...,tagN=valueN]name2[...]...nameN[...]
```

Here is an example:

```
-requiretags testplugin[User='admin',Pwd='secret',Len=64]
```

The command line above would pass three additional tags to the the initialization routine of the plugin `testplugin.hwp`, namely `User`, `Pwd`, and `Len`. `User` is set to "admin", `Pwd` to "secret", and `Len` is passed as an integer value of 64. Please note that you also have to use `-requireplugins` if you use `-requiretags` console argument. Otherwise, the plugins' initialization code won't be executed at all.

`-resourcemonitor:`
: Specifying this argument will open Hollywood's resource monitor right at the start of your script. The resource monitor is useful to keep track of your resources while developing your script. Please read the documentation of `OpenResourceMonitor()` for more information on this topic.

`-scalefactor s:`
: This argument can be used in connection with either `-autoscale` or `-layerscale` to apply a global scaling factor to your whole script. The scaling factor must be specified as a fractional number indicating the desired scaling coefficient, e.g. a value of 0.5 shrinks everything to half of its size whereas a value of 2.0 scales everything to twice its size. Note that setting `-scalefactor` will make the script behave slightly different than setting `-scalewidth` and `-scaleheight` does. The latter will enforce a fixed display size for the script which will never be changed unless the user manually uses the mouse to change the display size. Setting `-scalefactor`, however, will apply the scale factor to all new BGPics and display sizes so the display size may change if the BGPic size changes or the script changes the display size. Thus, using `-scalefactor` is perfect for scaling a script for a high dpi display because it makes sure that the script behaves exactly the same but just appears larger (or smaller if you want!). You can also use the `-systemscale` argument to automatically apply the host system's scaling factor to your display (see below). Please also read the documentation of `-autoscale`/`-layerscale` for more information on the Hollywood scaling engines. See Section 25.18 [Scaling engines], page 401, for details.

`-scalepicture:`
: Only possible if `-backfill` is set to `picture`. This argument tells Hollywood to scale the specified backfill picture to the actual size of the backfill window so that it fills it completely.

`-scaleswitch:`
: When the user switches between windowed and full screen mode using the `CMD+RETURN` hotkey (on Windows it is `LALT+RETURN`), Hollywood might not change the monitor's screen mode but just simulate full screen mode by scaling

the display to the monitor's current resolution. This is only done if the system Hollywood is running on supports hardware-accelerated scaling. On older systems or platforms that don't support hardware-accelerated scaling Hollywood will switch the monitor to the new resolution instead. If you want Hollywood to always simulate full screen mode by just scaling the display to the monitor's current resolution instead of changing its physical mode, pass this argument.

`-scalewidth width:`
: This argument can be used in connection with either `-autoscale` or `-layerscale` to specify the initial scaling engine dimensions. You can specify a numeric pixel value (e.g. `-scalewidth 1280 -scaleheight 1024`) or a scaling percentage (e.g. `-scalewidth 200% -scaleheight 200%`). Please read the documentation of `-autoscale`/`-layerscale` for more information on the Hollywood scaling engines. See Section 25.18 [Scaling engines], page 401, for details.

`-scaleheight height:`
: Same as `-scalewidth` but specifies the scaling height.

`-scrdepth d:`
: This argument can be specified to set the screen depth when you want Hollywood to open in full screen mode. This argument tells Hollywood which color depth to choose for the full screen mode (valid depths are 15, 16, 24 and 32). If you do not specify this argument, Hollywood will open in the same depth as the desktop screen.

`-scrwidth width:`
: This argument can be used in connection with `-fullscreen` to specify the dimensions of the full screen that Hollywood shall open. If you do not specify this argument, Hollywood will choose a full screen mode which fits best for your display. If you specify 0 in `-scrwidth` and `-scrheight`, Hollywood will use the dimensions of the desktop screen for the full screen mode.

`-scrheight height:`
: Same as `-scrwidth` but specifies the screen height.

`-setconstants list:`
: You can use this argument to declare one or more constants. Normally, constants are declared using the Const statement. Sometimes, however, it can be convenient to be able to declare constants from the command line as well. This is especially useful when using the `@IF` preprocessor command. You have to pass a string which contains one or more constants to be declared to this argument. If you want to declare multiple constants, you have to separate the individual constants by using the vertical bar character (`|`). You can use the equal sign (=) to assign a value to a constant. If you leave out the equal sign, the constant will automatically be given a value of 1. Note that the constant name must not include the hash tag prefix but just the constant's name. Here is an example string: `"MYCONSTANT|MYCONSTANT2=1000"`. If you pass this string to `-setconstants`, `#MYCONSTANT` will be defined as 1 and `#MYCONSTANT2` will be defined as 1000. If you need to define a string constant, you need to enclose the

string in square brackets, e.g. "MYSTRINGCONSTANT=[Test123]". If you need to store square brackets within a string constant, simply duplicate them so that they cannot be confused with the string constant delimiters.

`-sizeable:`
: If you specify this argument, Hollywood will open its window with a size widget at the bottom right side of the window. This widgets will be invisible if your window does not have borders but it will still be accessible.

`-skipplugins mask:`
: This argument can be used to tell Hollywood which plugins it should not load on startup. You can specify multiple plugins by separating them using a vertical bar (|) as a separator. If you want Hollywood to load no plugins at all on startup, specify an asterisk (*) here. You can load plugins later using the `@REQUIRE` preprocessor command or the `LoadPlugin()` function.

`-smoothscale:`
: If `-autoscale` or `-layerscale` is active and you specify `-smoothscale`, all scaling operations are interpolated using anti-aliased pixel smoothing. This looks better but is also slower. Please read the documentation of `-autoscale`/`-layerscale` for more information on the Hollywood scaling engines.

`-softtimer:`
: If you specify this argument, Hollywood will use a low resolution software timer instead of the high resolution hardware timer. This is sometimes necessary because with certain older Windows XP hardware, the timer may occassionally leap which can cause unexpected behaviour. On newer hardware and Windows versions you should never have to use this. [Windows only]

`-softwarerenderer:`
: Specify this argument to disable Hollywood's GPU-accelerated Direct2D renderer on Windows systems. If `-softwarerenderer` is set, Hollywood will use its CPU-based renderer for maximum compatibility. [Windows only]

`-startcolor color:`
: Only required in connection with `-backfill` set to `gradient` or `color`. `color` is a color specified in RGB format (e.g. $00FF00 for green). You can also use this argument together with `-backfill` set to `picture`; the color will fill the picture background then.

`-stayactive:`
: (removed in Hollywood 2.0)

`-systemscale:`
: If you set this argument, the host system's scaling factor will automatically be applied to your display. This can be useful on systems with high-DPI monitors. For example, if your display normally opens in 640x480 pixels and you run it on a monitor that uses twice as many dots per inch (DPI), specifying the `-systemscale` option will automatically scale your script to 1280x960 pixels so that it doesn't look tiny just because the system uses a high-DPI monitor.

By default, `-systemscale` will activate the auto scaling engine. If you want it to use layer scaling instead, just pass the `-layerscale` argument. Note that `-systemscale` uses the same scale mode as `-scalefactor` internally, so scripts using `-systemscale` will behave as if `-scalefactor` was specified. It is even possible to use the `-scalefactor` argument on top of `-systemscale`, in that case the value specified in `-scalefactor` is multiplied by host system's default scaling factor. Please also read the documentation of `-autoscale`/`-layerscale` for more information on the Hollywood scaling engines. See Section 25.18 [Scaling engines], page 401, for details. Note that on Windows you must also set the `DPIAware` tag to `True` in the `@OPTIONS` preprocessor command in order to use `-systemscale`. See Section 52.25 [OPTIONS], page 1088, for details.

`-tempdir path:`
: This argument can be used to specify the path where Hollywood should store its temporary files. This is especially useful on AmigaOS and compatibles since Hollywood will store temporary files in the RAM disk on these systems. This can lead to problems on systems that are short on memory or when working with very large projects. To specify the current directory as the location for temporary files, pass `"."` here.

`-usequartz:`
: Tells Hollywood to use Quartz 2D for all graphics output. By default, Hollywood uses QuickDraw because that is much faster (though deprecated). If you experience any graphics problems on macOS, you might want to try this argument. Note that this argument is only supported by the PowerPC version of Hollywood. The x86/x64 versions of Hollywood for macOS will always use Quartz 2D. [macOS only]

`-usewpa:`
: Tells Hollywood to use device independent bitmaps instead of standard OS bitmaps. Device independent bitmaps are normally slower than the standard OS bitmaps with the exception of WinUAE and AROS which both can lock OS bitmaps only pretty inefficiently. Thus, on WinUAE and AROS, `-usewpa` is activated automatically to speed up Hollywood. If you want to turn this off, specify `-wpamode 0`. Please note: `-usewpa` is a lowlevel argument which is primarily here for testing purposes. Normally, you should not deal with this directly. [Amiga OS only]

`-videofps fps:`
: Only used together with `-videoout`. If `-videoout` is active, `-videofps` can be used to tell Hollywood how many frames per second (FPS) the video to be recorded shall have. If not specified, 50 frames per second will be used as a default value. See Section 4.6 [Hollywood video recorder], page 62, for details.

`-videoout file:`
: Enables Hollywood's built-in video recording feature. If `-videoout` is specified Hollywood will save your script as an AVI video file which you then could burn on a DVD, for instance. See Section 4.6 [Hollywood video recorder], page 62, for details.

`-videopointer:`
: Only used together with `-videoout`. If you specify `-videopointer` the mouse pointer will always be rendered into the video stream. By default, when in video recording mode, no mouse pointer will appear in the video. If you need to have a mouse pointer in the video (e.g. to demonstrate user interaction), specify this argument. See Section 4.6 [Hollywood video recorder], page 62, for details.

`-videoquality quality:`
: Only used together with `-videoout`. If `-videoout` is active, `-videoquality` can be used to specify the compression level for the video frames. Quality is specified in percent so valid quality levels range from 0 to 100. The default is 90 which results in a pretty high quality video file which needs quite some disk space. If you want to have a smaller video file, you can try to use a lower quality level. See Section 4.6 [Hollywood video recorder], page 62, for details.

`-videostrategy strategy:`
: Only used together with `-videoout`. If the `-videoout` argument is active, `-videostrategy` can be used to specify the strategy Hollywood shall use when converting a Hollywood script into a video file. Currently, you can specify `wait` and `raw` here. By default Hollywood uses `wait` strategy. See Section 4.6 [Hollywood video recorder], page 62, for details.

`-vsync:`
: On Windows systems, this argument can be used to force Hollywood's renderer to throttle refresh to the monitor's refresh rate. This means that you'll no longer have to use functions like `VWait()` to throttle drawing. However, do note that if you set this to `True`, you must make sure to draw in full frames only otherwise drawing will become extremely slow. Full frame drawing can be achieved e.g. by either using a double buffer or by using `BeginRefresh()` and `EndRefresh()`. Also note that `VSync` is currently only supported on Windows and only if Hollywood uses its Direct2D backend. Direct2D is not available before Windows Vista SP2. [Windows only]

`-window:`
: If you specify this argument, Hollywood will open its window on the desktop instead of full screen mode. This is the default setting.

`-winwidth width:`
: This argument allows you to set the initial display width without activating one of the scaling engines. This has the same effect as if the user resized your display to the specified width. That's why your script will also receive a `"SizeWindow"` event right after Hollywood has been started if you pass this console argument. Note that this doesn't active any scaling engines, so your script needs to be prepared to adapt to the new dimensions itself.

`-winheight height:`
: Same as `-winwidth` but specifies the window height.

`-wpamode mode:`
: If `-usewpa` is active, this argument can be used to define the device independent bitmap mode. Passing 0 here turns off `-usewpa`, 1 activates 32-bit DIB mode, and 2 activates Workbench compliant DIB mode. Defaults to 1 which should

give the best performance. Please note: `-wpamode` is a lowlevel argument which is primarily here for testing purposes. Normally, you should not deal with this directly. [Amiga OS only]

`-xserver name:`
: This argument can be used to specify the X Server that Hollywood should try to connect to. By default, Hollywood will use the X Server that is specified in the `DISPLAY` environment variable. If you want Hollywood to connect to a different X Server, use this argument. [Linux only]

3.3 Console emulation

Hollywood and programs compiled with Hollywood recognize many console arguments that can be used to control various functions of the program. Most of the time, though, you won't start Hollywood or Hollywood-compiled programs from the console. So how can you pass these arguments to Hollywood programs when you don't have a console? This differs from platform to platform. Below is an overview of how to pass console arguments to Hollywood programs without using a console.

Note that by default, all programs compiled by Hollywood will handle console arguments and arguments passed to them using the mechanisms described below. If you would like to forbid this behaviour, you have to compile your programs using the `-locksettings` argument. Otherwise, the user will be able to change the appearance of your compiled program by passing arguments to it.

Here is how to specify console arguments to Hollywood programs without using the console:

1) AmigaOS:

Simply create an icon for your program and add the console arguments you want to use as tooltypes to that icon. For example, to add a black-to-blue gradient your program in `RAM:MyCoolProg`, create an icon named `RAM:MyCoolProg.info` and add the following tooltypes to it:

```
BACKFILL=GRADIENT
STARTCOLOR=$000000
ENDCOLOR=$0000ff
BORDERLESS
FIXED
(FULLSCREEN)
```

Note that tooltypes enclosed in parentheses are ignored so in the above case, `FULLSCREEN` is not handled.

2) Windows:

Under Windows, you can put the console arguments you want to use in an `*.ini` file accompanying your program. Let's assume you have compiled a program named `MyCoolProg.exe` with Hollywood. You have installed this program in

```
C:/Program Files/MyCoolProg/MyCoolProg.exe
```

You can now create an `*.ini` file which holds further options for this program. The `*.ini` file must have the same name as your program. Otherwise your program will not be able to detect that there is an `*.ini` file. Thus, you have to create the `*.ini` file as follows:

```
C:/Program Files/MyCoolProg/MyCoolProg.ini
```

You can then use your favorite text editor to add options to this `*.ini` file. For example, to create a black to blue gradient backfill for your program, you can put the following tags in `MyCoolProg.ini`:

```
Backfill=Gradient
StartColor=$000000
EndColor=$0000ff
Borderless=True
Fixed=True
```

See Section 3.2 [console arguments], page 33, for more information on which arguments can be specified in the *.ini file.

3) macOS:

Under macOS, you can choose between two different ways of passing console arguments without using a console:

1. Open the `Info.plist` inside the application bundle that was compiled by Hollywood with your favorite text editor. Now search for the dictionary entry named `CFBundleExecutableArgs`. Here you can add all the console arguments you like now. For example:

   ```
   <key>CFBundleExecutableArgs</key>
   <string>
   -backfill gradient -startcolor $000000 -endcolor $0000ff
   -borderless -fixed
   </string>
   ```

   With these console arguments specified, your program will open with a black-to-blue gradient.

2. Alternatively, you can also create an `*.ini` file just as you can do under Windows. See above for instructions on how to create such an `*.ini` file. The only difference under macOS is that you have to put the `*.ini` file inside the `Resources` directory of your application bundle. For instance, if your app bundle is located in this path:

   ```
   /Programs/MyCoolProg.app
   ```

   Your `*.ini` file has to go here then:

   ```
   /Programs/MyCoolProg.app/Contents/Resources/MyCoolProg.ini
   ```

   The rest is exactly the same as under Windows.

4) Linux:

On Linux you can also put all console arguments in an `*.ini` file. See above for detailed instructions.

4 Compiler and linker

4.1 Compiling executables

Hollywood's compiler can either be used from the GUI or by using the `-compile` argument from the console.

Once invoked, the compiler will read the specified script file, compile it and link a special player-only version of Hollywood to it. All external data that is declared using preprocessor commands is linked, too (unless it is explicitly declared that a file should not be linked). The output executable format can be defined by specifying the `-exetype` argument. This argument can be set to the following executable types:

`amigaos4` AmigaOS 4 executable (PowerPC)

`aros` AROS executable (x86)

`classic` AmigaOS 3.x executable (68020+)

`classic881`
AmigaOS 3.x executable (68020+) with math co-processor (68881/2 or 68040/68060)

`linux` Linux executable (x86)

`linux64` Linux executable (x64)

`linuxarm` Linux executable (arm)

`linuxppc` Linux executable (PowerPC)

`macos` macOS application bundle (PowerPC)

`macosarm64`
macOS application bundle (arm64)

`macos86` macOS application bundle (x86)

`macos64` macOS application bundle (x64)

`morphos` MorphOS executable (PowerPC)

`warpos` WarpOS mixed-binary executable (68040/PowerPC)

`win32` Windows executable (x86)

`win64` Windows executable (x64)

`applet` Universal Hollywood applet which can be started on any system with a Hollywood player

In the 68k version of Hollywood, `-exetype` defaults to `classic`, in the AmigaOS4 version `-exetype` defaults to `amigaos4` and so on.

If your script uses a lot of external data, your executable might become very big because Hollywood will link all the files declared with preprocessor commands to it. If you do not want that, you can use the `Link` argument which all preprocessor commands support to

tell the linker not to link certain files. Alternatively, you could load the files using normal Hollywood commands instead of preprocessor commands.

The generated executable will accept the same console arguments as the main Hollywood program. Therefore you can start it for example with a borderless window by calling it with the `-borderless` argument.

You can also link plugins to your executables using the `-linkplugins` argument but you have to be very careful with the plugin license to see if static linking is allowed and the effect it can have on your project's license. See Section 3.2 [Console arguments], page 33, for details.

4.2 Compiling applets

Apart from stand-alone executables, you can also compile your scripts to Hollywood applets. These are much smaller because they do not contain the Hollywood Player. Hollywood applets can be started with the interpreter or the freely available Hollywood Player. The advantage of applets is that you save a lot of space. Imagine you want to compile your script for all platforms which Hollywood supports (AmigaOS 3, AmigaOS 4, WarpOS, MorphOS, AROS, Windows, macOS, Linux). The players for all that platforms alone take up more than 50 megabytes so you would have to distribute a rather large archive. In this case it is a much better idea to just compile your script to a Hollywood applet. Then, the user can simply download the freely available Hollywood Player for his platform from `http://www.hollywood-mal.com/` and use it to run your applet. And you have to distribute your applet only.

To compile applets with Hollywood, just pass `APPLET` in the `-exetype argument` or use the GUI. Hollywood applets carry the suffix `*.hwa`.

4.3 Linking data files

By default, Hollywood's linker will automatically link all external data files declared using preprocessor commands to the output executable or applet. If your script looks like below, for example, the file `test.jpg` will automatically be linked to your executable or applet:

```
@BGPIC 1, "test.jpg"
WaitLeftMouse
End
```

If you don't want that, you can set the `Link` tag, which is accepted by all preprocessor commands dealing with files, to `False`. In that case, the file referenced by the preprocessor command will not be linked. The code looks like this then:

```
@BGPIC 1, "test.jpg", {Link = False}
WaitLeftMouse
End
```

Sometimes, you might also want to link files that are loaded by your script at runtime into your executable or applet. Consider the following code for example:

```
LoadBGPic(1, "test.jpg")
LoadBrush(1, "title.png")
DisplayBGPic(1)
DisplayBrush(1, #CENTER, #CENTER)
```

```
WaitLeftMouse
End
```

By default, `test.jpg` and `title.png` won't be linked to your executable or applet because they haven't been declared in a preprocessor command but they are loaded at runtime using `LoadBGPic()` and `LoadBrush()` instead. Still, it is possible to link `test.jpg` and `title.png` to your executable or applet. This can be achieved by using either the `-linkfiles` compiler option or the `@LINKER` preprocessor command.

When using the `-linkfiles` console argument, you need to pass a database file to it. The database file is a simple UTF-8 text file which contains a list of files to be linked into the applet or executable that will be compiled by Hollywood. You must only specify one file per line in the database file. The file specification must be identical to the specification you use in your script. For example, if there is a command `LoadBrush(1, "data/menu.png")` in your script and you want the file `data/menu.png` to be linked into your applet or executable, you need to put it into the database you pass to `-linkfiles`. But you must use the same specification, i.e. you need to use `data/menu.png`! Specifying `MyScripts/CoolGame/data/menu.png` in the database will not work! The specification used in the link files database and in the script must be the same because otherwise Hollywood cannot know which file it must load.

So in order to link the files `test.jpg` and `title.png` to our executable or applet, the database file we pass to `-linkfiles` needs to look like this:

```
test.jpg
title.png
```

That is all! The Hollywood linker will then link `test.jpg` and `title.png` to the output executable or applet and the calls to `LoadBGPic()` and `LoadBrush()` in the script presented above will load `test.jpg` and `title.png`, respectively, directly from the executable or applet instead of from an external source.

The same can be achieved by using the `@LINKER` preprocessor command. The only difference is that the files to be linked don't have to be passed in an external database file to Hollywood, but they must be stored directly in your script as part of the `@LINKER` preprocessor command instead. All other rules are the same as with `-linkfiles`. So if you don't want to use `-linkfiles` like above, you could also just add the following line to your script and achieve the same:

```
@LINKER {Files = {"test.jpg", "title.png"}}
```

You can add as many files as you want to have linked to your applet or executable to the `Files` tag that is part of the `@LINKER` preprocessor command. Just make sure the path specification of the files you pass to `@LINKER` is identical to the path specification used later in the code so that Hollywood can correctly map the linked files to the individual files used in the script.

If you need to link lots of files to your applet or executable, you can put all those files into a directory and then tell Hollywood to link everything in that directory to the applet or executable. This is done by using `@DIRECTORY` preprocessor command. For example, the following line tells Hollywood to link all files inside the `data` directory to the applet or executable:

```
@DIRECTORY 1, "data"
```

Once you have done that, you can then access the individual files in the `data` directory by using the `GetDirectoryEntry()` function. For example, to load the files `data/test.jpg` and `data/title.png` using `LoadBGPic()` and `LoadBrush()`, you would write the following code:

```
LoadBGPic(1, GetDirectoryEntry(1, "data/test.jpg"))
LoadBrush(1, GetDirectoryEntry(1, "data/title.png"))
```

The `@DIRECTORY` preprocessor command is very flexible because it will archive the complete directory tree inside an applet or executable which also makes it possible to iterate through the directory (and all of its subdirectories!) as if it were a real one. See Section 26.13 [DIRECTORY], page 427, for details.

4.4 Linking fonts

By default, Hollywood's linker will automatically link all fonts declared using the `@FONT` preprocessor command to the output executable or applet. If your script looks like below, for example, the font `Arial` will automatically be linked to your executable or applet:

```
@FONT 1, "Arial", 36
WaitLeftMouse
End
```

If you don't want that, you can set the `Link` tag, which is accepted by the `@FONT` preprocessor command, to `False`. In that case, the font specified in the preprocessor command will not be linked. The code looks like this then:

```
@FONT 1, "Arial", 36, {Link = False}
WaitLeftMouse
End
```

Sometimes, you might also want to link fonts that are loaded by your script at runtime into your executable or applet. Consider the following code for example:

```
SetFont("Arial", 36)
WaitLeftMouse
End
```

By default, font `Arial` won't be linked to your executable or applet because it wasn't declared in a preprocessor command but it is loaded at runtime using `SetFont()` instead. Still, it is possible to link `Arial` font to your executable or applet. This can be achieved by using either the `-linkfonts` compiler option or the `@LINKER` preprocessor command.

If you choose to use the `-linkfonts` compiler option, you need to pass a database file to it. The database file is a simple UTF-8 text file which contains a list of fonts to link into the applet or executable that will be compiled by Hollywood. You must only specify one font per line in the database file. The font can be either the name of a font or a path to a *.ttf or *.otf file. Note that when passing paths to *.ttf or *.otf files directly, you must use the inbuilt font engine because only that is able to load fonts from files. A font database could look like the following:

```
Arial
"Times New Roman"
FuturaL
helvetica
```

```
data/arial.ttf
```

Do not forget to use quotes when passing font names that have spaces in them!

The same can be achieved by using the `@LINKER` preprocessor command. The only difference is that the fonts to be linked don't have to be passed in an external database file to Hollywood, but they must be stored directly in your script as part of the `@LINKER` preprocessor command instead. All other rules are the same as with `-linkfonts`. So if you don't want to use `-linkfonts` like above, you could also just add the following line to your script and achieve the same:

```
@LINKER {Fonts = {"Arial", "Times New Roman", "FuturaL", "helvetica",
    "data/arial.ttf"}}
```

You can add as many fonts as you want to have linked to your applet or executable to the `Fonts` tag that is part of the `@LINKER` preprocessor command.

Important note: Please note that most fonts are copyrighted and it is not allowed to link them into your programs without acquiring a licence. So make sure you check the licence of the font you are going to link into your program! If you do not want to pay for font licences, it is advised to use a free font such as DejaVu or Bitstream Vera or use one of the TrueType fonts that are inbuilt into Hollywood (`#SANS`, `#SERIF`, `#MONOSPACE`, cf. `SetFont()`)

4.5 Linking plugins

In contrast to data files and fonts, plugins aren't automatically linked to your executable when you require them in the preprocessor commands. The following code, for example, will not force the linker to link "jpeg2000" into your executable:

```
@REQUIRE "jpeg2000"
```

If you want to have `jpeg2000.hwp` linked into your executable, you have to set the `Link` tag to `True`. The code looks like this then:

```
@REQUIRE "jpeg2000", {Link = True}
```

In that case, `jpeg2000.hwp` will be linked to your executable and the user won't need to keep a copy of `jpeg2000.hwp` because it has already been linked into the executable.

Alternatively, you can also use the `-linkplugins` console argument to link plugins into your executable. See Section 3.2 [Console arguments], page 33, for details.

Note that plugins can only be linked to executables, not to applets, since applets are platform-independent and plugins are not.

Before you can use the plugin linker, you first have to copy the plugins you would like to link into a directory named `LinkerPlugins`. On AmigaOS and compatibles, this directory needs to be created in Hollywood's installation directory, i.e. you need to create `Hollywood:LinkerPlugins`. On all other systems, you have to create the `LinkerPlugins` directory in the directory where Hollywood has been installed, i.e. next to the Hollywood executable. Keep in mind that on macOS this will be inside the application bundle, i.e. in `HollywoodInterpreter.app/Contents/Resources/LinkerPlugins`. Furthermore, you have to create the following architecture subdirectories inside the `LinkerPlugins` directory:

```
arm-android-v7a
arm64-android-v8a
arm-ios
```

```
arm-linux
m68k-amigaos
m881-amigaos
ppc-amigaos
ppc-linux
ppc-macos
ppc-morphos
ppc-warpup
x86-aros
x86-macos
x86-linux
x86-windows
x86-windows-console
x64-linux
x64-macos
x64-windows
x64-windows-console
```

After that, you have to copy the plugins you want to link to these subdirectories. You need to copy plugins for all the architectures you want to compile executables for. If you don't do that, the linker won't be able to find the plugins to link. Note that the linker will look for plugins only inside the `LinkerPlugins` directory. It won't look anywhere else, in particular not in the standard plugins location.

Note that when creating executables for the `m881-amigaos` architecture, the linker will also look for plugins in the `m68k-amigaos` directory because both architectures are completely compatible. The same is true for `ppc-warpup` which will also take both, the `m68k-amigaos` and `m881-amigaos` architectures, into account. Also, `x86-windows-console` and `x86-windows` are compatible as are `x64-windows-console` and `x64-windows`.

Important note: Make sure to carefully read the license of every plugin you link to your executable because many licenses are very restrictive when it comes to static linking. For example, if you link a plugin that is licensed under the LGPL license, then your complete project automatically becomes LGPL as well and you must provide all sources and data files. So make sure to study plugin licenses before you link them to your executables. You have been warned.

4.6 Saving scripts as videos

Starting with Hollywood 4.0 it is possible to save Hollywood scripts as AVI video files. This is useful for example if you want to create DVDs of your Hollywood scripts or just run them on a platform that is currently not supported by Hollywood. Saving Hollywood scripts as video files also allows you to import them into video editing software for further processing or format conversion.

Hollywood's video recorder was designed with the idea in mind to reproduce the exact behaviour of the Hollywood script in a video file. Thus, you will most likely not notice any difference between the video file and the actual Hollywood script. Hollywood's video recorder tries to time the script exactly as it would appear in real time mode. Hence, it is no problem for the video recorder to deal with scripts that require exact timing - for

example for synchronization with music. The video recorder pays special attention to this and tries to time everything correctly.

To enable the video recorder mode you simply have to specify the `-videoout` argument together with a filename for the video to be created. Hollywood will then start in recording mode and graphics and sounds will now be redirected into the video stream. Thus, when in recording mode, no sounds will be played because sound data is immediately rendered into the video stream. Also, please note that certain options are disregarded when Hollywood is in video recording mode. For example, in video recording mode Hollywood will always open in windowed mode, never in full screen even if you specify so. Also the window will not be sizeable etc.

The video file written by the video recorder will be an AVI 2.0 stream adhering to the OpenDML standard so streams greater than 2 GB are possible. Hollywood currently uses the Motion JPEG codec to compress the video frames. Audio data is written to the video file without any compression. You can control the quality of the Motion JPEG by using the `-videoquality` argument.

To achieve the best result you may need to adjust some parameters in the video recorder with which we will deal now:

1. First, it is advised that you tell the video recorder how many frames per second shall be recorded. You can do this by using the `-videofps` argument. The value that you specify here should be identical to the frequency of your main loop. If your main loop runs at 25 frames per second, e.g. using the following code:

   ```
   SetInterval(1, p_MainLoop, 1000\25)
   ```

   Then your video file should also run at 25 fps. So you would have to specify

   ```
   -videofps 25
   ```

   on the command line to tell the video recorder that you want your video to have 25 frames per second.
2. You may want to specify a scaling resolution for the video file. Remember that the video resolution cannot change but must be static throughout the whole video. Hollywood can change the resolution of its display at any time but for video files this is not possible so if Hollywood's display size changes while the program is in video recording mode, graphics will be scaled to keep up the correct video resolution. By default, the video resolution will be set to the resolution of the first background picture. If you want a different resolution, however, you must specify the `-scalewidth` and `-scaleheight` arguments.
3. Your Hollywood script needs to follow a certain pattern in order for the recorder to save it as a video file. Particularly, the recorder needs to know when its frame buffer shall be flushed into the video file. Normally, this is done whenever it encounters a waiting command. For instance:

   ```
   VWait()
   Wait()
   WaitEvent()
   WaitTimer()
   etc.
   ```

   Thus, it is necessary that you use one of the commands above in your script! Your script needs to employ a timing mechanism, otherwise it cannot be converted properly

into a video file. Suggested timing mechanisms are either using an interval function which is called a certain number of times per second or using `WaitTimer()` or `VWait()`. See Section 15.3 [script timing mechanisms], page 161, for more information on the importance of using a correct timing mechanism.

Flushing the frame buffer whenever a wait command occurs is called the "wait strategy" which is also the default video strategy. Normally, the wait strategy should be suitable for all purposes. With correctly timed scripts, the wait strategy delivers the best results. In some very rare cases - or for debugging purposes - you might want to use the `raw` strategy instead. When `-videostrategy` is set to `raw`, the video recorder will render every frame to the stream no matter if waits are used or not. In most cases, of course, this results in wrongly timed videos so you will most likely never want to use the raw strategy.

4. Finally, you must decide whether or not the mouse pointer shall be rendered into the video stream. By default, this is disabled because rendering the mouse pointer into the video makes only sense in special situations, for instance if you are creating a demo video where user input shall be visible. To turn on mouse pointer recording, specify the `-videopointer` argument. All mouse pointer movements will then be recorded in the video file.

5 Plugins

5.1 Plugins

Hollywood's functionality can be greatly enhanced via plugins. Plugins can provide load and save support for additional video, audio, image, and sample formats, they can extend the command set of the Hollywood language as well as enable Hollywood to use real vector graphics and it is even possible to write plugins which replace core parts of Hollywood like its inbuilt display and audio driver with custom implementations provided by plugins. It is also possible to write plugins which convert project files of other applications like Scala or PowerPoint into Hollywood scripts so that Hollywood can run these project files directly although they are not in the `*.hws` format.

5.2 Installation

Hollywood plugins use the suffix `*.hwp`. On all systems except on AmigaOS and compatibles, plugins must be stored in a directory named `"Plugins"` that is in the same directory as the main Hollywood program. On AmigaOS and compatible systems, plugins must be installed to `LIBS:Hollywood` instead. On macOS, the `"Plugins"` directory must be inside the `"Resources"` directory of the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle itself, namely in `Hollywood.app/Contents/Resources`.

When distributing a compiled Hollywood program, plugins required by your program must simply be put into the same directory as your program. When compiling application bundles for macOS, plugins need to be put in the `"Resources"` directory of the application bundle, i.e. in `MyProject.app/Contents/Resources`. Alternatively, you can also choose to link plugins into your executable.

The Android version of Hollywood also supports Hollywood plugins. You have to copy them to the directory `Hollywood/Plugins` on your SD card. Hollywood will scan this location on every startup and load all plugins from there.

5.3 Usage

Plugins will be loaded automatically by Hollywood on startup. If you do not want this, you can disable automatic loading by renaming the plugin: Plugins whose filename starts with an underscore character ('_') will not be loaded automatically by Hollywood on startup. As an alternative, you can also use the `-skipplugins` console argument to tell Hollywood to skip automatic loading of certain plugins. Plugins which have not been loaded at startup, can be loaded later by using the `@REQUIRE` preprocessor command or the `LoadPlugin()` function. See Section 45.6 [REQUIRE], page 930, for details.

Please note that although Hollywood loads all plugins automatically on startup, many plugins require you to call `@REQUIRE` before they can be used. This is because these plugins need custom initialization code which is only run if you explicitly call `@REQUIRE` on them. For example, plugins which install a display adapter will not be activated unless you call `@REQUIRE` on them. Plugins which just add a loader or saver for additional file formats, however, will be automatically activated even if you don't call `@REQUIRE` on them.

5.4 Obtaining plugins

Many plugins are available from the official Hollywood portal which is online at http://www.hollywood-mal.com/. Here is an overview of plugins currently available from the official Hollywood portal:

AHX: Allows you to load and play AHX and HivelyTracker modules with Hollywood.

AIFF: Allows you to load and play AIFF samples with Hollywood.

APNG Anim:
Allows you to load and save APNG animations with Hollywood. This is useful because the APNG format supports anims with alpha channel.

AVCodec: Adds loaders for many video and audio formats provided by FFMPEG. This is very useful for playing modern video and audio formats but be careful that many of those formats are patented and require you to pay royalties or licensing fees if you use them in your products.

DigiBooster:
Load and play DigiBooster modules with Hollywood.

FLIC Anim:
Load FLI and FLC animations with Hollywood.

GL Galore:
OpenGL® wrapper for Hollywood. This plugin allows you to program in OpenGL using Hollywood. It also supports hardware-accelerated 2D drawing, i.e. it supports hardware double buffers and hardware brushes. Thus, it is very useful for hardware-accelerated on drawing on Windows, macOS, and Linux because by default, Hollywood only supports hardware-accelerated drawing on AmigaOS and compatibles.

HTTP Streamer:
This plugin allows you to load data from HTTP sources as normal files. This means that once this plugin is installed you can just pass URLs to functions like `LoadBrush()` and the files will be loaded from there. This plugin can also be used for video and audio streaming from HTTP sources.

Iconic: Iconic is the ultimate icon loader and saver plugin for Hollywood. It can load and save a large variety of different icon formats. Currently, the following icon formats are supported by Iconic: AmigaOS 1.x style icons, AmigaOS 2.x/3.x style icons, AmigaOS 3.5 icons (aka GlowIcons), AmigaOS 4.0 icons, macOS icons (*.icns format), MagicWB icons, MorphOS/PowerIcons icons (PNG), NewIcons, Windows icons (*.ico format).

hURL: hURL is a plugin for Hollywood that allows you to transfer data using many different protocols. Based on curl, hURL supports an incredibly wide range of transfer protocols, e.g. DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, Telnet and TFTP. Furthermore, hURL supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, HTTP/2, cookies, user+password authentication (Basic, Plain,

Digest, CRAM-MD5, NTLM, Negotiate and Kerberos), file transfer resume, proxy tunneling and more. It really is the ultimate data transfer engine for Hollywood, leaving nothing to be desired.

JPEG2000:
Load and save images in the JPEG2000 format.

Malibu: This plugin allows you to run Scala scripts with Hollywood. Malibu supports all Amiga versions of Scala until Scala MM400 and Scala InfoChannel 500.

Movie Setter:
Open and play animations that are in Gold Disk's MovieSetter format.

MUI Royale:
Create MUI GUIs with Hollywood. This is a powerful plugin which wraps almost the complete MUI API to Hollywood and allows you to conveniently create MUI GUIs via XML.

Ogg Theora:
Load and play video streams in the Ogg Theora format.

Ogg Vorbis:
Load and play Ogg Vorbis streams with Hollywood.

PCX: Load PCX images with Hollywood.

Plananarama:
Use Hollywood on Amigas without a graphics board. If you need to run Hollywood on a palette-based screen, you can do this with this plugin.

Polybios: Polybios is a plugin for Hollywood that allows you to easily create PDF documents from Hollywood scripts. On top of that, Polybios can also open existing PDF documents and convert their pages into Hollywood brushes. In fact, when converting PDF pages into Hollywood brushes, Polybios will create vector brushes for you which can be scaled, rotated and transformed without any losses in quality (unless bitmap graphics are embedded inside the PDF document of course).

RapaGUI: Cross-platform GUI toolkit for Hollywood. This is a very powerful plugin which turns Hollywood into a complete cross-platform GUI toolkit that allows you to create GUI-based applications very conveniently by defining the GUI layout in an XML file. On top of that, Hollywood displays can be embedded inside your GUI as widgets which allows you to use all of Hollywood's powerful graphics features in your GUI application as well.

RebelSDL:
This is a Hollywood wrapper plugin for the popular SDL library. The great benefit is that it supports hardware-accelerated 2D drawing, i.e. it supports hardware double buffers and hardware brushes. Thus, it is very useful for hardware-accelerated on drawing on Windows, macOS, and Linux because by default, Hollywood only supports hardware-accelerated drawing on AmigaOS and compatibles.

SID: This plugin allows you to load and play SID files.

SQLite3: Create and modify SQL databases with Hollywood.

SVG Image:
Load SVG vector images with Hollywood.

TIFF: Load and save TIFF images with Hollywood.

Vectorgraphics:
Draw real vector-based graphics with Hollywood.

XAD: This plugin allows you to unpack lots of different archiver formats like LhA, LZX, RAR, ZIP, TAR, etc. Very useful.

XLSX: The XLSX plugin allows you to conveniently read and write XLSX documents from Hollywood scripts. It offers a wide variety of functions to set and get cell values, cell types, cell formulas, document/worksheet properties and several other attributes. It also offers an iterator function for a high performance iteration of a large number of cells.

XML: Plugin for convenient parsing of XML documents.

XMP: This plugin can play lots of different tracker formats.

YAFA: Load and play animations in the YAFA format (created by Wildfire).

ZIP: This plugin allows you read and write ZIP archives.

5.5 Writing your own plugins

In case you are missing a certain feature or functionality in Hollywood, you can write a plugin which adds it to the language. Writing your own plugin can also be helpful in case your script needs to do certain CPU-intensive calculations which are best implemented in native code for an optimal performance.

Writing plugins is really easy. Hollywood's plugin interface is public, fully documented and all necessary files are available for free download from the official Hollywood portal. The Hollywood SDK comes with over 300 pages of detailed documentation and several examples that help you to get started with plugin development.

Please visit the official Hollywood portal at `http://www.hollywood-mal.com/` to download the latest Hollywood SDK. It contains all the developer materials you need for building your own Hollywood plugins.

6 History and compatibility

6.1 History

Please see the file `history.txt` for a complete change log of Hollywood.

6.2 Compatibility notes

Hollywood 9.0 API changes

There have been a few API changes in Hollywood 9.0. Most likely you won't have to adapt your scripts to work with 9.0. Just check the following notes to see if your script requires any adaption.

- On AmigaOS and compatibles, plugins installed in `LIBS:Hollywood` will no longer be loaded automatically by all executables compiled by Hollywood. Executables compiled by Hollywood will only load the plugins now that are stored next to the executable in its directory. If you want your executable to load all plugins in `LIBS:Hollywood` as well, you have to set the `GlobalPlugins` tag to `True` in the `@OPTIONS` preprocessor command or pass the `-globalplugins` console argument or tooltype to the program.
- On Windows, Hollywood's graphics engine has been completely rewritten and uses Direct2D now. This has many advantages and also makes it possible to use the GPU when the auto scaling engine is active. In some rare cases, however, your script might be slower with Direct2D than before, depending on how you do your drawing. If your script runs slower with Hollywood 9.0, you can either force Hollywood to use its old renderer, which is still supported for compatibility with Windows versions that don't support Direct2D (i.e. all Windows versions before Vista SP2) or adapt the way your script draws its graphics. To activate Hollywood's legacy renderer, just set the `SoftwareRenderer` tag to `True` in `@DISPLAY`. Alternatively, you can also adapt the way your script draws its graphics to improve performance with Direct2D. On Direct2D, every drawing operation will always result in a full display refresh, even if just a single pixel needs to be drawn! Thus, your script needs to draw its graphics in a way that minimizes full display refreshes. This can be achieved by either using a double buffer or by using `BeginRefresh()` and `EndRefresh()`. See Section 30.4 [BeginRefresh], page 591, for details.
- On Windows, Hollywood will automatically scale your scripts to fit to the monitor's DPI on high-DPI monitors now. This guarantees that they will appear in the correct size on high-DPI monitors as well. If you don't want that, set the new `DPIAware` tag to `True` in the OPTIONS preprocessor command. This is especially recommended for GUI applications because they will look much better when the program is DPI-aware.
- Loaders and adapters are now handled in the order they appear in the string that is passed to a `Loader` or `Adapter` table argument. For example, if you pass the string `digibooster|xmp` to `OpenMusic()`, it will first try to open the music using `digibooster.hwp` and only if that fails will `xmp.hwp` be asked to open the file. This also allows you to prioritize generic loaders like `Native`, `Inbuilt` and `Plugin`, e.g. if you want native and inbuilt loaders to be used before plugin ones, you could pass the

string `native|inbuilt|plugin` to achieve that. Note that although this is an API change there are probably no scripts which really depend on the old behaviour because the order was mostly random.

- When a non-existing path was passed to `ChangeDirectory()`, the function failed silently and didn't show an error. This has been changed now. `ChangeDirectory()` will trigger an error now if a non-existing path is passed to it.
- When `CopyFile()` couldn't copy a file because the file already existed and was write- or delete-protected, `CopyFile()` silently skipped the file. This is no longer done. An error will be thrown now if an existing file can't be overwritten. If you really need the old behaviour, use a callback function and return `False` whenever you get a `#COPYFILE_UNPROTECT` message.
- The optional `transcolor` argument in `SaveBrush()` now defaults to `#NOCOLOR` instead of `#BLACK`. Having `#BLACK` as a default didn't really make much sense because you wouldn't want `SaveBrush()` to mess with the image data but just write it to disk.
- `SaveAnim()`, `BeginAnimStream()` and `WriteAnimFrame()` no longer support the `Transparency` and `UseAlpha` tags because they just didn't make sense. Transparency and alpha settings are now determined solely by the format of the source data that is passed to these functions.
- When `KeepProportions` is active for a display and `DisplayBGPic()` or `ChangeDisplaySize()` is called, the scaling dimensions are now re-calculated to fit the aspect-ratio of the new BGPic or display size. This change also affects `SetDisplayAttributes()` when it is called with `Width` and `Height` parameters.
- If the `Monitor` tag isn't set in the optional table argument, `ChangeDisplayMode()` will now always automatically detect the monitor the display is on. So if it is on a second monitor and you call `ChangeDisplayMode(#DISPMODE_FULLSCREEN)`, that monitor will be put into fullscreen mode. Previously, always the first monitor was used (except if the `Monitor` tag was set for the display).
- Videos handled by the operating system's native video renderer (i.e. DirectShow and Media Foundation on Windows, AVFoundation and QuickTime on macOS) are now automatically resized and repositioned when a display is resized by the user, even if no scaling engine is active. This is done purely for aesthetic reasons because otherwise it looks really ugly. You can forbid this behaviour by setting the `NoLiveResize` tag in `@DISPLAY`. Note that videos managed by Hollywood's inbuilt video renderer won't be resized and repositioned automatically, it's only done for videos managed by the OS.

Hollywood 8.0 API changes

There have been no API changes for Hollywood 8.0.

Hollywood 7.1 API changes

There have been some small API changes in Hollywood 7.1. Most likely you won't have to adapt your scripts to work with 7.1. Just check the following notes to see if your script requires adaption.

- There have been some minor changes to Hollywood's platform-independent catalog format. Lines that start with semicolon are now considered comments and are ignored.

If you need to define a catalog string that starts with a semicolon, you need to prefix the semicolon with a backslash. Furthermore, empty lines are ignored now and strings ending in a single backslash are considered multi-line strings. It might be necessary to fix your catalogs to be compatible with the new format. See Section 36.1 [Using catalogs], page 739, for details.

Hollywood 7.0 API changes

New plugin and keyfile location on Windows, macOS, Linux, and Android

First of all, Windows, macOS, and Linux users should note that in Hollywood 7.0 plugins must now be stored in a `Plugins` subdirectory that must be in the same directory as the Hollywood executable on Windows and Linux. On macOS, the `Plugins` directory must be stored inside the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources/Plugins` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle, namely in `Hollywood.app/Contents/Resources`. On Android, plugins must now be stored inside the `Hollywood/Plugins` directory on your SD card (instead of `Hollywood/_Plugins` as in earlier versions). On AmigaOS and compatibles, plugins must be copied to `LIBS:Hollywood` as usual.

Note that executables compiled by Hollywood will still load plugins from the same directory as the executable (except on macOS where they must be inside the app bundle's `Resources` directory). Hollywood itself, however, will now need the plugins inside a `Plugins` subdirectory on Windows, macOS, and Linux as described above.

macOS users please do also note that the file `Hollywood.key` must now be copied to `HollywoodInterpreter.app/Contents/Resources` as well. It must no longer be in the `HollywoodInterpreter.app/Contents/MacOS` directory.

Important Unicode notes and other API changes

Since Hollywood 7.0 introduces Unicode support there might be some compatibility issues with your old scripts. If you don't want to adapt your scripts, you can simply run them in non-Unicode mode by disabling Unicode like this:

```
@OPTIONS {Encoding = #ENCODING_ISO8859_1}
```

If you add this as the very first line of your script, Hollywood will run your script in legacy mode and there shouldn't be any compatibility issues. However, your script will run in ISO 8899-1 mode then which means that it won't run correctly on non-Western European systems.

Thus, it is recommended that you don't run your script in legacy mode but use Unicode mode all the time. Most scripts will probably run just out of the box without any issues and without any need for adapting anything. If your script shows compatibility issues with Hollywood 7.0, please read the following list of API changes in Hollywood 7.0 to learn how to fix your scripts.

– First of all, make sure to save all your scripts in UTF-8 encoding now. When running your old scripts with Hollywood 7.0, Hollywood will first check if they contain only valid UTF-8 characters. If they don't, Hollywood will assume they are in ISO 8859-1

encoding (or the system's default encoding on Amiga) and convert them to UTF-8 automatically. Since this automatic conversion might lead to problems with scripts using a different encoding than ISO 8859-1 it is highly recommended to save all your scripts in UTF-8 now.

- Since Hollywood 7.0 runs in Unicode mode by default now, the default string encoding is set to `#ENCODING_UTF8` now as well. This means that you'll run into problems if your script tries to use the string library functions to access the raw binary data of strings. When the string encoding is set to `#ENCODING_UTF8`, the string library functions can only deal with strings that contain valid UTF-8 text. In Hollywood, however, strings can also contain binary data. For example, you could download a file into a string using `DownloadFile()` and then find out its length using `StrLen()`. This won't work when Hollywood is in Unicode mode (i.e. when the default string encoding is set to `#ENCODING_UTF8`) because `StrLen()` will expect valid UTF-8 data then. To work around this problem, you have to pass `#ENCODING_RAW` to `StrLen()` to tell it that the string you passed contains raw binary data instead of valid UTF-8 text. Likewise, most other functions of the string library accept an additional `encoding` parameter now too which you can use to set the character encoding of the string you pass. If your script doesn't use the string library functions to operate on raw binary data, you won't have to worry about anything and your script should work flawlessly in Unicode mode.
- The default text encoding is also set to `#ENCODING_UTF8` automatically by Hollywood 7.0. This means that functions like `TextOut()` and `Print()` will expect UTF-8 encoded text now. This isn't a problem if you just convert your script to UTF-8 but it could lead to problems if the text to be printed is read from a file (or other external source) that doesn't use UTF-8 encoding.
- `ReadChr()` and `WriteChr()` may now read and write up to 4 bytes instead of just a single byte if Hollywood is in Unicode mode. That is because they'll now really deal with characters, just as their names imply, and in UTF-8 a character may need up to 4 bytes for storage. If you want to read and write single bytes, you have to use the new `ReadByte()` and `WriteByte()` functions now.
- `ReadString()` and `WriteString()` can no longer be used for binary I/O because they read and write strings, i.e. a number of characters (not bytes!), just as their names imply. If you need to read and write raw binary data, use the new `ReadBytes()` and `WriteBytes()` functions now.
- `ReadChr()`, `WriteChr()`, `ReadString()`, and `WriteString()` now read and write UTF-8 characters by default. If you use them to read data from non-UTF-8 text files, there can be problems with non-ASCII characters. In that case, you have to tell those functions to read ISO 8859-1 characters instead by passing `#ENCODING_ISO8859_1` in the optional `encoding` argument.
- You are now discouraged from using the `OnKeyDown` and `OnKeyUp` event handlers for non-English characters. Non-English characters should be handled by the new `VanillaKey` event handler instead which has full Unicode support. `OnKeyDown` and `OnKeyUp` will continue to work as before but using non-English characters with them is generally unsafe. It might work on your system but not on systems with a different locale. Only use `VanillaKey` to handle non-English characters please.
- The `IsKeyDown()` and `WaitKeyDown()` functions no longer support non-English keys.

If you need to get the state of a non-English key, use the `VanillaKey` event handler instead.

- The platform-neutral format supported by `OpenCatalog()` now has to be in UTF-8 character encoding, with or without BOM. ISO 8859-1 files are no longer supported.
- Multi-byte character constants like 'ABCD' are no longer supported because they conflict with UTF-8 character constants supported by Hollywood 7.0. If your script uses multi-byte character constants, you have to rewrite your script to use the direct numeric value of the character constant instead.
- Hollywood 7.0 introduces the FallThrough statement which allows code to fall through to the next `Case` statement in a Switch-Case statement. This addition means that it is no longer allowed to use variables or functions which are called `FallThrough`. This will trigger an error because `FallThrough` is a reserved token now.

Hollywood 6.0 API changes

There have been some small API changes in Hollywood 6.0. Most likely you won't have to adapt your scripts to work with 6.0. Just check the following notes to see if your script requires adaption.

- Since Hollywood 6.0 comes with built-in support for vector-based drawing, the vectorgraphics library no longer automatically uses the first vectorgraphics plugin it can find. Instead, it will use Hollywood's inbuilt vectorgraphics renderer by default now. If you don't want this, you will have to call `SetVectorEngine()` to tell the vectorgraphics library which plugin to use when drawing vectorgraphics.
- The `ChangeDisplayMode()` command no longer puts all displays onto a single full screen but only the active one is switched to full screen mode. This change was necessary because Hollywood 6.0 introduces multi-monitor support which makes it possible to have multiple displays in full screen mode on separate monitors.
- Hollywood's display handler has been rewritten and does not support the display mode `OwnScreen` any longer. `OwnScreen` was a special mode that could be used on AmigaOS and compatibles to make Hollywood open in fullscreen mode but keep the traditional Amiga screen look, i.e. Hollywood would not open a shielding window that covered the Amiga screen decorations. If you want to achieve the look that the `OwnScreen` display mode gave you with Hollywood 6.0, you have to use the console arguments `-nobackfill` and `-nostyleoverride` together with `-fullscreen`. Then the appearance should be exactly the same as the old `OwnScreen` display mode which is no longer supported by Hollywood 6.0.
- Prior to Hollywood 6.0 display attributes specified in the `@DISPLAY` preprocessor command automatically overrode the display attributes that were specified on the command line, i.e. if the `Borderless` attribute was set to `False` in `@DISPLAY` and the script was started using `-borderless`, then the script would still appear with a bordered window because the specifications in `@DISPLAY` were given priority over the command line specifications. Starting with Hollywood 6.0 this behaviour has been turned around: Display attributes set on the command line will now override display attributes specified in preprocessor commands. If you do not want this behaviour, compile your script using the `-locksettings` mode. Then command line arguments won't be able to override your preprocessor display settings.

- Prior to Hollywood 6.0, command line arguments that affected the display style were applied to all displays defined in the preprocessor commands of your script. For example, if you started a script that defined four displays in the preprocessor commands with the `-borderless` argument, all four displays would be opened in borderless mode. In Hollywood 6.0, command line arguments that modify the display style will only be applied to display number 1 by default. If you want them to affect all displays, you have to use the new `-alldisplays` argument.
- Some command line arguments have been renamed for purely cosmetic reasons: `-audiodev` is now called `-audiodevice` and `-depth` is now called `-scrdepth`. Their functionality hasn't changed.
- Dropped support for `mpega.library` on AmigaOS and compatibles. `mpega.library` caused quite some trouble because it usually recognized every file format as an MPEG stream leading to several unwanted effects. If you need to play MP3s you can just use a plugin like `avcodec.hwp` instead.

Hollywood 5.0 API changes

There have been some small API changes in Hollywood 5.0. Most likely you won't have to adapt your scripts to work with 5.0. Just check the following notes to see if your script requires adaption.

- The plugin interface has been completely rewritten and is no longer compatible with old plugins. On Amiga systems, plugins must always be installed into `LIBS:Hollywood` now. The old plugin location `Hollywood:Plugins` is no longer supported by Hollywood 5.0. Alternatively, plugins can be installed into the program's directory (especially useful when you have to distribute plugins with executables compiled by Hollywood).
- Shadow and border effects on layers will look different (better!) in comparison to previous versions because Hollywood now uses real alpha channel compositing for this. The downside is that the new shadow and border effects are slower than in previous versions.
- Shadow and border are now global settings for every layer. This means that you can no longer have layers that have multiple shadow or border styles (e.g. a text layer where only part of the text has a shadow or a border). That is no longer supported. Either the layer has a shadow/border, or doesn't have one. But it is no longer possible to have a shadow/border for just a part of a layer.
- Layers with a border will be displayed differently now when they are shown or hidden with a transition effect. Because the border is no longer part of the main layer in Hollywood 5.0, the transition effect for the border cannot be combined with that of the main layer any more. Thus, Hollywood will now simply fade in/out the border while the main layer's transition effect is being displayed. If you do not want this behaviour, you can use the new `NoBorderFade` tag.
- For reasons of consistency, Hollywood no longer supports thick layers when the fill style is set to `#FILLNONE` because the thick layer concept clashes with the new layer border concept. Thus, instead of a thickness setting, layers will now get a border of the specified 'thickness' size to make them thicker. The main layer, though, will always have a thickness of 1. If you want to increase this thickness, just enable the layer border and set the desired thickness size in the `BorderSize` tag.

- As Hollywood 5.0 introduces support for vector images, `CreateGradientBGPic()` and `CreateTexturedBGPic()` will now create a vector BGPic for you. This means that you can no longer modify the graphics of these BGPics using the `SelectBGPic()` function. Another difference is that when a textured BGPic gets scaled (e.g. when the user resizes a window), Hollywood will not scale the textured BGPic but will remake it in the new resolution. Prior to 5.0, the textured BGPic just got scaled. Starting with 5.0, the BGPic will be completely remade.
- Prior to 5.0, the `CopyFile()` and `DeleteFile()` functions accepted wildcards in the filename argument. This is no longer supported. If you want to copy/delete files selectively, you have to use an optional argument now. See the documentation of these two functions for more information.
- Prior to 5.0, Hollywood used AmigaOS style pattern matching functions in `MatchPattern()`, `CopyFile()`, and `DeleteFile()`. Starting with version 5.0, Hollywood uses platform-independent pattern matching functions that differ from AmigaOS style patterns in some cases. See Section 26.42 [MatchPattern], page 449, for details.
- Long strings [[...]] behave differently in Hollywood 5.0 if your script was saved using carriage return plus linefeed encoding (CR+LF encoding is the text editor default on Windows; Amiga and Unix systems just use LF characters for line breaks). Previously, carriage return characters ('\r') were always included in the long string. This is no longer the case. All line breaks will be converted to single linefeeds now ('\n'). If a carriage return character is present, it will be dropped. This change has been made to prevent that scripts behave differently when saved on Windows and on AmigaOS/Unix/Mac.

Hollywood 4.5 API changes

There have been some small API changes in Hollywood 4.5. Most likely you won't need to adapt your scripts to work with 4.5. Just check the following notes to see if your script requires adaption.

- `RotateLayer()` will behave differently in 4.5 than it did in 4.0. This break was necessary because Hollywood 4.5 introduces anchor points for layers. In Hollywood 4.0, `RotateLayer()` rotated the layer around its center, thus assuming a 0.5/0.5 anchor point. In 4.5, however, all layers have a default anchor point of 0.0/0.0. Thus, if you would like to replicate the 4.0 behaviour, you need to change the layer's anchor point to 0.5/0.5 by calling `SetLayerAnchor()`.
- executables compiled for macOS will now look in the `"Resources"` folder of the application bundle ONLY for data files. This change was made to comply with macOS UI guidelines. All data files accompanying an application must be put into its app bundle
- when using `CreateSprite()` to create sprite links (i.e. source type `#SPRITE`), you previously could also create links from sprite links. This is no longer possible. When creating sprite links, you always have to specify a sprite that is not linked as the source sprite
- when using `#VANILLACOPY` with `SetAlphaIntensity()`, Hollywood previously sometimes did not draw anything at all. e.g. if you tried to draw a brush with mask to an alpha channel using `SetAlphaIntensity()` with `#VANILLACOPY` set. This behaviour

has changed now: Hollywood will draw the visible mask pixels as 255 alpha intensity now and the invisible mask pixels as 0.

– `SelectBGPic()` had a secret feature that was never documented anywhere (and thus not official): If you used `SelectBGPic()` with layers enabled on the current BGPic, all layers inserted before `EndSelect()` were inserted as hidden layers. This behaviour is gone now. `SelectBGPic()` will now insert normal layers and draw them when `EndSelect()` is called. If you want to have the previous behaviour, create your layers with `Hidden` set to `True`.

Hollywood 4.0 API changes

There have been some small API changes in Hollywood 4.0. Most likely your won't need to adapt your scripts to work with 4.0. Just check the following notes to see if your script requires adaption.

– `SetPointer()` syntax has completely changed. It does no longer accept a filename but requires you to call `CreatePointer()` first.

– all of the transition effects library functions as well as `PlayAnim()`, the MoveXXX() functions & `DisplayBGPicPart()` use a new syntax now. However, the old syntax is still supported for compatibility reasons.

Hollywood 3.1 API changes

There have been some small API changes in Hollywood 3.1. Most likely your won't need to adapt your scripts to work with 3.1. Just check the following notes to see if your script requires adaption.

– Colon is no longer supported as a command separator. In Hollywood 1.x the colon had to be used to separate multiple commands on the same line, e.g.

```
; Hollywood 1.x code - NO LONGER SUPPORTED
x=100:y=200:width=50:height=50:Box(x, y, width, height, #RED)
```

The 1.x emulator inside of Hollywood emulated this behaviour up to Hollywood 3.0. In Hollywood 3.1 it is now no longer supported because the colon is needed for object oriented programming. So you need to update your scripts if you are still using colons to separate multiple commands on a single line. Since Hollywood 2.0, you can put as many commands on a single line as you desire, so the above code could now be written as:

```
x=100 y=200 width=50 height=50 Box(x, y, width, height, #RED)
```

This does not look very nice so you should probably refrain from calling multiple commands on the same line altogether. Of course, the choice is with you. Just keep in mind that Hollywood 3.1 does not emulate the colon behaviour of 1.x any longer.

– The `#TYPEWRITER` transition effect is gone now. This was a special effect which could only be used on text objects. However, it made the font interface unnecessarily complex so it had to go. You can emulate the `#TYPEWRITER` behaviour by just using a series of `Print()` calls.

Hollywood 3.0 API changes

There have been some small API changes in Hollywood 3.0. Most likely your won't need to adapt your scripts to work with 3.0. Just check the following notes to see if your script requires adaption.

- If Hollywood 3 is started without any arguments, it will open in windowed mode. All previous versions opened full screen in that case, but I think it is much wiser to have Hollywood open in windowed mode because full screen mode might not work on every system.
- Command line arguments are now handled differently. You must prefix them with a dash character (-). In previous versions you would call Hollywood like this:

  ```
  Hollywood script.hws WINDOW BORDERLESS
  ```

  This will not work any longer! You now have to use dashes. The correct way to call Hollywood now is:

  ```
  Hollywood script.hws -window -borderless
  ```

  This change was necessary because of the new `GetCommandLine()` function which allows you to work with your own arguments.
- The second argument of `FileRequest()` has changed. Previously, it was a pattern in the AmigaDOS pattern format. Now it is merely a filter string that specifies which files shall be displayed by the requester. This change was necessary because of the new cross-platform nature of Hollywood. Operating systems as Windows and macOS just don't have such elaborate filter pattern handlers as the AmigaOS offers.
- In previous version the optional third argument of the `OpenFile()` function fell back to `#MODE_READWRITE` if it was not specified. This has been changed. Now the default mode is `#MODE_READ`. This is a vanity API break. I just think it makes much more sense to open files in read-only mode by default.

Hollywood 2.5 API changes

There have been some small API changes in Hollywood 2.5. Most likely your won't need to adapt your scripts to work with 2.5. Just check the following notes to see if your script requires adaption.

- Support for `ttengine.library` has been removed. Of course Hollywood does still support true type fonts. The only thing which you can do no longer is to use `SetFont()` on `*.ttf` files directly, i.e.

  ```
  SetFont("dh1:arial.ttf")  ; this is Hollywood 2.0 code!
  ```

  This does not work any longer. In Hollywood 2.5, you can only use true type fonts which have been installed into your system using FTManager or a similar tool. You open them as if they were normal fonts, i.e.

  ```
  SetFont("Arial Narrow.font")  ; OKAY in 2.5!
  ```

  You have to do it this way because Hollywood 2.5 loads all true types through the bullet.library compatible ft2 (OS4) or freetype2 (MorphOS, AROS, AmigaOS3) interfaces respectively.
- The `CheckEvent()` command has been removed. It did not fit into the concept any longer. Please always use `WaitEvent()` instead.

- The `Plot()` command does only work with disabled layers now. Layers of type `#PLOT` are no longer possible. It just does not make much sense to have 1x1 sized layers. If you really need that, you can use the `Box()` command to draw a pixel.
- Due to the new text rendering engine, it is now mandatory to use two square brackets in the strings you pass to `Print()`, `TextOut()` and `CreateTextObject()` when you want to print a single square bracket. For instance, the following code

  ```
  Print("[Hello World]")    ; this is Hollywood 2.0 code!
  ```

  would generate a syntax error in Hollywood 2.5 because the new text engine expects a formatting command after a square brackets. Thus, you would have to write it as follows:

  ```
  Print("[[Hello World]]")  ; OKAY in Hollywood 2.5!
  ```

  Then it will work as you expect it.
- If the fill style is set to `#FILLTEXTURE` or `#FILLGRADIENT` and you draw using a ARGB value, these fill styles will now also respect the alpha value. This was not the case in Hollywood 2.0.
- If layers are enabled and you call a command from the draw library (e.g. `Ellipse()`) and specify an ARGB color (i.e. you want to draw with transparency), Hollywood 2.0 would create a transparent layer for you as if you called the `SetLayerTransparency()` function with the A byte of the ARGB value as the transparency setting. This is no longer done in that way. If you draw with an ARGB color, Hollywood 2.5 will not give the layer a transparency setting, although the layer has of course a transparency now, but with 2.5 the transparency is already rendered to graphics data (i.e. the alpha channel) and is not kept dynamic as in the case of `SetLayerTransparency()`.
- Up to Hollywood 2.0, `RotateBrush()` always returned a brush of the maximum size that a rotation with the source brush could occupy, i.e. maxs = sqrt(width * width + height * height). The new brush allocated by Hollywood would then be of width and height 'maxs'. This is no longer done now. The brush is exactly as big as it needs to be to contain all graphics.
- In Hollywood 2.0, `WriteMem()` and `ReadMem()` always used unbuffered IO while all the other DOS functions used buffered IO. Now all functions have been unified and they use all buffered IO by default. Furthermore, in Hollywood 2.0 `WriteMem()` always automatically flushed buffers before starting the write operation. This is no longer done in 2.5. So, if you used `WriteMem()`/`ReadMem()` in your scripts and you need it to have unbuffered IO like in 2.0, you first have to call `SetIOMode()` to change the IO mode to unbuffered. Then it will work as you are used to it but remember that it does not flush the buffers as in Hollywood 2.0. And remember that once you call `SetIOMode()` all other DOS functions will also use the IO mode set here! If you only want unbuffered IO for `WriteMem()` or `ReadMem()` you have to use `SetIOMode()` again after your call. You also have to call `FlushFile()` manually if you switch from buffered to unbuffered IO on the same file. This all might sound a bit complicated, but it is really easy. In fact, it gives you full control over the DOS functions which can come in pretty handy at many times. Please see the documentation of `SetIOMode()` for more information.
- Up to Hollywood 2.0, `TextOut()` would automatically align the text if a special coordinate constant like `#CENTER` or `#RIGHT` was specified as `x`. This is no longer done in this way. There is a new argument which you can use to specify the desired alignment.

Hollywood 2.0 API changes

Although Hollywood 2.0 is a gigantic update, only little API changes were necessary. Here is a list of things you have to change in your script:

- If you call functions that do not accept any arguments but return a value, you have to use brackets. For example, the following code worked in 1.9 but does no longer work in 2.0:

```
; wrong!
x = MouseX
y = MouseY
```

  You have to write this as:

```
; correct!
x = MouseX()
y = MouseY()
```

  The wrong version will not trigger a compiler error by the way. It is correct Hollywood code but does something completely different: It assigns the function `MouseX()` to the variable `x` which is not what you want.

- `GetTimer()` always returns the value in milliseconds now. In Hollywood 1.x the default unit was seconds. This is a vanity API break. Of course, I could have kept the old implementation but honestly, there's noone who wants a return value in seconds because it is just too unprecise. Thus, I decided to do programmers a favour and make milliseconds the default, so you do not have to type the lengthy GetTimer(1, `#MILLISECONDS`) every time but just GetTimer(1).

- The `MoveBrush()`, `MoveTextObject()`, `MoveAnim()`... functions can no longer "grab" old objects. For example, the following code does not work correctly in 2.0:

```
MoveBrush(1, #LEFTOUT, #CENTER, #CENTER, #CENTER)
Wait(100)
MoveBrush(1, #CENTER, #CENTER, #RIGHTOUT, #CENTER)
```

  In Hollywood 1.x, this code moved the brush 1 from the outer left to the center, waited 100 ticks, and moved the brush to the outer right. In Hollywood 2.0 it will do the same, but the a copy of the brush will remain in the center of the display. This is due to major changes in the refresh system. If you want to imitate the 1.x behaviour, use `MoveSprite()` instead of `MoveBrush()`.

- `DisplayTransitionFX()` can no longer be used to display transparent background pictures; switching to transparent BGPics can only be done without an effect now. This is because Hollywood 2.0 uses real transparent windows on MorphOS, OS4 and AROS now. Those windows have a layer where no graphics can be drawn.

- In Hollywood 1.x, `MixBrush()` scaled the two brushes to the same size if they were of different dimensions. This is no longer done. `MixBrush()` just mixes the parts that match and drops the rest.

- `RotateBrush()` will now create a mask for the brush if it does not have one. You no longer have to do this on your own.

- If layers are enabled and you use `InKeyStr()` only one layer of type `#PRINT` will be installed by `InKeyStr()`. In Hollywood 1.x, `InKeyStr()` left a `#PRINT` layer for each character.

Hollywood 1.9 API changes

There were some minor API changes in Hollywood 1.9, which are listed here:

- the commands `EnableEventHandler()` and `DisableEventHandler()` were removed. They could cause much trouble because if you use them, you do not know when your event procedures are called. Please use the new `CheckEvent()` function now!
- `EnablePrecalculation()` and `DisablePrecalculation()` were removed because effect precalculation is no longer supported by Hollywood. The PRECALCULATION argument/tooltype is also gone now.
- `WhileMouseOn()` had some changes that you will most likely not notice but under certain circumstances you might get a problem with it now: In earlier versions, Hollywood would immediately jump back to your `WaitEvent()` loop after an ONBUTTONCLICK event occurred. This was a wrong behaviour! Now it will jump back to your `WhileMouseOn()` command because the mouse is still over your button after ONBUTTONCLICK occurred. If you want Hollywood 1.9 to behave like Hollywood 1.0 and 1.5 did, you need to use the new `BreakWhileMouseOn()` command

Hollywood 1.5 API changes

Unfortunately I had to make some API changes to the Hollywood language in the 1.5 update. If your script does not work correctly under Hollywood 1.5 but worked under 1.0, please read the following information and adapt your script.

- the constant syntax has changed. In Hollywood 1.0 you just specified constants by their name but now you will have to specify also a '#'-prefix. So you have to specify e.g. `#CENTER` instead of CENTER and `#BOLD` instead of BOLD. I'm sorry but this change was absolutely necessarily.
- `Undo()` will not work until you have called `EnableLayers()`. If you are using `Undo()` in your script, make sure you call `EnableLayers()` at the beginning.
- syntax of `PlaySample()` has changed. You can no longer specify a channel for playback. Hollywood will do everything for you. Just specify the sample number and if it shall be looped or not.
- syntax of `PlayAnim()` has changed. It runs now synchronously. This change was necessary because the old `PlayAnim()` implementation did no longer fit in the concept. If you need to play anims asynchronously, use brush links of frames and display them with `DisplayBrush()`. Because `PlayAnim()` is synchronous now, the commands `IsAnimPlaying()` and `WaitAnimEnd()` are no longer required and were removed.
- `ClearScreen()` was removed because it did no longer fit in the concept.
- `LoadModule()` does not load THX, P61 or MED modules any longer. Module support now concentrates on the Protracker format. Other module formats cannot be played back cleanly through AHI.
- `Print()` does no longer support anti alias for true type fonts. This change was currently necessary to stay compatible with layers. Anti aliasing will be re-introduced for all objects in Hollywood 2.0.

6.3 Future

Here are some ideas that are on my to do list:

- speed of all the transition fx functions should be passed in milliseconds instead of a custom type; would help to time scripts correctly
- text transition effects
- API for creating video streams with Hollywood
- faster drawing using polygon clipping when possible
- more features

Please drop me a mail if you have some nice ideas what shall be implemented in Hollywood.

7 Language overview

7.1 Your first Hollywood program

Hollywood's script language is easy to use but very powerful! The syntax is based mainly on BASIC but Hollywood is much more powerful because it is a dynamically typed language! We will figure out later what this means for the programmer. Hollywood incorporates the best elements of (Blitz-) BASIC, C, AmigaE, Pascal and Lua into one powerful, flexible language that allows you to do almost everything with little effort.

Hollywood scripts are just plain text files in UTF-8 encoding. So fire up your favorite text editor now and start creating your first script!

This is how the famous 'Hello World' program looks in Hollywood:

```
Print("Hello World!")
WaitLeftMouse()
End()
```

The little program above will open a 640x480 display. If you want Hollywood to open a display with other dimensions, you will need to use the `@DISPLAY` or the `@BGPIC` preprocessor command. 640x480 is the default display size that Hollywood uses when you do not specify anything else. The display size is not the same as the screen size. It is just the size of your display (your work area!). The screen size can be anything which is large enough to hold the display. Your display will be centered on the screen (you can use the `@DISPLAY` preprocessor command if you want a different initial display position). The window that is opened and holds the display will be larger than your display size if it has borders. If you specify the `-borderless` argument the window's size will match your display size.

If you want to have a fancy background picture instead of a plain black background, just place the `@BGPIC` preprocessor command at the beginning of your script:

```
@BGPIC 1, "FancyBackground.jpg"
Print("Hello World!")
WaitLeftMouse()
End()
```

You can also place multiple commands in one line, so the above code could also be written like this:

```
Print("Hello World!") WaitLeftMouse() End()
```

However, it is advised to use line feeds to make your code better readable. To achieve this, you can also use comments starting with a `/*` and ending with a `*/` or just a single line comment starting with `;`, e.g.:

```
/* this is a comment */
Print("Hello World!")   ; this one too
WaitLeftMouse()   ; Wait for left mouse
End()  ; Exit
```

If a Hollywood function does neither accept nor return any arguments, you can leave out the parentheses when you call the function. If you pass arguments to a function however or if you want to store the return value of a function, you have to use parentheses. In our

example, we could leave out the parentheses for the `WaitLeftMouse()` and `End()` commands because they do not take any arguments:

```
Print("Hello World!")
WaitLeftMouse
End
```

Of course, it is also possible to use variables instead of direct numbers or strings. You do not need to declare variables, they will be initialized to zero or an empty string respectively when you first use them. Variables have to start with a letter from A/a to Z/z or with an underscore. After that, they can also contain the numbers from 0 to 9, the dollar sign ($) and the exclamation mark (!). As a matter of style, variables that hold strings should have a dollar sign as their last character and variables that hold floating point values should have an exclamation mark as the last character. This makes your code better readable. The length of a variable name must not exceed 64 characters.

```
mystring$ = "Hello World!"
Print(mystring$)
WaitLeftMouse
End
```

Besides normal commands, there are also preprocessor commands available in Hollywood. These commands are processed before the script execution starts and they are always prefixed with an @-character (at). One of those preprocessor commands is `@VERSION`. It allows you to define the version of Hollywood that the script requires as a minimum. For example, the following script will only work with Hollywood 2.0 and higher:

```
@VERSION 2,0
Print("Hello World!")
WaitLeftMouse
End
```

You should always use this preprocessor command as the first action of your script to make sure the version is checked before anything else.

If you type the code above in your text editor and save it as `MyScript.hws`, you can then start it from a console by typing:

```
Hollywood MyScript.hws [ARGUMENTS]
```

`[ARGUMENTS]` can be any combination of console arguments supported by Hollywood. See Section 3.2 [Console arguments], page 33, for more information on supported arguments.

If you want to start your script through the GUI, start the GUI, click on "Display" and choose your script.

Congratulations, you have just created your first Hollywood script!

7.2 Reserved identifiers

The following identifiers are reserved by Hollywood and cannot be used as variable or function names:

```
And
Block
Break
```

```
Case
Const
Continue
Default
Dim
DimStr
Do
Else
ElseIf
EndBlock
EndFunction
EndIf
EndSwitch
FallThrough
False
For
Forever
Function
Global
Gosub
Goto
If
In
Label
Local
Next
Nil
Not
Or
Repeat
Return
Step
Switch
Then
To
True
Until
Wend
While
Xor
```

If you attempt to use one of those as a function or variable name, you will get an error from Hollywood.

7.3 Preprocessor commands

A preprocessor command is a command that Hollywood processes before actually running your script. In Hollywood they are mainly used to preload data before the script is started.

For example, if your script requires the files `mainmenu.png`, `gamescreen.png` and `music.mod` in any case, you could simply preload them by using the following code:

```
@BGPIC 1, "mainmenu.png"
@BGPIC 2, "gamescreen.png"
@MUSIC 1, "music.mod"
```

Hollywood will then load all those files before actually running your script. All files loaded via preprocessor commands are immediately ready for use when your script starts. Most of the `LoadXXX()` commands have their preprocessor command equivalent in Hollywood. For instance, the preprocessor equivalent of `LoadBrush()` is `@BRUSH`, the equivalent of `LoadBGPic()` is `@BGPIC` and so on.

Preprocessor commands are always prefixed by an at character (@). You should also write them in capital letters so that they can be distinguished better from normal commands. Preprocessor commands can be placed anywhere in the script, but for readability reasons it is suggested to put them at the beginning of your script.

An elementary preprocessor command is the `@VERSION` command. You should use it as the first thing in each of your scripts! `@VERSION` checks if the Hollywood version used is sufficient for running the script. Otherwise, Hollywood will abort.

Most preprocessor commands take several arguments which are separated by commas just like with normal commands. You can also use expressions in the preprocessor commands. For instance, the following declaration would be uncommon but perfectly valid:

```
@BRUSH 5+5, "MyBrush.png"
```

This would load `MyBrush.png` as brush number 10. What you cannot do, however, is using variables in your expressions. When Hollywood parses the preprocessor commands, it does not know anything about variable states because the script has not been started yet. Thus, all expressions you use must be constant.

Another advantage of the preprocessor commands is that all files specified here will be automatically linked into the executable when you compile your script. This behaviour can be changed by using the `Link` tag that is accepted by all preprocessor commands that work with files. This tag tells the Hollywood linker whether or not the file of that preprocessor command should be linked into the executable or applet when you compile a script. The `Link` tag always defaults to `True` which means that by default all files loaded through preprocessor commands will be linked to your executable or applet. If you do not want certain files to be linked, for example because they are too large, you have to specify this explicitly in the corresponding preprocessor commands.

The following preprocessor commands are available:

```
@ANIM            Preload an animation
@APPAUTHOR       Declare application author
@APPCOPYRIGHT    Declare application copyright
@APPDESCRIPTION  Declare application description
@APPENTRY        Declare application entry script
@APPICON         Declare application icon
@APPIDENTIFIER   Declare application identifier
@APPTITLE        Declare application title
@APPVERSION      Declare application version
@BACKFILL        Choose a backfill for your script
```

```
@BGPIC          Preload a background picture
@BRUSH          Preload a brush
@CATALOG        Preload a catalog
@DIRECTORY      Link whole directory into applet or executable
@DISPLAY        Configure display settings
@ELSE           Block to enter if all conditions failed
@ELSEIF         Test for another condition
@ENDIF          Declare end of conditional block
@ERROR          Abort compilation with an error message
@FILE           Open a file
@FONT           Preload a font
@ICON           Preload an icon
@IF             Test for condition
@INCLUDE        Include code from another file
@LINKER         Pass options to the linker
@MENU           Create a menu strip
@MUSIC          Preload a music file
@OPTIONS        Configure miscellaneous options
@PALETTE        Preload a palette
@REQUIRE        Declare a plugin dependency
@SAMPLE         Preload a sample
@SCREEN         Configure screen mode for your script
@SPRITE         Preload a sprite
@VERSION        Define which Hollywood version your script requires
@VIDEO          Preload a video
@WARNING        Send warning message to debug device
```

7.4 String and number conversion

Hollywood supports automatic string to number and number to string conversion. That means that if a function expects a string in an argument and you pass a number then Hollywood will automatically convert this number into a string and pass it to the function as a string.

For example: `StrLen()` returns the length of the specified string. Now if we call

```
a = StrLen(256)
```

Hollywood will automatically convert the number 256 to the string "256" and therefore the variable `a` receives the value of 3 because the string "256" consists of three characters.

This works the same way vice versa. If you pass a string to a function that expects a number then Hollywood will try to convert this string to a number. The difference to the number to string conversion is now that the string to number conversion might fail. For example: Hollywood cannot convert a string like "Hello" to a number. The string must contain decimal or hexadecimal digits only. Mixed alphabetical and number strings cannot be converted either, even if the digits come before the characters. Hexadecimal numbers must be prefixed with a dollar sign ($) or 0x. An example:

```
LoadBrush("1", "Brush.iff")
```

`LoadBrush()` expects a number as the identifier. Thus, the string `"1"` will be automatically converted to a number by Hollywood. This in contrast will not work:

```
LoadBrush("Test", "Brush.iff")
LoadBrush("1Test", "Brush.iff")
```

The strings "Test" or "1Test" cannot be converted to a number.

You can also use all of the operators with numbers and strings except the relational operators. They can only compare two values of the same data type. For example, the following code works fine:

```
a = "5" * 10 + 100 / "10" + ("100" - 60)  ; a is 100
```

But this code will give you an error because you use relational operators with values of different types:

```
If "10" < 20       ---> Error!
```

If you want to do something like this, you have to use `Val()` or `StrStr()` to convert the number manually to string or the string to number. E.g.

```
If Val("10") < 20  ---> Works!
```

7.5 Comments

Hollywood supports two types of comments: A newline terminated and a user terminated comment. The newline terminated comment starts if Hollywood discovers a semicolon in your code. Hollywood will ignore everything after that semicolon then and continue parsing in the next line. For example:

```
DebugPrint("Hello")   ; Hi I'm a newline terminated comment!
```

The second version needs to be terminated by the user. You start this comment with the character sequence `/*` and end it with a `*/`. Because this comment is user terminated, it can run over several lines. But it can also be in the middle or at the beginning of a line. Examples:

```
/*
Everything in here will be ignored by Hollywood!
*/
DebugPrint("Hello") /* Hello I'm a comment */ DebugPrint("World")
```

Please comment your code! You do not have to comment every little local variable but giving functions a short description does not hurt and makes it easier for other people to understand the program.

7.6 Includes

Includes can be used to import code from a separate file to the current Hollywood script. You can import Hollywood source code (`.hws` files) as well as Hollywood applets (`.hwa` files). The code that you import from these external files will be linked into your current project so that these files are not required by compiled Hollywood projects. Importing code is especially useful for bigger projects because it can easily get quite complex to overlook if you have only one source code file with lots of code in it. The idea of include files is to split your program into several pieces. For instance, a jump'n'run game could be split into the pieces Intro, Menu, MapEngine, Level and Game. Now you create source code files for

every piece, e.g. `Intro.hws`, `Menu.hws`, `MapEngine.hws`, `Level.hws` and `Game.hws`. One of the source code files must be the main source code, that is the source code that you start with Hollywood.

Another use could be to create libraries for Hollywood in the form of Hollywood applets. You could then publish these applets so that other programmers can benefit from them by importing the applet into their own projects. The advantage of publishing your library as a Hollywood applet is that you will not have to expose the source code of your library. Hollywood applets contain only precompiled bytecode that is not human readable. So if you want to protect your code but still want to share it with other users, then you can simply publish it as a Hollywood applet.

Let us return to the example of a jump'n'run game now which spreads its code over several files. We assume that `Intro.hws` will be our main source code because the intro is the first thing, that the end-user will see. Our `Intro.hws` header will look like the following then:

```
@INCLUDE "Menu.hws"
@INCLUDE "MapEngine.hws"
@INCLUDE "Level.hws"
@INCLUDE "Game.hws"

ShowIntro()
ShowMenu()          ; Function ShowMenu() declared in Menu.hws
RunGame()           ; RunGame() declared in Game.hws
DrawMap()           ; DrawMap() declared in MapEngine.hws
NextLevel()         ; NextLevel() declared in Level.hws
```

You see that we use the `@INCLUDE` preprocessor command to include the other four source files in our `Intro.hws` file. This allows us to call all functions that are declared in those four files from our main source code, i.e. from `Intro.hws`.

Included files contain only functions, variable or constant declarations in most cases. If there are immediate statements in your include files, e.g. `DebugPrint("Hello")`, they will be executed before any code from the main source code because all include files are inserted in the order they are declared into the main source code file. In our example from above, Hollywood would first open `Menu.hws` and insert its code, then `MapEngine.hws`, then `Level.hws` and finally `Game.hws`. So what Hollywood compiles would look like the following:

```
@INCLUDE "Menu.hws"
@INCLUDE "MapEngine.hws"
@INCLUDE "Level.hws"
@INCLUDE "Game.hws"

<...contents of file Menu.hws...>
<...contents of file MapEngine.hws...>
<...contents of file Level.hws...>
<...contents of file Game.hws...>
ShowIntro()
ShowMenu()
...
```

You see that all include files are inserted before the code section of your main source code file. Therefore all immediate statements will be executed before the code of the main source code too.

If you want to include applets, simply pass an applet file to the `@INCLUDE` preprocessor command:

```
@INCLUDE "Test.hwa" ; import functions from Test.hwa

LibFunc() ; call LibFunc() which was defined in Test.hwa
```

7.7 Error handling

There are several ways of dealing with errors in Hollywood. The easiest is to let Hollywood do everything for you, which is the default behaviour. By default, Hollywood will always terminate your script when an error occurs inside a Hollywood function. Consider the following code:

```
LoadBrush(1, "xyz")
```

If the file `xyz` does not exist, Hollywood will terminate your script and show an error that says: "Cannot read file xyz!"

If you do not like this behaviour, you can also tell Hollywood to call a function provided by you whenever an error occurs. This is possible by calling the `RaiseOnError()` function and providing a callback function that Hollywood should run whenever an error occurs. Here is how you can replace Hollywood's default error handler with a custom error handler:

```
Function p_ErrorFunc(code, msg$, cmd$, line)
   DebugPrint(code, msg$, cmd$, line)
EndFunction
RaiseOnError(p_ErrorFunc)
LoadBrush(1, "xyz")
```

If you use the code above, calling `LoadBrush()` with a brush that doesn't exist, won't trigger Hollywood's default error handler but will instead call the user function `p_ErrorFunc()` and pass further information about the error that has just occurred to it. See Section 28.7 [RaiseOnError], page 546, for details.

Sometimes, however, it can be useful to know if a single call succeeded or not. This can be achieved by temporarily disabling Hollywood's error handler and getting the error code from the last function call, for example like this:

```
ExitOnError(False)       ; disable default error handler
LoadBrush(1, "xyz")
error = GetLastError()
ExitOnError(True)        ; enable default error handler again
```

The code above temporarily disables Hollywood's default error handler just for the duration of the `LoadBrush()` call. Right after the `LoadBrush()` call we use `GetLastError()` to find out if the `LoadBrush()` call has succeeded or not. It is important to call `GetLastError()` immediately after `LoadBrush()` because the internal error flag will be reset whenever a Hollywood command is executed so if you call another function after `LoadBrush()` `GetLastError()` will return the error state of this function instead of `LoadBrush()`.

Since the code above requires lots of typing for a rather simple thing, there is also some syntactic sugar which does the same as the code above while dramatically reducing the amount of typing that is required. Instead of calling `ExitOnError()` and `GetLastError()` manually like shown above, you can also have Hollywood do all that automatically for you by simply prefixing function calls with a question mark. Thus, the code above could also be written like this:

```
error = ?LoadBrush(1, "xyz")
```

In case a function returns other values and you use a question mark to obtain an error code from a function call, all other return values are simply shifted down. The error code will always be the first return value. For example, if we want to use automatic ID selection with `LoadBrush()` and combine this with the question mark syntax, we have to write the code like this:

```
error, id = ?LoadBrush(Nil, "xyz")
```

Normally, `id` would be the first return value but since we use the question mark syntax to obtain an error code from `LoadBrush()`, the first return value is shifted down and becomes the second return value now because the error code will always be in the first return value.

Finally, to check whether an error has occurred or not, you just have to compare the error code `error` against `#ERR_NONE`, which is defined as 0 for convenience, i.e. whenever `error` is not 0 you know that something went wrong. You could then use `GetErrorName()` to convert the error code into a human-readable string or implement some custom error handling depending on the error code that has been set. See Section 28.3 [Error codes], page 507, for a list of all error codes.

Please note that there are some errors that cannot be caught. For example, if you pass the wrong number of arguments to a function or you pass wrong variable types to a function, Hollywood will always exit immediately with a fatal error and your script won't be given a chance to catch such errors. Even though they occur at runtime, Hollywood will consider such errors syntax errors and will immediately exit. Here is an example where we pass a string in the first argument of `LoadBrush()` which is forbidden because `LoadBrush()` expects a number:

```
ExitOnError(False)
LoadBrush("Hello", "xyz")
ExitOnError(True)
```

Although we have disabled Hollywood's error handler by passing `False` to `ExitOnError()`, Hollywood will still immediately halt the script's execution because passing `"Hello"` to `LoadBrush()` is just plain wrong and Hollywood will consider this a major mistake and won't allow your script to intercept this error in any way.

7.8 Automatic ID selection

You can pass `Nil` to all functions that ask you to specify an identifier for the new Hollywood object. In that case, Hollywood will automatically choose an identifier and return it to you. This is especially useful for larger projects. If your project is small it is more convenient to use hard-coded ids, e.g.

```
LoadBrush(1, "brush1.iff")
LoadBrush(2, "brush2.iff")
```

```
LoadSample(1, "sample.wav")
OpenFile(1, "file.txt")
```

However, when your project grows larger id management can get quite confusing and nobody wants to mess around with a myriad of different ids. Thus, you can simply pass `Nil` instead of an id and Hollywood will return an id for the new object that is guaranteed to be unique because it uses the special variable type `#LIGHTUSERDATA`. That way, it is ensured that no id conflicts will arise because if you pass `Nil`, Hollywood will not choose an id from the id pool (i.e. integer numbers from 1 to n) but it will create unique ids. Thus, all normal ids will still be available for use, e.g.

```
brush1 = LoadBrush(Nil, "brush1.iff")
brush2 = LoadBrush(Nil, "brush2.iff")
sample1 = LoadSample(Nil, "sample.wav")
file1 = OpenFile(Nil, "file.txt")
```

The variables `brush1`, `brush2`, `sample1`, and `file1` will not receive any human readable ids but special ids of type `#LIGHTUSERDATA`. Thus, all human readable ids from 1 to n will still be available. Therefore, you do not have to worry about any id conflicts when passing `Nil` to object creation function, because they cannot occur as Hollywood uses two separate id dimensions: One human readable that is only used when you pass an id to the object creation functions and one opaque id mechanism that is used when you pass `Nil` to the object creation functions.

7.9 Loaders and adapters

Many Hollywood functions support loaders and adapters. The difference between a loader and an adapter is the following: A loader adds support for additional image, sound, video, icon, font, or animation formats while an adapter replaces certain parts of Hollywood with an own implementation. For example, there are display adapters which can be used to replace Hollywood's inbuilt display handler with a custom one (e.g. displays managed by SDL or OpenGL), there are network adapters which allow plugins to override Hollywood's inbuilt network implementation and of course there are file adapters.

File adapters can, for example, be used to add support for a new container formats. They can be tied to loaders in a way that the adapter provides the raw data which is then interpreted by a loader later in the process. For example, an adapter could provide support for reading files compressed by gzip. The data thus extracted by an adapter could then be handled by a loader. For example, there could be a BMP picture inside a file compressed by gzip: Hollywood would then first ask the adapter to provide the uncompressed data of the gzip file and then ask the loader to load the actual BMP picture. A file adapter could also implement data streaming from a random source, e.g. from HTTP server or other sources.

Starting with Hollywood 6.0 almost all functions that deal with files allow you to specify a loader and/or an adapter in their optional table argument. The idea behind this design is to speed up loading of external data. If you do not specify a loader or an adapter, Hollywood will ask all loaders and all adapters that are currently installed whether or not they want to open this file. Depending on how many plugins you have installed, this can slow down things quite considerably when many files need to be loaded. If you know the loader that should load your external data or the adapter that should handle it, you can pass its name to the loading function to speed up the loading process.

The string you pass to the `Loader` or `Adapter` tags accepted by the optional table argument of almost all functions that deal with files needs to be composed of at least a single loader or adapter name, or a reserved keyword describing a special loader or adapter. Multiple names and keywords have to be separated by a vertical bar character (`|`). The following reserved keywords are currently recognized:

`Default:` This is the default operation mode. This cannot be combined with any other keywords or loader and adapter names. It must always be used independently of the others. In default operation mode, Hollywood will first ask all the loaders and adapters made available by plugins whether or not they want to handle a file. If there is no plugin which wants to handle the file, Hollywood's inbuilt handlers will be asked to deal with it. If no inbuilt handlers recognize the file, native loaders of the host OS loaders will be asked to load the file.

`Inbuilt:` If this keyword is specified, Hollywood's inbuilt loaders will be asked whether or not they want to load the file. Hollywood's inbuilt loaders support the following file formats:

`Inbuilt image loaders:`
IFF ILBM, JPEG, PNG, GIF, and BMP.

`Inbuilt anim loaders:`
IFF ANIM, GIF ANIM, and AVI MJPEG.

`Inbuilt sound loaders:`
IFF 8SVX, IFF 16SV, RIFF WAVE, and Protracker.

`Inbuilt video loaders:`
CDXL video.

`Native:` If this keyword is specified, Hollywood will ask the host OS to try to load the file. This is only supported for certain types and operating systems. Here is an overview:

`AmigaOS:` Passing `Native` as the loader will use datatypes to load images, animations, and sounds. There is no native video loader.

`Windows:` There are native sound and video loaders based on DirectShow and Media Foundation and native image and anim loaders based on the Windows Imaging Component.

`macOS:` There are native sound, video, and image loaders based on macOS technologies. There is no native anim loader.

`Linux:` There are no native loaders at all.

`iOS:` There are native sound, video, and image loaders based on iOS technologies. There is no native anim loader.

`Android:` There are no native loaders at all.

`Plugin:` If this keyword is specified, Hollywood will ask all plugins whether they want to handle the file. Plugins will be asked in the order they were loaded by Hollywood which is rather random because it depends on the order they are returned to Hollywood by the file system. If you want to make sure a certain plugin is asked

to handle a file before another plugin, you need to explicitly include the name of this plugin in the string instead of using the generic `Plugin` keyword.

If you use a general keyword like `Plugin` and you need to find out which loader or adapter was used to load a file, you can query the `#ATTRLOADER` or `#ATTRADAPTER` attributes using `GetAttribute()` to find out which loader or adapter opted to handle the file. Loaders and adapters may also provide a format name for the file they loaded. You can get this by querying the `#ATTRFORMAT` attribute.

Here are some example specifications:

```
LoadBrush(1, "test.png", {Loader = "inbuilt"})
```

The code above will load the specified file using the inbuilt PNG image loader. Neither image plugins nor host OS loaders will ever be asked whether they want to load this file.

```
LoadBrush(1, "test.png", {Loader = "myplugin"})
```

The code above will ask `myplugin.hwp` to load the file `test.png`. If `myplugin.hwp` fails to load the file, `LoadBrush()` will fail as well. It will not fall back to the inbuilt image loader. If you want `LoadBrush()` to fall back to the inbuilt image loader, you will have to add it to the string you pass in the `Loader` tag, e.g.:

```
LoadBrush(1, "test.png", {Loader = "myplugin|inbuilt"})
```

In that case, `LoadBrush()` will use the inbuilt image loader in case the loader provided by `myplugin.hwp` fails. The following code will work on AmigaOS, Windows and macOS but will fail on all the other platforms since they do not have a native image loader (see above):

```
LoadBrush(1, "test.png", {Loader = "native"})
```

On AmigaOS, `test.png` will be loaded via datatypes, on Windows using the Windows Imaging Component and on macOS and iOS it will be loaded via CoreGraphics. On Linux and Android, however, it will fail because Hollywood does not have a native image loader on these platforms.

You can also use the `Adapter` and `Loader` tags together, for example like this:

```
LoadBrush(1, "test.bmp.gz", {Adapter = "gzip", Loader = "inbuilt"})
```

The code above will first pass the file `test.png.gz` to `gzip.hwp` so that it can unzip it and then the unzipped BMP picture will be loaded using the inbuilt image loader. Of course, you could also just write the following code and it would work as well:

```
LoadBrush(1, "test.bmp.gz")
```

However, there is some overhead here because Hollywood will first ask all file adapters whether they want to handle `test.bmp.gz` and after that Hollywood will ask all image loader plugins whether they want to load the file or not. Depending on how many plugins you have installed, this can take quite some time. So if you know which adapter and loader you want to use, it will increase the loading speed if you specify loader and adapter names directly.

Also note that if you pass multiple loaders or adapters to a Hollywood function and separate them using a vertical bar character (`|`), the order of the individual loaders and adapters will matter. For example, the following code will first ask the plugin `digibooster.hwp` to open the file, and then it will ask the plugin `xmp.hwp` to open it:

```
OpenMusic(1, "shades.dbm", {Loader = "digibooster|xmp"})
```

This also allows you to prioritize certain generic loaders like `Native`, `Inbuilt` and `Plugin`. As described above, by default Hollywood will first ask plugin loaders, then inbuilt loaders, then native loaders to open a file. If you want native and inbuilt loaders to be asked before ones provided by plugins, you could pass the string `native|inbuilt|plugin` to achieve that. With such a loader string, native loaders would be asked first and plugin loaders would be asked last. You could also change the default order globally by using the `SetDefaultLoader()` and `SetDefaultAdapter()` functions, respectively. See Section 52.27 [SetDefaultLoader], page 1092, for details.

Another advantage of directly specifying a loader or an adapter is that it allows you to access loaders and adapters which are hidden from general usage. Plugin authors can decide to write loader or adapter plugins that are not automatically available once Hollywood has loaded the plugin but can only be used either by explicitly calling `@REQUIRE` on the plugin or by directly passing the plugin's name to the `Loader` or `Adapter` tags. So these two tags can also be used to address hidden plugins directly.

Starting with version 8.0, Hollywood also supports network adapters now. Those adapters follow the same principle as file and directory adapters, i.e. they can be used to route the functions of Hollywood's network library through custom handlers implemented as Hollywood plugins. Network adapters, for example, can enhance the functionality of `DownloadFile()` and `UploadFile()` to support TLS/SSL connections as well. They could also be used to implement support for completely different network types and protocols because Hollywood's network adapter interface is completely abstracted from any kind of specific networking API, which makes it very flexible to adapt to new environments.

Starting with version 10.0, Hollywood also supports filesystem adapters. Filesystem adapters can be used to replace core filesystem features like renaming files and directories, creating directories, moving files and directories, etc. with customized ones. Several Hollywood functions support filesystem adapters, e.g. `CopyFile()`, `MakeDirectory()`, or `DeleteFile()`.

7.10 User tags

User tags are a way of passing additional information from Hollywood scripts to plugins. They can be used to pass an unlimited amount of additional data to plugins directly from the Hollywood script. Most of the Hollywood commands that support plugins also allow you to pass user tags that should be forwarded to plugins. Of course, this makes only sense if the user data is actually recognized by the respective plugin.

For example, let's suppose there's a plugin that can load PDF pages as images. This makes it possible to use Hollywood's `LoadBrush()` command to create a brush from a PDF page. However, functions like `LoadBrush()` don't support specifying a page number or a password because they're not designed to load pages from PDF documents. A plugin could deal with this limitation by simply defining two new user tags, e.g. `Page` and `Password`, and then scripts could use these two tags to pass the information to the plugin.

From the Hollywood script's point of view, user tags are simply passed in an optional `UserTags` table accepted by many Hollywood functions, e.g. `LoadBrush()`. The `UserTags` table can contain an unlimited amount of key-value pairs that define individual user tags. Note that the key must always be a named table index like `Page` or `Password`. It's not

possible to use numeric table indices as user tags. The value can be a string or a numeric value. If it is a string, it can also contain binary data.

To come back to our example from above, to pass a page and a password to an image plugin via user tags, a Hollywood script could simply do the following:

```
; load page 5 of test.pdf as a brush, passing "mypwd" as the password
LoadBrush(1, "test.pdf", {UserTags = {Page = 5, Password = "mypwd"}})
```

A plugin could then look for the tags `Page` and `Password` to find out the page number to load and the password for the PDF (if any). This makes it possible to pass all kinds of additional information to Hollywood plugins.

Furthermore, user tags are also supported by many preprocessor commands so you could also do the following to load page 5 of `test.pdf` as a brush:

```
; load page 5 of test.pdf as a brush, passing "mypwd" as the password
@BRUSH 1, "test.pdf", {UserTags = {Page = 5, Password = "mypwd"}}
```

Note that user tags are supported by all kinds of plugins: They are supported by image loaders, anim loaders, sound loaders, video loaders, icon loaders, font loaders, file adapters, directory adapters, display adapters, network adapters, and serializers. Loaders typically forward the user tags to adapters as well so that file adapters will be able to listen to user tags passed through loaders.

7.11 Styleguide suggestions

Here are some suggestions to keep your code readable. As you have read before, Hollywood does not distinguish between capitals and small letters but to keep your code readable, we suggest the following styleguide rules:

- always write inbuilt commands like they appear in this documentation but at least begin them with a capital
- write constants in capitals to distinguish them from variable names
- write all preprocessor commands in capitals to highlight them
- one command per line is usually enough!
- use the "$" character only in string variables to avoid confusion
- use the "!" character only in variables that carry floating point values to avoid confusion
- some comments won't hurt either
- when using If blocks and loops, you should use tabs to structure the different levels
- you should prefix your own functions with a "p_" to distinguish them from Hollywood functions (there might also be Hollywood functions in future versions which have the same name as your functions which could lead to unexpected results)

8 Data types

8.1 Overview

This chapter covers all data types that are available in Hollywood. The following five data types are offered by Hollywood:

```
Number    Numeric values like 1, 2, $FF, 3.5, 1.7e8, True, False, 'a'
String    Sequences of characters; usually used for text e.g. "Hello"
Table     Collections of data items of the same or different types
Function  User-callable routines
Nil       Means that a variable does not have any value
```

You can find out the data type of a variable by using the `GetType()` command. E.g.

```
GetType(1)            ---> returns #NUMBER
GetType(2.5)          ---> returns #NUMBER
GetType(True)         ---> returns #NUMBER
GetType('x')          ---> returns #NUMBER
GetType(#STRING)      ---> returns #NUMBER
GetType("What am I?") ---> returns #STRING
GetType({1, 2, 3})    ---> returns #TABLE
GetType(DebugPrint)   ---> returns #FUNCTION
GetType(Nil)          ---> returns #NIL
```

8.2 Numbers

The number type can be used to store integer and real numbers. Internally, all numbers are stored as 64-bit floating point values which means that it can represent very large integers and very precise real numbers. The number type can store numbers ranging from 1.7*10^-308 to 1.7*10^308. The integer range is from -9007199254740992 to 9007199254740992.

You can also specify hexadecimal numbers by using the prefix $ or 0x, e.g.:

```
a = $FF     ; a = 255
```

Floating point numbers can also be specified by using the exponential notation, e.g.

```
a = 2.5e5   ; a = 2.5 * 10^5  => a = 250000
```

The 0 is optional for floating point values between -1 and 1. So the following code would also work:

```
a = .25 * 2 ; a = 0.5
```

Although Hollywood does not have separate data types for integer and floating point numbers, there is still the style-guide suggestion to suffix variables that are expected to hold floating point values with an exclamation mark. E.g.

```
a! = 3.14159265
```

This makes it easier to read your code because you know exactly which variables will get integer values only and which variables will get floating point values. Of course, you can use floating point values without the exclamation mark, but it is suggested that you use it.

8.3 Strings

The string type can be used to store a sequence of characters or binary data. By default, text is stored in the UTF-8 character encoding in strings which means that up to 4 bytes may be necessary to store one Unicode character. Strings are specified by enclosing them in double quotes. As a matter of style, you should always suffix string variables with the `$` dollar sign so that a reader of your source code can easily see which variables carry strings and which carry numbers. For example:

```
a$ = "Hello World!"
```

This could also be written as:

```
a = "Hello World!"
```

But with the dollar sign at the end the code is more readable because we know that `a` is a string.

You can concatenate strings by using the `..` operator. The code above could also be written as:

```
a$ = "Hello" .. " " .. "World!"
```

This will concatenate three strings into one string and write it to `a$`. See Section 9.6 [String concatenation], page 109, for details.

If your string needs to contain a double quote, you can use escape code `\"` for that, e.g.:

```
; this will print Hello, "Mr. John Doe"!
DebugPrint("Hello, \"Mr. John Doe\"!")
```

Escape codes are always specified after one backslash character (`\`). If you need to put a backslash into a string, use a backslash character as the escape code (`\\`). The following escape sequences are supported by Hollywood:

```
\a    Ring the system bell
\b    Back space
\f    Form feed
\n    Newline character
\r    Carriage return
\t    Horizontal tab
\v    Vertical tab
\\    Backslash
\"    Double quote
\'    Single quote
\?    Question mark
\[    Square bracket open
\]    Square bracket close
\xxx  Code point
```

The last escape sequence allows you to insert characters directly by simply specifying their code point value after the backslash. The code point value must be specified in decimal notation only and may occupy up to three digits. Only Latin 1 code points in the range of 0 to 255 are allowed here. Every value greater than 255 will not be accepted. Using this escape sequence, you could insert a zero character in a string:

```
a$ = "Hello\0World"
```

In many programming languages a zero character defines the end of the string. Not so in Hollywood. Hollywood allows you to use as many zero characters as you want in your strings. All functions of the string library are zero character safe. For example, this code would return 11:

```
DebugPrint(StrLen("Hello\0World"))
```

However, that does not apply to functions that output text. The following example will print "Hello" because of the zero character:

```
; this will print "Hello" because a zero char terminates the string
DebugPrint("Hello\0World")
```

If a newline character follows a backslash, Hollywood will insert a newline character into the string also and will continue parsing the string on the next line. For example, the following two statements create the same string:

```
a$ = "Hello\nWorld!"
a$ = "Hello\
World!"
```

If you are using this feature, make sure the newline character is right behind the backslash. There must be no spaces/tabs between the backslash and the newline!

Another way to specify strings is to use a pair of double square brackets. This is especially useful if you have multiple lines of text that should be placed inside the string. An example:

```
a$ = [[
<HTML>
<HEAD>
<TITLE>My HTML Page</TITLE>
</HEAD>
<BODY>
<A HREF="http://www.airsoftsoftwair.de/" TARGET="_NEW">
http://www.airsoftsoftwair.de/</A>
</BODY>
</HTML>
]]
```

The above string initialization is equal to this code:

```
a$ = "<HTML>\n<HEAD>\n<TITLE>My HTML Page</TITLE>\n</HEAD>\n" ..
     "<BODY>\n<A HREF=\"http://www.airsoftsoftwair.de/\"" ..
     " TARGET=\"_NEW\">http://www.airsoftsoftwair.de/</A>\n" ..
     "</BODY>\n</HTML>\n"
```

You see that the first version is much more readable. So if you want to use multiple line strings, it is advised to use the `[[...]]` version. If a newline character follows after the initial `[[` then this newline is ignored. Carriage return characters ('\r') are never included inside the long string. Every line break inside the long string will be converted to just a linefeed character ('\n'). You can also freely use double quotes in a string delimited by `[[...]]`. That is another advantage.

You can also store raw binary data in strings. For example, the `DownloadFile()` function can be used to download a file directly into a string. When using binary data inside strings, you have to be careful when calling functions of the string library. Functions of the

string library normally expect valid UTF-8 data within the strings that are passed to them. Obviously, this won't be the case when you use strings as containers for raw binary data. To make strings containing raw binary data work with the functions of the string library as well, you need to explicitly tell those functions not to interpret the string data as UTF-8. This is done by passing the special character encoding constant `#ENCODING_RAW` in the optional encoding parameter most of the string library functions accept. Then the string library functions can also be used with strings containing raw binary data. See Section 13.2 [Character encodings], page 149, for details.

Finally, there is no string length limit. Strings can be as large as system memory permits but when storing large amounts of data inside a string you should take some care and set the string to `Nil` when you no longer need it so that the garbage collector knows that it can free the memory allocated for this string. Consider the following example:

```
data$ = DownloadFile("http://www.airsoftsoftwair.de/images/" ..
                     "products/hollywood/47_shot1.jpg")
...do something with data$...
data$ = Nil
```

This will download the file at the specified URL and store the binary data in `data$`. Once the binary data in `data$` has been processed, `data$` is set to `Nil` to tell the garbage collector that it can release the memory occupied by `data$`. This is very important because otherwise it could happen that your script constantly consumes more memory.

8.4 Tables

A table is a collection of many different data items which can be of any type. A table is the universal data structure in Hollywood. It can be used in many different forms, e.g. as an array, as a list or as a record. Tables are created by using the constructor {...}. For example, the following code creates an empty table:

```
a = {}
```

An empty table is of no use because there is no data in it and Hollywood requires that all fields of a table must be initialized before they are used. Thus, if you tried to access a field of this empty table now by stating for instance

```
b = a[0]
```

you would get an error stating that this field 0 has not been initialized yet. You may only access fields of the table, that you have initialized before. The correct version would therefore be:

```
a[0] = 5      ; assign 5 to a[0]
b = a[0]      ; assign 5 to b
```

Or you could also use the constructor to initialize the table:

```
a = {5}       ; create a table with 5 as the element 0
b = a[0]      ; assign 5 to b
```

You can also use the constructor to initialize the table with multiple items. The constructor assigns the specified values to the table starting at index 0. For example:

```
a = {1, 2, 4, 8, 16, 32, 64, 128, 256} ; create table with 9 elements
b = a[7]      ; assign 128 to b (128 is at index 7 in a)
```

Additionally, you can use the Dim and DimStr statements to create and initialize a table of a specified size.

One thing that is important to know when dealing with tables is that when you assign a table to a new variable, the new variable receives only a reference to the table. It does not receive an independent copy of the table. Consider the following code:

```
a = {1, 2, 3, 4, 5}       ; create a table with 5 elements
b = a                     ; create a REFERENCE of a in b
b[0] = 2                  ; change element 0 to 2
DebugPrint(a[0], b[0])    ; will print "2 2"
```

If you want to create an independent copy of a table, you can use the `CopyTable()` function for this.

In Hollywood, indices cannot only be positive integers, but also negative integers, floating point values and even strings. For example, you can also initialize negative elements of the table:

```
a       = {}  ; create empty table
a[-5]   = 3   ; assign 3 to index -5
a[1.5]  = 2   ; assign 2 to index 1.5
```

If you want to do this custom initialization in the constructor, you will have to use square brackets. The three lines above could also be written as:

```
a = {[-5] = 3, [1.5] = 2}   ; initialize new table
```

If you want to use strings as indices, you can use the following statements:

```
a         = {}          ; create empty table
a["name"] = "John Doe"  ; assign "John Doe" to index "name"
a["age"]  = 20          ; assign "20" to index "age"
a["sex"]  = "male"      ; assign "male" to index "sex"
```

An easier way to use strings as indices is to use the '.' expression. The following code does the same as the code above:

```
a         = {}          ; create empty table
a.name    = "John Doe"  ; assign "John Doe" to index "name"
a.age     = 20          ; assign "20" to index "age"
a.sex     = "male"      ; assign "male" to index "sex"
```

Last but not least, you can also use the constructor to initialize a table with named indices. The following code does the same as the two snippets above:

```
a = {["name"] = "John Doe", ["age"] = 20, ["sex"] = "male"}
```

Or the easier way:

```
a = {name = "John Doe", age = 20, sex = "male"}
```

You can access named elements of a table also in two ways:

```
b = a["name"]
b = a.name
```

Both lines will assign the same value to `b`. The most common way to access and initialize named elements of a table is to use the dot method. Please note that Hollywood does not distinguish between upper and lower case names, so you could also access the elements above by using `a.NAME` or even `a.nAmE`.

There is, however, an exception: When using brackets to initialize or access table fields, Hollywood distinguishes between upper and lower case string indices. Further details on this topic can be found in the documentation of the `RawGet()` command. See Section 53.17 [RawGet], page 1110, for details.

You can add elements to a table by simply assigning a value to them. If you want to remove elements, you have to set their value to `Nil`. That is another big advantage of Hollywood's programming language which is dynamically typed. Tables (arrays) are not limited to a specific size: You can grow and shrink them as you like.

You can also use tables which combine named and numbered elements, for instance:

```
a = {x = 1, y = 2, 10, 11, 12, 13, z = 3, [6] = 16, 14, 15, obj="Cube"}
```

This creates a new table and initializes elements 0 to 6 with the numbers 10 to 16. Additionally, it creates four elements named `x`, `y`, `z` and `obj` and initializes them to 1, 2, 3, and "Cube".

We are now going to have a look at some more complicated table constructions. You might want to skip the following section if you are just starting out with Hollywood.

It is also possible to use tables within tables. Have a look at the following example:

```
buts = { {x1 = 0, y1 = 0, x2 = 100, y2 = 50},
    {x1 = 100, y1 = 0, x2 = 80, y2 = 50},
    {x1 = 180, y1 = 0, x2 = 100, y2 = 50} }
```

This code creates a new table called `buts` and initializes the first three elements with tables which contain the start and end position of each button. We could now use the following code to create those three buttons:

```
For k = 0 To 2
    CreateButton(k + 1, buts[k].x1, buts[k].y1, buts[k].x2, buts[k].y2)
Next
```

Multi-dimensional tables are also no problem. The following code creates a matrix of size 50x100 and initializes it to zero:

```
N = 50
M = 100
mtx = {}                            ; create an empty table
For i = 0 To N - 1
    mtx[i] = {}                     ; create a new row
    For j = 0 To M - 1
            mtx[i][j] = 0           ; initialize element
    Next
Next
```

You can also use the Dim and DimStr statements to create multi-dimensional tables.

You do not have to use constants when initializing a table using a constructor. You can use variables whereever you want. For example:

```
s$ = "test"
i = 5
a = {[s$] = "An element", [i * 5 + 1] = "Another element"}
```

This code will create the element a.test (which is the same as a["test"]) and assign the string "An element" to it. In addition, it creates the element a[26] and assigns the string "Another element" to it.

Do not get confused when you see something like this:

```
x = 5
y = 4
a = {x = x, y = y}     ; assign 5 to "x" and 4 to "y"
```

The table declaration above is no nonsense. It creates a table with two elements named x and y. The element x gets the value of the variable x which is 5 and the element y gets the value of the variable y which is 4. An other way to write the code above would be for instance:

```
x = 5
y = 4
a = {}         ; empty table
a.x = x        ; assign 5 to a.x
a.y = y        ; assign 4 to a.y
```

Both snippets do the very same.

Finally, you can place functions in your tables. Here is an example:

```
a = {Add = Function(v1, v2) Return(v1 + v2) EndFunction,
     ShowBrush = DisplayBrush}
a.ShowBrush(1, #CENTER, #CENTER)  ; calls DisplayBrush()
b = a.Add(15, 16)                 ; returns 31 to b
```

The code above creates a table with two functions. The first function is a custom function which adds two values and the second function simply refers to the Hollywood function DisplayBrush(). You could also write this code in the following way:

```
a = {["Add"] = Function(v1, v2) Return(v1 + v2) EndFunction,
     ["ShowBrush"] = DisplayBrush}
a["ShowBrush"](1, #CENTER, #CENTER)  ; calls DisplayBrush()
b = a["Add"](15, 16)                 ; returns 31 to b
```

8.5 Functions

Yes, that is right: Functions are part of our data types chapter too. In Hollywood every function is just a variable. That means that you can initialize them just like variables, you can pass functions as parameters to other functions and functions can also be the return values of other functions. For example, the following code

```
p_Print = Function(s) DebugPrint(s) EndFunction
```

is just another way for writing:

```
Function p_Print(s)
    DebugPrint(s)
EndFunction
```

Because functions are variables you can also assign new values to them, for instance:

```
DebugPrint = Print
```

Now all calls to DebugPrint() will call the Print() command instead.

There is a lot more to know about the function data type. Therefore it has its own chapter in this manual. See Section 12.1 [Functions], page 137, for details..

8.6 Nil

If you use a variable without assigning a value to it, the variable will have the type `Nil` which practically means that the variable does not exist. Hollywood only keeps variables which have a value. If you pass an uninitialized variable to a function or use it with an operator, it will be automatically converted to zero or - if the function expects a string - to an empty string (`""`).

If you do not need a variable any longer, you can also set it to `Nil` and it will be deleted in the next cycle of the garbage collector then.

You can also delete an element of a table by setting it to `Nil`.

Be careful when checking variables against `Nil` because 0=`Nil` is actually `True` in Hollywood. Thus, `IsNil()` and `GetType()` are the only reliable way to find out if a variable is really `Nil`. Simply checking against `Nil` will also result in `True` if the variable is 0.

9 Expressions and operators

9.1 Overview

An expression is a combination of operands and operators. If there is at least one operand and one operator we speak of an expression. Hollywood needs to evaluate expressions before it can pass their result to a function. An expression can be constant or variable, depending on whether it contains variables or not. For example, 5 + 3 is an expression. The operands are 5 and 3 and the operator is +. -1 is also an expression because we have one operand and one operator. Usually operators are binary which means that they require two operands but there are exceptions: For example, the negation operator (-) is unary and therefore requires only one operand.

You can use parentheses in expressions to tell Hollywood what shall be evaluated first. In the following line

```
a = (3 + 4) * 5
```

Hollywood will first add 3 and 4 and then multiply the result of the addition by 5. If you did not include the parentheses in the code above, Hollywood would first evaluate 4 * 5 and then add 3 to it because the multiplication operator (*) has a higher priority than the addition operator (+). See Section 9.7 [Operator priorities], page 110, for details.

9.2 Arithmetic operators

Hollywood supports the following arithmetic operators:

BINARY		
Operator	**Description**	**Example**
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Real division	a / b
\	Integer division	a \ b
%	Division remainder	a % b
^	Power	a ^ b

UNARY		
Operator	**Description**	**Example**
-	Negation	-a

It should be pretty self-explaining how to use these operators so here is only a brief description of every operator:

Addition: a + b
Adds a and b, e.g. 5 + 3 = 8

Subtraction: a - b
: Subtracts b from a, e.g. 10 - 5 = 5

Multiplication: a * b
: Multiplies a by b, e.g. 10 * 8 = 80

Real division: a / b
: Does an exact division. Result might be a floating point value, e.g. 5 / 2 = 2.5. b must not be 0

Integer division: a \ b
: Divides a by b. Decimal places will be deleted, e.g. 5 \ 2 = 2 b must not be 0

Division remainder: a % b
: Returns the integer remainder of the division a \ b, e.g. 5 % 2 = 1 b must not be 0

Power: a ^ b
: Calculates a to the power of b, e.g. 2 ^ 8 = 256

Negation: -a
: Negates a, e.g. –5 = 5

9.3 Relational operators

Hollywood supports the following relational operators:

BINARY		
Operator	**Description**	**Example**
=	Equal	a = b
<>	Not equal	a <> b
<	Less than	a < b
>	Greater than	a > b
<=	Less or equal to	a <= b
>=	Greater or equal to	a >= b

Each of the operators compares the operands a and b and returns `True` if the condition matches and `False` otherwise. Please note that you can only compare values of the same type. The automatic number to string conversion does not apply here!

The equality operators can be used with all types, i.e. you can also compare functions and tables with them. The order operators (< > <= >=) only work with numbers and strings. If you compare strings with them Hollywood will do an alphabetical comparison. For example:

```
r = ("Hello" < "World")     -> True because H is before W alphabetically
r = ("Commodore" < "Amiga") -> False because C is after A alphabetically
```

Note that for compatibility reasons, comparing strings with the relational operators is only supported for ASCII characters. To compare strings with full Unicode collation, use the `CompareStr()` function instead. See Section 51.15 [CompareStr], page 1032, for details.

9.4 Logical operators

Hollywood supports the following logical operators:

BINARY		
Operator	**Description**	**Example**
And	Logical And	a And b
Or	Logical Or	a Or b

UNARY		
Operator	**Description**	**Example**
Not	Logical Not	Not a

The binary logical operators allow you to make decisions based on multiple conditions. Each binary logical operator needs two operands which are used to evaluate the result of the logical condition. All values that are not 0, `Nil` or the empty string (`""`) are considered `True`.

The `And` and `Or` operators use short-cut evaluation. This means that if the first operand already defines the result, the second operand is not evaluated at all. For example, if the first operand of an `And` expression is `False` (zero), then the second operand does not need to be evaluated because the whole expression cannot be `True` anyway. The same applies to an `Or` expression if the first operand of it is `True` (non-zero). Then the whole expression will always be `True` - no matter what value the second operand has.

Please note: `And` and `Or` do not return the constant `True` (1) if they are true. `And` returns the second operand if it is true and `Or` returns the first operand if it is true. For example:

```
a = 5 And 4     ; a = 4
a = 5 And 0     ; a = 0
a = 0 And 4     ; a = 0
b = 5 Or 4      ; b = 5
b = 5 Or 0      ; b = 5
b = 0 Or 4      ; b = 4
```

The unary `Not` operator will negate its operand. The result will always be `True` (1) or `False` (0). If used on a string it will result in `True` if the string is empty (`""`). For example:

```
a = Not True     ; a = 0 (False)
a = Not False    ; a = 1 (True)
a = Not 5        ; a = 0 (False)
a = Not Not True ; a = 1 (True)
a = Not "Hello"  ; a = 0 (False)
a = Not ""       ; a = 1 (True)
```

Please note: The `Not` operator has a high priority. You will need parentheses in most cases. For example, this does not work:

```
If Not a = -1        ; wrong!
```

Hollywood will translate it to

```
If (Not a) = -1
```

because the `Not` operator has a higher priority than the equality operator! But obviously this translation does not make any sense because the result of the expression in parentheses (Not a) will always be 0 or 1 but never ever -1. Therefore, if you would like to check if a is not -1, you will have to use parentheses around the expression with the lower priority:

```
If Not (a = -1)      ; correct!
```

See Section 9.7 [Operator priorities], page 110, for details.

9.5 Bitwise operators

Hollywood supports the following bitwise operators:

BINARY		
Operator	**Description**	**Example**
<<	Left shift (logical)	a << b
>>	Right shift (logical)	a >> b
&	Bitwise And	a & b
~	Bitwise Xor	a ~ b
|	Bitwise Or	a | b

UNARY		
Operator	**Description**	**Example**
~	Bitwise negation	~a

The bitwise operators allow you to work with expressions on bit level. Those operations are all limited to 32-bit values. Here is a description of the bitwise operators:

The left shift operator (`<<`) shifts all bits of the operand `a` left `b` times. The bit holes on the right side of the number created by this operation will be padded with zeros (logical shift). `b` must not be negative. Shifting `a` `x` times left is equal to multiplying this number by 2^x. But of course, shifting is much faster than multiplying. Examples:

```
7 << 1 = %111 << 1 = %1110 = 14 (7 * 2^1 = 14)
256 << 4 = %100000000 << 4 = %1000000000000 = 4096 (256*2^4=4096)
```

The right shift operator (`>>`) shifts all bits of the operand `a` right `b` times. The bit holes on the left side of the number will be padded with zeros (logical shift). If you need an arithmetic shift (bit holes will be padded with the most significant bit), please use the `Sar()` function instead. `b` must not be negative. Shifting `a` right `x` times is equal to dividing this number by `2^x`. But of course shifting is much faster than dividing if an integer result is precise enough for your purpose. Here are some examples:

```
65 >> 1 = %1000001 >> 1 = %100000 = 32 (65\2^1=32)
256 >> 4 = %100000000 >> 4 = %10000 = 16 (256\2^4=16)
```

The bitwise `And`, `Xor` and `Or` operators are basically the same as the logical `And` / `Xor` / `Or` operator with the difference that &, ~ and | work on bit level, i.e. they compare all 32 bits of operands a and b and set the bits in the result according to this comparison. The `And` operator will set the bit in the return value if both bits in operands a and b are set on this position. The `Xor` operator will set the bit in the return value if one of the two bits are 1

but not if both are 1. The `Or` operator will set the bit in the return value if one or both of the operands have the bit set on that position. A table:

Bit 1	Bit 2	Bit1 & Bit2	Bit1 ~ Bit2	Bit1 | Bit2
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Examples:

```
%10011001 & %11101000 = %10001000   ; Bitwise And
%10011001 ~ %11101000 = %01110001   ; Bitwise Xor
%10011001 | %11101000 = %11111001   ; Bitwise Or
```

The unary negation operator (~) will do a bitwise inversion of the number it is used on. All bits will be inverted. Please note that the value will always be converted to a 32-bit integer before the inversion. Thus, you might get a lot of leading ones. For example:

```
~%00000000 = %11111111111111111111111111111111
~%10111001 = %11111111111111111111111101000110
```

To get rid of these 1s, simply use the bitwise `And` operator (&) on the resulting value. For instance, if you only want to have 8 bits inverted like in the example above, use the bitwise `And` with 255:

```
~%00000000 & %11111111 = %11111111
~%10111001 & %11111111 = %01000110
```

9.6 String concatenation

BINARY		
Operator	**Description**	**Example**
..	Concatenation	a .. b

The string concatenation operator can be used to concatenate two strings to a new one. Because Hollywood offers automatic number to string conversion you can even concatenate two numbers. The result will always be a string though. Examples:

```
DebugPrint("Hello" .. " World")    ; prints "Hello World"
DebugPrint(5 .. " + " .. 5 .. " = " .. 10)   ; prints "5 + 5 = 10"
```

This operator is also useful if you want to spread a string over multiple lines. For example:

```
DebugPrint("My Program v1.0\n" ..
           "(c) by me 2005\n" ..
           "Press RETURN to start!")
```

9.7 Operator priorities

Here is a complete list of all available operators and their priorities. You do not have to know this by heart. When in doubt use parentheses. It does not hurt and makes your program more readable because not everyone knows that the left shift operator has a higher priority than the bitwise or operator.

Priority	Operator	Description
12	^	Power
11	-	Negation
11	~	Bitwise Negation
11	Not	Logical Not
10	*	Multiplication
10	/	Real division
10	\	Integer division
10	%	Division remainder
9	+	Addition
9	-	Subtraction
8	<<	Left shift
8	>>	Right shift
7	&	Bitwise And
6	~	Bitwise Xor
5	|	Bitwise Or
4	..	Concatenate
3	=	Equality
3	<>	Inequality
3	<	Less than
3	>	Greater than
3	<=	Less/equal to
3	>=	Greater/equal to
2	And	Logical And
1	Or	Logical Or

9.8 Metamethods

Metamethods can be used to define how Hollywood's operators shall behave when used with tables. Normally, you cannot use any of Hollywood's operators with tables as operands. For example, the following is not possible:

```
table_A = {1, 2, 3, 4, 5}
table_B = {5, 4, 3, 2, 1}
result = table_A + table_B   ; generates compiler error!
```

The code above tries to add `table_A` to `table_B` but this does not work because tables may contain any random data (functions, subtables, strings, etc.) so there is no generic way of saying how the add operator should behave on a table. This is where metamethods come into play. Metamethods allow you to define how an operator shall behave when it receives a table operand. In other words, metamethods allow you to define a function that gets executed whenever an operator is used with a table operand. This function then computes the result and it is called a metamethod.

Metamethods are not a global setting but they are private to every table. When you create a table it will not have any metamethods attached. Thus, trying to use an operator on this table will fail because it does not have any metamethods. To assign metamethods to a table you need to use the `SetMetaTable()` command. A metatable is a table containing a set of metamethods. `SetMetaTable()` accepts two table argument: The first argument is the table whose metamethods you would like to set, and the second argument is the actual metatable, i.e. the table that contains the metamethods that you would like to set.

Let's have a look at an example now. We will rewrite the code from above using metamethods so that we can add the two tables.

```
mt = {}  ; create our metatable

Function mt.__add(a, b)
    Local sizeA = ListItems(a)   ; number of elements in table A
    Local sizeB = ListItems(b)   ; number of elements in table B
    Local result = {}            ; create resulting table

    For Local k = 0 To Min(sizeA, sizeB) - 1
       result[k] = a[k] + b[k]                ; add elements
    Next

    Return(result)   ; return resulting table
EndFunction

table_A = {1, 2, 3, 4, 5}
table_B = {5, 4, 3, 2, 1}

SetMetaTable(table_A, mt)   ; set "mt" as table_A's metatable
result = table_A + table_B
```

The resulting table will have five elements that are all set to 6. Now what did we do in the code above? We first create an empty table that serves as our metatable. Then we add a function called `__add` (using two underscores) to that table. This function will be the metamethod for the `+` operator. Note that we must use the name `__add` for this function because Hollywood uses the function name to detect the operator that is served by the metamethod. Using `__add` as name defines a metamethod for the add (`+`) operator. The code in our metamethod simply calculates the length of the two tables, adds the table elements, and stores them in a resulting table that it returns.

Note that the implementation of our `__add` metamethod above requires that both arguments are tables. And the tables must only contain numbers (or strings that can be converted

to numbers). E.g. the following expressions would not work using the above metamethod implementation:

```
result = table_A + 10       ; --> error because "10" is not a table
result = table_A + "Hello"  ; --> same error
```

Of course, it is possible to write metamethods which can handle these situations. You would just have to check the types of the parameters that are passed to your metamethod and then you can take custom actions depending on the variable types specified.

Now we have covered the metamethod for the add (+) operator only. Of course, you can set a metamethod for every other Hollywood operator, too. You can also create metamethods for all relational operators (= <> < > <= >=) so that you can compare tables directly. All you need to know is the correct name for the metamethod of the operator so that you can install it. Here is a list of all available metamethods and to their corresponding operators:

Metamethod	Operator	Description
__pow	^	Power
__unm	-	Negation
__not	~	Bitwise Negation
__mul	*	Multiplication
__div	/	Real division
__divint	\	Integer division
__mod	%	Division remainder
__add	+	Addition
__sub	-	Subtraction
__lsh	<<	Left shift
__rsh	>>	Right shift
__and	&	Bitwise And
__xor	~	Bitwise Xor
__or	|	Bitwise Or
__concat	..	Concatenate
__eq	=	Equality
__lt	<	Less than
__le	<=	Less/equal to
__index	[]	Read value from table
__newindex	[]=	Write value to table
__call	()	Call a table

As you can see, there are no metamethods for the >, >=, and <> operators. This is because Hollywood handles them by simply reformulating the condition in the following way:

```
a <> b          is the same as      Not (a = b)
a > b           is the same as      b < a
a >= b          is the same as      b <= a
```

If you would like to compare two tables that both have associated metatables, but you would like to compare them without invoking the `__eq` metamethod, you have to use the `RawEqual()` function. This function will compare both tables just by reference without invoking any metamethod.

9.8.1 Differing metatables with binary operators

As you have seen above every table has its own private metatable setting. When using binary operators, however, it could happen that the two operands do not use the same metatable but different ones. So how does Hollywood choose the metatable for the operator now? This depends on several conditions:

a. If the operator is a relational operator (= <> < <= > >=), the metamethod will only be called if the two tables that shall be compared use the same metatable. If they have different metatables, the comparison will fail.

b. If the operator is an arithmetic operator, a bitwise operator, or the string concatenation operator (..), Hollywood will first look in operand A. If operand A has a metatable then this metatable will be used. If operand A does not have a metatable Hollywood will look in operand B. If operand B has a metatable it will be used. If neither operand has a metatable an error message will be raised.

9.8.2 Limitations of the relational metamethods

You have already read above that the relational metamethods will only be called if the two operands use the same metatables. However, there is another limitation when using relational metamethods: They will only be called if the two operands are tables. It is not possible to compare a table with a number, or comparing a string with a table, etc. The arithmetic and bitwise metamethods can be made to work with any variable type but the relational metamethods are limited to comparisons of tables.

9.8.3 Advanced metamethods

So far we have only covered the relational, arithmetic, bitwise and concatenation metamethods. There are, however, a few more metamethods that you can use, namely `__index`, `__newindex`, `__call` and `__tostring`. Here is a detailed description of these metamethods:

`__index`: This metamethod is called whenever you try to read from a table index that does not exist. This metamethod could be used to create a default value for all uninitialized table fields. Normally, Hollywood will fail when you read from uninitialized fields. This behaviour could be changed using this metamethod. Here is a code snippet that sets the default value to 0:

```
mt = {}
Function mt.__index(t, idx)
   Return(0)
EndFunction

t = {x = 10, y = 20}
SetMetaTable(t, mt)
NPrint(t.x, t.y, t.z)  ; --> prints 10 20 0
```

Without our metatable, the call to `NPrint()` would fail because `z` has not been initialized. By using the metatable, however, `z` will automatically fall back to 0 because it does not exist.

Sometimes it might become necessary to read from a table without invoking any metamethod. You can do this using the `RawGet()` function. `RawGet()` will never invoke any metamethod. If an index does not exist it will return `Nil` to you.

`__newindex:`
This metamethod is called whenever you try to write a value to a table index that does not yet exist in the table. You could use this metamethod for example to create tables that are read-only. The following code will issue an error whenever you try to write to a protected table:

```
mt = {}
Function mt.__newindex(t, idx, val)
   NPrint("Blocked writing", val, "at index", idx)
EndFunction

t = {x = 10, y = 20}
SetMetaTable(t, mt)
t.z = 45    ; --> "Blocked writing 45 at index z"
```

The code above sets table `t` as write-protected. You will not be able to make any modifications to the table.

Sometimes it might become necessary to write to a table without invoking any metamethod. You can do this using the `RawSet()` function. `RawSet()` will never invoke any metamethod. You could even write to write-protected tables using the `RawSet()` function.

`__call:` This metamethod is called whenever you try to call a table. Normally, trying to call a table will fail because tables are obviously just types of data storage and not a function. However, there are cases where it can come handy if you could also call a table. The following example demonstrates a metamethod that will calculate the average of all table values:

```
mt = {}
Function mt.__call(t)
   Local c = ListItems(t)
   Local sum = 0

   For Local k = 0 To c - 1 Do sum = sum + t[k]

   Return(sum / c)
EndFunction

t = {10, 23, 45, 5, 107, 45, 18, 46}
SetMetaTable(t, mt)
NPrint(t())   ; --> 37.375
```

The code above will return 37.375 which is the average of the eight values stored in table `t`.

`__tostring`:

This metamethod is used by commands like `Print()` or `DebugPrint()`. Normally, when you pass a table to `Print()` you will receive something as "Table: 1acd432f" as the output. This is the handle Hollywood uses internally to refer to the table and is obviously of not much use for you. Using the `__tostring` metamethod, however, you can easily change this behaviour. Here is a metamethod which creates a string representation of a table:

```
mt = {}
Function mt.__tostring(t)
   Local r$
   For Local k=0 To ListItems(t)-1 Do r$=r$..t[k].." "
   Return(r$)
EndFunction

t = {"Jeff", "Andy", "Mike", "Dave"}
SetMetaTable(t, mt)
NPrint(t)   ; --> Jeff Andy Mike Dave
```

The code above prints "Jeff Andy Mike Dave" because our `__tostring` metamethod has simply concatenated all elements of the table.

10 Variables and constants

10.1 Variables and constants

A variable can be used to store a piece of data under a given name. In Hollywood variables do not have to be declared. This means that you can simply assign a value to a variable without having to define the type for the variable first. Hollywood will do this automatically. Your variable name must start with a character from the English alphabet (a-z) or an underscore (_). After the first character, you may also use numbers, the exclamation mark (!) and the dollar sign ($). It is suggested that you use the dollar sign only in variables of the type string. As Hollywood is a case insensitive language, all variable names are case insensitive too. This means that, for example, the names "MYVAR" and "myvar" refer to the same variable. Hollywood is a dynamically typed language, which means that variables are also dynamic. For instance, the following does work without problems:

```
myvar = 1.5
myvar = "Hallo"
myvar = {1, 2, 3}
myvar = Function(s) DebugPrint(s) EndFunction
```

You can change the type of a variable on-the-fly. But this is not good programming practice!

Constants are fixed values that are globally available from everywhere in your script. They are prefixed with a hash character (#) so that you can distinguish them from variables and functions.

10.2 Global variables

If you assign a value to a variable for the first time, then this variable will automatically become global if you did not explicitly tell Hollywood that it shall be local by using the `Local` statement. Global variables can be accessed from anywhere in your script. They are globally available to all functions. However, if there is a local variable that has the same name as a global variable, then Hollywood will always use this local variable first.

Global variables are slower than local variables and they cannot be easily collected by the garbage collector unless you explicitly set them to `Nil` when you do not need them any longer. Thus, you should only use globals when really necessary. In functions you should try to work with local variables only.

Here is an example:

```
; bad code!
Function p_Add(a, b)
    tmp = a + b
    Return(tmp)
EndFunction
```

The variable `tmp` will be created as a global variable. This does not make much sense here because you only need the variable `tmp` in this function. So you should better make it local to this function, e.g.

```
; good code!
Function p_Add(a, b)
```

```
    Local tmp = a + b
    Return(tmp)
EndFunction
```

To improve the readability of your program, you can use the `Global` statement to clearly mark certain variables as globals. This is of course optional, because all variables that are not explicitly declared as local will become global automatically. But using the `Global` statement makes your program more readable.

See Section 10.4 [Local variables], page 118, for details.

10.3 Global statement

```
Global <var1> [, <var2>, ...] [= <expr1> [, <expr2>, ...]]
```

The `Global` statement is used to tell Hollywood that the specified variable should be global. Additionally, it can also initialize your variable. This statement is only included to improve the readability of your program. You could also leave it out and the code would work the same way. The `Global` statement works exactly like the `Local` statement.

If you use the `Global` statement, it is advised that you place all statements at the beginning of your code. So everyone can clearly see which variables are globally available. Using `Global` elsewhere in your code is generally not suggested and can be quite confusing to read.

See Section 10.2 [Global variables], page 117, for details.

10.4 Local variables

You should use local variables whenever and wherever possible. They improve the memory management of your program because the garbage collector knows automatically when it can delete them. Additionally, access to local variables is much faster than to globals because Hollywood does not need to traverse through the whole global enviroment and finally, they increase the readability of your program.

Local variables have a limited lifetime. They will only be available in the block where you have declared them. A block will usually be the body of a function but it can also be the body a control structure. You can even declare blocks by using the `Block` and `EndBlock` statements.

Important: Local variables have to be declared explicitly. If you do not do this, the variable will be automatically global. For example, if you write

```
Function p_Add(a, b)
    tmp = a + b
    Return(tmp)
EndFunction
```

the variable `tmp` will automatically be created as a global variable. But this is just a waste of resources because you only need the variable inside the function so you should better write:

```
Function p_Add(a, b)
    Local tmp = a + b
    Return(tmp)
```

```
EndFunction
```

Now the variable `tmp` is explicitly declared local and will be deleted when the function `p_Add()` exits.

As you have already seen now, local variables are declared by using the `Local` statement. To declare a local variable, simply place the identifier `Local` before the declaration:

```
a = 10        ; global variable
Local b = 10  ; local variable
```

If you want to initialize multiple variables with one `Local` statement, simply uses commas as you would do in a normal assignment:

```
a, b = 10, 5        ; global variables! a receives 10, b receives 5
Local x, y = 10, 5  ; local variables! x receives 10, y receives 5
```

Once you have declared a local variable, you do not need to use the Local statement any longer:

```
; if x > 10, multiply it by 2, else divide it by 2
If x > 10
    Local tmp = x     ; declare local variable "tmp"
    tmp = tmp * 2     ; multiply local variable "tmp" by 2
    x = tmp           ; assign value of local "tmp" to "x"
Else
    Local tmp = x     ; declare local variable "tmp"
    tmp = tmp / 2     ; divide local variable "tmp" by 2
    x = tmp           ; assign value of local "tmp" to "x"
EndIf

Print(tmp)            ; this will print 0 because "tmp" is gone
```

The code above creates the local variable `tmp` in the two blocks of the `If` statement. After that it multiplies or divides it by 2. The identifier `Local` is no longer required there because Hollywood already knows at this point that `tmp` is a local variable. `tmp` will be deleted at the end of the block so it is not available in the line `Print(tmp)` any more. `tmp` becomes `Nil` after the block ends.

If you do not assign a value to the variable, it will get the value `Nil` but Hollywood will know that it is a local variable, e.g.:

```
If True         ; block is always entered
    Local x     ; declare local variable x
    Print(x)    ; prints 0 because x is Nil
    x = 5       ; assign 5 to local x
    Print(x)    ; prints 5 now
EndIf           ; scope of x ends here

Print(x)        ; prints 0 because x is Nil now again
```

You can also use the name of a global variable (or a local variable of the superior block) for a new local variable. For instance:

```
a = 50
Block             ; delimit the next two lines
```

```
    Local a = 40  ; create local "a" and assign 40
    NPrint(a)     ; prints 40
EndBlock          ; scope of "a" ends here

NPrint(a)         ; prints 50
```

A more complex example which uses many variables with the same name follows:

```
x = 10                      ; global x (x1 = 10)
Block                       ; open new block
    Local x = x + 1         ; assign 11 to local x (x2 = x1 + 1)
    Block                   ; open new block
        Local x = x + 1     ; assign 12 to local x (x3 = x2 + 1)
        Block               ; open new block
            Local x = x + 1 ; assign 13 to local x (x4 = x3 + 1)
            NPrint(x)       ; prints 13 (= x4)
        EndBlock            ; scope of x4 ends here
        NPrint(x)           ; prints 12 (= x3)
    EndBlock                ; scope of x3 ends here
    NPrint(x)               ; prints 11 (= x2)
EndBlock                    ; scope of x2 ends here
NPrint(x)                   ; prints 10 (= x1)
```

This code might look a bit confusing but it makes perfect sense. In every new block Hollywood will look up the variable `x` starting from the current scope and traversing through all superior blocks.

It should be noted that you cannot use local variables together with `Gosub()` because Hollywood will jump out of the current block on a `Gosub()` and return to it later with totally different data on the stack. Thus, the following code will not work:

```
; invalid code
Block
    Local a = 50      ; create local "a"
    Gosub(SUBROUTINE) ; jump out of the block; "a" will be trashed
    Print(a)          ; local "a" is some random stack value now
EndBlock
```

This shouldn't be much of a problem because `Gosub()` is deprecated anyway and shouldn't be used in your code.

You can also use local functions. They work in almost the same way and are also preferable to global functions although you will most likely do not use them as excessively as you use local variables. But they can be handy from time to time. See Section 12.8 [Local functions], page 145, for details.

10.5 Local statement

```
Local <var1> [, <var2>, ...] [= <expr1> [, <expr2>, ...]]
```

The `Local` statement is used to tell Hollywood that the specified variable should be local. Additionally, it can also initialize your variable.

```
Local myvar           ; tell Hollywood that myvar will be local
```

```
r = GetType(myvar)     ; returns #NIL
DebugPrint(myvar)      ; prints zero
<other code>
myvar = 5              ; now myvar is created as a local variable!
```

The code above simply tells Hollywood that myvar shall be local if a value is assigned to it. The statement `Local myvar` will not initialize the variable. The variable will still be of type `Nil`, i.e. it does not even exist. `myvar` is created when you set it to a specific value. Normally, if the initial value is different from 0, you will do the initialization in the Local statement, e.g.

```
Local myvar = 5        ; create local variable
```

You can create and initialize as many variables as you like. Just use a comma on each side of the equal sign for that. Example:

```
Local myvar, myvar2, myvar3 = 5, 4, 3
```

The code above creates three new local variables and assigns the value 5 to `myvar`, 4 to `myvar2` and 3 to `myvar3`.

Please note that the `Local` statement does not have to be placed at the beginning of a function/block as it is the case with variable declarations in other programming languages. You can place it wherever you want and it is no bad programming style to use `Local` in the middle of a function. For example, this code is fine:

```
Block
    DebugPrint("Now calling TestFunc()")
    Local r = TestFunc()
    DebugPrint("Now calling TestFunc2()")
    Local r2 = TestFunc2()
    DebugPrint("Results:", r, r2)
EndBlock
```

This code uses `Local` in the middle of a new block which is no problem with Hollywood.

See Section 10.4 [Local variables], page 118, for details.

10.6 Garbage Collector

Hollywood will invoke its garbage collector from time to time while your script is running. The garbage collector manages all resources allocated by your script and frees all memory that is no longer needed. For example:

```
Print("Hello World")
```

After Hollywood has called the `Print()` command the memory allocated for the string "Hello World" can be released because it is no longer needed. You can support the garbage collector by setting variables to `Nil` when you do not need them any longer. This is especially useful for long strings or extensive tables, e.g.

```
a = {}
For k = 1 To 1000
    a[k] = {e1 = x, e2 = y}
    x = x + 5
    y = y + 5
```

```
Next
```

This code creates a pretty extensive table which occupies some memory of your system. If you do not need this table any longer, simply set it to `Nil`, e.g.

```
a = Nil
```

The garbage collector will then free the memory occupied by this table.

It is also strongly suggested that you use local variables whenever and wherever it is possible because the garbage collector can automatically release them when their scope ends (e.g. at the end of a function). See Section 10.4 [Local variables], page 118, for details.

10.7 Constants

Constants, as the name implies, are values which cannot be changed after their first initialization. They can be accessed through a user-specified name but their values are fixed during the script's execution. Like all language elements, Hollywood does not distinguish between lower and upper case constants, but they should be written in capitals for style guide reasons. Constants also need to have a hash character (#) prefix to distinguish them from variables. The constants `True` and `False` are exceptions here, they do not need to have a hash character prefix because they are elementary parts of the Hollywood script language. All other constants are just additions for commands and therefore need to be prefixed.

Constants must be either numbers or strings. You can also declare your own constants by using the `Const` statement, for example:

```
Const #MYCONSTANT = 5 * 5
```

If you use this statement, you should always use it at the beginning of your script because it is a global declaration which cannot be changed during your script's execution.

10.8 Const statement

```
Const #<name> = <expr>
```

The `Const` statement allows you to declare a new constant. The name you specify in `name` must be prefixed with a hash character (#). `expr` must be a constant (!) expressions, i.e. you must not use any variables here. For example:

```
Const #MYCONSTANT  = (5 * 10) / 2      ; #MYCONSTANT  = 25
Const #MYCONSTANT2 = #MYCONSTANT * 10  ; #MYCONSTANT2 = 250
Const #MYCONSTANT3 = b * 5             ; does not work!
```

The last example will not work because a variable is used and the expression must be constant.

Alternatively, `expr` can also be a constant string expression. E.g.

```
Const #PRGVERSTRING = "$VER: MyProgram 1.0 (13.04.2005)"
```

Constants can also be declared from the command line by using the `-setconstants` console argument. This is especially useful in connection with the `@IF` preprocessor command. See Section 3.2 [Console arguments], page 33, for details.

10.9 Inbuilt constants

The inbuilt constants are used by many functions as descriptors for a special action. Therefore, their function is different from constant to constant. If a function requires a special constant as an argument (e.g. `SetFontStyle()` accepts the constants `#BOLD`, `#ITALIC`, `#NORMAL` and `#UNDERLINED`), then those constants are described in the documentation for that command.

Additionally there are some constants that can be specified everytime a Hollywood function asks you for a x or y coordinate. These constants are the so-called "position constants". They allow you to easily specify some often used positions. The following position constants are inbuilt in Hollywood:

The following constants can be used as a x-coordinate:

`#CENTER:` Specifies the center of your display (= (displaywidth-objectwidth) / 2)

`#LEFT:` Specifies the left edge of your display (= 0)

`#LEFTOUT:`
Specifies the outer left of your display (= -objectwidth)

`#RIGHT:` Specifies the right edge of your display (= displaywidth-objectwidth)

`#RIGHTOUT:`
Specifies the outer right of your display (= displaywidth+objectwidth)

`#USELAYERPOSITION:`
Specifies the current x-position of the layer.

The following constants can be used as a y-coordinate:

`#CENTER:` Specifies the center of your display (= (displayheight-objectheight)/2)

`#TOP:` Specifies the top edge of your display (= 0)

`#TOPOUT:` Specifies the outer top of your display (= -objectheight)

`#BOTTOM:` Specifies the bottom edge of your display (= displayheight-objectheight)

`#BOTTOMOUT:`
Specifies the outer bottom of your display (= displayheight+objectheight)

`#USELAYERPOSITION:`
Specifies the current x-position of the layer.

These constants make it very easy for you to position your objects. For example if you want to display brush 1 in the center of the display, just call `DisplayBrush()` with the arguments 1, `#CENTER`, `#CENTER` et voila!

You can even fine-tune the positions by subtracting and adding values to these constants! For example, `DisplayBrush(1, #CENTER, #CENTER + 25)` displays brush one 25 pixels below the vertical center of the display.

There are also some constants that allow you to easy access some basic colors. The following color constants are currently declared by default: `#BLACK`, `#MAROON`, `#GREEN`, `#OLIVE`, `#NAVY`, `#PURPLE`, `#TEAL`, `#GRAY`, `#SILVER`, `#RED`, `#LIME`, `#YELLOW`, `#BLUE`, `#FUCHSIA`, `#AQUA`, `#WHITE`.

#BLACK	$000000	
#MAROON	$800000	
#GREEN	$008000	
#OLIVE	$808000	
#NAVY	$000080	
#PURPLE	$800080	
#TEAL	$008080	
#GRAY	$808080	
#SILVER	$C0C0C0	
#RED	$FF0000	
#LIME	$00FF00	
#YELLOW	$FFFF00	
#BLUE	$0000FF	
#FUCHSIA	$FF00FF	
#AQUA	$00FFFF	
#WHITE	$FFFFFF	

Finally, Hollywood defines some platform-specific constants depending on the platform it is currently running on or compiling for. You can use the `@IF` preprocessor command to test for those constants and take desired action. You can find these platform-specific constants in the section on the `@IF` preprocessor command in this documentation. See Section 52.17 [IF], page 1080, for details.

10.10 Character constants

Character constants are usually used to get the code point value of a character in an easy way. If you embed a character in single quotes (''), Hollywood will replace this specification with the code point value of the character. Thus, character constants are always of the data type Number. Here is an example:

```
DebugPrint('A')    ; prints 65
```

You can also put escape sequences into a character constant. For example:

```
DebugPrint('\n')  ; prints 10
```

See Section 8.3 [String data type], page 98, for more information on which escape sequences Hollywood supports.

11 Program flow

11.1 Statements controlling the program flow

This chapter describes all statements offered by Hollywood, which are used to control the program flow. It is very important that you know these control structures because they can make your program much more readable. We can categorize the Hollywood control structures into two groups:

1) Conditional blocks: They are used to check if a specific expression is true (non-zero) or false (zero). This is very important because your program needs to make decisions all the time. The following kinds of conditional blocks are available:

```
If-Else-ElseIf-EndIf
Switch-Case-Default-EndSwitch
```

2) Loops: They are used to repeat certain portions of your code. Imagine you want to print the numbers from 1 to 100. You could type in the Print() command a hundred times for that, but you could also use a simple For loop that calls Print() a hundred times. The following loop structures are available:

```
While-Wend
For-Next
Repeat-Until
```

11.2 If-EndIf statement

There are two versions of the `If` statement: A long and a short version.

1) Long version If statement:

```
If <expr> <block> [ElseIf <expr> <block> ...] [Else <block>] EndIf
```

The `If` statement checks if the given expression is true (non-zero). If this is the case, the commands following the statement are executed. If the given expression is false (zero), `If` jumps to the next `ElseIf` statement (if there is any) and checks if the expression given there is true. This is repeated until the `Else` statement is reached. If none of the expressions before was true, the code following `Else` will be executed.

The statements `If` and `EndIf` are obligatory. `ElseIf` and `Else` statements are optional. You can use as many `ElseIf`'s as you like but there must be only one `Else` in your `If` statement. Furthermore, the `Else` statement must be the last condition before `EndIf`. Here is an example:

```
If a > 5                        ; check if a is greater than 5
    DebugPrint("a > 5")
ElseIf a < 5                    ; check if a is less than 5
    DebugPrint("a < 5")
Else                            ; else a must be 5
    DebugPrint("a = 5")
EndIf
```

You can also use more complex expressions as the condition:

```
If country$ = "USA" And age < 21
```

```
    DebugPrint("No alcohol permitted under 21 in the USA!")
EndIf
```

2) Short version If statement:

```
If <expr> Then <true-stat> [ElseIf <expr> <true-stat> ...] [Else <stat>]
```

The short version of the `If` statement works in the same way as the long version but has the advantage that you do not need to include an `EndIf`. The short `If` statement has the restriction, that all of its parts have to be placed on a single line. Another restriction is, that only one statement must follow the `Then` / `ElseIf` / `Else` identifiers. If you want to execute multiple statements you have to use the long version.

Using the short `If` statement we could write the example from above in the following way:

```
If a>5 Then Print("a>5") ElseIf a<5 Print("a<5") Else Print("a=5")
```

You can see that the result is not very readable, so in the case of the example from above, it is not recommended to use the short version. The short `If` statement does better fit if you just have one condition, e.g.

```
If a = True Then b = 5
```

This is better readable than

```
If a = True
    b = 5
EndIf
```

Another version of the `If` statement is the so called immediate-if statement `IIf()`. This version is implemented as a command in Hollywood and it is part of the system library. See Section 52.18 [IIf], page 1083, for details.

11.3 While-Wend statement

There are two versions of the `While` statement: A long and a short version.

1) Long version While statement:

```
While <expr> <loop-block> Wend
```

The `While` statement enters the loop if the given expression is true (non-zero). If the expression is false (zero) the loop will not be entered at all and execution will continue after the `Wend` statement. If `While` entered the loop it will repeat the loop as long as the given expression is true.

```
i = 0
While i < 100
    i = i + 1
Wend
DebugPrint(i)   ; prints 100
```

The loop above will be repeated until the expression `i < 100` becomes false. This is the case when `i` is equal or greater to 100. Because we start from 0 and add 1 to `i` after each loop cycle, `i` has the value of 100 when the loop exits.

You may also want to have a look at the documentation of the `Break` and `Continue` statements. These can be used to exit from a loop or to jump to the end of it.

2) Short version While statement:

```
While <expr> Do <stat>
```

The short version behaves exactly like the long version but you do not have to include the `Wend` statement. The short `While` statement has the restriction that the loop block must only consist of one statement. If you need to execute multiple statements in the loop block, you have to use the long version. The identifier `Do` signals Hollywood that you want to use the short version.

The example from above could be written in the following way using the short `While` statement:

```
While i < 100 Do i = i + 1
```

11.4 For-Next statement

There are three versions of the `For` statement: A long version, a short version, and a generic version. The generic version of the `For` statement is also available in long and short versions so that there are actually four different versions of the `For` statement. For reasons of clarity, however, we stick with just the three different versions in this documentation.

1) Long version For statement:

```
For [Local] <var> = <expr1> To <expr2> [Step <expr3>] <loop-block> Next
```

The first thing the `For` statement does, is to set the variable specified by `var` to the expression specified by `expr1`. Now the value of the `Step` statement passed in `expr3` defines how to continue. This value is optional. If you do not specify the `Step` statement, `expr3` defaults to the value 1.

If `expr3` is positive, the `For` statement will check if the value of the variable `var` is less or equal to `expr2`. If this is the case, the loop will be entered and repeated until the value of `var` is greater than `expr2`. At the end of each loop, `expr3` is added to the value of the variable `<var>`

If `expr3` is negative, the `For` statement will check if the value of the variable `var` is greater or equal to `expr2`. If this is the case, the loop will be entered and repeated until the value of `var` is less than `expr2`. At the end of each loop, `expr3` is added to the value of the variable `var`.

In case `expr3` is zero, the loop will be repeated forever. Please do also note that the expressions specified in `expr2` and `expr3` are only evaluated once, namely at the start of the loop. Thus, the loop limit and step are constant while the loop is active and cannot be modified.

An example:

```
For i = 1 To 100
    DebugPrint(i)
Next
```

This code prints the numbers from 1 to 100. `DebugPrint()` is executed a hundred times. When the loop exits, the variable `i` has the value 101. You see that we did not specify a `Step` statement. This means that 1 is added each time the loop is repeated. If we would like to progress with factor 2, we could use the following code:

```
For i = 1 To 100 Step 2
    DebugPrint(i)
Next
```

This will print "1 3 5 7 9 ... 95 97 99". The variable i will have a value of 101 when the loop exits.

If we wanted to count down from 100 to 0, we would have to use a negative step value just as in the following example:

```
For i = 100 To 0 Step -1
    DebugPrint(i)
Next
```

This calls DebugPrint() a hundred and one times. After the loop exits, the variable i has the value -1.

If you add the Local identifier before the variable initialization, the For statement will create the iterator variable locally to the loop block. This means, that it cannot be accessed from outside the loop block. An example:

```
For Local i = 1 To 50
    DebugPrint(i)      ; prints 1, 2, 3 ... 49, 50
Next
DebugPrint(i)          ; prints 0 (i is only available inside the loop)
```

The advantage of For loops that use a local iterator variable is that they run faster than loops that use a global variable. If you do not need to access the variable of the For statement from outside the loop, you should always use the Local identifier. A limitation of For loops with the Local identifier is that you must not assign a new value to the local iterator value. If you need to exit the loop, use Break. Modifying the iterator variable during the loop's execution works only without the Local identifier.

You may also want to have a look at the documentation of the Break and Continue statements. These can be used to exit from a loop or to jump to the end of it.

2) Short version For statement:

```
For [Local] <var> = <expr1> To <expr2> [Step <expr3>] Do <stat>
```

The short version behaves exactly like the long version but you do not have to include the Next statement. The short For statement has the restriction that the loop block must only consist of one statement. If you need to execute multiple statements in the loop block, you have to use the long version. The identifier Do signals Hollywood that you want to use the short version.

The first example from above could be written in the following way using the short For statement:

```
For i = 1 To 100 Do DebugPrint(i)
```

3) Generic version For statement:

```
For <var1> [, <var2>, ...] In <expr> [Do <stat>] or [<loop-block> Next]
```

The generic version of the For statement is different from the other two versions through the fact that it calls a user-defined function to retrieve the values for each iteration. This fact makes the generic For statement suitable for a wide variety of purposes. You can write your own iterator functions but for most cases you will likely use the inbuilt iterator functions that are provided by functions such as Pairs(), IPairs(), or PatternFindStr().

The expression specified in expr is evaluated only once. It has to return three values: An iterator function, a state value, and an initial value for var1. The iterator function and the

state value are private values and they are neither visible nor accessible as variables during the `For` loop's runtime. Once the generic `For` loop has retrieved these three values, it will start calling the iterator function with the state value and current value of `var1` as the two arguments. The loop will be terminated as soon as `var1` becomes `Nil`.

Most iterator functions return multiple values for each iteration. That is why you can also specify multiple variables in the generic `For` statement. The ability to have multiple variables initialized to different iteration states makes the generic `For` statement very flexible.

Let's have a look at an example now. Consider the following table:

```
months = {"January", "February", "March", "April", "May", "June",
     "July", "August", "September", "October", "November", "December"}
```

We now want to be able to find out the index of a month (1 to 12) by using its name as a reference. Of course, we could iterate over the table, compare the name of the month to the one we are looking for, and thus find out the appropriate index. But when dealing with larger amounts of data, it is often faster to create a reverse table to find out the desired information. In our case, we want a table that uses the name of the months as indices so that table["January"] returns 1, table["February"] returns 2, and so on. We can easily create this reverse table using the generic `For` loop together with the iterator function provided by `IPairs()`. The `IPairs()` function will return an iterator function that returns two values: the index value as well as the key value for each table element that is passed to it. We can use this iterator function to reverse the table very easily:

```
revmonths = {}
For i,v In IPairs(months)
   revmonths[v] = i + 1
Next
```

Alternatively, we could also use the short version of the generic `For` statement for this code because there is only one statement in the `For` loop. Using the short version of the generic `For` statement the code would look like this:

```
revmonths = {}
For i,v In IPairs(months) Do revmonths[v] = i + 1
```

The `IPairs()` function will only iterate over all integer indices in a table. If you want to traverse all fields of a table, you can use the `Pairs()` function instead.

Another command that is often used in conjunction with the generic `For` statement is the `PatternFindStr()` function. It will return an iterator function that can be used to parse a string. For example, the following code will iterate over all words in a string:

```
s$ = "Hello World This is a test"
For w$ In PatternFindStr(s, "%a+") Do Print(w$)
```

Of course, it is also possible to write own iterator functions. This, however, can get quite complicated. That is why it is not explained here. Please consult the book "Programming in Lua (second edition)" by Roberto Ierusalimschy for more information on how to write own iterator functions.

11.5 Repeat-Until statement

There are two versions of the `Repeat` statement: A conditional version and an endless version.

1) Conditional version of the Repeat statement:

```
Repeat <loop-block> Until <expr>
```

The conditional `Repeat` statement will repeat the specified loop block until the given expression becomes true (non-zero). In other words: The block will be looped while `expr` is false (zero). This is just the other way round as the `While` statement behaves: `While` loops the code while the expression is true and `Repeat` loops the code while the expression is false.

Here is an example:

```
i = 1
Repeat
    i = i + 1
Until i = 100
```

This code counts from 1 to 100. When the loop exits, the variable `i` will have the value 100.

You may also want to have a look at the documentation of the `Break` and `Continue` statements. These can be used to exit from a loop or to jump to the end of it.

2) Endless version of the Repeat statement:

```
Repeat <loop-block> Forever
```

The endless version can be used to repeat a specific portion of code forever. You can still jump out of the loop by using the `Break` statement though. The endless version is mostly used in the main loop of a script that calls `WaitEvent()`, e.g.

```
Repeat
    WaitEvent
Forever
```

11.6 Switch-Case statement

```
Switch <ex1> Case <ex2>[:] <blk> [...] [Default[:] <blk>] EndSwitch
```

The `Switch` statement can be used to compare the expression specified in `ex1` with all other expressions specified after the `Case` identifiers. You can use as many `Case` statements as you want but there must be at least one `Case` identifier in your `Switch` statement. If the expression after a `Case` matches `ex1`, the code after that `Case` statement is executed. After the execution, Hollywood continues with your program after the `EndSwitch` statement. If none of the `Case` expressions match `ex1`, the code following the `Default` statement will be executed. Note that using the `Default` statement is optional. If you do not need it, you do not have to use it. If you do use it, however, it always needs to be the last statement of the `Switch` statement. It isn't allowed to have additional `Case` statements after the `Default` statement. The colon after the `Case` and `Default` statements is optional too.

Please note that the expression following the `Case` statement must always be constant. You cannot use variables or function return values here. Also you must not mix variable types

in this statement: If `ex1` is a string, all other expressions must be strings too. If `ex1` is a number, all other expressions must be numbers as well.

Here is an example:

```
Switch x
Case 1:
    DebugPrint("x = 1")
Case 2:
    DebugPrint("x = 2")
Default:
    DebugPrint("x <> 1 And x <> 2")
EndSwitch
```

The above code looks at the variable `x` and enters the first `Case` statement if `x` is one. The second `Case` statement is entered if `x` is 2. Otherwise the `Default` statement is entered.

Every Switch-Case statement can also be written as a normal `If` statement. The example from above would look like the following then:

```
If x = 1
    DebugPrint("x = 1")
ElseIf x = 2
    DebugPrint("x = 2")
Else
    DebugPrint("x <> 1 And x <> 2")
EndIf
```

C and Java programmers should note that Hollywood's `Switch` statement does not automatically fall through to the next `Case` block after reaching the end of the previous `Case` block. Instead, Hollywood will automatically jump to the end of the statement after a `Case` block has been executed. Thus, you do not have to use `Break` at the end of a `Case` block either. But you can use it earlier to exit from the `Switch` statement.

It is, however, possible to manually force a fall through using the `FallThrough` statement. Whenever Hollywood encounters this statement inside a `Case` block, it will fall through to the next `Case` block (or the `Default` block), i.e. it will jump directly into this block. Therefore, `FallThrough` may only be used if there is a `Case` or `Default` block following. Otherwise an error will be generated. Since the `Default` block must always be the last block of a `Switch` statement, it is not allowed to use `FallThrough` in the `Default` block because there is no subsequent block to fall through to.

Here is an example:

```
Switch msg.action
Case "OnKeyDown":
    FallThrough
Case "OnKeyUp":
    DebugPrint("Key event:", msg.key)
Default:
    DebugPrint("Other event")
EndSwitch
```

The code above will print the key that has been pressed in case `msg.action` is `OnKeyDown` or `OnKeyUp`. If `msg.action` is `OnKeyDown`, the `FallThrough` statement is used to jump into the `OnKeyUp` block.

Alternatively, the `FallThrough` statement can also be placed directly after the expression after the `Switch` statement. In that case, each `Case` block will automatically fall through to the next one, unless there is a `Break` statement forbidding falling through.

Here is an example:

```
Switch msg.action FallThrough
Case "OnKeyDown":
Case "OnKeyUp":
    DebugPrint("Key event:", msg.key)
    Break
Case "OnMouseDown":
Case "OnMouseUp":
    DebugPrint("Left mouse event")
    Break
Case "OnRightMouseDown":
Case "OnRightMouseUp":
    DebugPrint("Right mouse event")
    Break
Default:
    DebugPrint("Other event")
EndSwitch
```

The code above uses `FallThrough` globally to use the same code for `OnKeyDown` and `OnKeyUp`, and for `OnMouseDown` and `OnMouseUp`, and for `OnRightMouseDown` and `OnRightMouseUp`. The `Break` statements are necessary because otherwise Hollywood would fall through all the way to the very last line of code in the `Default` block.

11.7 Break statement

```
Break [(<level>)]
```

The `Break` statement can be used to exit from a loop or from the `Switch` statement. If you call `Break` inside a loop or a `Switch` statement, then Hollywood will exit from this control structure. An example:

```
For k = 1 To 100
    DebugPrint(k)
    If IsKeyDown("ESC") = True Then Break
Next
```

The above loop counts from 1 to 100 but can be aborted at any time by pressing the escape key.

Using the optional argument `level`, you can also finish higher loops. If you do not specify `level` Hollywood will assume 1 for it. This means that the nearest loop will be finished. If you use higher values for `level` Hollywood will traverse the loops upwards. An example:

```
For x = 1 To 100
    For y = 1 To 100
```

```
            DebugPrint(x, y)
            If IsKeyDown("ESC") Then Break(2)
        Next
    Next
```

This code uses two nested `For` loops and checks in the second loop if the escape key was pressed. To finish both loops, we have to use a `Break(2)` statement now because the normal `Break` would only finish the inner loop which would be started right again because there is still the outer loop.

Please note: If you specify the optional argument `level` it is obligatory to put parentheses around it.

11.8 Continue statement

```
Continue [(<level>)]
```

The `Continue` statement can be used to jump to the end of a loop structure. An example:

```
While i < 100
    i = i + 1
    If i > 50 Then Continue
    j = j + 1
Wend
```

The code above counts `i` from 0 to 100. The variable `j` is also incremented each loop but only while `i` is less than or equal to 50. If `i` is greater than 50 `j` will not be incremented any more. At the end of the loop `i` has the value of 100 and `j` has the value of 50.

Using the optional argument `level` you can also jump to the end of higher loops. If you do not specify `level` Hollywood will assume 1 for it which means that it will jump to the end of the nearest loop. If you use higher values for `level` Hollywood will traverse the loops upwards.

Please note: If you specify the optional argument `level` it is obligatory to put parentheses around it.

11.9 Return statement

```
Return [(<retval1>, <retval2>, ...)]
```

The `Return` statement is used to exit from a user defined function. The program control will return to the position in the script from where the function was called.

Optionally, `Return` can return as many values as you like back to the caller. If you return values from a function, it is obligatory to put these values into parentheses. For example:

```
; wrong!
Function p_Min(a, b)
    If a < b Then Return a
    Return b
EndFunction

; right!
Function p_Min(a, b)
```

```
    If a < b Then Return(a)
    Return(b)
EndFunction
```

See Section 12.1 [Functions], page 137, for details.

Compatibility note: `Return` can also be used after a `Label` statement so that this label code can be called by `Gosub`. This feature is only included for compatibility with Hollywood 1.x scripts. Please do not use it any longer.

11.10 Block-EndBlock statement

```
Block <block-code> EndBlock
```

The `Block` statement simply executes the following code in a separate scope. This function is of rare use. Normally you will not need this. Here is an example:

```
For k = 1 To 100
    Block
        Local k
        For k = 1 To 2
            DebugPrint(k)
        Next
    EndBlock   ; local "k" will be deleted now
Next
```

The code above uses two variables with the name `k` in two nested loops. This is only possible because we put the inner loop in its own block and created a new local variable `k` in that block. This local variable is only accessible in this block. After the `EndBlock` statement the local variable `k` will be deleted and the global `k` will be used again.

11.11 Dim and DimStr statements

```
Dim <varname>[<dim1-size>] ([<dim2-size>], ...), ...
DimStr <varname>[<dim1-size>] ([<dim2-size>], ...), ...
```

The `Dim` and `DimStr` statements can be used to create a n-dimensional table with the specified sizes and initialize all its elements to 0 (`Dim`) or `""` (`DimStr`).

As you already know from the documentation of the table data type, table fields need to be initialized before they can be used. Even if a field shall only carry a zero or an empty string, you have to initialize it with that value before you can access it. The `Dim` statement can help you here. It will create the table specified by `varname` with the size specified in `size`. The `size` parameter must be a constant value, not a variable. Please note, that `size` specifies really the table's size and not the last element that is to be initialized. Thus, if you use 50 as `size`, Hollywood will initialize table fields 0 to 49. Table field 50 will not be initialized.

Here is an example:

```
Dim mytable[100]
```

This statement translates to the following Hollywood code:

```
Local mytable = {}
For k = 0 To 99 Do mytable[k] = 0
```

The `Dim` statement comes really handy if you want to create multi-dimensional tables. You can use as many of the square brackets after the `varname` specification as you like. Each new square bracket will create a new table dimension of the specified size. For example:

```
Dim vector[10][10][10]
```

The statement above creates a three dimensional vector table and initializes it with all zeros. This statement translates to the following, quite a bit more complex, Hollywood code:

```
Local vector = {}
For i = 0 To 9
    vector[i] = {}
    For j = 0 To 9
       vector[i][j] = {}
       For k = 0 To 9
          vector[i][j][k] = 0
       Next
    Next
Next
```

You can also create and initialize more than one table with this statement. Just use a comma after the last dimension specification and you can repeat the whole procedure as many times as you like. Example:

```
Dim table1[50], table2[50], table3[50]
```

The `DimStr` statement works in the same way than `Dim` but initializes all fields with empty strings (`""`).

Please note that `Dim` / `DimStr` will always create local tables if you are not in the main block of your script. So if you want a table to be global, be sure to use the `Dim` / `DimStr` statement in the script's main block.

Keep also in mind that `Dim` / `DimStr` do not limit the table to the specified size. The table can still grow because Hollywood is a dynamically typed language! To grow a table, simply initialize the fields you need and Hollywood will automatically grow it. For example:

```
Dim table[50]
...
For k = 50 To 59 Do table[k] = 0   ; grow table by 10 fields
```

The code above creates a table with space for 50 fields and grows it to 60 fields then. If you want to shrink a table, set the corresponding fields to `Nil`. For instance:

```
Dim table[50]
...
For k = 40 To 49 Do table[k] = Nil ; shrink table by 10 fields
```

The code above shrinks the table from 50 initialized fields to 40 initialized fields. Hollywood is a dynamically typed language in which tables do not have a fixed size. You simply grow and shrink your tables as you need it.

12 Functions

12.1 Overview

Functions can be used to break down your program into several smaller code sections which increases the readability and structure of your code. A function can be regarded as a little program of its own. It can use variables that are local to the function, which means that they will only be available inside the function and cannot be accessed from the outside. Of course, you can also access global variables from a function. Synonyms for the term "function" are the terms "procedure", "subroutine" or "statement". Functions can return nothing or any number of values of any type.

You can declare your own functions by using the identifiers `Function` and `EndFunction`:

```
Function p_Add(a, b)
    Return(a + b)
EndFunction
c = p_Add(5, 2)        ; c receives the value 7
```

You should always use the prefix `p_` in your own function names. This helps to distinguish between your own functions and built-in Hollywood functions. Also, in future versions of Hollywood there might be functions that have the same name as functions in your code. This could lead to unexpected results. So you should always use `p_` in your function names so that there will not be any confusion. The `p_` stands for "private function".

Functions have to be declared before they are called, so the following code will give you an error:

```
c = p_Add(5, 2)
Function p_Add(a, b)
    Return(a + b)
EndFunction
```

Hollywood will try to call the function `p_Add()` but will not find it because it has not been declared yet. The two variables `a` and `b` will be local to the function's scope which means that you can only access them inside the `p_Add()` function. If you pass more arguments to the function than it expects, for instance

```
c = p_Add(5, 2, 4)     ; c receives the value 7
```

then all superfluous arguments will be discarded. In this case the argument number 3 will be thrown away. If you pass less arguments to the function than it expects, for instance

```
c = p_Add(5)           ; c receives the value 5 because 5 + Nil = 5
```

then Hollywood will pass the special value `Nil` for all arguments, which the function expects but which were not specified.

Functions can return values by using the `Return()` statement. It is obligatory that the return values are enclosed by parentheses. If you need to return multiple values, simply separate them by using commas. For example:

```
Function p_SomeValues()
    Return(5, 6, 7, 8, 9, 10)
EndFunction
```

When you call functions which return multiple arguments and you do not specify enough variables to hold all the return values, all return values you did not specify variables for will be discarded, e.g.:

```
a, b, c = p_SomeValues()
```

The line above assigns 5 to `a`, 6 to `b` and 7 to `c`. The return values 8, 9 and 10 will be discarded. If you specify more variables than there are return values, the superfluous variables will get the special value `Nil`:

```
a, b, c, d, e, f, g, h = p_SomeValues()  ; "g" and "h" will be Nil
```

This line has two superfluous variables `g` and `h`. They will be assigned the value `Nil` because the `p_SomeValues()` function returns only six values.

Of course, you can also define functions which do not return any value. These functions are also called statements. For example:

```
Function p_WaitSecs(s)
    Wait(s, #SECONDS)
EndFunction
```

If you try to get a return value from statements, you will just receive `Nil` as shown in the following line:

```
a = p_WaitSecs(5)
```

The variable `a` will be set to `Nil` because `p_WaitSecs()` does not return any value.

12.2 Functions are variables

In Hollywood functions are just variables of the type function. Therefore, you can easily assign them to other variables, e.g.:

```
myfunc = DisplayBrush          ; assign DisplayBrush to "myfunc"
myfunc(1, #CENTER, #CENTER)    ; calls DisplayBrush(1, #CENTER, #CENTER)
```

You can even write the definition of a function as an assignment:

```
p_Add = Function(a, b) Return(a + b) EndFunction
c = p_Add(5, 2)   ; c receives 7
```

The definition of `p_Add()` in the first line is the same as if you wrote:

```
Function p_Add(a, b)
    Return(a + b)
EndFunction
```

You could also replace Hollywood functions with your own ones, e.g. if you want all `Print()` calls to use `DebugPrint()` instead, the following code could do this:

```
Function p_Print(...)
    DebugPrint(Unpack(arg))   ; redirect arguments to DebugPrint()
EndFunction
Print = p_Print        ; all calls to Print() will call p_Print() now
Print("Hello World!")  ; Print() refers to p_Print() now
```

Or an even simpler solution:

```
Print = DebugPrint    ; redirect all calls to Print() to DebugPrint()
Print("Hello World!") ; calls DebugPrint() directly
```

12.3 Callback functions

Several Hollywood commands allow you to specify callback functions. Callback functions are normal Hollywood functions with the difference that they are not called by the script but by Hollywood commands. They are an integral part of Hollywood and make your program much more flexible. The whole button and event handler system in Hollywood relies heavily on callback functions. For example, if the user presses your button, then Hollywood will run the callback function you provided for that specific event. Callback functions are simply passed as normal arguments to the corresponding Hollywood commands.

An example of a Hollywood command that uses callback functions is the `MakeButton()` command. This command expects a table in the seventh parameter, which defines callback functions for the different events that can occur. Possible events for a button are `OnMouseOver`, `OnMouseOut`, `OnMouseDown`, `OnMouseUp`, `OnRightMouseDown` and `OnRightMouseUp`. If you simply want to react on a button press, use the `OnMouseUp` event. This event will be triggered when the user releases the left mouse button while the mouse pointer is still over the button. Here is an example:

```
MakeButton(1, #SIMPLEBUTTON, 0, 0, 100, 100, {OnMouseUp = p_MyFunc})
```

This command creates a new button with the identifier 1 and defines that the function `p_MyFunc()` shall be called, when the user presses this button. In this case, `p_MyFunc()` is a callback function. It will not be called by you but by Hollywood when the user presses the button. This works automatically. More precisely, the callback functions are actually called by the Hollywood command `WaitEvent()` which you should use in every script. The callback function itself could look like this now:

```
Function p_MyFunc()
    DebugPrint("Button 1 pressed!")
EndFunction
```

You could also place this function directly in the argument list of `MakeButton()`. This would look like this then:

```
MakeButton(1, #SIMPLEBUTTON, 0, 0, 100, 100, {OnMouseUp =
    Function() DebugPrint("Button 1 pressed!") EndFunction})
```

You see that Hollywood is very flexible. Remember that if you declare functions within an argument list of a call, you must not provide a function name because those functions are anonymous. So the following code would be invalid:

```
; invalid code!
MakeButton(1, #SIMPLEBUTTON, 0, 0, 100, 100, {OnMouseUp =
    Function p_MyFunc() DebugPrint("Button 1 pressed!") EndFunction})
```

Callback functions usually receive a message table in parameter 1. In the above example we did not fetch this message because we declared the function without any arguments. For the above code this is fine but imagine the following example:

```
MakeButton(1, #SIMPLEBUTTON, 0, 0, 100, 100, {OnMouseUp = p_MyFunc})
MakeButton(2, #SIMPLEBUTTON, 200, 0, 100, 100, {OnMouseUp = p_MyFunc})
```

Now we have declared two buttons but they both call the same function when the user presses them. The function `p_MyFunc()` needs to know now which button was pressed when it gets called. `p_MyFunc()` can find this out by looking at the message it receives in argument 1:

```
Function p_MyFunc(msg)
    If msg.id = 1
            DebugPrint("Button 1 pressed!")
    ElseIf msg.id = 2
            DebugPrint("Button 2 pressed!")
    EndIf
EndFunction
```

You see that `p_MyFunc()` checks the id-field of the message it got in argument 1 and so it can distinguish between button 1 and 2. Of course, you could extend that to any number of buttons. But there is more to look at. Consider the following situation:

```
evttable = {OnMouseUp = p_MyFunc, OnRightMouseUp = p_MyFunc}
MakeButton(1, #SIMPLEBUTTON, 0, 0, 100, 100, evttable)
MakeButton(2, #SIMPLEBUTTON, 200, 0, 100, 100, evttable)
MakeButton(3, #SIMPLEBUTTON, 400, 0, 100, 100, evttable)
```

Now we have declared three buttons and they all use the same event table. Thus Hollywood will call the same function for all of them. Furthermore, these buttons react on another event, namely `OnRightMouseUp`. Now `p_MyFunc()` needs to be able to distinguish not only between several buttons but also between different events. But that is no problem at all because the message passed to `p_MyFunc()` has another field from which you can read the event which caused the function call. Our `p_MyFunc()` function would look like this now:

```
Function p_MyFunc(msg)
    Switch msg.action
    Case "OnMouseUp":
            DebugPrint("Left mouse button pressed:", msg.id)
    Case "OnRightMouseUp":
            DebugPrint("Right mouse button pressed:", msg.id)
    EndSwitch
EndFunction
```

So you see that it is no problem to handle multiple buttons and events with the very same callback function. This increases the readability of your program a lot! There is much more to be discovered, so make sure you read the documentation about `MakeButton()` too.

If you want to be notified when the user closes or moves the window, you can install a callback function for that using `InstallEventHandler()`. The function you pass to this Hollywood command will then be called every time the user presses the window's close box or moves the window. But `InstallEventHandler()` supports more event types: You can also install a callback function that will be invoked if the user presses or releases a key and much more.

The Hollywood command `SetInterval()` also uses callback functions. The function passed to `SetInterval()` will be called again and again at the specified interval. This is useful if you want to make sure that your script runs at the same speed on every system. To realise this, simply use `SetInterval()` to tell Hollywood that it should run your callback function 25 times a second. So you can make sure that it does not run faster on faster machines. See Section 29.26 [SetInterval()], page 582, for a good overview of the interval technique.

`SetTimeout()` is another example of a Hollywood command that works with callback function. You pass a function as well as a timeout value to `SetTimeout()`. Your callback

function will then be called exactly after the specified time has elapsed. This is very useful for correct timing of your script, e.g. timing of your script with the music.

Last but not least, the Hollywood command `CopyFile()` accepts a function in the fourth parameter. This function will be called from time to time while `CopyFile()` is copying files. This is a difference to the callback functions of `MakeButton()`, `SetInterval()`, `SetTimeout()` etc. These are always called by `WaitEvent()` and not by the Hollywood command itself. `CopyFile()`, however, will call the specified function while it is running. So you could delete your callback function after `CopyFile()` is done (you can delete function by setting them to `Nil`). This is not possible with `MakeButton()` or `SetInterval()` because those functions just install the callbacks but they do not invoke them. This task is left to `WaitEvent()`. The callback function of `CopyFile()` is usually used to update a progress bar or abort the copy operation at any time. See Section 26.6 [CopyFile()], page 413, for details.

12.4 Return values

If your function returns one or more values it is required to specify these values in parentheses. These are required because otherwise the parser would treat them as separate statements. Consider the following code:

```
; wrong code!
Function p_Max(a, b)
    If a > b Then Return a
    Return b
EndFunction
```

The Hollywood parser would interpret this code in the following way:

```
Function p_Max(a, b)
    If a > b Then Return   ; if a > b, then return no value !!
    a                      ; execute function a() !!
    Return                 ; return no value !!
    b                      ; execute function b() !!
EndFunction
```

You see that this does not make much sense. The parentheses are obligatory because Hollywood allows you to type as many commands as you wish in one line without any delimiters. And Hollywood allows you to call functions that do not accept arguments without specifying parentheses. Therefore a statement like `Return a` is converted into two statements, namely `Return` and `a()`. If you want to return the variable `a` you have to write `Return(a)`. The correct version of our `p_Max()` function thus has to look like this:

```
Function p_Max(a, b)
    If a > b Then Return(a)
    Return(b)
EndFunction
```

By using the parentheses you signal to Hollywood that the variables `a` and `b` belong to the `Return()` calls and are not separate functions.

If a function returns more than one value but you want to have only the first return value, you need to put a pair of parentheses around the function call. This will cast the result of this function to one single value. For example, the following function returns three values:

```
Function p_ThreeVals()
    Return(1, 2, 3)
EndFunction
```

If you pass this function now to a function that accepts a multiple number of arguments, e.g. `DebugPrint()` all three return values will be passed to `DebugPrint()` as well:

```
DebugPrint(p_ThreeVals()) ; prints "1 2 3"
```

If you want `DebugPrint()` to receive only the first return value of `p_ThreeVals()` you need to put parentheses around the `p_ThreeVals()` call so that it looks like this:

```
DebugPrint((p_ThreeVals()))  ; prints "1"
```

Functions cannot only return numbers, but also strings, tables and even other functions. For example, the following code is completely legal:

```
Function p_Return_a_Table()
    Return({1, 2, 3, 4, 5})
EndFunction
a = p_Return_a_Table()
DebugPrint(a[3])         ; prints 4
```

In practice, you will probably not use this feature very much but you should know that it is at least possible to have functions that return tables or other functions. Another example:

```
Function p_Return_Func()
    Return(Function(s) DebugPrint(s) EndFunction)
EndFunction
myfunc = p_Return_Func()
myfunc("Hello World!")   ; calls DebugPrint("Hello World!")
```

12.5 Recursive functions

Hollywood supports recursive functions, i.e. you can write functions which call themselves. For example, here is a function which calculates the faculty of `n`:

```
Function p_Fac(n)
    If n = 0 Then Return(1)   ; 0! = 1
    Return(n * p_Fac(n - 1))  ; multiply n with n - 1 until n = 0
EndFunction
```

As you can see above, the `p_Fac()` function calls itself again and again until the `n` counter is zero. This is what we call a recursive function.

12.6 Variable number of arguments

You can also write functions which accept any number of arguments. To do this you have to use the `...` identifier as the last parameter. Your function will then get a local table called `arg` which contains all parameters that were passed to your function including an element called `n` which carries the number of parameters that were passed to the function. Please also note that the arguments will be stored in the `arg` table starting at index 0. For example, here is a function that calculates the average of all parameters that are passed to it:

```
Function p_Average(...)
```

```
    Local pars = arg.n            ; how many parameters were passed
    Local avg, k                  ; temporary locals

    For k = 1 To pars
            avg = avg + arg[k-1]  ; sum up all parameters
    Next

    Return(avg / pars)            ; and divide the sum by their quantity

EndFunction

a = p_Average(10, 20, 30, 40, 50) ; (10 + 20 + 30 + 40 + 50) / 5 = 30
b = p_Average(34, 16, 27, 39)     ; (34 + 16 + 27 + 39) / 4 = 29
c = p_Average(10, 10)             ; (10 + 10) / 2 = 10
Print(a, b, c)                    ; prints "30 29 10"
```

It is important to note that the `...` identifier must be specified as the last entry of your parameter list. You cannot do things like:

```
; invalid code
Function p_Test(a, b, ..., c)
    ...
EndFunction
```

This will obviously not work because Hollywood could never know which parameter belongs to `c`. Using parameters before the `...` identifier works fine though:

```
Function p_MinMax(ismin, ...)

    Local pars = arg.n            ; number of parameters passed
    Local k

    If ismin = True               ; find out smallest element?
            Local min = arg[0]    ; store the smallest element here
            For k = 2 To pars     ; iterate over all elements
                    If arg[k-1] < min Then min = arg[k-1] ; smaller ?
            Next
            Return(min)           ; and return the smallest
    Else
            Local max = arg[0]    ; store the greatest element here
            For k = 2 To pars     ; iterate over all elements
                    If arg[k-1] > max Then max = arg[k-1]  ; greater ?
            Next
            Return(max)          ; and return the greatest element
    EndIf

EndFunction

a = p_MinMax(True, 4, 8, 2, 3, 10, 1, 7, 9, 5, 6)   ; returns 1
```

```
b = p_MinMax(False, 4, 8, 2, 3, 10, 1, 7, 9, 5, 6)  ; returns 10
```

This function will return the smallest number of the specified parameters if the first argument is `True` or the greatest number if the first argument is set to `False`.

If you need to pass all arguments over to another function, the `Unpack()` function can become handy. It will return all elements of a table. For example, if you want to write your own `Print()` function:

```
Function p_Print(...)
    Print(Unpack(arg))
EndFunction
```

All arguments passed to `p_Print()` will be passed over to `Print()` using the `Unpack()` function.

12.7 Functions as table members

As we have already learnt before functions in Hollywood are just variables of the type "function". Therefore, you can use them everywhere where you can use variables. This includes tables. You can store functions just like normal strings or values inside a table and call them from there. Let us look at an example:

```
mathlib = {}  ; create an empty table

Function mathlib.add(a, b)
    Return(a + b)
EndFunction

Function mathlib.sub(a, b)
    Return(a - b)
EndFunction

Function mathlib.mul(a, b)
    Return(a * b)
EndFunction

Function mathlib.div(a, b)
    Return(a / b)
EndFunction

a = mathlib.mul(5, 10)  ; a receives the value 50
```

The table mathlib contains four functions now that can be called from it. Of course, we could also declare the functions during the initialization of the table. This would look like the following:

```
mathlib = {add = Function(a, b) Return(a + b) EndFunction,
           sub = Function(a, b) Return(a - b) EndFunction,
           mul = Function(a, b) Return(a * b) EndFunction,
           div = Function(a, b) Return(a / b) EndFunction}
a = mathlib.mul(5, 10)  ; a receives the value 50
```

This code does the very same as the code above but is more compact. Functions inside a table are also often refered to as "methods". This is a term from the object-oriented programming world.

12.8 Local functions

Because functions in Hollywood are just variables of type "function", you can also use local functions which have a limited lifetime. They work pretty much the same way than local variables and have the same advantages too. Here is an example of a local function:

```
Block
    Local p_Add = Function(a, b) Return(a + b) EndFunction
    Print(p_Add(5, 6))     ; prints 11
EndBlock
```

In the above code, the function `p_Add()` will be local to the block it has been declared in. Thus, any attempts to call `p_Add()` after the `EndBlock` statement will lead to an error.

You could also use the more common function definition to create local functions. This code does the same as the code above but uses the common way of declaring functions:

```
Block
    Local Function p_Add(a, b) Return(a + b) EndFunction
    Print(p_Add(5, 6))     ; prints 11
EndBlock
```

Using local functions can also become handy if you want to temporarily replace a Hollywood function. For example, the following code replaces the `DebugPrint()` function with the `Print()` function but only for the lifetime of the block where it has been defined (and in subordinate blocks):

```
If error = True
    Local Function DebugPrint(...) Print(Unpack(arg)) EndFunction
    DebugPrint("An error occurred!")  ; redirects to Print()
EndIf
DebugPrint("Hello")                   ; points to DebugPrint() again
```

The string "An error occurred!" will be rendered to your display in the code above because we have defined a local function called `DebugPrint()` which calls the Hollywood function `Print()`. This local `DebugPrint()` will be killed when the block is left. The following call to `DebugPrint()` will then call the real Hollywood `DebugPrint()` function.

12.9 Methods

It is also possible to use Hollywood for object-oriented programming. Hollywood does not have the concept of class but you can easily emulate the behaviour using tables and metatables. One thing that is important for object-oriented programming is that object functions usually receive a handle to themselve as the first parameter. This parameter is usually called `self` or `this`. Of course, you can emulate this behaviour by simply declaring a `self` or `this` parameter in your function and passing to it the object whenever you call the function, but you can also use a special syntax for object-oriented programming that Hollywood offers.

If you declare your functions using the colon operator Hollywood will automatically initialize a hidden `self` parameter for you. You do not have to declare it explicitly. If you use the colon syntax, it is always there. Functions that are declared using the colon syntax are called methods because they are dependent on a root object.

Here is a simple example of a methods in Hollywood:

```
cart = {items = {}, numitems = 0}

Function cart:AddItem(n$, p)
    self.items[self.numitems] = {name = n$, price = p}
    self.numitems = self.numitems + 1
EndFunction

Function cart:RemoveItem(n$)
    For Local k = 0 To self.numitems - 1
       If self.items[k].name = n$
         RemoveItem(self.items, k)
         self.numitems = self.numitems - 1
         Return
       EndIf
    Next
EndFunction

Function cart:CheckOut()
    Local total = 0

    For Local k = 0 To self.numitems - 1
       NPrint(self.items[k].name, self.items[k].price)
       total = total + self.items[k].price
    Next

    NPrint("Your total is", total)
EndFunction

cart:AddItem("DVD", 10)
cart:AddItem("Blizzard PPC", 1000)
cart:AddItem("AAA Chipset", 100000)
cart:AddItem("68070", 500)
cart:CheckOut()
cart:RemoveItem("Blizzard PPC")
cart:CheckOut()
```

The code above creates a simple class that represents a cart. The class has three methods: Add item, remove item, and check out. Furthermore, it has two properties: A table containing a list of all elements in the cart, and a count value that contains how many elements are currently in the cart. You can see that each of the three methods works with a `self` variable which has not been declared. This is because all methods have been declared using

the colon operator and thus Hollywood will always pass a the `self` parameter to them automatically.

Of course there is more much more to object-oriented programming than covered in this brief excursion. Going into the depths of OOP (inheritance, multiple inheritance, privacy, etc.) would be too much for this guide, but it is all possible with Hollywood tables and metatables. If you are interested in learning more about this topic, you should consult a book about the Lua programming language because Hollywood uses a Lua kernel. For example, the book "Programming in Lua (second edition)" by Roberto Ierusalimschy has an extensive chapter about OOP in Lua, which you usually can adapt straight into Hollywood code.

13 Unicode support

13.1 Overview

Hollywood 7.0 finally introduces full Unicode support. Before Hollywood 7.0 the program was limited to ISO 8859-1 on Windows, Linux, and macOS, and to the system's default charset on AmigaOS and compatible systems. Hollywood 7.0 now comes with full Unicode support which is implemented using the UTF-8 character encoding. Thus, starting with Hollywood 7.0, all your scripts should be saved in the UTF-8 character encoding, either with or without BOM.

All text stored inside strings will now be stored as UTF-8 and all the functions in the string and text libraries will now expect UTF-8 formatted text by default. It is possible to put Hollywood in legacy mode, i.e. to force it to use ISO 8859-1 or the system's default charset on AmigaOS, by using the `-encoding` console argument or its counterpart in the `@OPTIONS` preprocessor command, but this is not recommended because in legacy mode, your script isn't guaranteed to run flawlessly on different locales.

All string and text library functions that need to operate on characters within strings accept an optional encoding parameter which allows you to set the character encoding the string uses. Normally, it is not necessary to use this optional encoding parameter because it is highly recommended to always use UTF-8. In some cases, however, it might be handy to be able to have string and text functions operate on encodings different from UTF-8. This is especially so if you need to operate on raw binary data stored in a string. In that case, you can just pass the `#ENCODING_RAW` constant to the respective functions to tell them that you wish to operate on the raw binary data inside the string instead of characters stored in UTF-8 encoding. The string functions won't perform any integrity checks on the string that is passed to them and will just operate on the raw binary data stored inside the string.

To change the default character encoding for the string and text libraries, you can use the `SetDefaultEncoding()` function. See Section 54.30 [SetDefaultEncoding], page 1138, for details. However, this is normally not needed and you should just keep `#ENCODING_UTF8` as the default encoding.

If you need to listen to non-English keys, you have to use the new `VanillaKey` event handler with `InstallEventHandler()`. `VanillaKey` supports the complete Unicode range of characters whereas `OnKeyDown` and `OnKeyUp` only support control keys and standard English keys. See Section 29.13 [InstallEventHandler], page 553, for details.

In the course of the transition to Unicode in Hollywood 7.0 there might be some compatibility issues with older scripts. All potential issues are discussed in the compatibility section. See Section 6.2 [Compatibility], page 69, for details.

Finally, please note that Hollywood's text renderer currently only supports traditional text, i.e. text that is laid out from left to right on horizontal lines. The text renderer currently doesn't support text that runs from right to left or vertical text.

13.2 Character encodings

Most of the string and text library functions accept an optional parameter specifying the character encoding to use. This parameter tells the function how the strings you pass to it are internally formatted, i.e. which character encoding they use.

Normally, you shouldn't have to use this parameter at all because starting with Hollywood 7.0 all text should be stored as UTF-8. Under certain circumstances, however, it might be necessary to use the optional character encoding parameter. For example, Hollywood strings can also contain raw binary data. This data of course isn't valid UTF-8 and thus the string functions will reject it. The only way to operate on this data then is to tell the respective functions that this isn't UTF-8 encoded data but just a raw sequence of bytes. This can be done by passing the `#ENCODING_RAW` constant in the character encoding parameter.

Here is an overview of the different encodings available in Hollywood:

`#ENCODING_UTF8:`
: This is the default encoding since Hollywood 7.0 and should be used whenever you work with text.

`#ENCODING_ISO8859_1:`
: This was the default encoding before Hollywood 7.0. It is still supported for compatibility reasons but it isn't recommended to use it.

`#ENCODING_RAW:`
: This is a synonym for `#ENCODING_ISO8859_1`. It can be used to tell the string library functions to treat the string as raw binary data instead of text.

`#ENCODING_AMIGA:`
: This specifies the system's default character set on AmigaOS and compatible systems. This constant is only supported by `ConvertStr()` and only on AmigaOS and compatible systems, obviously. `#ENCODING_AMIGA` allows you to convert between AmigaOS' default character set and UTF-8 (both ways).

You can use the `SetDefaultEncoding()` function to change the default character encoding for the string and text libraries. See Section 54.30 [SetDefaultEncoding], page 1138, for details.

14 Troubleshooting

14.1 Troubleshooting

This section covers some common problems and presents their solutions.

1. **Table initialization:** Be careful when trying to create a table field by assigning a variable to it that has not been used before, i.e. is `Nil`. If you do that, the table field will not be created. For example, the following will not work:

   ```
   t = {}    ; create a table
   t.x = y   ; assign 'y' to field x; note that y is Nil
   DebugPrint(t.x)  ; ---> Error! Field 'x' not initialized!
   ```

   The solution is to initialize `y` first, e.g.:

   ```
   t = {}    ; create a table
   y = 0     ; set y to 0
   t.x = y   ; assign 'y' to field x
   DebugPrint(t.x)  ; Works! Prints '0'
   ```

2. **Checking a variable against Nil:** Be careful when checking a variable against `Nil`! `GetType()` is the only reliable way to find out if a variable is `Nil` or not. Checking the variable against the `Nil` identifier is not a good idea because that would also result in `True` if the variable was zero instead of `Nil`. Example:

   ```
   a = 0
   b = Nil
   DebugPrint(GetType(a) = #NIL, a = Nil)  ; prints "0 1"
   DebugPrint(GetType(b) = #NIL, b = Nil)  ; prints "1 1"
   ```

   You see that "a = Nil" returns `True` although `a` is zero. That is because `Nil` is always regarded as zero when used in expressions. Thus, if you want to find out whether a variable really is `Nil`, use `GetType()` and compare the result against `#NIL`. Starting with Hollywood 6.0 you can also use the dedicated `IsNil()` function to check a variable against `Nil`. See Section 52.20 [IsNil], page 1084, for details.

3. **Wrong variable initialization:** In Hollywood initializing multiple variables is a bit different than in most other languages because Hollywood expects only one equal sign. For example, this might look correct but it is wrong:

   ```
   ; bad code!
   Local a = 5, b = 6, c = 7
   ```

   Unfortunately, this bad code would not even trigger an error but it would be interpreted in a wrong way. The code above would make Hollywood assign 5 to `a` and simply drop the rest because there is only one variable on the left side of the equal sign. So you have to be careful with multiple variable initialization. Accordingly, the correct version would be the following code:

   ```
   ; good code!
   Local a, b, c = 5, 6, 7
   ```

   This code will assign 5 to a, 6 to b, and 7 to c.

4. **Returning values:** Be careful with functions that return something. The return value has to be enclosed in parentheses. Code like this is wrong and does not trigger an error:

```
Function p_Add(a, b)
    Local r = a + b
    Return r   ; OUCH!!!
EndFunction
```

This code would be interpreted as "Return, and then call the function r()". Of course, the call to `r()` will never be reached but the function as written above will return `Nil`, i.e. nothing, in every case. The correct version is this:

```
Function p_Add(a, b)
    Local r = a + b
    Return(r)
EndFunction
```

14.2 Frequently asked questions

This section covers some frequently asked questions. Please read them first before asking on the forum because your problem might have been covered here.

Q: Is it possible to compile *.apk Android packages directly with Hollywood so that I can publish them on Google Play?

A: Yes, that's possible with the Hollywood APK Compiler which is available as a Hollywood add-on. Please visit the official Hollywood portal at http://www.hollywood-mal.com for more information. Alternatively, you can also use to freely available Hollywood Player for Android if you want to run your Hollywood projects on Android. See Section 2.4 [Mobile platforms], page 24, for details.

Q: Is it possible to create GUIs in Hollywood which use the native widgets of the operating system?

A: Yes, this is now possible with the RapaGUI plugin. RapaGUI allows you to create native GUIs for AmigaOS and compatibles, Windows, macOS, and Linux. GUI layouts can be conveniently defined in XML. RapaGUI is available for free and can be downloaded from http://www.hollywood-mal.com. If you only target AmigaOS and compatibles, you can also use the MUI Royale plugin. This allows you to use almost the complete MUI API from Hollywood and creating GUIs with MUI Royale is also very convenient because GUI layouts can be defined in XML.

Q: I have compiled my project for macOS but macOS refuses to start it. What can I do?

A: macOS is very strict when it comes to executing apps that haven't been signed by a registered Apple developer. Typically, unsigned apps end up in "quarantine" which means that you can't open them. You can, however, clear the quarantine flag of an app by executing the following line on the terminal:

```
xattr -dr com.apple.quarantine /path/to/your/App.app
```

Then you should be able to open your app without problems.

Q: 2D drawing is too slow. What can I do to make this faster?

A: On some platforms Hollywood does all of its drawing using the CPU. This ensures maximum compatibility with a wide range of hardware. Using hardware-accelerated drawing increases the risk of glitches with buggy graphics drivers of the host OS. Nevertheless, Hollywood supports hardware-accelerated drawing. To do that, you need to set up a hardware double buffer and create your brushes as hardware brushes. Then 2D drawing will be done using the GPU and this will be extremely quick. Note that the Windows, macOS, and Linux versions don't have inbuilt support for hardware double buffers and hardware brushes at the moment. You need to use a plugin like GL Galore or RebelSDL in order to be able to use hardware double buffers are hardware brushes on those systems. See Section 21.37 [Hardware brushes], page 280, for details.

Q: I'm using the auto scaling engine (or the "FullScreenScale" display mode) to scale my script to a higher resolution. The performance is very poor. Can't Hollywood use the GPU for scaling?

A: Hollywood doesn't support hardware-accelerated scaling on all platforms by default. On Windows, this is only available on Windows 7 and better. On macOS, it is only available on 10.10 and better. If you're running Hollywood on a system where the auto scaling engine shows a poor performance, you might be able to get massive speed improvements by using a plugin which supports hardware-accelerated scaling, e.g. the GL Galore or RebelSDL plugins. Plugins which support hardware-accelerated scaling can apply auto scaling in almost no time. So if Hollywood's inbuilt auto scaling performance is too poor for your requirements, you might want to use a plugin which supports hardware-accelerated scaling. See Section 5.4 [Obtaining plugins], page 66, for details.

Q: The Windows version of Hollywood comes with a nice IDE. Why is there no such IDE on the Amiga platform?

A: It would be too much work to create such an IDE for Amiga compatibles. There are already several other programs that you can use on the Amiga to create scripts for Hollywood. Check out Dietmar Eilert's Cubic IDE (Link: `http://www.softwareandcircuits.com/`) or Simon Archer's Codebench (Link: `http://codebench.co.uk`). Both programs support Hollywood through plugins.

Q: How can I link images, sounds, fonts, etc. into my compiled executable?

A: If you are using the preprocessor commands like `@BRUSH`, `@BGPIC`, `@MUSIC`, `@FONT`, etc. then Hollywood will link all external data declared using these commands to your executable automatically. If you are loading your external data manually using `LoadBrush()`, `OpenMusic()`, `SetFont()` etc., then you have to use the `-linkfiles` console argument to specify which files should be linked to your executable.

Q: How do I switch between windowed and full screen mode?

A: There is a hotkey that can switch your scripts between windowed and full screen mode: Just press `CMD+RETURN` on AmigaOS and macOS, or `LALT+RETURN` on Windows. If you need to switch modes from your script, use the `ChangeDisplayMode()` command.

Q: How can I change the icon in executables compiled using Hollywood?

A: Use the `@APPICON` preprocessor command for that.

Q: When using TrueType fonts, I noticed that the text looks slightly different between AmigaOS and Windows, or Windows and macOS, or AmigaOS and macOS. How can I fix that?

A: If you want TrueType text to look exactly the same on every platform, you have to use Hollywood's inbuilt font engine. You can enable the inbuilt font engine by passing `#FONTENGINE_INBUILT` to the `SetFont()`, `OpenFont()` or `@FONT` commands. By default, Hollywood will use the host operating system's native font engine (`#FONTENGINE_NATIVE`) and this leads to a slightly different look on each platform. If you don't want that, use `#FONTENGINE_INBUILT`.

Q: How can I increase the raw performance of my script?

A: You might want to disable the line hook for brief periods of time to increase the raw performance of Hollywood's virtual machine. See Section 52.3 [DisableLineHook], page 1072, for details.

Q: I have compiled my script for macOS but when I start it under macOS, I get an error indicating that the data files for my program could not be found!

A: Make sure that you put all data files required by your program inside the `Resources` folder of the application bundle compiled by Hollywood. For example, if Hollywood compiled a bundle called `MyCoolProgram.app`, then you need to put all data files that are required by `MyCoolProgram.app` inside the following bundle folder: `MyCoolProgram.app/Contents/Resources`. Then it will work.

Q: I have compiled my script for macOS/Linux but it won't start. What's wrong there?

A: Make sure that the main program inside the application bundle has the executable flag set. When cross-compiling programs for macOS/Linux on Windows or AmigaOS, the executable flag often is not set correctly because macOS and Linux use a different file system. So you sometimes need to set this flag manually.

Q: Is there visual designer for Hollywood scripts or do I have to edit every script using a text editor?

A: Yes, there is a program called Hollywood Designer which has a powerful WYSIWYG interface to create your Hollywood projects. Have a look at `http://www.hollywood-mal.com` for more information about this great program. Please note that Hollywood Designer is currently only available for Amiga compatible systems.

Q: Is there a Hollywood forum where I can get in touch with other users?

A: Yes, please check out the "Community" section of the official Hollywood Portal online at http://www.hollywood-mal.com.

Q: When I compile my script for Windows/macOS, I'm getting an error message that Hollywood can't open the fonts I'm using! What am I doing wrong?

A: See Section 54.42 [Working with fonts], page 1157, for a detailed explanation of how to deal with fonts in multi-platform scripts.

Q: I see that programs compiled by Hollywood support many different console arguments. But I never start my programs compiled by Hollywood from the console! Can I still pass console arguments to them somehow?

A: Yes, that's possible. See Section 3.3 [Passing console arguments without a console], page 54, for details.

Q: When trying to load an animation, I'm always getting an "Out of memory!" error although I have 512 MB RAM.

A: Make sure that you enable disk-buffered playback. This can be done by using the `FromDisk` tag in `LoadAnim()` or `@ANIM`. If you don't specify `FromDisk`, Hollywood will buffer the entire anim in memory and because Hollywood always uses 32-bit graphics, 512 MB are used up pretty soon.

Q: When I'm trying to access a non-existing field in my table, Hollywood immediately exits with an error message! Can I somehow check if a table field exists before accessing it?

A: That is possible using the `HaveItem()` function. It will return `False` if the specified table field does not exist.

Q: The sound Hollywood outputs is distorted under AmigaOS. What is wrong there?

A: Check your AHI settings. You will have to set the master volume in your AHI advanced settings to "With Clipping". If this does not help, try to reduce the master volume in Hollywood, by specifying the `-mastervolume` argument. You can also reduce the master volume in the GUI preferences. You should also turn off the echo and surround modes if there are any sound problems. Also make sure that you have set the frequency for your sound driver correctly. It should be at least 22050 Hz.

Q: I would like to add my own commands to Hollywood using a plugin. Is there a SDK available?

A: Yes, the Hollywood SDK is available for download from the official Hollywood portal at http://www.hollywood-mal.com. It comes with many examples and extensive documentation that should get you started.

Q: How do I interrupt scripts that run in a window that has no close box?

A: Just press CTRL+C. This will always work except when CTRL+C has been disabled using `CtrlCQuit()`.

Q: Where can I ask for help?

A: The best place to ask for help is the official Hollywood forum `http://forums.hollywood-mal.com`. There is also a newsletter which is used for announcements. So if you want to stay up to date about the latest Hollywood releases, new plugins, updates, and everything else about Hollywood visit `http://www.hollywood-mal.com` and sign up for the newsletter.

Q: I have found a bug.

A: Please post about it in the "Bugs" section of the forum.

15 Tutorials

15.1 Tutorial

This little tutorial shows you how to create your own slide show in ten easy steps. Try to understand every step that is taken here and you will soon be able to create your own scripts.

The following things are required for this tutorial:

1. Background picture called `BG.png` with two arrows. Rectangle embracing arrow 1 is X: 4, Y: 430, W: 35, H: 19. Rectangle embracing arrow 2 is X: 591, Y: 430, W: 35, H: 19. The area where the pictures are displayed is at coordinates X: 29, Y: 41.
2. 11 pictures named `0.jpg`, `1.jpg`, `2.jpg` ... `10.jpg` of size 571x377 pixels
3. Protracker module named `MyMusic.mod`

Of course I have prepared all these things for this tutorial. They are in your Hollywood directory under Help/Tutorial. Please copy those files to the directory where you will create your script. Then follow these steps:

1. Start up your favorite text editor
2. The background for your slideshow is a picture that you have created with your favorite paint program. In our example, I have already prepared a background.
3. Now we need to tell Hollywood that it shall use the file `BG.png` as the first background picture. This is done by specifying the preprocessor command `@BGPIC` together with `BG.png`. So you have to write the following code now in your script file:

   ```
   @BGPIC 1, "BG.png"
   ```

   This command tells Hollywood to use `BG.png` as the initial background picture. The initial background picture must always have the identifier 1. If there is no background picture with the identifier 1, Hollywood will create a blank display.
4. Our slideshow also shall have some background music. This music is a Protracker module with the name `MyMusic.mod`. So we add the following line to our script:

   ```
   @MUSIC 1, "MyMusic.mod"
   ```
5. Now we have to define areas in our background picture that shall be accessible as buttons. As you can see, there are two arrows in the background picture. As all buttons need to be defined as a rectangle, we need to find out the coordinates as well as the width and height of each arrow. You can use a paint program like PPaint to find out the coordinates. For our background picture, the left arrow is in a rectangle with the coordinates 4:430 (top left corner) and the width/height of 35/19. So we can add the left arrow now as button 1 to our script by writing the following code in the script file:

   ```
   MakeButton(1, #SIMPLEBUTTON, 4, 430, 35, 19,
     {OnMouseUp = p_Back})
   ```

   We can do the same now with the right arrow which is in a rectangle starting at 591:430 with the same dimensions as arrow 1. So we write the following keyword in our script:

   ```
   MakeButton(2, #SIMPLEBUTTON, 591, 430, 35, 19,
     {OnMouseUp = p_Forward})
   ```

Now we have defined two buttons that can be clicked. If button 1 gets clicked Hollywood will call the function `p_Back()`, and if button 2 gets clicked Hollywood will call the function `p_Forward()`.

6. Now we can start adding a bunch of commands that tell Hollywood what to do. At first, we want that the background music starts to play. As we have declared `MyMusic.mod` as the music object with number 1, we now call `PlayMusic()` with argument 1. Add the following line to your script and Hollywood will play your Protracker module:

   ```
   PlayMusic(1)
   ```

   We also need to define which picture shall be the last one. In our example we will have 11 pictures ranging from `0.jpg` to `10.jpg`, so the last picture is number 10. Therefore we add the following line to our script:

   ```
   lastpic = 10
   ```

   We also add the following line because our first pic is `0.jpg`:

   ```
   pic = 0
   ```

7. The next command shall load the next picture and display it. As there is no command which does all this, we need to write a little function. This routine will be called `p_NextPic()`. You will see in point 10 how to write this routine. Let us pretend that it was already there and therefore we will now say that Hollywood shall execute this routine:

   ```
   p_NextPic()
   ```

8. Now we add the main loop to our script. The popular format of this loop is the following:

   ```
   Repeat
       WaitEvent
   Forever
   ```

   The command `WaitEvent()` holds the script execution until an event occurs, e.g. a button is pressed. If an event occurs `WaitEvent()` will jump to the function that handles this event, e.g. if button 1 is clicked, `WaitEvent()` will jump to the function `p_Back()`. When the function has finished its action, it jumps back in our main loop and `WaitEvent()` gets called again. This is repeated until the user closes the window.

9. Now the structure of our script is complete. What we still need to do now is adding the functions `p_Back()` which is called by `WaitEvent()` when button 1 (backward button) was pressed and `p_Forward()` which is called by `WaitEvent()` when button 2 (forward button) was pressed. It is important that you define the functions before you reference them. Thus, you need to add the following code before the calls to `MakeButton()` which you added in step 5. So let's add the functions for the buttons now:

   ```
   Function p_Back()
     If pic =0
       pic = lastpic
     Else
       pic = pic - 1
     EndIf
     p_NextPic()
   EndFunction
   ```

```
Function p_Forward()
  If pic = lastpic
    pic = 0
  Else
    pic = pic + 1
  EndIf
  p_NextPic()
EndFunction
```

As you can see in the above code, the variable `pic` contains the actual picture number. If the user clicks the forward button, `pic` is increased by one, if the backward button is clicked it is decreased by one. The variable is also checked against 0 and `lastpic` so that it always stays in the range of our pictures.

10. Now the only thing left to do is our function `p_NextPic()` which shall load and display the picture with the number that is stored in the variable `pic`. Here is the code. Remember to insert this code before the calls to `MakeButton()`.

```
Function p_NextPic()
    LoadBrush(1, pic .. ".jpg")
    DisplayBrush(1, 29, 41)
EndFunction
```

So what does the routine `p_NextPic()` do? It simply adds the `".jpg"` extension to the variable `pic` and after that it loads the file and displays the brush at coordinates 29:41. So the pictures must be named like this `0.jpg` (first pic), `1.jpg` (second pic), `2.jpg` (third pic) and so on.

Altogether our script looks now like this:

```
@BGPIC 1, "BG.png"
@MUSIC 1, "MyMusic.mod"

Function p_NextPic()
   LoadBrush(1, pic .. ".jpg")
   DisplayBrush(1, 29, 41)
EndFunction

Function p_Back()
   If pic =0
     pic = lastpic
   Else
     pic = pic - 1
   EndIf

   p_NextPic()
EndFunction

Function p_Forward()
   If pic = lastpic
     pic = 0
```

```
   Else
     pic = pic + 1
   EndIf

   p_NextPic()
EndFunction

MakeButton(1, #SIMPLEBUTTON, 4, 430, 35, 19,
   {OnMouseUp = p_Back})
MakeButton(2, #SIMPLEBUTTON, 591, 430, 35, 19,
   {OnMouseUp = p_Forward})
PlayMusic(1)

lastpic = 10
pic = 0

p_NextPic()

Repeat
  WaitEvent
Forever
```

Now you can save your script and start it through the Hollywood GUI or from the console. Congratulations, you have just created your first Hollywood script! Wasn't that easy? Only 35 lines of code!

Now you can go and extend it if you want. For example, if you want that the picture gets displayed with a transition effect just replace the line

```
DisplayBrush(1, 29, 41)
```

with the line

```
DisplayBrushFX(1, 29, 41, #RANDOMEFFECT)
```

and your picture will appear with a nice transition effect from Hollywood's wide palette of transition effects.

15.2 Animation techniques

When it comes to animation, you have to choose between three techniques: Sprites, double buffering, and layers. This section is designed to give you an overview of the three techniques so that the decision is easier for you.

1. Sprites: Sprites are especially useful when there are not much graphics to be drawn. For example, if you only need to move some blocks or player and enemy sprites around. In this case, it is better to use sprites because Hollywood can refresh the display pretty fast because not much changes. See Section 50.1 [Sprite introduction], page 1011, for details.
2. Double buffering: Using a double buffer Hollywood always needs to refresh the whole display. Although hardware acceleration is used here when possible this can still be quite expensive when you have a 640x480 display which needs to be refreshed 25 times

a second. Thus, a double buffer is only recommended when a lot of custom graphics have to be drawn. For instance, the Hollywood examples that draw a real sine scroller use a double buffer because they need to draw a lot of different tiles. Such things would not be possible with sprites because the drawing operations are heavily customized and change every frame. See Section 30.3 [BeginDoubleBuffer()], page 590, for details.

3. Layers: Hollywood comes with a powerful layers system which allows you to access every graphics item on the display as its own layer and modify its position, size, and looks on the fly. The layer system is extremely flexible and powerful at the cost of speed so if you need to draw a lot of graphics it might be faster to use double buffering instead.

Here is a recommendation of animation techniques that are suitable for common types of applications:

Board/card games:
: Sprites or layers because fast graphics aren't required.

Tetris:
: Sprites or layers because there's not much action and screen updates do not have to be very fast.

PacMan:
: Sprites or layers. The only thing that moves are the enemies and the player.

2D shooter:
: Double buffering because the background is scrolling. Hence, the whole screen has to be updated every frame.

Jump'n'Run:
: Double buffer if there is a scrolling background. If the game doesn't scroll then sprites or layers.

Scene demo:
: Double buffer by any means. A lot of custom graphics have to be drawn. This is a classical double buffer case.

If you use sprites or layers you should also encapsulate all commands required to draw a single frame of your project within a `BeginRefresh()` and `EndRefresh()` section. This will allow Hollywood to use optimized drawing on systems that do not support partial screen refresh like Android. As a welcome side effect using `BeginRefresh()` and `EndRefresh()` will also improve drawing speed when autoscaling is active. See Section 30.4 [BeginRefresh], page 591, for details.

15.3 Script timing

Correct timing is a crucial issue for every good script that should be able to run on many different systems. As Hollywood is available for a multitude of platforms you have to think about script timing if you plan to give your script to others. The basic problem is that if you do not add speed limiters to your script, it will run as fast as possible. That might not be a problem on old 200 Mhz systems but on a gigahertz machine it surely is a problem. Imagine you have a game and use the following loop:

```
/* Bad code */
While quit = False
```

```
        dir = p_QueryInput()
        If dir = "Left" Then p_MoveSpriteLeft()
        If dir = "Right" Then p_MoveSpriteRight()
        ....
        p_RedrawDisplay()
Wend
```

This loop has two serious problems:

1. There is no timing in this loop. The loop will always run as fast as the CPU of the host system allows. This is really bad. It means that the timing will only be correct on your system and nowhere else.
2. This loop will eat 100% of your CPU power because there is no limit that says "Execute this loop 25 times a second and that's enough!". Consuming 100% of the CPU power might not be a problem on very slow systems but when running the script on a newer system the OS could noticeably slow down and the CPU fan might start up because of the heat generated by the CPU. Additionally, remember that Hollywood runs in a multitasking environment. In such an environment you have to be a good citizen and only take as much CPU time as you really need. If you take all the CPU time without really needing it, all other processes will get less CPU time.

The solution to the problem is pretty simple: We just need to tell Hollywood to execute this loop only a certain amount of times per second. For most games, it is completely sufficient to query for input and draw graphics 25 times per second. There are two methods how you can implement such a throttle:

1. Using `WaitTimer()`. This function accepts a timer and a timeout value. It pauses the script until the specified time has passed. Then the timer is reset and you can use it again. Our loop from above would look like the following using `WaitTimer()`:

   ```
   /* Good code */
   StartTimer(1)    ; start timer #1

   While quit = False
        dir = p_QueryInput()
        If dir = "Left" Then p_MoveSpriteLeft()
        If dir = "Right" Then p_MoveSpriteRight()
        ....
        p_RedrawDisplay()
        WaitTimer(1, 40)    ; do not run faster than 40 milliseconds
   Wend
   ```

   Now our loop will never run faster than 40 milliseconds. Thus, it will never be executed more than 25 times per second because 25 * 40 = 1000. Hence, a game using this loop will run at the same speed on every system - no matter if the CPU has 50mhz or 1ghz.
2. The second method is `SetInterval()`. This function allows you to install an interval function that Hollywood will call at the frequency you specify. Thus, you can tell Hollywood to call your game loop 25 times a second. The code to do this looks like this:

   ```
   /* Good code */
   Function p_MainLoop()
   ```

```
        ; this does the same code as our While-Wend loop above
        dir = QueryInput()
        If dir = "Left" Then MoveSpriteLeft()
        If dir = "Right" Then MoveSpriteRight()
        ....
        RedrawDisplay()
EndFunction

; call MainLoop() 25 times a second -> 40 * 25 = 1000 milliseconds
SetInterval(1, p_MainLoop, 40)

While quit = False
     WaitEvent
Wend
```

This code does the same as the code above using `WaitTimer()`. The only difference is that you have to use `WaitEvent()` with `SetInterval()` because interval functions trigger Hollywood events.

Both of the methods discussed above are easy to use and efficient. It is up to you to decide which one you prefer.

16 Amiga support library

16.1 AmiDock information

Hollywood has native support for AmigaOS 4's AmiDock system. You can make your script appear in AmiDock by setting the `RegisterApplication` tag in the `@OPTIONS` preprocessor command to `True`. See Section 52.25 [OPTIONS], page 1088, for details.

By default, Hollywood will show the icon obtained from the script's or application's `*.info` file in AmiDock. If you want Hollywood to show a custom icon in AmiDock, you can do so by specifying a number of icons using the `@APPICON` preprocessor command and then you have to tell Hollywood which icon to show by setting the `DefaultIcon` tag. See Section 18.5 [APPICON], page 209, for details. Alternatively, you can use the `DockyBrush` tag with the `@OPTIONS` preprocessor command.

There are two different types of dockies that Hollywood supports:

1. Standard docky: This is the default docky type. Your application will appear in AmiDock as an icon that has two different states. The icon's second state will be shown every time the user clicks on it. Standard dockies have the disadvantage that they cannot have a context menu associated with them and it also takes a lot of time to change the standard docky icon at runtime using `ChangeApplicationIcon()`. There will be a clearly noticeable relayout if you change the icon of a standard docky. If you do not need a context menu and you never need to update your docky icon, however, standard dockies are the best choice.
2. App docky: App dockies are more flexible than standard dockies as they can have a context menu associated with them and it is also possible to change their icons really quickly using `ChangeApplicationIcon()`. This makes it possible to show animations in AmiDock, for example. The downside of app dockies is that app docky icons can only have a single state, i.e. it is impossible to associate a second icon that is to be shown whenever the user clicks on the docky with app dockies.

By default, Hollywood will create a standard docky for you. App dockies are only created if you attach a context menu to your docky by specifying the `DockyContextMenu` tag in `@OPTIONS` or if you call `ChangeApplicationIcon()` and pass only one instead of two images to the function. An alternative way to make your application start up as an app docky is to use the `DockyBrush` tag with the `@OPTIONS` preprocessor command.

If you want to have your script registered as an OS4 application without an icon in AmiDock, you will have to set the `NoDocky` tag to `True` with the `@OPTIONS` preprocessor command.

16.2 CloseAmigaGuide

NAME

CloseAmigaGuide – close current AmigaGuide window (V6.1)

SYNOPSIS

```
CloseAmigaGuide()
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION
This function closes an AmigaGuide window that has been opened using `OpenAmigaGuide()`. When Hollywood shuts down, this function is called automatically.

INPUTS
none

16.3 CreateRexxPort

NAME
CreateRexxPort – create a Rexx port for your script (V2.5)

SYNOPSIS
`CreateRexxPort(name$)`

PLATFORMS
AmigaOS and compatibles only

FUNCTION
This function will create a rexx port for your script and assign the specified name to it. In order to receive ARexx messages, your script needs to have an ARexx port. Other applications can then communicate with your script by sending messages to this port. All messages that arrive at your Rexx port will be forwarded to the callback function which you need to install using the `InstallEventHandler()` function (use the OnARexx event handler). If you do not install this event handler, you will not get any notifications on incoming messages.

Please remember that Rexx port names are always given in case sensitive notation. Thus, "MYPORT" and "myport" denote two different Rexx ports. For style reasons it is suggested that you use only upper case characters for your port name. Furthermore, each Rexx port must be unique in the system. If you specify a port name which is already in use, this function will fail. Thus, make sure that you use a unique name.

Please note that every Hollywood script can only have one ARexx port. Hence, this function can only be called once in your script. You cannot delete the port created by this function. It will be automatically destroyed when Hollywood exits.

An example how to catch ARexx messages is provided below. See Section 29.13 [InstallEventHandler], page 553, for more information on how the user callback function will be called.

INPUTS

`name$` desired name for your Rexx port

EXAMPLE

```
Function p_EventFunc(msg)
  Switch msg.action
  Case "OnARexx"
    Switch msg.command
    Case "RealFunc"
      Return(100)
```

```
    Default
      Local t = SplitStr(msg.args, "\0")
      DebugPrint(msg.command, "called with", msg.argc, "argument(s)")
      For Local k = 1 To msg.argc
         DebugPrint("Argument", k .. ":", t[k - 1])
      Next
    EndSwitch
  EndSwitch
EndFunction
CreateRexxPort("MY_COOL_REXX_PORT_123")
InstallEventHandler({OnARexx = p_EventFunc})
Repeat
  WaitEvent
Forever
```

Save the code above as a Hollywood script and run it with Hollywood. Then save the following code as a Rexx script and execute it from a Shell with "RX test.rx":

```
/* remember the first line of every Rexx script must be a comment */
OPTIONS RESULTS

/* the port of our Hollywood script is now the host */
ADDRESS MY_COOL_REXX_PORT_123

/* send commands from Rexx to Hollywood and watch the debug output */
DummyFunc_1 '"Dummy Arg 1"'
DummyFunc_2 1 2 3
DummyFunc_3 '"First arg"' '"Second arg"' '"Third arg"'
DummyFunc_4  /* no args */
DummyFunc_5 "These will be handled as separate arguments"
DummyFunc_6 '"This is a single argument (use double quotes!)"'

'RealFunc'
SAY RESULT   /* this will print 100; it is the result from RealFunc */
```

16.4 GetApplicationList

NAME

GetApplicationList – get a list of all registered applications (V6.0)

SYNOPSIS

```
t = GetApplicationList()
```

PLATFORMS

AmigaOS 4 only

FUNCTION

This function returns a table containing a list of all applications that have been registered through application.library.

INPUTS
none

RESULTS

`t` table containing a list of strings describing all registered applications

EXAMPLE

```
t = GetApplicationList()
For Local k = 0 To ListItems(t) - 1 Do DebugPrint(t[k])
```

The code above prints all registered applications.

16.5 GetFrontScreen

NAME
GetFrontScreen – get name of frontmost screen (V8.0)

SYNOPSIS

```
n$ = GetFrontScreen()
```

PLATFORMS
AmigaOS and compatibles only

FUNCTION
This function can be used to get the name of the screen that is currently up front. In case the screen isn't a public screen, an empty string is returned.

Note that in a multitasking environment like AmigaOS, all screens that aren't owned by your application can disappear at any time and their stack order can change as well, so you need to be prepared that at the time this function returns the screen is no longer front or even doesn't exist any more.

INPUTS
none

RESULTS

`n$` the screen that is currently front (see warning above concerning the reliability of this information)

EXAMPLE

```
ShowScreen("WORKBENCH")
Print(GetFrontScreen())
```

This is not guaranteed to return "WORKBENCH", although it will normally do so. See warning above.

16.6 GetPubScreens

NAME
GetPubScreens – return a list of all available public screens (V5.2)

SYNOPSIS

```
t, info = GetPubScreens()
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This function can be used to query the system for a list of all available public screens. It will return a table that contains one string element for each public screen that is currently open.

Starting with Hollywood 5.3 this function will return a second table containing information about the screen dimensions and color depth. This second return table will contain as many elements as the first return table and there will be one subtable for each public screen that is currently open. Each subtable will contain the following fields:

`Width:` Initialized to the width of the public screen.

`Height:` Initialized to the height of the public screen.

`Depth:` Initialized to the depth of the public screen.

You can use `ShowScreen()` to switch to a public screen. If you want to move your display to a specific public screen, use `SetDisplayAttributes()`.

Note that in a multitasking environment like AmigaOS, all screens that aren't owned by your application can disappear at any time so you need to be prepared that this function returns screens that don't exist any longer because they have been closed already.

INPUTS

none

RESULTS

`t` table containing a number of strings describing all open public screens

`info` additional table containing information about the screen dimensions and depth (V5.3)

EXAMPLE

```
t = GetPubScreens()
For Local k = 0 To ListItems(t) - 1 Do DebugPrint(t[k])
```

This code lists all public screens.

16.7 HideScreen

NAME

HideScreen – hide public screen (V7.1)

SYNOPSIS

```
HideScreen([s$])
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This function can be used to move the public screen specified in `s$` to the back. If the `s$` argument is omitted, the screen that is currently at the front will be moved to the back.

Note that in a multitasking environment like AmigaOS, all screens that aren't owned by your application can disappear at any time so you need to be prepared that this function fails because the screen doesn't exist any longer.

INPUTS

`s$` optional: name of public screen to move to back (defaults to an empty string which means move the current screen to the back)

EXAMPLE

```
HideScreen()
```

This code moves the currently active screen to the back.

16.8 OpenAmigaGuide

NAME

OpenAmigaGuide – open AmigaGuide document in new window (V6.1)

SYNOPSIS

```
OpenAmigaGuide(file$[, node$])
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This command will open the AmigaGuide file specified by `file$` and display it in its own window. If the optional argument `node$` is specified, Hollywood will show this particular node of the AmigaGuide file, otherwise the table of contents is shown.

You can close the AmigaGuide file by calling the `CloseAmigaGuide()` command. Open AmigaGuide files will also be closed automatically when Hollywood quits.

Note that there can be only one open AmigaGuide file at a time. If you call this function and an AmigaGuide file is already visible, Hollywood will close that old AmigaGuide file first.

INPUTS

`file$` AmigaGuide file to show

`node$` optional: node to show (defaults to `""` which means show the table of contents)

EXAMPLE

```
OpenAmigaGuide("Hollywood:Help/Hollywood.guide", "OpenAmigaGuide")
```

The code above shows this page.

16.9 RunRexxScript

NAME

RunRexxScript – run an ARexx script from file or memory (V2.5)

SYNOPSIS

```
res$ = RunRexxScript(script$[, nofile])
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

You can use this function to run the ARexx script specified in `script$`. Additionally, you can also run ARexx code directly by setting the optional `nofile` argument to `True`. In that case, `script$` must not be a path to an ARexx script but must contain the ARexx code to execute. The function will return the result from the ARexx script. The return value will always be a string - even if it contains just a number. If ARexx does not return anything, you will receive an empty string.

You have to start RexxMast prior to using this function. It is, however, not necessary to create a Rexx port in order to use this function. This function works also if your script does not have a Rexx port. The script will always be started with "REXX" being the host port. Thus, if you want to address an other port, you have to use the "ADDRESS" command of ARexx first.

If you use this function to start external ARexx scripts, make sure that the first line of your ARexx script is a comment. Otherwise you will receive a "program not found" error. As a matter of syntax, the first line of all ARexx scripts must be a comment.

INPUTS

`script$` path to an external ARexx script or ARexx code directly; in the latter case, `nofile` must be `True`

`nofile` optional: `False` is `script$` contains a path to an ARexx script and `True` if `script$` is ARexx code (defaults to `False`)

RESULTS

`res$` return value from ARexx; this is always a string

EXAMPLE

```
RunRexxScript("dh0:MyScript.rx")
```

The above code runs the script "dh0:MyScript.rx".

```
r$ = RunRexxScript("SAY 'Hello'\nRETURN 5\n", True)
```

The above code prints "Hello" to the console and returns 5 to Hollywood. The variable `r$` thus will contain "5" after the call.

16.10 SendApplicationMessage

NAME

SendApplicationMessage – send message to another application (V6.0)

SYNOPSIS

```
SendApplicationMessage(app$, msg$)
```

PLATFORMS

AmigaOS 4 only

FUNCTION

This function can be used to send a message to a registered AmigaOS 4 application. The name of the receiving application has to be passed in `app$` and the message itself is passed in `msg$`.

Please note that this function can only be used if you have set the `RegisterApplication` tag in `@OPTIONS` to `True`. See Section 52.25 [OPTIONS], page 1088, for details.

INPUTS

`app$`	name of the application that should receive the message
`msg$`	the message to send

16.11 SendRexxCommand

NAME

SendRexxCommand – send command to Rexx port (V2.5)

SYNOPSIS

```
res$ = SendRexxCommand(port$, cmd$)
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This function sends the command specified in `cmd$` to the Rexx port specified in `port$`. The function will then return the result from the command. The return value will always be a string - even if it contains just a number. If the command does not return anything, you will receive an empty string. You can also send multiple commands with this function. Just separate the statements with `";"` or you can also use new line characters (`"\n"`) for separation.

If you do not want to address a specific Rexx port, simply pass `"REXX"` in `port$`. In that case, the system's standard ARexx port will be your host port. Please also remember that port names are case sensitive, i.e. `"MYPORT"` and `"myport"` denote two different Rexx ports. For style guide reasons, port names are usually in upper case only.

You have to start RexxMast prior to using this function. It is, however, not necessary to create a Rexx port in order to use this function. This function works also if your script does not have a Rexx port.

INPUTS

`port$`	name of the port you want to address
`cmd$`	the command(s) you want to send to that port

RESULTS

`res$` return value from ARexx; this is always a string

EXAMPLE

```
SendRexxCommand("WORKBENCH", "WINDOW 'Sys:' OPEN")
```

The above code will open the SYS: drawer on your Workbench. Please note that the ARexx interface of the Workbench is a feature introduced in OS3.5. Thus, you will require OS3.5 or better. MorphOS does probably not support the Workbench ARexx interface because it is only rarely used. See the OS3.9 NDK for documentation on the available commands.

16.12 SetScreenTitle

NAME

SetScreenTitle – change the screen title of the current display (V6.0)

SYNOPSIS

```
SetScreenTitle(title$)
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This function can be used to change the text that should be shown in the screen's title bar whenever the current display is active. By default, "Workbench screen" will be shown.

INPUTS

`title$` new screen title

EXAMPLE

```
SetScreenTitle("My cool program")
```

The above code changes the screen title to "My cool program".

16.13 ShowRinghioMessage

NAME

ShowRinghioMessage – show a Ringhio notification (V6.0)

SYNOPSIS

```
ShowRinghioMessage(title$, text$[, table])
```

PLATFORMS

AmigaOS 4 only

FUNCTION

This function can be used to show a notification through AmigaOS 4's Ringhio system. You need to pass a title for the notification in the first argument and the actual text in the second argument.

The optional table argument can be used to specify additional parameters for the way the Ringhio notification should be handled. The following tags are accepted here:

`PubScreen:`
: This tag can be used to specify the name of the public screen the notification should appear on.

`ImageFile:`
: This tag can be used to specify the path to an image file that should be displayed inside the Ringhio notification. For the best results, you should use only PNG images with alpha channel here.

`DoubleClickClose:`
: If you set this tag to `True`, the Ringhio notification window can be closed by double-clicking on it. In that case a message will be sent to your application containing the string you specify in the `DoubleClickMessage` tag (see below).

`DoubleClickMessage:`
: If the user double-clicks the Ringhio notification window in order to close it, the string you specify here will be sent to your application using the `OnApplicationMessage` event handler. This tag is only supported if you have also set the `DoubleClickClose` tag to `True` (see above). If the string specified here has the following format `"URL:http://www.example.com/"` the Ringhio server will not send a message back to your application but it will automatically show the specified URL in the default browser if the user double-clicks on the notification window.

Please note that this function can only be used if you have set the `RegisterApplication` tag in `@OPTIONS` to `True`. See Section 52.25 [OPTIONS], page 1088, for details.

INPUTS

`title$` title for the Ringhio notification

`text$` text to show in the Ringhio notification

`table` optional: table containing further parameters (see above)

16.14 ShowScreen

NAME

ShowScreen – switch to specified public screen (V5.2)

SYNOPSIS

```
ShowScreen(s$)
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This function can be used to bring the public screen specified in `s$` to the front. If you want to move your display to a specific public screen, use `SetDisplayAttributes()` with the `PubScreen` tag.

Note that in a multitasking environment like AmigaOS, all screens that aren't owned by your application can disappear at any time so you need to be prepared that this function fails because the screen doesn't exist any longer.

INPUTS

`s$` name of public screen to bring to front

EXAMPLE

```
ShowScreen("WORKBENCH")
```

This code brings Workbench screen back to the front.

17 Anim library

17.1 Overview

Animations are Hollywood objects that contain several frames of image data. They can be streamed from disk or they can be buffered entirely in memory. To stream an animation from disk, you can use the `OpenAnim()` command to open the animation file or, alternatively, set the `FromDisk` tag to `True` in `LoadAnim()` or the `@ANIM` preprocessor command to `True`. To load an animation entirely into memory, use the `LoadAnim()` command. It is recommended, though, to always stream larger animations because buffering all frames will require lots of memory.

Here is how you can open an animation for streaming from disk using the `@ANIM` preprocessor command (of course, you could also use the `OpenAnim()` command instead):

```
@ANIM 1, "test.anim", {FromDisk = True}
```

By default, Hollywood can open the animation formats IFF ANIM and GIF ANIM. There are, however, several plugins which extend the number of animation formats you can open with Hollywood. For example, you can download plugins for the APNG and FLI/FLC formats from the official Hollywood portal.

Animations can be played using the `PlayAnim()` command. `PlayAnim()` will block the script execution, but you set the `Async` tag to `True` to get an asynchronous draw object from `PlayAnim()` which you can then use to play the animation asynchronously using `AsyncDrawFrame()`. See Section 17.18 [PlayAnim], page 196, for details.

Alternatively, you can also draw individual anim frames by using the `DisplayAnimFrame()` command. If layers are enabled, it's very easy to show the next frame by just using the `NextFrame()` command after you have created an anim layer by using `DisplayAnimFrame()`. See Section 17.7 [DisplayAnimFrame], page 184, for details.

Normally, animations contain raster pixel data, but Hollywood also supports special vector animations which consist of vector path data instead and can thus be freely transformed. See Section 17.25 [Vector anim information], page 204, for details.

17.2 ANIM

NAME

ANIM – preload an animation for later use (V2.0)

SYNOPSIS

```
@ANIM id, filename$[, table]
```

FUNCTION

This preprocessor command preloads the animation specified in `filename$` and assigns the identifier `id` to it.

Anim formats that are supported on all platforms are IFF ANIM, GIF ANIM, AVI (uncompressed or using Motion JPEG compression), and formats you have a plugin for. Depending on the platform Hollywood is running on, more anim formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open

all anim formats you have datatypes for as well. On Windows, `@ANIM` can also load anim formats supported by the Windows Imaging Component.

Starting with Hollywood 4.5, `@ANIM` can also automatically create animations from an image file. If you want to use an image file with `@ANIM`, you need to specify the optional `Frames` argument. See below for more information.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall be made transparent in the animation.

`Link:`
: Set this field to `False` if you do not want to have this animation linked to your executable/applet when you compile your script. This field defaults to `True` which means that the animation is linked to your to your executable/applet when Hollywood is in compile mode.

`FromDisk:`
: If you set this field to `True`, Hollywood will not load the whole animation into memory but it will load the single frames directly from disk when needed. This is slower but requires much less memory. For the functions of the anim library it does not matter whether the animation is completely in memory or loaded dynamically from disk. You can use all anim functions like `ScaleAnim()` also with anims that are loaded from disk. Anim layers are also correctly handled with disk anims. (V3.0)

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the anim shall be loaded, too. Please note that most anim formats do not support alpha channels. Thus, it is advised that you create the anim manually from a PNG picture using `CreateAnim()` if you need to have an alpha channel in your animation. This field defaults to `False`. (V4.5)

`X, Y, Width, Height, Frames, FPR:`
: This group of fields is only used when you specify an image file source. In that case, you have to use these arguments to tell `@ANIM` how it shall create the animation from the image. `Width` and `Height` define the dimensions for the animation and `Frames` specifies how many frames `@ANIM` shall read from the source image. If the frames are aligned in multiple rows in the source image, you will also have to pass the argument `FPR` (abbreviation for frames per row) to tell `@ANIM` how many frames there are in each row. Finally, you can tell `@ANIM` where in the image file it should start scanning by specifying the fields `X` and `Y` (they both default to 0). `@ANIM` will then start off at position `X` and `Y` and read `Frames` number of images with the dimensions of `Width` by `Height` from the picture specified by `filename$`. After it has read `FPR` number of images, it will advance to the next row. (V4.5)

`SkipLoopFrames:`
: If you set this to `True`, Hollywood will automatically skip the last two frames of the anim. This is only required for IFF ANIMs that have two loop frames

at the end of the anim. Auto detection of loop frames is not possible because it would require Hollywood to decode the whole anim first. That is why you have to tell Hollywood manually whether the anim has loop frames or not. (V5.3)

`Deinterlace:`
This tag allows you to specify how Hollywood should deinterlace interlaced anims. This can be set to either `#DEINTERLACE_DEFAULT` or `#DEINTERLACE_DOUBLE`. If set to `#DEINTERLACE_DEFAULT` (which is as the name implies also the default), Hollywood will combine two half-frames into one full frame. This mostly results in the best quality but can lead to visual artefacts when there is a lot of movement in the anim. If you use `#DEINTERLACE_DOUBLE` instead, Hollywood will double the lines of a half-frame to get a full frame. This leads to some quality loss but can make the anim look more smooth. The best deinterlace mode to use always depends on the anim. Note that mostly you should not have to care about this tag at all because deinterlacing is actually only required for some obscure IFF ANIM formats which store interlaced frames like ANIM16i and ANIM32i. (V5.3)

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this anim. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
If this tag is set to `True`, the monochrome transparency of the anim will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based anim. If you want to load the alphachannel of an anim, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
If this tag is set to `True`, Hollywood will load the anim as a palette anim. This means that you can get and modify the anim's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette animations also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`. (V9.0)

`TransparentPen:`
If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made

transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the anim file (if the anim format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. An animation cannot have a mask and an alpha channel!

Starting with Hollywood 9.0, this preprocessor command can also load vector anim formats if you have an appropriate plugin installed. Keep in mind, though, that if you load vector anim formats using `@ANIM`, the anim may not support all features of normal anims. See Section 17.25 [Vector animations], page 204, for more information on vector anims.

If you want to load anims manually, please use the `LoadAnim()` command.

INPUTS

`id` a value that is used to identify this animation later in the code

`filename$`
the animation file you want to load

`table` optional: a table that can contain a combination of the fields discussed above

EXAMPLE

```
@ANIM 1, "MyAnim.gif"
```

Load "MyAnim.gif" as animation number 1.

```
@ANIM 1, "MyAnim.gif", {Transparency = $FF0000}
```

Does the same like above but the animation is now transparent (transparency color is red=$FF0000).

```
@ANIM 1, "Huge_Animation.iff", {Link = False}
```

The code above loads the specified animation and tells Hollywood that it should never link this anim because it is so big.

17.3 BeginAnimStream

NAME

BeginAnimStream – begin sequential anim creation (V4.5)

SYNOPSIS

```
[id] = BeginAnimStream(id, file$, width, height[, format, table])
```

FUNCTION

This function allows you to create an empty animation object on disk that you can then subsequently append frames to using `WriteAnimFrame()`. The advantage of `BeginAnimStream()` over `SaveAnim()` is that SaveAnim() requires you to provide an animation object as the source. If you use `BeginAnimStream()`, you can append frames to your animation from individual brush objects. This gives you the utmost flexibility. Because of its sequential design, `BeginAnimStream()` can be used to create new animations of virtually unlimited size and length. You could easily create a 2 hour AVI video with this function.

The first argument to `BeginAnimStream()` must be an id for the new write animation object. Alternatively, you can specify `Nil` and `BeginAnimStream()` will return a handle to the object to you. The second argument specifies a path to a file that shall be created for this anim. Arguments three and four specify the desired dimensions of the animation. The fifth argument specifies the format of the animation. This can either be one of the following animation types or an anim saver provided by a plugin:

`#ANMFMT_GIF:`
: GIF format. Because GIF anims are always palette-based, RGB graphics have to be quantized before they can be exported as GIF. When calling `WriteAnimFrame()` you can use the `Colors` and `Dither` tags to specify the number of palette entries to allocate for the frame and whether or not dithering shall be applied. When using `#ANMFMT_GIF` with a palette frame, no quantizing will be done. `#ANMFMT_GIF` also supports palette anims with a transparent pen. `#ANMFMT_GIF` is the default format used by `BeginAnimStream()`.

`#ANMFMT_MJPEG:`
: AVI with Motion JPEG compression. This is a lossy anim format so you can set the `Quality` tag (see below) to control the level of compression that should be used.

`#ANMFMT_IFF:`
: IFF anim. Hollywood will use mode 5 compression (the most common compression mode) for IFF anims. Because IFF anims are always palette-based, RGB graphics have to be quantized before they can be exported as IFF. When calling `WriteAnimFrame()` you can use the `Colors` and `Dither` tags to specify the number of palette entries to allocate for the frame and whether or not dithering shall be applied. When using `#ANMFMT_IFF` with a palette frame, no quantizing will be done. `#ANMFMT_IFF` also supports palette anims with a transparent pen. (V9.0)

The optional table argument allows you to configure further parameters:

`Quality:`
: Here you can specify a value between 0 and 100 indicating the compression quality for lossy compression formats. A value of 100 means best quality, 0 means worst quality. This is only available for anim formats that support lossy compression. Defaults to 90 which means pretty good quality.

`FPS:`
: Video formats like AVI do not support an individual delay value for each frame but require a global value indicating how many frames per second

shall be displayed. This field allows you to set the FPS. This is only handled for video file formats. Defaults to 25 frames per second.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Here is a table that shows an overview which table elements can be used with the different animation formats:

	GIF	IFF	MJPEG AVI
Quality	No	No	Yes
FPS	No	No	Yes

When you have successfully obtained a handle to a new animation object, you can then sequentially append frames to it using `WriteAnimFrame()`. When you are done adding frames, you have to call `FinishAnimStream()` to finalize the animation file on disk and make it ready for use.

INPUTS

`id` id for the animation object or `Nil` for auto id selection

`file$` destination file

`width` desired width for the animation

`height` desired height for the animation

`format` optional: which anim format to use (defaults to `#ANMFMT_GIF`)

`table` optional: further arguments for save operation; see above

RESULTS

`id` optional: identifier of the animation; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateBrush(1, 320, 240)
SelectBrush(1)
SetFillStyle(#FILLCOLOR)
BeginAnimStream(1, "test.gif", 320, 240)
For Local k = 1 To 100
   Circle(#CENTER, #CENTER, k * 2, #RED)
   WriteAnimFrame(1, 1)
Next
```

```
FinishAnimStream(1)
EndSelect
```

The code above creates a new GIF animation with 100 frames. The animation will show a red circle zooming into the screen.

17.4 CloseAnim

NAME
CloseAnim – close an animation (V9.0)

SYNOPSIS
`CloseAnim(id)`

FUNCTION
This function closes the animation specified by `id`. The animation must have been previously opened using `OpenAnim()`. See Section 17.17 [OpenAnim], page 194, for details.

INPUTS

`id`	identifier of the animation

17.5 CopyAnim

NAME
CopyAnim – clone an animation (V2.0)

SYNOPSIS
`[id] = CopyAnim(src, dst)`

FUNCTION
This function clones the animation specified by `src` and creates a copy of it in the new animation with id `dst`. If you specify `Nil` in the `dst` argument, this function will choose an identifier for the cloned animation automatically and return it to you. The new animation is fully independent from the old one so you could free up the source anim after it has been cloned.

INPUTS

`src`	source animation
`dst`	destination animation id or `Nil`

RESULTS

`id`	optional: identifier of the cloned anim; will only be returned when you pass `Nil` as argument 2 (see above)

17.6 CreateAnim

NAME

CreateAnim – create animation from a brush (V2.0)

SYNOPSIS

```
[id] = CreateAnim(id, brush, width, height, frames, fpr[, sx, sy])
```

FUNCTION

This function can be used to create a new animation from a brush source. If you specify `Nil` in the `id` argument, this function will choose an identifier for this animation automatically and return it to you. The single frames will be read from the specified brush and will be put together in a new animation. You need to specify the width and height of the frames as well as the number of frames to read from the brush and how many frames are in one row. Optionally, you can define a position from where in the brush the conversion shall start.

If the source brush is transparent, the new animation will also have transparent areas. If the source brush uses an alpha channel, the animation will get an alpha channel, too.

INPUTS

`id`	identifier for the new animation
`brush`	source brush
`width`	width of each anim frame
`height`	height of each anim frames
`frames`	how many frames should the animation have
`fpr`	how many frames are in one row; if this is the same than frames, then all frames must be in the same row
`sx`	optional: x-offset in the source brush (defaults to 0)
`sy`	optional: y-offset in the source brush (defaults to 0)

RESULTS

`id`	optional: identifier of the new anim; will only be returned when you pass `Nil` as argument 1 (see above)

17.7 DisplayAnimFrame

NAME

DisplayAnimFrame – display a single frame of an animation (V4.0)

SYNOPSIS

```
DisplayAnimFrame(id, x, y, frame[, table])
```

FUNCTION

This function displays a single frame of an animation at the specified coordinates.

If layers are enabled, this command will add a new layer of the type `#ANIM` to the layer stack.

`DisplayAnimFrame()` also recognizes an optional table argument which allows you to specify one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`id`	identifier of the animation to use
`x`	destination x coordinate
`y`	destination y coordinate
`frame`	animation frame to display (1 = first frame)
`table`	optional: table specifying further options

EXAMPLE

```
DisplayAnimFrame(1, #CENTER, #CENTER, 5)
```

The code above display frame 5 of animation 1 on the center of the screen.

17.8 FinishAnimStream

NAME

FinishAnimStream – finalize sequential anim object (V4.5)

SYNOPSIS

```
FinishAnimStream(id)
```

FUNCTION

This function must be used to finalize a sequential anim object when you are done appending frames to it. When `FinishAnimStream()` returns, the new animation will be ready to use on your hard disk.

See Section 17.3 [BeginAnimStream], page 180, for more information on sequential anim objects.

INPUTS

`id`	identifier of the animation object to finalize; must have been obtained using `BeginAnimStream()`

EXAMPLE

See Section 17.3 [BeginAnimStream], page 180.

17.9 FreeAnim

NAME

FreeAnim – free an animation

SYNOPSIS

```
FreeAnim(id)
```

FUNCTION

This function frees the memory of the animation specified by `id`. To reduce memory consumption, you should free animations when you do not need them any longer.

Note that this command should be used with animations allocated by `LoadAnim()`, `CreateAnim()` or `@ANIM`. Animations allocated by `OpenAnim()` should be freed using `CloseAnim()`.

INPUTS

`id`	identifier of the animation

17.10 GetAnimFrame

NAME

GetAnimFrame – copy animation frame to brush (V3.0)

SYNOPSIS

```
GetAnimFrame(id, frame, animid)
```

FUNCTION

This function can be used to convert a single frame of an animation to a brush. The animation must have been loaded using `LoadAnim()` or the `@ANIM` preprocessor command. If you want to load a frame directly from an animation file, use `LoadAnimFrame()` instead. `GetAnimFrame()` is preferred, however, because it is faster. In the first argument, pass an identifier for the brush you want this function to create. In the second argument you have to specify which frame of the animation should be loaded, and the third argument finally specifies the identifier of the animation to use as the source.

INPUTS

`id`	identifier of brush to be created by this function
`frame`	frame to load (ranges from 1 to number of frames); specify -1 if you want to load the last frame
`animid`	identifier of the animation to use as source

EXAMPLE

```
LoadAnim(1, "TestAnim.anim")
GetAnimFrame(1, 15, 1)
```

The code above converts frame 15 of animation 1 into brush 1.

17.11 IsAnim

NAME

IsAnim – determine if an animation is in a supported format

SYNOPSIS

```
ret = IsAnim(file$[, table])
```

FUNCTION

This function will check if the file specified `file$` is in a supported animation format. If it is, this function will return `True`, otherwise `False`. If this function returns `True`, you can load the animation by calling `LoadAnim()`.

Starting with Hollywood 6.0 this function accepts an optional table argument which allows you to configure further options:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this anim. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

See Section 17.13 [LoadAnim], page 188, for a list of supported anim formats.

INPUTS

`file$` file to check

`table` optional: table configuring further options (V6.0)

RESULTS

`ret` `True` if the animation is in a supported format, `False` otherwise

17.12 IsAnimPlaying

NAME

IsAnimPlaying – check if an animation is playing (V1.0 only)

SYNOPSIS

```
playing = IsAnimPlaying(id)
```

FUNCTION

Attention: This command was removed in Hollywood 1.5.

This function checks if the animation specified by `id` is currently playing and returns `True` if it is, `False` otherwise.

INPUTS

`id` identifier of an animation

RESULTS

`playing` `True` if the animation specified by `id` is playing, `False` otherwise

EXAMPLE

```
LoadAnim(1, "Gfx/Anims/CoolAnim.anm")
PlayAnim(1, #PLAYONCE)
playing = IsAnimPlaying(1)
While playing = TRUE
   playing = IsAnimPlaying(1)
Wend
FreeAnim(1)
```

The above code loads the animation `"Gfx/Anims/CoolAnim.anm"`, plays it and then waits for it to finish. After that, the animation is freed. If you just want to do something like above, it is easier for you to use the `WaitAnimEnd()` command. But if you want to do some things during the animation is playing, you will have to do it this way (using `IsAnimPlaying()` and a loop).

17.13 LoadAnim

NAME

LoadAnim – load an animation

SYNOPSIS

```
[id] = LoadAnim(id, filename$[, table])
```

FUNCTION

This function loads the animation specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadAnim()` will automatically choose an identifier and return it.

Note that by default, this command will load all anim frames into memory which may take a while and significant amounts of memory. If you want to create an animation that dynamically loads frames as needed and only keeps the current frame in memory, use the `OpenAnim()` command instead or set the `FromDisk` tag to `True` (see below). See Section 17.17 [OpenAnim], page 194, for details.

Anim formats that are supported on all platforms are IFF ANIM, GIF ANIM, AVI (uncompressed or using Motion JPEG compression), and formats you have a plugin for. Depending on the platform Hollywood is running on, more anim formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all anim formats you have datatypes for as well. On Windows, `LoadAnim()` can also load anim formats supported by the Windows Imaging Component.

Starting with Hollywood 4.5, `LoadAnim()` can also automatically create animations from an image file. If you want to load an image file with `LoadAnim()`, you need to specify the optional `Frames` argument. See below for more information.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`

This field can be used to specify a color in RGB notation that shall be made transparent in the animation.

`FromDisk:`

If you set this field to `True`, Hollywood will not load the whole animation into memory but it will load the single frames directly from disk when needed. This is slower but requires much less memory. For the functions of the anim library it does not matter whether the animation is completely in memory or loaded dynamically from disk. You can use all anim functions like `ScaleAnim()` also with anims that are loaded from disk. Anim layers are also correctly handled with disk anims.

`LoadAlpha:`

Set this field to `True` if the alpha channel of the anim shall be loaded, too. Please note that most anim formats do not support alpha channels. Thus, it is advised that you create the anim manually from a PNG picture using `CreateAnim()` if you need to have an alpha channel in your animation. This field defaults to `False`. (V4.5)

`X, Y, Width, Height, Frames, FPR:`

This group of fields is only used when you specify an image file source. In that case, you have to use these arguments to tell `LoadAnim()` how it shall create the animation from the image. `Width` and `Height` define the dimensions for the animation and `Frames` specifies how many frames `LoadAnim()` shall read from the source image. If the frames are aligned in multiple rows in the source image, you will also have to pass the argument `FPR` (abbreviation for frames per row) to tell `LoadAnim()` how many frames there are in each row. Finally, you can tell `LoadAnim()` where in the image it should start scanning by specifying the fields `X` and `Y` (they both default to 0). `LoadAnim()` will then start off at position `X` and `Y` and read `Frames` number of images with the dimensions of `Width` by `Height` from the picture specified by `filename$`. After it has read `FPR` number of images, it will advance to the next row. (V4.5)

`SkipLoopFrames:`

If you set this to `True`, Hollywood will automatically skip the last two frames of the anim. This is only required for IFF ANIMs that have two loop frames at the end of the anim. Auto detection of loop frames is not possible because it would require Hollywood to decode the whole anim first. That is why you have to tell Hollywood manually whether the anim has loop frames or not. (V5.3)

`Deinterlace:`

This tag allows you to specify how Hollywood should deinterlace interlaced anims. This can be set to either `#DEINTERLACE_DEFAULT` or `#DEINTERLACE_DOUBLE`. If set to `#DEINTERLACE_DEFAULT` (which is as the name implies also the default), Hollywood will combine two half-frames into one full frame. This mostly results in the best quality but can lead to visual artefacts when there is a lot of movement in the anim. If you use `#DEINTERLACE_DOUBLE`

instead, Hollywood will double the lines of a half-frame to get a full frame. This leads to some quality loss but can make the anim look more smooth. The best deinterlace mode to use always depends on the anim. Note that mostly you should not have to care about this tag at all because deinterlacing is actually only required for some obscure IFF ANIM formats which store interlaced frames like ANIM16i and ANIM32i. (V5.3)

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this anim. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
If this tag is set to `True`, the monochrome transparency of the anim will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based anim. If you want to load the alphachannel of an anim, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
If this tag is set to `True`, Hollywood will load the anim as a palette anim. This means that you can get and modify the anim's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette animations also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`. (V9.0)

`TransparentPen:`
If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the anim file (if the anim format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. An animation cannot have a mask and an alpha channel!

Starting with Hollywood 9.0, this function can also load vector anim formats if you have an appropriate plugin installed. Keep in mind, though, that if you load vector anim formats using `LoadAnim()`, the anim may not support all features of normal anims. See Section 17.25 [Vector animations], page 204, for more information on vector anims.

This command is also available from the preprocessor: Use `@ANIM` to preload animations!

INPUTS

`id` identifier for the animation or `Nil` for auto id selection

`filename$`
file to load

`table` optional: further options (see above) (V2.5)

RESULTS

`id` optional: identifier of the animation; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadAnim(2, "MyAnim.gif", {Transparency = #RED})
```

This loads "MyAnim.gif" as anim 2 with the color red being transparent.

17.14 LoadAnimFrame

NAME

LoadAnimFrame – load a single animation frame (V1.5)

SYNOPSIS

```
LoadAnimFrame(id, frame, anim$[, table])
```

FUNCTION

This function loads a single anim frame into the brush specified by `id`. The animation file is specified by the string `anim$`. The `frame` argument specifies which frame to load. If you want to load the last frame, set `frame` to -1.

The fourth argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
This field can be used to specify a color in RGB notation that shall be made transparent in the brush.

`LoadAlpha:`
Set this field to `True` if the alpha channel of the anim frame shall be loaded, too. Please note that not all animations have an alpha channel and that not all animation formats are capable of storing alpha channel information. This field defaults to `False`.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this anim. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using

`SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
If this tag is set to `True`, the monochrome transparency of the anim frame will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based anim. If you want to load the alphachannel of an anim, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. An anim frame can only have one transparency setting!

See Section 17.13 [LoadAnim], page 188, for details on supported anim formats.

INPUTS

`id` brush which shall contain the anim frame

`frame` frame to load (ranges from 1 to number of frames); specify -1 if you want to load the last frame

`anim$` animation file

`table` optional: further options (see above) (V5.1)

EXAMPLE

```
LoadAnimFrame(1, 5, "Animations/HugeAnim.gif")
DisplayBrushFX(1, #CENTER, #CENTER, #CROSSFADE)
```

The above code loads frame 5 of the animation "Animations/HugeAnim.gif" into brush 1 and crossfades brush 1 onto the display.

17.15 ModifyAnimFrames

NAME

ModifyAnimFrames – add or remove animation frames (V4.5)

SYNOPSIS

```
ModifyAnimFrames(id, frames[, pos])
```

FUNCTION

This function can be used to extend or shrink an existing animation. If you specify a positive value in `frames`, then the animation is extended by this number of frames. If you specify a negative value, the number of frames specified are removed from the animation.

The optional argument `pos` can be used to specify where the new frames shall be inserted or from where the frames shall be removed, respectively. If you do not specify the optional argument or set it to 0, frames are added at the end of the animation or removed from the end of the animation, respectively.

This command works only with animations buffered entirely in memory. You cannot use it for animations that are played directly from disk.

INPUTS

`id`	identifier of the animation to modify
`frames`	number of frames to insert (if value is positive) or number of frames to remove (if value is negative)
`pos`	optional: where to insert or remove frames (defaults to 0 which means insert at/remove from the end)

EXAMPLE

```
ModifyAnimFrames(1, -5, 1)
```

The code above removes the first five frames from animation number 1.

17.16 MoveAnim

NAME

MoveAnim – move an animation from a to b

SYNOPSIS

```
MoveAnim(id, xa, ya, xb, yb[, table])
```

FUNCTION

This function moves (scrolls) the animation specified by `id` softly from the location specified by `xa`,`ya` to the location specified by `xb`,`yb`.

Further configuration options are possible using the optional argument table. You can specify the move speed, special effect, and whether or not the move shall be asynchronous. See Section 21.46 [MoveBrush], page 287, for more information on the optional table argument.

Besides the table elements mentioned in the `MoveBrush()` documentation, `MoveAnim()` accepts one additional table element named `AnimSpeed`: The anim speed value defines after how many draws the frame number should be increased; therefore a higher number means a lower playback speed of the animation.

It should also be mentioned that starting with Hollywood 4.5, you can specify the new `#DEFAULTSPEED` constant in the `Speed` table argument. (see `MoveBrush()`). If you use `#DEFAULTSPEED`, Hollywood will use the playback speed as defined in the animation file.

Note that not all animations define such a speed but if they do, it should be respected because otherwise the playback looks wrong.

INPUTS

`id`	identifier of the animation to use as source
`xa`	source x position
`ya`	source y position
`xb`	destination x position
`yb`	destination y position
`table`	optional: further configuration for this move

EXAMPLE

```
MoveAnim(1, 100, 50, 0, 50, {Speed = 5, AnimSpeed = 4})
```

Moves the animation 1 from 100:50 to 0:50 with move speed 5 and anim playback speed 4.

17.17 OpenAnim

NAME

OpenAnim – open an animation (V9.0)

SYNOPSIS

```
[id] = OpenAnim(id, filename$[, table])
```

FUNCTION

This function opens the animation specified by `filename$` and assigns the identifier `id` to it. If you pass `Nil` in `id`, `OpenAnim()` will automatically choose an identifier and return it.

In contrast to `LoadAnim()`, `OpenAnim()` won't load any frames into memory. Thus, it will return control to the script quickly and won't use much memory. Using `OpenAnim()` is basically the same as calling `LoadAnim()` with the `FromDisk` table argument set to `True`.

Anim formats that are supported on all platforms are IFF ANIM, GIF ANIM, AVI (uncompressed or using Motion JPEG compression), and formats you have a plugin for. Depending on the platform Hollywood is running on, more anim formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all anim formats you have datatypes for as well. On Windows, `OpenAnim()` can also load anim formats supported by the Windows Imaging Component.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall be made transparent in the animation.

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the anim shall be loaded, too. This field defaults to `False`.

`SkipLoopFrames:`
: If you set this to `True`, Hollywood will automatically skip the last two frames of the anim. This is only required for IFF ANIMs that have two loop frames at the end of the anim. Auto detection of loop frames is not possible because it would require Hollywood to decode the whole anim first. That is why you have to tell Hollywood manually whether the anim has loop frames or not.

`Deinterlace:`
: This tag allows you to specify how Hollywood should deinterlace interlaced anims. This can be set to either `#DEINTERLACE_DEFAULT` or `#DEINTERLACE_DOUBLE`. If set to `#DEINTERLACE_DEFAULT` (which is as the name implies also the default), Hollywood will combine two half-frames into one full frame. This mostly results in the best quality but can lead to visual artefacts when there is a lot of movement in the anim. If you use `#DEINTERLACE_DOUBLE` instead, Hollywood will double the lines of a half-frame to get a full frame. This leads to some quality loss but can make the anim look smoother. The best deinterlace mode to use always depends on the anim. Note that mostly you should not have to care about this tag at all because deinterlacing is actually only required for some obscure IFF ANIM formats which store interlaced frames like ANIM16i and ANIM32i.

`Loader:`
: This tag allows you to specify one or more format loaders that should be asked to load this anim. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`Adapter:`
: This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`LoadTransparency:`
: If this tag is set to `True`, the monochrome transparency of the anim will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based anim. If you want to load the alphachannel of an anim, set the `LoadAlpha` tag to `True`. This tag defaults to `False`.

`LoadPalette:`
: If this tag is set to `True`, Hollywood will load the anim as a palette anim. This means that you can get and modify the anim's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette animations also have the advantage of requiring less

memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`.

`TransparentPen:`
If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the anim file (if the anim format supports the storage of transparent pens). This tag defaults to `#NOPEN`.

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. An animation cannot have a mask and an alpha channel!

This command is also available from the preprocessor: Use `@ANIM` to preload animations from the preprocessor but note that you have to set `FromDisk` to `True` in that case to get the same behaviour as `OpenAnim()`. If you don't set `FromDisk` to `True`, `@ANIM` will load the entire animation into memory!

Starting with Hollywood 9.0, this function can also open vector anim formats if you have an appropriate plugin installed. Keep in mind, though, that if you open vector anim formats using `OpenAnim()`, the anim may not support all features of normal anims. See Section 17.25 [Vector animations], page 204, for more information on vector anims.

To free an animation allocated by `OpenAnim()`, use the `CloseAnim()` command. See Section 17.4 [CloseAnim], page 183, for details.

INPUTS

`id` identifier for the animation or `Nil` for auto id selection

`filename$`
file to load

`table` optional: further options (see above)

RESULTS

`id` optional: identifier of the animation; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenAnim(2, "MyAnim.gif", {Transparency = #RED})
```

This opens "MyAnim.gif" as anim 2 with the color red being transparent.

17.18 PlayAnim

NAME

PlayAnim – play an animation

SYNOPSIS

```
[handle] = PlayAnim(id[, x, y, table])
```

FUNCTION

This function starts playing a preloaded animation specified by `id`. Optionally you can specify the `x` and `y` coordinates on the screen where the animation should be displayed.

If layers are enabled, this command will add a new layer of the type `#ANIM` to the layer stack.

As of Hollywood 4.0, `PlayAnim()` accepts an additional optional table argument which can be used to configure several further playback options:

`Speed:` Defines the playback speed for the animation. The higher the number, the slower the playback speed. You can also specify a constant for the speed argument (`#SLOWSPEED`, `#NORMALSPEED` or `#FASTSPEED`). New in Hollywood 4.5: You can specify the new `#DEFAULTSPEED` constant here. If you use `#DEFAULTSPEED`, Hollywood will use the speed as defined in the animation file. Note that not all animations define such a speed but if they do, it should be respected because otherwise the playback looks wrong.

`Times:` Specifies how many times the animation shall be played (defaults to 1 which means play anim just once). If you want the animation to loop infinitely, specify 0 here.

`Async:` You can use this field to create an asynchronous draw object for this playback. If you pass `True` here `PlayAnim()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

INPUTS

`id` identifier of the animation to play

`x` optional: x playback position (defaults to 0)

`y` optional: y playback position (defaults to 0)

`table` optional: further configuration for the anim playback

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
PlayAnim(1)
```

Play the animation 1.

17.19 PlayAnimDisk

NAME

PlayAnimDisk – play an animation directly from disk (V1.5)

SYNOPSIS

```
PlayAnimDisk(anim$[, x, y, frameskip, speed, transcolor, times])
```

FUNCTION

This function plays the animation specified by `anim$` directly from disk. This is useful if you want to display large animations which do not fit into your memory. Of course, this function is a lot of slower than `PlayAnim()` which plays an animation from memory.

New in Hollywood 1.9 is the optional `times` argument which allows you to specify how many times the animation shall be played. This defaults to 1 which means that the animation is only played once. If you want to loop the animation indefinitely, specify 0.

Important note: Since Hollywood 2.5 it is better to use `LoadAnim()` with `FromDisk` set to `True` for disk anims. This gives you more flexibility because you can also use the other commands from the anim library (like `ScaleAnim()` etc.) and you can also access the anims as layers using `NextFrame()` - all of which is not possible with `PlayAnimDisk()`.

See Section 17.13 [LoadAnim], page 188, for supported animation formats.

INPUTS

`anim$` animation file to play

`x` optional: x-position for the animation on the display (defaults to 0)

`y` optional: y-position for the animation on the display (defaults to 0)

`frameskip` optional: frame skip (defaults to 0)

`speed` optional: delay after each frame in ticks (defaults to 0)

`transcolor` optional: transparent color for the animation (defaults to `#NOTRANSPARENCY`)

`times` optional: specifies how many times the animation should be played (defaults to 1) (V1.9)

EXAMPLE

```
PlayAnimDisk("Animations/LargeAnim.gif", 0, 0, 3)
```

The above code plays the animation "Animations/LargeAnim.gif" and skips 3 frames per run.

17.20 SaveAnim

NAME

SaveAnim – save animation to disk (V4.5)

SYNOPSIS

```
SaveAnim(id, file$[, format, table])
```

FUNCTION

This function saves the animation specified by `id` to the file specified by `file$` in animation format specified by `format`. This can either be one of the following constants or an anim saver provided by a plugin:

`#ANMFMT_GIF:`
: GIF format. Because GIF anims are always palette-based, RGB graphics have to be quantized before they can be exported as GIF. You can use the `Colors` and `Dither` tags (see below) to specify the number of palette entries to allocate for the anim and whether or not dithering shall be applied. When using `#ANMFMT_GIF` with a palette anim, no quantizing will be done. `#ANMFMT_GIF` also supports palette anims with a transparent pen. `#ANMFMT_GIF` is the default format used by `SaveAnim()`.

`#ANMFMT_MJPEG:`
: AVI with Motion JPEG compression. This is a lossy anim format so you can set the `Quality` tag (see below) to control the level of compression that should be used.

`#ANMFMT_IFF:`
: IFF anim. Hollywood will use mode 5 compression (the most common compression mode) for IFF anims. Because IFF anims are always palette-based, RGB graphics have to be quantized before they can be exported as IFF. You can use the `Colors` and `Dither` tags (see below) to specify the number of palette entries to allocate for the anim and whether or not dithering shall be applied. When using `#ANMFMT_IFF` with a palette anim, no quantizing will be done. `#ANMFMT_IFF` also supports palette anims with a transparent pen. (V9.0)

The optional table argument allows you to configure further parameters:

`Dither:` Set to `True` to enable dithering. This field is only handled when the destination format is palette-based and the source data is RGB. GIF anims and IFF anims always use a color palette. Defaults to `False` which means no dithering.

`Depth:` Specifies the desired anim depth. This is only handled when the format is palette-based and the source data is in RGB format. Valid values are between 1 (= 2 colors) and 8 (= 256 colors). Defaults to 8. (V9.0)

`Colors:` This is an alternative to the `Depth` tag. Instead of a bit depth, you can pass how many colors the anim shall use here. Again, this is only handled when the format is palette-based and the source data is in RGB format. Valid values are between 1 and 256. Defaults to 256.

`Optimize:`
: Specifies whether or not Hollywood shall try to optimize the animation. Optimized saving is slower but usually leads to smaller animations. Defaults to `True`.

`Quality:` Here you can specify a value between 0 and 100 indicating the compression quality for lossy compression formats. A value of 100 means best quality, 0 means worst quality. This is only available for anim formats that support lossy compression. Defaults to 90 which means pretty good quality.

`FPS:` Video formats like AVI do not support an individual delay value for each frame but require a global value indicating how many frames per second

shall be displayed. This field allows you to set the FPS. This is only handled for video file formats. Defaults to 25 frames per second.

`FillColor:`
When saving an RGB anim that has transparent pixels, you can specify an RGB color that should be written to all those transparent pixels here. This is probably of not much practical use. Defaults to `#NOCOLOR` which means that transparent pixels will be left as they are. (V9.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Here is a table that shows an overview which table elements can be used with the different animation formats:

	GIF	IFF	MJPEG AVI
Dither	Yes	Yes	No
Colors	Yes	Yes	No
Optimize	Yes	No	No
Depth	Yes	Yes	No
FillColor	Yes	Yes	Yes
Quality	No	No	Yes
FPS	No	No	Yes

`SaveAnim()` can be used both with animations buffered completely in memory or with disk-based animations.

If you want to save an animation from individual frames (e.g. a series of brushes), you can do so by using `BeginAnimStream()`, `WriteAnimFrame()` and `FinishAnimStream()`.

INPUTS

`id`	animation which shall be saved
`file$`	destination file
`format`	optional: which anim format to use (defaults to `#ANMFMT_GIF`)
`table`	optional: further arguments for save operation; see above

EXAMPLE

```
SaveAnim(1, "my_anim.gif", #ANMFMT_GIF, {Colors = 64, Dither = True})
```

The code above saves anim 1 as "my_anim.gif" in 64 colors with dithering enabled.

17.21 ScaleAnim

NAME

ScaleAnim – scale an animation

SYNOPSIS

```
ScaleAnim(id, width, height[, smooth])
```

FUNCTION

This function scales the animation specified by `id` to the desired `width` and `height`. Please note that scaling an animation on a 68k processor can take quite some time. Optionally, you can choose to have the scaled graphics interpolated by passing `True` in the `smooth` argument. The graphics will then be scaled using anti-alias.

New in V2.0: You can pass `#KEEPASPRAT` as either `width` or `height`. Hollywood will calculate the size then automatically by taking the aspect-ratio of the anim into account.

Starting with Hollywood 2.0, `width` and `height` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id`	identifier of the animation scale
`width`	desired new width for the animation
`height`	desired new height for the animation
`smooth`	optional: whether or not anti-aliased scaling shall be used (V2.5)

EXAMPLE

```
ScaleAnim(1, 320, 240)
```

The above code scales animation 1 to a format of 320x240.

17.22 SelectAnim

NAME

SelectAnim – select animation frame as output device (V4.5)

SYNOPSIS

```
SelectAnim(id, frame[, mode, combomode])
```

FUNCTION

This function selects the specified animation frame as the current output device. This means that all graphics data that is rendered by Hollywood will be written to this animation frame. You have to specify an animation identifier as well as the single frame that shall be used as output device.

The optional `mode` argument defaults to `#SELMODE_NORMAL` which means that only the color channels of the anim will be altered when you draw to it. The transparency channel of the anim (can be either a mask or an alpha channel) will never be altered. You can change this behaviour by using `#SELMODE_COMBO` in the optional `mode` argument. If you use this mode, every Hollywood graphics command that is called after `SelectAnim()`

will draw into the color and transparency channel of the anim. If the anim does not have a transparency channel, `#SELMODE_COMBO` behaves the same as `#SELMODE_NORMAL`.

Starting with Hollywood 5.0 you can use the optional `combomode` argument to specify how `#SELMODE_COMBO` should behave. If `combomode` is set to 0, the color and transparency information of all pixels in the source image are copied to the destination image in any case - even if the pixels are invisible. This is the default behaviour. If `combomode` is set to 1, only the visible pixels are copied to the destination image. This means that if the alpha value of a pixel in the source image is 0, i.e. invisible, it will not be copied to the destination image. Hollywood 6.0 introduces the new combomode 2. If you pass 2 in `combomode`, Hollywood will blend color channels and alpha channel of the source image into the destination image's color and alpha channels. When you draw the destination image later, it will look as if the two images had been drawn on top of each other consecutively. Please note that the `combomode` argument is only supported together with `#SELMODE_COMBO`. It doesn't have any effect when used with the other modes. Please note that the `combomode` argument is only supported together with `#SELMODE_COMBO`. It doesn't have any effect when used with the other modes.

An alternative way to draw into the transparency channels of an anim is to do this separately using `SelectMask()` or `SelectAlphaChannel()`. These two commands, however, will write data to the transparency channel only. They will not touch the color channel. So if you want both channels, color and transparency, to be affected, you need to use `SelectAnim()` with `mode` set to `#SELMODE_COMBO`.

When you are finished with rendering to your animation and want to use your display as output device again, just call `EndSelect()`.

Note that you must not call any commands which modify your animation while it is selected as the output device. Specifically, you must not call `ScaleAnim()`, `FreeAnim()`, or `ModifyAnimFrames()`.

Only commands that output graphics directly can be used after `SelectAnim()`. You may not call animated functions like `MoveAnim()` or `DisplayBrushFX()` while `SelectAnim()` is active.

This command works only with animations buffered entirely in memory. You cannot use it for animations that are played directly from disk.

INPUTS

`id` animation which shall be used as output device

`frame` frame of the animation that graphics shall be drawn to

`mode` optional: rendering mode to use (see above); this can be either `#SELMODE_NORMAL` or `#SELMODE_COMBO`; defaults to `#SELMODE_NORMAL`

`combomode` optional: mode to use when `#SELMODE_COMBO` is active (see above); defaults to 0 (V5.0)

EXAMPLE

```
SelectAnim(1, 5)
SetFillStyle(#FILLCOLOR)
Box(0, 0, 320, 256, #RED)
```

```
EndSelect()
```

The code above draws a 320x256 rectangle to frame number 5 of animation 1.

17.23 SetAnimFrameDelay

NAME

SetAnimFrameDelay – set delay after anim frame (V4.5)

SYNOPSIS

```
SetAnimFrameDelay(id, frame, delay)
```

FUNCTION

This function can be used to set the delay of a frame of an existing animation. Pass the identifier of the animation as well as the frame you want to modify. The delay time has to be passed in milliseconds.

INPUTS

`id`	identifier of the animation to use
`frame`	frame you want to delay
`delay`	new delay time in milliseconds

EXAMPLE

```
SetAnimFrameDelay(1, 5, 1500)
```

Sets the delay of frame 5 of animation 1 to 1.5 seconds.

17.24 StopAnim

NAME

StopAnim – stop a playing animation (V1.0 only)

SYNOPSIS

```
StopAnim(id)
```

FUNCTION

This function was removed in Hollywood 1.5. It can no longer be used.

This function stops the animation specified by `id`. If the animation specified by `id` is played back multiple times, the animation that was last started will be stopped by Hollywood.

INPUTS

`id`	identifier of the animation to stop

17.25 Vector animations

Hollywood's anim library also supports a special type of anim: a vector anim. To find out if `OpenAnim()`, `LoadAnim()`, or `@ANIM` have loaded a vector anim, you need to query the `#ATTRTYPE` attribute for the anim using `GetAttribute()`.

The advantage of a vector anim in contrast to traditional raster anim formats like GIF ANIM and IFF ANIM is that you can scale and/or transform it without any quality losses. For example, the `ScaleAnim()` command will produce high-quality anim frames when used with vector anims. Also, when layers and the layer scaling engine are enabled, vector anim layers will be automatically scaled and transformed without any quality losses. Therefore, if you only use vector anims and TrueType text in your script, it can be scaled to any resolution and will still appear perfectly crisp.

The disadvantage of vector anims is that they are not supported by all Hollywood functions. For example, you can't draw to them using `SelectAnim()`.

17.26 WaitAnimEnd

NAME

WaitAnimEnd – halt until animation has finished playing (V1.0 only)

SYNOPSIS

```
WaitAnimEnd(id)
```

FUNCTION

Attention: This command was removed in Hollywood 1.5.

This function halts the program flow until the animation specified by `id` has finished playing. After that, the execution of your script is continued. If you need to do something while your animation is playing, use the `IsAnimPlaying()` command in connection with a loop.

INPUTS

`id` identifier of an animation that is currently playing

EXAMPLE

```
PlayAnim(1,#PLAYONCE)
WaitAnimEnd(1)
```

The above code plays animation 1 and waits for it to finish.

17.27 WriteAnimFrame

NAME

WriteAnimFrame – append frame to sequential anim (V4.5)

SYNOPSIS

```
WriteAnimFrame(id, brush_id[, table])
```

FUNCTION

This function can be used to append a single new frame to a sequential animation object created with `BeginAnimStream()`. The frame that shall be appended to the animation

must be provided as a brush. Ideally, the brush's size should match the dimensions specified in `BeginAnimStream()` but this function will do automatic padding if the sizes do not match.

The optional table argument allows you to configure further parameters:

`X, Y:` These two allow you to configure the position at which the brush shall be copied in the frame. This is useful when adding a frame that is smaller than the anim bounding box. You could for example center this frame in the anim then. Defaults to 0,0 which means top left corner.

`Delay:` Set this field if you want to attach a time delay to this frame. The time must be specified in milliseconds. Not all anim formats support frame delays. See the table below. Defaults to 0 which means no delay.

`Dither:` Set to `True` to enable dithering. This field is only handled when the format is palette-based and the source data is in RGB format. GIF and IFF anims always use a color palette. Defaults to `False` which means no dithering.

`Depth:` Specifies the desired frame depth. This is only handled when the format is palette-based and the source data is in RGB format. Valid values are between 1 (= 2 colors) and 8 (= 256 colors). Defaults to 8. (V9.0)

`Colors:` This is an alternative to the `Depth` tag. Instead of a bit depth, you can pass how many colors the frame shall use here. Again, this is only handled when the format is palette-based and the source data is in RGB format. Valid values are between 1 and 256. Defaults to 256.

`Optimize:`
Specifies whether or not Hollywood shall try to optimize this frame. Optimized saving is slower but usually leads to smaller animations. Defaults to `True`.

`FillColor:`
When saving an RGB frame that has transparent pixels, you can specify an RGB color that should be written to all those transparent pixels here. This is probably of not much practical use. Defaults to `#NOCOLOR` which means that transparent pixels will be left as they are. (V9.0)

Here is a table that shows an overview which table elements can be used with the different animation formats:

	GIF	**IFF**	**MJPEG AVI**
X, Y	Yes	Yes	Yes
Delay	Yes	Yes	No
Dither	Yes	Yes	No
Colors	Yes	Yes	No
Optimize	Yes	No	No
Depth	Yes	Yes	No
FillColor	Yes	Yes	Yes

INPUTS

`id`	identifier of the animation object to append to; must be obtained using `BeginAnimStream()`
`brush_id`	identifier of brush to append to animation
`table`	optional: further arguments for save operation; see above

EXAMPLE

See Section 17.3 [BeginAnimStream], page 180.

18 Application library

18.1 APPAUTHOR

NAME
APPAUTHOR – declare application author (V2.0)

SYNOPSIS
```
@APPAUTHOR author$
```

FUNCTION
This preprocessor command simply allows you to include a string containing the script's author into the application. This command does not have any actual function. It just takes the specified string and saves it in the applet/executable.

INPUTS
`author$` script author

EXAMPLE
See Section 18.8 [APPVERSION], page 212.

18.2 APPCOPYRIGHT

NAME
APPCOPYRIGHT – declare application copyright (V2.0)

SYNOPSIS
```
@APPCOPYRIGHT copyright$
```

FUNCTION
This preprocessor command simply allows you to include a string containing the script's copyright into the application. This command does not have any actual function. It just takes the specified string and saves it in the applet/executable.

INPUTS
`copyright$`
script copyright

EXAMPLE
See Section 18.8 [APPVERSION], page 212.

18.3 APPDESCRIPTION

NAME
APPDESCRIPTION – declare application description (V2.0)

SYNOPSIS
```
@APPDESCRIPTION desc$
```

FUNCTION

This preprocessor command simply allows you to include a string containing the script's description into the application. This command does not have any actual function. It just takes the specified string and saves it in the applet/executable.

Under AmigaOS, the description specified here is used as the description of the commodity that is added to Exchange.

INPUTS

`desc$` script description

EXAMPLE

See Section 18.8 [APPVERSION], page 212.

18.4 APPENTRY

NAME

APPENTRY – declare application entry script (V10.0)

SYNOPSIS

```
@APPENTRY file$
```

FUNCTION

This preprocessor command allows you to define the entry script for a project. This is only useful if you have a project that consists of multiple scripts. In such a project you typically have a main script which uses `@INCLUDE` to include several other helper scripts. By adding `@APPENTRY` to your helper scripts you can tell Hollywood the name of the main script. Whenever you run one of the helper scripts and Hollywood encounters the `@APPENTRY` definition, it will run the main script instead of the helper script.

For example, let's suppose your project consists of the following three scripts:

```
main.hws
engine.hws
utils.hws
```

You could then place the following line at the top of the helper scripts, i.e. at the top of `engine.hws` and `utils.hws`:

```
@APPENTRY "main.hws"
```

With that definition Hollywood will run `main.hws` whenever you start `engine.hws` or `utils.hws` because `main.hws` has been set as the entry script. This can be very convenient, e.g. when editing helper scripts in the Hollywood IDE. If you use `@APPENTRY` you can just click on "Run" and the IDE will run the main script instead.

INPUTS

`file$` entry script file

18.5 APPICON

NAME

APPICON – declare application icon (V4.7)

SYNOPSIS

```
@APPICON table
@APPICON f$     (V8.0)
```

FUNCTION

This preprocessor command can be used to specify an icon for your application. On Windows, macOS, and Linux this icon will appear in the window's border and in several elements of the window manager like the task bar. On AmigaOS 4, the icon will appear in AmiDock if your script runs as a registered AmigaOS 4 application. The icon you specify here will also be linked into the applets and executables you compile with Hollywood. By default, executables compiled by Hollywood will always use the standard Hollywood icon (the clapperboard). If you prefer to use your own icon instead, use this preprocessor command.

Starting with Hollywood 8.0 there are two different ways of using this preprocessor command: You can either specify individual images for the different icon sizes in a table (see below) or you can simply pass a Hollywood icon that has been created using the `SaveIcon()` function to this preprocessor command. In that case, the icon has to be passed in the `f$` parameter. Specifying a single icon instead of a whole table results in code that is more readable but of course it requires you to generate the icon using Hollywood's `SaveIcon()` command first.

If you want to pass individual images for the different icon sizes, you have to pass a table in the `table` argument to this preprocessor command. The table can contain one or more of the following tags:

`Ic16x16:` Custom icon in the resolution of 16x16 pixels.

`Ic16x16s:`
Selected state icon for size 16x16. (V6.0)

`Ic24x24:` Custom icon in the resolution of 24x24 pixels.

`Ic24x24s:`
Selected state icon for size 24x24. (V6.0)

`Ic32x32:` Custom icon in the resolution of 32x32 pixels.

`Ic32x32s:`
Selected state icon for size 32x32. (V6.0)

`Ic48x48:` Custom icon in the resolution of 48x48 pixels.

`Ic48x48s:`
Selected state icon for size 48x48. (V6.0)

`Ic64x64:` Custom icon in the resolution of 64x64 pixels.

`Ic64x64s:`
Selected state icon for size 64x64. (V6.0)

`Ic96x96:` Custom icon in the resolution of 96x96 pixels.

`Ic96x96s:`
Selected state icon for size 96x96. (V6.0)

`Ic128x128:`
Custom icon in the resolution of 128x128 pixels.

`Ic128x128s:`
Selected state icon for size 128x128. (V6.0)

`Ic256x256:`
Custom icon in the resolution of 256x256 pixels.

`Ic256x256s:`
Selected state icon for size 256x256. (V6.0)

`Ic512x512:`
Custom icon in the resolution of 512x512 pixels.

`Ic512x512s:`
Selected state icon for size 512x512. (V6.0)

`Ic1024x1024:`
Custom icon in the resolution of 1024x1024 pixels. (V7.0)

`Ic1024x1024s:`
Selected state icon for size 1024x1024. (V7.0)

`DefaultIcon:`
This tag lets you specify which icon should be the default icon. You need to pass a string here that describes an icon size from the sizes listed above, e.g. `"64x64"` designates the icon specified in the `Ic64x64` tag as the default icon. The default icon is the icon that Hollywood will show in AmiDock on AmigaOS4 if the `RegisterApplication` tag in `@OPTIONS` has been set to `True`. Thus, currently, the `DefaultIcon` tag only has an effect on AmigaOS 4. (V6.0)

The reason why this preprocessor command does not simply accept just a single 512x512 icon and then scales it down to all other resolutions is that very small icons like 16x16 or 24x24 do not really look very good when scaled down from a larger icon. They look much better when they are handcrafted for each size. That is why this preprocessor command accepts so many different tags.

Please note that not all sizes are currently supported on all platforms but you should make sure to provide icons for all these sizes. If you leave a size out, Hollywood might fall back to its default icon (clapperboard) for the size. So if you intend to use your own icons, make sure that you always provide it in all sizes listed above.

The image file that is required as a parameter by the tags listed above should be a PNG image with alpha channel. Images without alpha channel are supported as well, but this is not recommended because it doesn't look too good.

Please note that it is mandatory to use the `DefaultIcon` tag on AmigaOS 4 to indicate which icon size should be shown in AmiDock. If you do not pass the `DefaultIcon`

tag, Hollywood will display its default icon in AmiDock (the clapperboard). If you want to show an icon in AmiDock that uses a custom size not listed above, you can use the `DockyBrush` tag that is supported by the `@OPTIONS` preprocessor command. See Section 52.25 [OPTIONS], page 1088, for details. Please note that Hollywood supports two different docky types on AmigaOS 4: Standard dockies and app dockies. See Section 16.1 [AmiDock information], page 165, for more information on the difference between the two.

Starting with Hollywood 6.0 you can specify two images for every icon size because on AmigaOS and compatibles icons usually contain two different states whereas on all other platforms icons are just single static images. If you only target non-Amiga systems, you do not have to provide icons for the selected state because they won't be used anyway.

Alternatively, you can also use one of the `-iconXXX` console arguments instead of `@APPICON`.

Note that this preprocessor command currently does not have any effect on Amiga systems. If you would like to change the icon that Hollywood displays when it is minimized on Workbench, use the `SetWBIcon()` command.

INPUTS

`f$` name of a Hollywood icon OR

`table` table containing one or more of the supported tags (see above)

EXAMPLE

```
@APPICON "icon.png"
```

The code above sets the icon "icon.png" as the icon for the application. This icon must have been created using `SaveIcon()`.

```
@APPICON {Ic16x16 = "my16x16icon.png",
Ic24x24 = "my24x24icon.png",
Ic32x32 = "my32x32icon.png",
Ic48x48 = "my48x48icon.png",
Ic64x64 = "my64x64icon.png",
Ic96x96 = "my96x96icon.png",
Ic128x128 = "my128x128icon.png",
Ic256x256 = "my256x256icon.png",
Ic512x512 = "my512x512icon.png",
Ic1024x1024 = "my1024x1024icon.png"}
```

The code above will replace all inbuilt icons with custom ones provided in the specified png images.

18.6 APPIDENTIFIER

NAME

APPIDENTIFIER – declare application identifier (V6.1)

SYNOPSIS

```
@APPIDENTIFIER id$
```

FUNCTION

Specify a unique identifier for your application in reverse DNS notation. Some Hollywood commands like `LoadPrefs()` and `SavePrefs()` require you to provide a unique identifier for your application in reverse DNS notation, e.g. `com.airsoftsoftwair.hollywood`. This can be done using the `@APPIDENTIFIER` preprocessor command.

Also, when compiling application bundles for macOS, the identifier specified in `@APPIDENTIFIER` will automatically be written to the `Info.plist` file of the app bundle.

INPUTS

`id$` application identifier in reverse DNS notation

EXAMPLE

See Section 18.8 [APPVERSION], page 212.

18.7 APPTITLE

NAME

APPTITLE – declare application title (V2.0)

SYNOPSIS

```
@APPTITLE title$
```

FUNCTION

This preprocessor command simply allows you to include a string containing the script's title into the application. This command does not have any actual function. It just takes the specified string and saves it in the applet/executable.

Under AmigaOS, the title specified here is used as the title of the commodity that is added to Exchange.

INPUTS

`title$` script title

EXAMPLE

See Section 18.8 [APPVERSION], page 212.

18.8 APPVERSION

NAME

APPVERSION – declare application version (V2.0)

SYNOPSIS

```
@APPVERSION version$
```

FUNCTION

This preprocessor command simply allows you to include a string containing the script's version into the application. This command does not have any actual function. It just takes the specified string and saves it in the applet/executable.

Under AmigaOS, the version specified here is used as the information text of the commodity that is added to Exchange.

INPUTS

`version$` script version

EXAMPLE

```
@APPTITLE "Run-away Randy"
@APPAUTHOR "Andreas Falkenhahn"
@APPCOPYRIGHT "(C) Copyright 2002-2005 by Airsoft Softwair."
@APPVERSION "$VER: Run-away Randy 1.0 (23-Dec-05)"
@APPDESCRIPTION "A really cool jump'n'run game done with Hollywood."
@APPIDENTIFIER "com.airsoftsoftwair.runawayrandy"
```

This is what a complete application information sector looks like. Hollywood will save all this information in the applet/executable. When the user enter "Version Run-away-Randy.exe", the version string "$VER: Run-away Randy 1.0 (23-Dec-05)" will be returned.

18.9 DeletePrefs

NAME

DeletePrefs – delete user preferences (V6.1)

SYNOPSIS

```
DeletePrefs()
```

FUNCTION

This function can be used to delete user preferences that have been saved using `SavePrefs()`. You can use this function to implement a restoration of the default settings. By calling `DeletePrefs()` all user preferences will be physically erased from the user's hard drive.

Note that this function will only work if you have specified a unique identifier for your application by using the `@APPIDENTIFIER` preprocessor command. See Section 18.6 [APPIDENTIFIER], page 211, for details.

INPUTS

none

18.10 GetApplicationInfo

NAME

GetApplicationInfo – get information about the application (V6.0)

SYNOPSIS

```
t = GetApplicationInfo()
```

FUNCTION

This function can be used to obtain the information specified in the @APPXXX preprocessor commands. `GetApplicationInfo()` returns a table that contains the following fields:

`Title:` This is set to the value of `@APPTITLE`. See Section 18.7 [APPTITLE], page 212, for details.

`Version:` This is set to the value of `@APPVERSION`. See Section 18.8 [APPVERSION], page 212, for details.

`Author:` This is set to the value of `@APPAUTHOR`. See Section 18.1 [APPAUTHOR], page 207, for details.

`Copyright:`
This is set to the value of `@APPCOPYRIGHT`. See Section 18.2 [APPCOPYRIGHT], page 207, for details.

`Description:`
This is set to the value of `@APPDESCRIPTION`. See Section 18.3 [APPDESCRIPTION], page 207, for details.

`Identifier:`
This is set to the value of `@APPIDENTIFIER`. See Section 18.6 [APPIDENTIFIER], page 211, for details. (V6.1)

INPUTS

none

RESULTS

`t` a table containing the fields described above

18.11 GetCommandLine

NAME

GetCommandLine – get the arguments from the command line (V3.0)

SYNOPSIS

```
t, n, console = GetCommandLine()
```

FUNCTION

This function allows you to obtain the arguments that your script has been started with. This makes it possible for your script to take arguments from the user and then react on it accordingly. All arguments that are not recognized by Hollywood will be forwarded to your script. Please note that arguments must be prefixed by a dash character (-). A parameter may also follow after each argument.

`GetCommandLine()` returns three values: The second return value is a number which simply specifies how many arguments have been passed to your script from the command line. The first return value is a table which contains all arguments and their parameters. The table is an array of "n"-tables with the items `arg` and `param`. `arg` will be initialized to the argument excluding the dash character. `param` will receive the parameter for that

argument if there is one, else it will be receive an empty string (""). The third return value is new in Hollywood 4.7 and indicates whether or not Hollywood was started from a console. In that case, the third return value will be `True`, else it will be `False`.

Under AmigaOS this function will also take the tooltypes of the script's or applet's icon into account if the program was started from Workbench.

Under macOS this function will look into the `CFBundleExecutableArgs` dictionary entry in the application bundle's Info.plist if the program was started from Finder.

INPUTS

none

RESULTS

`t`	table containing all arguments and their parameters
`n`	number of arguments that have been passed to the program
`console`	`True` if program was started from a console, otherwise `False` (V4.7)

EXAMPLE

```
args, count = GetCommandLine()
NPrint("Number of arguments:", count)
For Local k = 0 to count - 1
   NPrint("Arg #", k, ":", args[k].arg, "Param:", args[k].param)
Next
```

The code above gets all arguments from the command line and prints them to the screen.

18.12 GetFileArgument

NAME

GetFileArgument – get file argument passed to a compiled script (V5.0)

SYNOPSIS

```
f$ = GetFileArgument([path])
```

FUNCTION

`GetFileArgument()` allows you to retrieve the file argument that was passed to a compiled Hollywood script. Thus, this function will only work correctly when used by executables that have been compiled with Hollywood because in script mode the file argument is obviously always the Hollywood script itself. `GetFileArgument()` extends the `GetCommandLine()` function because the latter only allows you to retrieve option arguments, not the file argument itself.

This function is especially useful when writing tools that should be able to act as viewers for certain file formats with Hollywood. By using `GetFileArgument()` you could create programs which can be used as a default tool for certain file formats.

Starting with Hollywood 9.0, there is an optional argument named `path`. If this is set to `True`, `GetFileArgument()` will return a fully qualified path to the file instead of just the file name. For compatibility reasons, this argument defaults to `False`.

INPUTS

`path` optional: set this to `True` to get a fully qualified to the file (defaults to `False`) (V9.0)

RESULTS

`f$` the file argument passed to the compiled Hollywood script or an empty string if Hollywood is running in interpreter mode

18.13 GetProgramInfo

NAME

GetProgramInfo – get information about the current program (V3.0)

SYNOPSIS

```
type, name$[, hw$] = GetProgramInfo()
```

FUNCTION

This function can be used to obtain some information about the currently running Hollywood program. `GetProgramInfo()` will return two values: The first return value specifies the program type currently running inside Hollywood. This variable can be `#PRGTYPE_SCRIPT` for Hollywood scripts, `#PRGTYPE_APPLET` for Hollywood applets, or `#PRGTYPE_PROGRAM` for compiled executables. The second return value is a string which contains the file name of the currently running program. If the currently running program is an applet or a script, its name will be returned in `name$`. If the currently running program is a compiled executable, `name$` will receive the file name of this executable.

If the currently running program is an applet or a script, there will be a third return value `hw$`. This return value will contain the file name of the Hollywood interpreter used to run this applet or script.

INPUTS

none

RESULTS

`type` type of the currently running program (`#PRGTYPE_SCRIPT`, `#PRGTYPE_APPLET` or `#PRGTYPE_PROGRAM`)

`name$` file name of the currently running program

`hw$` optional: this value is only returned if the currently running program is a script or an applet; it specifies the path to the Hollywood interpreter used to run this script or applet

18.14 GetRawArguments

NAME

GetRawArguments – get all arguments passed to program (V10.0)

SYNOPSIS

```
args = GetRawArguments()
```

FUNCTION

This function can be used to get all arguments passed to your program or script either via the console or via the host system's desktop manager. The arguments will be returned in the `args` table and won't be formatted in any way. They will be returned in their raw form as they were passed to your program or script. This means that the returned arguments may also include special arguments handled by Hollywood, e.g. `"-window"` or `"-quiet"`.

Note that the first table entry will always contain the name of the program but this will not necessarily include a qualified path but just the name of the program as it was specified by the caller.

`GetRawArguments()` can be useful if your script should be capable of handling multiple file arguments. `GetFileArgument()` only allows you to get the very first file passed to your program and `GetCommandLine()` expects arguments to be formatted in a certain way which makes it impossible to use it to get al file arguments. `GetRawArguments()` allows you to get all file arguments because arguments are never formatted in any way but they are simply forwarded to your script without any modification by Hollywood.

INPUTS

none

RESULTS

`args`	all arguments in a table; the first entry in that table will contain the name of the program (not necessarily with its path)

EXAMPLE

```
t = GetRawArguments()
DebugPrint("Program:", t[0])
For Local k = 1 To ListItems(t) - 1
   DebugPrint("Argument", k .. ":", t[k])
Next
```

The code above gets all arguments passed to the script and prints them.

18.15 LoadPrefs

NAME

LoadPrefs – load user preferences (V6.1)

SYNOPSIS

```
LoadPrefs(prefs[, t])
```

FUNCTION

This function loads user preferences that have been stored using the `SavePrefs()` command into the table specified by `prefs`. Prior to calling `LoadPrefs()`, the `prefs` table should have been initialized to the default preferences. `LoadPrefs()` will then overwrite all table items for which custom user preferences exist and keep the default values of the items for which there are no user preferences.

Note that this function will only work if you have specified a unique identifier for your application by using the `@APPIDENTIFIER` preprocessor command. See Section 18.6 [APPIDENTIFIER], page 211, for details.

Starting with Hollywood 9.0, this function accepts a new optional table argument that can be used to specify further options.

The following tags are currently recognized in the optional table argument:

`Adapter:` This table tag can be used to specify the deserializer that should be used to import the preferences. This can be the name of an external deserializer plugin (e.g. `xml`) or it can be one of the following inbuilt deserializers:

`Default:` Use Hollywood's default deserializer. This will deserialize data from the JSON format to a Hollywood table.

`Inbuilt:` Use Hollywood's legacy deserializer. Using this deserializer is not recommended any longer as the data is in a proprietary, non-human-readable format. Using JSON is a much better choice.

If the `Adapter` tag isn't specified, it will default to the default set using `SetDefaultAdapter()`.

`UserTags:`

This tag can be used to specify additional data that should be passed to serializer plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`prefs` table to load user preferences into

`t` optional: table containing further options (see above) (V9.0)

EXAMPLE

```
@APPIDENTIFIER "com.airsoftsoftwair.example"
prf = {lastfile$ = "Unnamed", lastxpos = 0, lastypos = 0}
LoadPrefs(prf)
```

This initializes the table `prf` to default values and then uses `LoadPrefs()` to read user preferences saved using `SavePrefs()` into the table. See Section 18.16 [SavePrefs], page 218, for details.

18.16 SavePrefs

NAME

SavePrefs – save user preferences (V6.1)

SYNOPSIS

```
SavePrefs(prefs[, t])
```

FUNCTION

This function saves the table `prefs`, containing user preferences for your application, to an external file. You can then load these preferences back into your program the next

time your program is started by using the `LoadPrefs()` function. The actual location where the preferences file will be stored is platform-dependent.

Note that this function will only work if you have specified a unique identifier for your application by using the `@APPIDENTIFIER` preprocessor command. See Section 18.6 [APPIDENTIFIER], page 211, for details.

Starting with Hollywood 9.0, this function accepts a new optional table argument that can be used to specify further options.

The following tags are currently recognized in the optional table argument:

`Adapter:` This table tag can be used to specify the serializer that should be used to export the preferences. This can be the name of an external serializer plugin (e.g. `xml`) or it can be one of the following inbuilt serializers:

- `Default:` Use Hollywood's default serializer. This will serialize the preferences to the JSON format.
- `Inbuilt:` Use Hollywood's legacy serializer. This will serialize the table into a custom, proprietary format.

If the `Adapter` tag isn't specified, it will default to the default set using `SetDefaultAdapter()`.

`UserTags:`

This tag can be used to specify additional data that should be passed to serializer plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`prefs` table containing user preferences to save

`t` optional: table containing further options (see above) (V9.0)

EXAMPLE

```
@APPIDENTIFIER "com.airsoftsoftwair.example"
prf = {lastfile$ = "test.txt", lastxpos = 100, lastypos = 200}
SavePrefs(prf)
```

This saves the table `prf` to a platform-dependent location. At the next program start you can load the preferences by using the `LoadPrefs()` function. See Section 18.15 [LoadPrefs], page 217, for details.

19 Asynchronous operation library

19.1 AsyncDrawFrame

NAME

AsyncDrawFrame – draw the next frame of an asynchronous object (V4.0)

SYNOPSIS

```
finish = AsyncDrawFrame(id[, frame])
```

FUNCTION

This function can be used to draw the next frame of an asynchronous draw object created by one of functions from the transition effect or move object libraries. `AsyncDrawFrame()` will return `False` if there are frames left in its queue. Thus, you should normally call this function until it returns `True`, which means that you have now drawn all frames that were in the asynchronous draw object's queue.

When this function returns `True`, it automatically deletes the asynchronous draw object so that is no longer valid and can no longer be used. If you want to stop an asynchronous drawing sequence before all frames have been drawn, you can use the `CancelAsyncDraw()` function. If you want to stop an asynchronous drawing object and make it execute its finishing code, you have to use the `FinishAsyncDraw()` function.

Starting with Hollywood 4.5, `AsyncDrawFrame()` accepts an optional argument which allows you to explicitly specify which frame you want to have drawn. To find out the number of frames of an asynchronous draw object, you have to call `GetAttribute()` on it using `#ATTRNUMFRAMES`. The value you get from this call is the largest valid frame number. To draw the first frame, you have to pass 1. To draw the next frame, pass the value 0 which is also the default and used if the frame argument is omitted.

If you manually specify the frame to draw, you also need to pay attention that your asynchronous drawing object is freed correctly. If you do not use the optional argument, the async drawing object is automatically freed when `AsyncDrawFrame()` returns `True`. If you specify a frame manually, the async draw object is never freed. Even if you specify the last frame, Hollywood will not free the async draw object because it must be possible to seek to the last frame and back to the another frame. If Hollywood automatically freed the async draw object if you chose to draw the last frame, then this would not be possible. So if you are manually setting the current frame, make sure that you call `FinishAsyncDraw()` on the asynchronous drawing object when you are done.

Please note that currently some restrictions apply to this function:

1. Asynchronous frames can only be drawn to the main display window. You cannot use this function to draw to brushes currently.
2. While there are asynchronous draw objects, you cannot switch the layer mode. Calls to `EnableLayers()` or `DisableLayers()` will be disabled while asynchronous draw objects are active.
3. You cannot switch background pictures while there asynchronous draw objects.
4. Frame seeking only works with asynchronous drawing objects that are associated with a layer.

5. Frame seeking does only work with asynchronous drawing objects of type `#ADF_FX`.

INPUTS

`id` identifier of the asynchronous draw object to use

`frame` optional: the frame you wish to draw; works only with async draw objects associated with a layer (V4.5)

RESULTS

`finish` `False` if there are more frames left to be drawn or `True` if the asynchronous draw object has finished

EXAMPLE

```
obj = DisplayBrushFX(1, #CENTER, #CENTER, {Type = #WATER1, Async = True})
Repeat
   done = AsyncDrawFrame(obj)
   VWait
Until done = True
```

The code above displays brush 1 in the center of the screen with the `#WATER1` transition effect. As you can see, the effect is not displayed by `DisplayBrushFX()` but in the `AsyncDrawFrame()` loop below so that you could do some other things as well during the transition effect.

```
EnableLayers
DisplayBrush(1, #CENTER, #CENTER)
obj = ShowLayerFX(1, {Type = #WALLPAPERTOP, Async = True})
frames = GetAttribute(#ASYNCDRAW, obj, #ATTRNUMFRAMES)
For Local k = frames To 1 Step -1
   AsyncDrawFrame(obj, k)
   VWait
Next
For Local k = 1 To frames
   AsyncDrawFrame(obj, k)
   VWait
Next
FinishAsyncDraw(obj)
```

The code above shows brush number 1 using `#WALLPAPERTOP`. The effect is first displayed the other way round and then in normal order. Note that we have to manually free the draw object using `FinishAsyncDraw()`.

19.2 CancelAsyncDraw

NAME

CancelAsyncDraw – cancel asynchronous drawing object (V4.0)

SYNOPSIS

```
CancelAsyncDraw(id)
```

FUNCTION

This function can be used to preliminary cancel an asynchronous draw object. See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

If the asynchronous draw object you want to stop is associated with a layer, then calling `CancelAsyncDraw()` will not execute the finishing code for your draw object. For example, if you are removing a layer with an asynchronous effect (i.e. you retrieved your asynchronous draw object from `RemoveLayerFX()`), then the layer won't be removed if you call `CancelAsyncDraw()`. The same applies to `HideLayerFX()` (layer won't be hidden if you cancel the asynchronous draw object halfway through) and `ShowLayerFX()` (layer won't be shown if you cancel the asynchronous draw object).

If you do not want this behaviour, you have to use `FinishAsyncDraw()` instead. `FinishAsyncDraw()` will first call the finishing code for the asynchronous draw object and then it will free the draw object. This means that if you execute `FinishAsyncDraw()` on a drawing object received from `RemoveLayerFX()`, the layer will be removed even if the effect is not finished yet. `FinishAsyncDraw()` will jump to the last frame, call the finishing code (i.e. remove layer, or hide layer, or show layer) and free the drawing object.

Please note that there is no difference at all between `CancelAsyncDraw()` and `FinishAsyncDraw()` if layers are disabled. In case layers are off, you should always use `CancelAsyncDraw()`.

INPUTS

`id`	identifier of the object to cancel

19.3 CancelAsyncOperation

NAME

CancelAsyncOperation – cancel an asynchronous operation (V9.0)

SYNOPSIS

```
CancelAsyncOperation(id)
```

FUNCTION

This function cancels the asynchronous operation specified by `id`. `id` must be set to an asynchronous operation handle created by functions which support asynchronous operations, e.g. `CopyFile()` or `DownloadFile()`.

INPUTS

`id`	asynchronous operation handle obtained from a function that supports asynchronous operations

EXAMPLE

See Section 19.4 [ContinueAsyncOperation], page 224.

19.4 ContinueAsyncOperation

NAME

ContinueAsyncOperation – continue an asynchronous operation (V9.0)

SYNOPSIS

```
done, ... = ContinueAsyncOperation(id)
```

FUNCTION

This function continues processing the asynchronous object specified by `id`. `id` must be set to an asynchronous operation handle created by functions which support asynchronous operations, e.g. `CopyFile()` or `DownloadFile()`.

Once the asynchronous operation has finished, `ContinueAsyncOperation()` will free the asynchronous operation handle and return `True`. If `ContinueAsyncOperation()` returns `False`, the operation hasn't finished yet and you need to call `ContinueAsyncOperation()` again until it returns `True`.

If the Hollywood call that created the asynchronous operation handle returns values to the script on completion, `ContinueAsyncOperation()` will forward those values to your script as soon as the asynchronous operation has finished, i.e. when `done` becomes `True`. In that case, `ContinueAsyncOperation()` may return additional values depending on the command that created the asynchronous operation handle. For example, `DownloadFile()` will return the downloaded data as well as its length to the script on completion.

To abort an asynchronous operation, you can use the `CancelAsyncOperation()` function. See Section 19.3 [CancelAsyncOperation], page 223, for details.

INPUTS

`id`	asynchronous operation handle obtained from a function that supports asynchronous operations

RESULTS

`done`	`True` if the operation has finished, `False` otherwise
`...`	optional: on completion, i.e. when `done` is `True`, all additional return values from the command that created the asynchronous operation handle

EXAMPLE

```
handle = CopyFile("images", "sounds", {Async = True})
Repeat
    NextFrame(1)
Until ContinueAsyncOperation(handle) = True
```

The code above demonstrates how to show an animation while copying the `images` directory into the `sounds` directory.

19.5 FinishAsyncDraw

NAME

FinishAsyncDraw – finish asynchronous drawing object (V4.5)

SYNOPSIS

```
FinishAsyncDraw(id)
```

FUNCTION

This function can be used to preliminary finish an asynchronous drawing object. When calling this function, Hollywood will skip to the last frame of the asynchronous drawing object and finish it. "Finishing" an asynchronous drawing object means drawing its last frame and running the finishing code (for example, removing the layer if the asynchronous drawing object belongs to a `RemoveLayerFX()` call).

There is a difference between `FinishAsyncDraw()` and `CancelAsyncDraw()`. See Section 19.2 [CancelAsyncDraw], page 222, for details.. An exception is when layers are enabled. In that case there is no difference between `CancelAsyncDraw()` and `FinishAsyncDraw()`. In case layers are off, you should always use `CancelAsyncDraw()`.

INPUTS

`id`	identifier of the object to finish

20 BGPic library

20.1 Overview

Background pictures (BGPics) are very important in Hollywood because every display needs to have a background picture that is attached to it. The background picture is what will be initially shown to the user when your display becomes visible. The background picture will then act as the "work area" of your script, i.e. the area that you can use for drawing any graphics you like. This area is fully yours and you can use it like you want.

The background picture must always be of the same size as the display. Thus, if you change the display size, e.g. by using `ChangeDisplaySize()` your background picture will automatically be scaled to fit the new dimensions because as already said before the display size is always the same as the current background picture size. If your window is resizeable, then the user may also adjust your display size. If he does, Hollywood will internally call `ChangeDisplaySize()` to adjust to the new dimensions.

If you choose to display a new background picture, e.g. by using the `DisplayBGPic()` command, and the dimensions of the new background picture differ from the dimensions of your current background picture, then your display will also be resized to fit the new dimensions.

At startup, Hollywood will display the background picture that has been assigned the identifier 1. If you haven't declared a background picture that uses the identifier 1 using the `@BGPIC` preprocessor command, Hollywood will create this background picture automatically for you and attach it to your display. The background picture will use the fill style and dimensions specified in the `@DISPLAY` preprocessor command for display 1 in your script.

Here is a minimal script which creates a display that shows the image file `test.jpg`:

```
@BGPIC 1, "test.jpg"
WaitLeftMouse
End
```

20.2 BGPIC

NAME

BGPIC – preload a background picture for later use (V2.0)

SYNOPSIS

`@BGPIC id, filename$[, table]`

FUNCTION

This preprocessor command preloads the background picture specified in `filename$` and assigns the identifier `id` to it. If you specify 1 as the identifier, then this picture will be used as the initial background picture when Hollywood opens your display.

Image formats that are supported on all platforms are PNG, JPEG, BMP, IFF ILBM, GIF, and image formats you have a plugin for. Depending on the platform Hollywood is running on, more image formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all image formats you have datatypes for as well. On Windows, `@BGPIC` can also load image formats supported by the Windows Imaging Component.

Starting with Hollywood 5.0, this function can also load vector formats like SVG if you have an appropriate plugin installed. Using a vector image as a BGPic has the advantage that when the size of the display changes (e.g. because the user is resizing the window), the BGPic can be adapted to the new size without any losses in quality because vector BGPics can be infinitely scaled without any sacrifices in quality. See Section 20.16 [Vector BGPics], page 248, for more information on vector BGPics.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall appear transparent in the BGPic.

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the image shall be loaded, too. Please note that not all pictures have an alpha channel and that not all picture formats are capable of storing alpha channel information. It is suggested that you use the PNG format if you need alpha channel data. This tag defaults to `False`. (V4.5)

`Link:`
: Set this field to `False` if you do not want to have this BGPic linked to your executable/applet when you compile your script. This field defaults to `True` which means that the BGPic is linked to your executable/applet when Hollywood is in compile mode.

`FillStyle:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`FillColor:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`TextureBrush:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`TextureX, TextureY:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientStyle:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientAngle:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientStartColor, GradientEndColor:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientCenterX, GradientCenterY:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientBalance:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientBorder:`
: See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`GradientColors:`
See Section 20.13 [LoadBGPic], page 242, for details. (V5.0)

`ScaleWidth, ScaleHeight:`
See Section 20.13 [LoadBGPic], page 242, for details. (V5.3)

`SmoothScale:`
See Section 20.13 [LoadBGPic], page 242, for details. (V5.3)

`Loader:` See Section 20.13 [LoadBGPic], page 242, for details. (V6.0)

`Adapter:` See Section 20.13 [LoadBGPic], page 242, for details. (V6.0)

`LoadTransparency:`
See Section 20.13 [LoadBGPic], page 242, for details. (V6.0)

`LoadPalette:`
See Section 20.13 [LoadBGPic], page 242, for details. (V9.0)

`FillPen:` See Section 20.13 [LoadBGPic], page 242, for details. (V9.0)

`TransparentPen:`
See Section 20.13 [LoadBGPic], page 242, for details. (V9.0)

`UserTags:`
See Section 20.13 [LoadBGPic], page 242, for details. (V10.0)

If you use `Transparency`, `LoadTransparency` or `LoadAlpha`, your display will automatically adopt the transparency settings of the BGPic when it is displayed. In other words, if you want to have a display with transparency, simply load a transparent BGPic and display it.

If you set the `LoadPalette` tag to `True`, your display will become a palette display as soon as the BGPic is shown. Palette displays behave differently than normal true colour displays and there are some things to be considered when using them. See Section 25.16 [Palette displays], page 400, for details.

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` tags are mutually exclusive. A BGPic can only have one transparency setting!

If you want to load background pictures manually, please use the `LoadBGPic()` command.

INPUTS

`id` a value that is used to identify this background picture later in the code; if this is 1 then the picture will be the initial display background

`filename$`
the picture file you want to load

`table` optional: a table for setting further options; see above for explanation

EXAMPLE

```
@BGPIC 1, "MyBG.png"
```

Declare "MyBG.png" as the initial background picture (will be displayed when Hollywood starts up).

```
@BGPIC 1, "MyBG.png", {Transparency = $FF0000}
```

Does the same like above but the picture is now transparent (transparency color is red=$FF0000).

```
@BGPIC 1, "4MB_uncompressed_picture.bmp", {Link = False}
```

The code above loads the specified picture and tells Hollywood that it should never link this image because it is so big.

20.3 BrushToBGPic

NAME

BrushToBGPic – convert a brush to a background picture

SYNOPSIS

`BrushToBGPic(brushid, bgpicid)`

FUNCTION

This function makes a copy of the brush specified by `brushid` and converts it to a new background picture that has the identifier `bgpicid`. Everything will be cloned so the background picture is independent of the brush (you could free the brush after this operation for example and the background picture would still be usable!).

INPUTS

`brushid` brush to clone

`bgpicid` id for the new background picture

EXAMPLE

```
BrushToBGPic(1,2)
DisplayBGPic(2)
```

The above code copies the brush 1 to background picture 2 and displays it then.

20.4 CopyBGPic

NAME

CopyBGPic – clone a background picture (V4.0)

SYNOPSIS

`[id] = CopyBGPic(source, dest)`

FUNCTION

This function clones the background picture specified by `source` and creates a copy of it as background picture `dest`. If you specify `Nil` in the `dest` argument, this function will choose an identifier for this background picture automatically and return it to you. The new background picture is independent from the old one so you could free the source background picture after it has been cloned.

Please note that only the plain graphics data of the background picture will be cloned. `CopyBGPic()` will not clone any layers, sprites, or buttons attached to the background picture.

INPUTS

`source` source background picture id

`dest` identifier of the new BGPic or `Nil` for auto id select

RESULTS

`id` optional: identifier of the BGPic; will only be returned when you pass `Nil` as argument 2 (see above)

EXAMPLE

```
CopyBGPic(1, 10)
FreeBGPic(1)
```

The above code creates a new background picture 10 which contains the same graphics data as background picture 1. Then it frees background picture 1 because it is no longer needed.

20.5 CreateBGPic

NAME

CreateBGPic – create a blank background picture (V1.5)

SYNOPSIS

```
[id] = CreateBGPic(id, width, height[[, color], table])
```

FUNCTION

This function creates a new background picture with the specified width and height and initializes it to the specified color. If no color is specified, the background picture is initialized to black. If you specify `Nil` in the `id` argument, `CreateBGPic()` will choose an identifier for this background picture automatically and return it to you.

Starting with Hollywood 9.0, there is an optional table argument that allows you to specify further options. The following table tags are currently supported:

`Palette:` If this tag is set to the identifier of a palette, Hollywood will create a palette background picture for you. Palettes can be created using functions like `CreatePalette()` or `LoadPalette()`. Alternatively, you can also set this tag to one of Hollywood's inbuilt palettes, e.g. `#PALETTE_AGA`. See Section 44.36 [SetStandardPalette], page 918, for a list of inbuilt palettes.

`FillPen:` If the `Palette` tag is set (see above), you can use this tag to set the pen that should be used for filling the background picture's background. Note that the `color` parameter that is passed to `CreateBGPic()` is ignored if `Palette` is `True`. That's why this tag is here to allow you to specify a pen that will be used when initializing the background picture's pixels. Defaults to 0.

`TransparentPen:`
If `Palette` is set to `True`, this tag can be used to specify a pen that should be made transparent in the new background picture. Defaults to `#NOPEN` which means that there should be no transparent pen.

INPUTS

`id` id for the new background picture

`width` width for the background picture

`height` height for the background picture

`color` optional: RGB color for background (defaults to `#BLACK`)

`table` optional: table containing further options (see above) (V9.0)

RESULTS

`id` optional: identifier of the BGPic; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateBGPic(2, 640, 480)
```

The above code creates a new black BGPic with the id 2 and the dimension of 640x480.

20.6 CreateGradientBGPic

NAME

CreateGradientBGPic – create a new background picture with a gradient (V2.0)

FORMERLY KNOWN AS

CreateRainbowBGPic (V1.0 - V1.9)

SYNOPSIS

```
[id] = CreateGradientBGPic(id, type, startcolor, endcolor[, width,
                           height, angle, table])
```

FUNCTION

This function can be used to create a new background picture with a gradient on it. If you specify `Nil` in the `id` argument, this function will choose an identifier for this background picture automatically and return it to you. `type` specifies the type of the gradient you want to use. The following gradient types are currently available: `#LINEAR`, `#RADIAL`, and `#CONICAL`. If `width` and `height` are omitted, the dimensions will be set to the same as the current display's dimensions. The `angle` parameter allows you to specify a rotation angle (in degrees) for the gradient. The angle argument is only supported by gradients of type `#LINEAR` and `#CONICAL`. Radial gradients cannot be rotated.

The optional table argument can be used to specify advanced options. The following tags are currently recognized:

`CenterX, CenterY:`
These two tags can be used to specify the center point of the gradient. As linear gradients do not have a center point, these two tags are only handled when you use gradients of type `#RADIAL` or `#CONICAL`. The center point must be specified as a floating point value that is between 0.0 (left/top corner) and 1.0 (right/bottom corner). If not specified, both tags default to 0.5 which means that the center point of the gradient is in the center of the image. (V5.0)

`Border:` This tag can be used to set the border size for gradients of type `#RADIAL`. For the other gradient types this tag is ignored. The border size of the radial

gradient must be a floating point value between 0.0 and 1.0. Defaults to 0.0 which means no border. (V5.0)

`Balance:` This tag can be used to set the balance point for gradients of type `#CONICAL`. For the other gradient types this tag is ignored. The balance point of the conical gradient must be floating point value between 0.0 and 1.0. Defaults to 0.5. Note that this is only used when creating a two-color gradient. When creating a multi-color gradient using the `Colors` table, `Balance` is ignored because the `Colors` table allows you to individually balance the colors in the gradient using color stops. (V5.0)

`Colors:` This tag allows you to create gradients that contain multiple colors. This tag must be set to a table that contains a sequence of alternating color and stop values. The colors must be specified in RGB format. The stop value is a floating point value between 0.0 and 1.0 and defines the position where the corresponding color should be merged into the gradient. A position of 0.0 means the start position of the gradient, and a position of 1.0 means the end position. Please note that the stop positions must be sorted in ascending order, i.e. starting from 0.0 to 1.0. If you specify this tag, the colors specified in the `startcolor` and `endcolor` arguments are ignored, and Hollywood will only use the colors specified in this tag. (V5.0)

INPUTS

`id` id for the new background picture or `Nil` for auto ID select

`type` type of the gradient; see above for available types

`startcolor`
RGB value defining the start color

`endcolor` RGB value defining the end color

`width` optional: desired width for the background picture (default: current display width)

`height` optional: desired height for the background picture (default: current display height)

`angle` optional: rotation angle for the gradient (default: 0)

`table` optional: table argument specifying further options; see above for a description of available options

RESULTS

`id` optional: identifier of the BGPic; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateGradientBGPic(2, #LINEAR, #BLACK, #BLUE)
DisplayBGPic(2)
```

Creates a top to bottom gradient as background picture 2 with a color fade from black to blue and displays it.

```
CreateGradientBGPic(2, #LINEAR, 0, 0, 640, 480, 0, {Colors = {#RED, 0,
    #BLUE, 0.25, #GREEN, 0.5, #YELLOW, 0.75, #BLACK, 1}})
DisplayBGPic(2)
```

The code above creates a gradient containing multiple color stops. This gradient tries to replicate the look of the famous Amiga copper bars.

20.7 CreateTexturedBGPic

NAME

CreateTexturedBGPic – create a new background picture textured with a brush

SYNOPSIS

```
[id] = CreateTexturedBGPic(id, brushid[, width, height, x, y])
```

FUNCTION

This function will create a new background picture for you and it will texture it with the brush specified by `brushid`. If you specify `Nil` in the `id` argument, this function will choose an identifier for this background picture automatically and return it to you. If `width` and `height` are omitted, the dimensions will be the same as the current display.

The optional `x` and `y` parameters are new in Hollywood 4.6. They allow you to specify an offset into the texture brush. Texturing will then start from this offset in the brush. The default for these arguments is 0/0 which means start at the top-left corner inside the texture brush.

INPUTS

`id`	id for the new background picture or `Nil` for auto ID select
`brushid`	identifier of the brush to be used as the texture
`width`	optional: desired width for the background picture (default: current display width)
`height`	optional: desired height for the background picture (default: current display height)
`x`	optional: start x offset in the texture brush (V4.6)
`y`	optional: start y offset in the texture brush (V4.6)

RESULTS

`id`	optional: identifier of the BGPic; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateTexturedBGPic(2,1)
DisplayBGPic(2)
```

Creates a background picture that will be textured with brush 1 and displays it.

20.8 DisplayBGPic

NAME

DisplayBGPic – change the background picture

SYNOPSIS

```
DisplayBGPic(id[, args])
```

FUNCTION

This function changes the background picture to the one specified by `id`. If the dimensions of this picture differ from the current one, the display size will be adjusted.

New in Hollywood 4.0: You can pass a table in the optional second argument to specify further options. Currently, the table can contain the following fields:

`X:` Specifies the new x position for the display. If you want the display to keep its current x position, specify the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

`Y:` Specifies the new y position for the display. If you want the display to keep its current y position, specify the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

INPUTS

`id` identifier of the background picture to display

`args` optional: specifies further configuration options (V4.0)

EXAMPLE

```
DisplayBGPic(2)
```

Displays background picture 2 and adjusts the window size if necessary.

```
DisplayBGPic(2, {X = #RIGHT, Y = #BOTTOM})
```

Displays background picture at the bottom-right of the current desktop.

20.9 DisplayBGPicPart

NAME

DisplayBGPicPart – display a part of a background picture

SYNOPSIS

```
DisplayBGPicPart(id, x, y, width, height[, dx, dy, table])
```

FUNCTION

This function displays a tile of the background picture specified by `id` on the screen. The tile is defined by `x`, `y` and its `width` and `height`.

If layers are enabled, this command will add a new layer of the type `#BGPICPART` to the layer stack.

As of Hollywood 4.0, this command uses a new syntax, although the old syntax is still supported for compatibility reasons. New scripts should use the new syntax, though.

The new syntax accepts a table as the last argument which allows you to specify several further options:

`Layers:` If you set this to `True`, the layers (in case layers are enabled) or the foreground graphics (in case layers are disabled) of the background picture are drawn, too. This is useful if you want to create an exact copy of a background picture in a brush, for example. Please note that if layers are disabled, you can use this argument only if `id` specifies the identifier of the current background picture because Hollywood does not keep the entire foreground contents of all background pictures if layers are disabled.

Furthermore, the optional table argument can also contain one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`id`	identifier of the background picture to use as source
`x`	left corner
`y`	top corner
`width`	width of the tile
`height`	height of the tile
`dx`	optional: destination x-position for the tile (defaults to x) (V1.5)
`dy`	optional: destination y-position for the tile (defaults to y) (V1.5)
`table`	optional: further configuration table

EXAMPLE

```
DisplayBGPicPart(2,0,0,100,100)
```

Display the first 100 pixels and rows from background picture 2 on the screen at position 0:0.

```
width = GetAttribute(#DISPLAY, 0, #ATTRWIDTH)
height = GetAttribute(#DISPLAY, 0, #ATTRHEIGHT)
id = GetAttribute(#DISPLAY, 0, #ATTRBGPIC)
CreateBrush(1, width, height)
SelectBrush(1)
DisplayBGPicPart(id, 0, 0, width, height, 0, 0, {Layers = TRUE})
EndSelect
```

This code makes a copy of the current display contents in brush 1.

20.10 DisplayBGPicPartFX

NAME

DisplayBGPicPartFX – display a part of a background picture with transition

SYNOPSIS

```
[handle] = DisplayBGPicPartFX(id, x, y, width, height[, table])
```

FUNCTION

This is an extended version of the `DisplayBGPicPart()` command. It does the same but displays the part with a transition effect.

If layers are enabled, this command will add a new layer of the type `#BGPICPART` to the layer stack.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type:` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of all supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed:` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter:` Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async:` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `DisplayBGPicPartFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`DX:` Destination x position for the tile. (defaults to the x specified as parameter 2)

`DY:` Destination y position for the tile. (defaults to the y specified as parameter 3)

`Layers:` Specify `True` here if the layers of the background picture shall also be displayed (requires enabled layers). (defaults to `False`)

INPUTS

`id` identifier of the background picture to use as source

`x` left corner

`y` top corner

`width` width of the tile

`height` height of the tile

`table` optional: transition effect configuration

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
DisplayBGPicPartFX(2, 0, 0, 100, 100, #HSTRIPES32, 10)   ; old syntax

OR

DisplayBGPicPartFX(2, 0, 0, 100, 100, {Type = #HSTRIPES32,
     Speed = 10})  ; new syntax
```

Display the first 100 pixels and rows from background picture 2 on the screen with the transition effect `#HSTRIPES32` at speed 10.

20.11 DisplayTransitionFX

NAME

DisplayTransitionFX – change the background picture using a transition effect

SYNOPSIS

```
[handle] = DisplayTransitionFX(id[, table])
```

FUNCTION

This function displays a new background picture with the specified transition effect. A list of all available effects is appended below. You also have to specify the speed of the transition which can be either one of the special speed constants (`#SLOWSPEED`, `#NORMALSPEED`, `#FASTSPEED`) or a custom fine-tuned numeric value. The rule of thumb for the speed parameter: the higher the value the faster the transition will run.

For the best effect, the new background picture should have the same dimensions as the old. If this is not the case, the old background picture will be scaled to the size of the new one.

Note that transparent BGPics cannot be displayed using a transition effect. It is also not possible to display a non-transparent BGPic with transition effect if the current BGPic is a transparent one. For this function to work, two conditions must be met: The current BGPic as well as the new BGPic must both be non-transparent.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type:` Specifies the desired effect for the transition. See the list below for possible effects. (defaults to `#RANDOMEFFECT`)

`Speed:` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter:`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async:` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `DisplayTransitionFX()` will exit immediately,

returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`X:` Specifies the new x position for the display. If you want the display to keep its current x position, specify the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

`Y:` Specifies the new y position for the display. If you want the display to keep its current y position, specify the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

The following effects are currently available:

- Horizontal stripes: `#HSTRIPES2`, `#HSTRIPES4`, `#HSTRIPES16`, `#HSTRIPES32`
- Vertical stripes: `#VSTRIPES2`, `#VSTRIPES8`, `#VSTRIPES16`, `#VSTRIPES32`
- Fast horizontal blinds: `#HBLINDS8`, `#HBLINDS16`, `#HBLINDS32`, `#HBLINDS64`, `#HBLINDS128`
- Fast vertical blinds: `#VBBLINDS8`, `#VBLINDS16`, `#VBLINDS32`, `#VBLINDS64`, `#VBLINDS128`
- Horizontal curtain: `#HOPENCURTAIN`, `#HCLOSECURTAIN`
- Vertical curtain: `#VOPENCURTAIN`, `#VCLOSECURTAIN`
- Horizontal lines: `#HLINES`, `#HLINES2`
- Vertical lines: `#VLINES`, `#VLINES2`
- Reveal: `#REVEALLEFT`, `#REVEALRIGHT`, `#REVEALTOP`, `#REVEALBOTTOM`
- Bars: `#BARS`
- Quarters: `#QUARTERS`
- Crossfade: `#CROSSFADE`
- Fade: `#FADE`; optional argument specifies color to use
- Blend: `#BLEND`; optional argument specifies color to use for blending

Starting with Hollywood 1.5 there are some new effects:

- Rectangle zoom in: `#RECTCENTER`, `#RECTNORTH`, `#RECTNORTHEAST`, `#RECTEAST`, `#RECTSOUTHEAST`, `#RECTSOUTH`, `#RECTSOUTHWEST`, `#RECTWEST`, `#RECTNORTHWEST`
- Rectangle zoom out: `#RECTBACKCENTER`, `#RECTBACKNORTH`, `#RECTBACKNORTHEAST`, `#RECTBACKEAST`, `#RECTBACKSOUTHEAST`, `#RECTBACKSOUTH`, `#RECTBACKSOUTHWEST`, `#RECTBACKWEST`, `#RECTBACKNORTHWEST`
- Scroll in: `#SCROLLLEFT`, `#SCROLLRIGHT`, `#SCROLLTOP`, `#SCROLLBOTTOM` (optional argument specifies a special effect to apply to the scroll process, you can use the same effects here like in `MoveBrush()`)
- Stretch image in: `#STRETCHLEFT`, `#STRETCHRIGHT`, `#STRETCHTOP`, `#STRETCHBOTTOM`, `#HSTRETCHCENTER`, `#VSTRETCHCENTER`
- Zoom image in: `#ZOOMCENTER`, `#ZOOMNORTH`, `#ZOOMNORTHEAST`, `#ZOOMEAST`, `#ZOOMSOUTHEAST`, `#ZOOMSOUTH`, `#ZOOMSOUTHWEST`, `#ZOOMWEST`, `#ZOOMNORTHWEST`
- Flow: `#HFLOWTOP`, `#HFLOWBOTTOM`, `#VFLOWLEFT`, `#VFLOWRIGHT`
- Gates: `#HOPENGATE`, `#HCLOSEGATE`, `#VOPENGATE`, `#VCLOSEGATE` (B)

- Pushes: `#PUSHLEFT`, `#PUSHRIGHT`, `#PUSHTOP`, `#PUSHBOTTOM` (B)
- Puzzle: `#PUZZLE`
- Diagonal: `#DIAGONAL`
- Roll on: `#ROLLTOP`
- Wallpaper: `#WALLPAPERTOP`
- General vertical stripes: `#VSTRIPES`; optional argument specifies the number of stripes to display
- General horizontal stripes: `#HSTRIPES`; optional argument specifies the number of stripes to display

Starting with Hollywood 1.9 there are a number of new effects:

- Scroll image in: `#SCROLLNORTHEAST`, `#SCROLLSOUTHEAST`, `#SCROLLSOUTHWEST`, `#SCROLLNORTHWEST` (optional argument specifies a special effect to apply to the scroll process, you can use the same effects here like in `MoveBrush()`)
- Reveal clock wise: `#CLOCKWIPE`
- Star zoom in: `#STAR`
- Strange pushes: `#HSTRANGEPUSH`, `#VSTRANGEPUSH` (B)
- Slide projector: `#SLIDELEFT`, `#SLIDERIGHT`, `#SLIDETOP`, `#SLIDEBOTTOM` (B)
- Spiral reveal: `#SPIRAL`
- Swiss cross effect: `#SWISS`
- Quad rectangles: `#QUADRECT`
- Split effects: `#HSPLIT`, `#VSPLIT`
- Up'n'down: `#UPNDOWN`
- Register card effect: `#CARDTOP`, `#CARDBOTTOM` (B)
- Sun zoom in: `#SUN`
- Water ripples: `#WATER1`, `#WATER2`, `#WATER3`, `#WATER4` (!)
- Strudel effect: `#STRUDEL` (!)
- Dissolve picture: `#DISSOLVE`
- Zoom to pixels: `#PIXELZOOM1`
- Zoom to pixels 2: `#PIXELZOOM2` (B)
- Large zoom effects: `#ZOOMIN`, `#ZOOMOUT` (B)
- Crush effects: `#CRUSHLEFT`, `#CRUSHRIGHT`, `#CRUSHTOP`, `#CRUSHBOTTOM` (B)
- Flip coins: `#VFLIPCOIN`, `#VLOWFLIPCOIN`, `#HFLIPCOIN`, `#HLOWFLIPCOIN` (B)
- Turn down picture effect: `#TURNDOWNTOP`, `#TURNDOWNBOTTOM`, `#TURNDOWNLEFT`, `#TURNDOWNRIGHT` (B)
- Type writer effect: `#TYPEWRITER` (T) [no longer supported since V3.1]
- Wallpaper: `#WALLPAPERLEFT` (!)
- Roll on: `#ROLLLEFT`

If you choose `#RANDOMEFFECT`, Hollywood will randomly choose any effect from all possible effects. Very useful when doing slideshows. When using the 68k version of Hollywood, `#RANDOMEFFECT` will not choose any high-end effects automatically.

Legend:

(B): effect can only be used with background pictures

(O): effect can only be used with objects (brushes, layers etc. - but not background pictures!)

(T): effect can only be used with text objects

(!): high-end effects which means that it needs a lot of cpu power to run smoothly. You can run them on 68k, but it is no fun at all because they will take like 4 minutes for a transition or so. You should only use "!"-effects on PPC systems, e.g. MorphOS or WarpOS.

INPUTS

`id` identifier of the background picture to display

`table` optional: transition effect configuration

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
DisplayTransitionFX(2, #HSTRIPES32, 10)   ; old syntax

OR

DisplayTransitionFX(2, {Type = #HSTRIPES32, Speed = 10})  ; new syntax
```

Display background picture 2 using the `#HSTRIPES32` effect and speed 10.

20.12 FreeBGPic

NAME

FreeBGPic – free a background picture

SYNOPSIS

```
FreeBGPic(id)
```

FUNCTION

This function frees the memory of the background picture specified by `id`. To reduce memory consumption, you should free background picture when you do not need them any longer.

INPUTS

`id` identifier of the background picture

20.13 LoadBGPic

NAME

LoadBGPic – load a background picture

SYNOPSIS

```
[id] = LoadBGPic(id, filename$[, table])
```

FUNCTION

This function loads the picture specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadBGPic()` will automatically choose an identifier and return it.

Image formats that are supported on all platforms are PNG, JPEG, BMP, IFF ILBM, GIF, and image formats you have a plugin for. Depending on the platform Hollywood is running on, more image formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all image formats you have datatypes for as well. On Windows, `LoadBGPic()` can also load image formats supported by the Windows Imaging Component.

Starting with Hollywood 5.0, this function can also load vector formats like SVG if you have an appropriate plugin installed. Using a vector image as a BGPic has the advantage that when the size of the display changes (e.g. because the user is resizing the window), the BGPic can be adapted to the new size without any losses in quality because vector BGPics can be infinitely scaled without any sacrifices in quality. See Section 20.16 [Vector BGPics], page 248, for more information on vector BGPics.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall appear transparent in the BGPic.

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the image shall be loaded, too. Please note that not all pictures have an alpha channel and that not all picture formats are capable of storing alpha channel information. It is suggested that you use the PNG format if you need alpha channel data. This tag defaults to `False`. (V4.5)

`FillStyle:`
: This tag allows you to define a background fill style for this BGPic. This fill style is only used when the BGPic has transparent areas, i.e. when you use either the `Transparency` or the `LoadAlpha` tags, or when loading an image in a format that always contains an alpha channel. `FillStyle` is useful especially in the latter case because some formats always return an alpha channel but most of the time you will not want to use this alpha channel when loading such an image into a background picture. See Section 27.14 [SetFillStyle], page 498, for information on all available fill styles. (V5.0)

`FillColor:`
: If the `FillStyle` tag was set to `#FILLCOLOR`, you can use this tag to define the RGB color that shall be used for backfilling. (V5.0)

`TextureBrush:`
: If the `FillStyle` tag was set to `#FILLTEXTURE`, you can use this tag to specify the identifier of the brush that shall be used for texturing. (V5.0)

`TextureX, TextureY:`
: These tags control the start offset inside the texture brush and are only supported if `FillStyle` was set to `#FILLTEXTURE`. See Section 27.14 [SetFillStyle], page 498, for details. (V5.0)

`GradientStyle:`
: If the `FillStyle` tag was set to `#FILLGRADIENT`, you can use this tag to specify the gradient type to use. This can be `#LINEAR`, `#RADIAL`, or `#CONICAL`. (V5.0)

`GradientAngle:`
: Specifies the orientation of the gradient if filling style is set to `#FILLGRADIENT`. The angle is expressed in degrees. Only possible for `#LINEAR` and `#CONICAL` gradients. (V5.0)

`GradientStartColor, GradientEndColor:`
: Use these two to configure the colors of the gradient if filling style is set to `#FILLGRADIENT`. (V5.0)

`GradientCenterX, GradientCenterX:`
: Sets the center point for gradients of type `#RADIAL` or `#CONICAL`. Must be a floating point value between 0.0 and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientBalance:`
: This tag controls the balance point for gradients of type `#CONICAL`. Must be a floating point value between 0.0 and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientBorder:`
: This tag controls the border size for gradients of type `#RADIAL`. Must be a floating point value between 0.0 and and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientColors:`
: This tag can be used to create a gradient between more than two colors. This has to be set to a table that contains sequences of alternating color and stop values. See Section 20.6 [CreateGradientBGPic], page 232, for details. If this tag is used, the `GradientStartColor` and `GradientEndColor` tags are ignored. (V5.0)

`ScaleWidth, ScaleHeight:`
: These fields can be used to load a scaled version of the image. If the image driver supports scaled loading, this will give you some significant speed-up for example in case you just want to load a thumbnail-sized version of a large image. If the image driver does not support scaled loading, the full image will be loaded first before it is scaled. This is not much faster than manually scaling the image after loading. You can pass an absolute pixel value or a string containing a percent specification here. (V5.3)

`SmoothScale:`
: If `ScaleWidth` or `ScaleHeight` is set, you can use this item to specify whether or not Hollywood shall use anti-aliased scaling. Defaults to `False` which means no anti-aliasing. Note that anti-aliased scaling is much slower than normal scaling. (V5.3)

`Loader:`
: This tag allows you to specify one or more format loaders that should be asked to load this BGPic. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:`
: This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
: If this tag is set to `True`, the monochrome transparency of the image will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based image. If you want to load the alphachannel of an image, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
: If this tag is set to `True`, Hollywood will load the BGPic as a palette BGPic. This means that you can get and modify the BGPic's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette BGPics also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. Note that if you set the `LoadPalette` tag to `True`, your display will become a palette display as soon as the BGPic is shown. Palette displays behave differently than normal true colour displays and there are some things to be considered when using them. See Section 25.16 [Palette displays], page 400, for details. This tag defaults to `False`. (V9.0)

`FillPen:`
: If the `LoadPalette` tag has been set to `True` (see above) and there is a transparent pen in the image, you can use the `FillPen` tag to specify the filling color for all transparent areas in the image. This is the palette equivalent to the `FillColor` tag which is only used for non-palette images. (V9.0)

`TransparentPen:`
: If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the image file (if the image format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
: This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

If you use `Transparency`, `LoadTransparency` or `LoadAlpha` your display will automatically adopt the transparency settings of the BGPic when it is shown. In other words, if you want to have a display with transparency simply load a transparent BGPic and display it.

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. A BGPic can only have one transparency setting!

This command is also available from the preprocessor: Use `@BGPIC` to preload background pictures!

INPUTS

`id` identifier for the background picture or `Nil` for auto id select

`filename$`
: file to load

`table` optional: further configuration options for loading operation

RESULTS

`id` optional: identifier of the background picture; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadBGPic(2, "MyBG.iff", {Transparency = $00FF00})
```

This loads "MyBG.iff" as background picture 2 with the color green as transparency mask.

20.14 ScaleBGPic

NAME

ScaleBGPic – scale a background picture (V1.5)

SYNOPSIS

```
ScaleBGPic(id, width, height[, smooth])
```

FUNCTION

This function scales the background picture specified by `id` to the specified width and height. You cannot use this function with the background picture that is currently displayed. If you want to change the size of this picture, you will have to change the display size using `ChangeDisplaySize()`.

Hollywood keeps the original picture of every background picture so that you do not have to do that. Every scaling done with `ScaleBGPic()` is made with the original picture and not with a scaled versions (in contrast to `ScaleBrush()`). The original picture will only be deleted if you modify the background picture's contents using `SelectBGPic()`.

New in V2.0: You can pass `#KEEPASPRAT` as either `width` or `height`. Hollywood will calculate the size then automatically by taking the aspect-ratio of the background picture into account.

Starting with Hollywood 2.0, `width` and `height` can also be a string containing a percent specification, e.g. `"50%"`.

Starting with Hollywood 2.5 you can choose to have the scaled graphics interpolated by passing `True` in the `smooth` argument. The graphics will then be scaled using anti-alias.

INPUTS

`id`	background picture to scale
`width`	new width for the background picture
`height`	new height for the background picture
`smooth`	optional: whether or not anti-aliased scaling shall be used (V2.5)

20.15 SelectBGPic

NAME

SelectBGPic – select a background picture as output device (V1.5)

SYNOPSIS

```
SelectBGPic(id[, mode, combomode])
```

FUNCTION

This function selects the BGPic specified by `id` as the current output device. This command be used in various different modes. The usual mode to use `SelectBGPic()` is layers mode (`#SELMODE_LAYERS`) which is also the default mode. Layers mode means that all graphics data that are output by Hollywood will be added as layers to this background picture. Therefore you will have to enable layers before using this command in layers mode. Your background picture will never be modified in layers mode, it will just get more and more layers.

Alternatively, you can use the `#SELMODE_NORMAL` or `#SELMODE_COMBO` modes. These modes will modify your BGPic's data. They can only be used on BGPics that are currently not associated with a display. `#SELMODE_NORMAL` means that only the color channels of the BGPic will be altered when you draw to it. The transparency channel of the BGPic (can be either a mask or an alpha channel) will never be altered. You can change this behaviour by using `#SELMODE_COMBO` in the optional `mode` argument. If you use this mode, every Hollywood graphics command that is called after `SelectBGPic()` will draw into the color and transparency channel of the BGPic. If the BGPic does not have a transparency channel, `#SELMODE_COMBO` behaves the same as `#SELMODE_NORMAL`.

Starting with Hollywood 5.0 you can use the optional `combomode` argument to specify how `#SELMODE_COMBO` should behave. If `combomode` is set to 0, the color and transparency information of all pixels in the source image are copied to the destination image in any case - even if the pixels are invisible. This is the default behaviour. If `combomode` is set to 1, only the visible pixels are copied to the destination image. This means that if the alpha value of a pixel in the source image is 0, i.e. invisible, it will not be copied to the

destination image. Hollywood 6.0 introduces the new combomode 2. If you pass 2 in `combomode`, Hollywood will blend color channels and alpha channel of the source image into the destination image's color and alpha channels. When you draw the destination image later, it will look as if the two images had been drawn on top of each other consecutively. Please note that the `combomode` argument is only supported together with `#SELMODE_COMBO`. It doesn't have any effect when used with the other modes.

Note that when you use `#SELMODE_NORMAL` or `#SELMODE_COMBO`, the original graphics of the BGPic are modified. You will always be drawing to the original graphics of the BGPic. Imagine that you have a 640x480 BGPic that is currently scaled to 800x600 because you called `ChangeDisplaySize()`. If you call `SelectBGPic()` with `#SELMODE_NORMAL` or `#SELMODE_COMBO` now on this BGPic, you will actually be drawing to the 640x480 picture. The 800x600 picture will be updated when `EndSelect()` is called. On `EndSelect()`, Hollywood will scale the original graphics to the current output size of the BGPic, but your initial drawing will always occur on the original BGPic.

An alternative way to draw into the transparency channels of a BGPic is to do this separately using `SelectMask()` or `SelectAlphaChannel()`. These two commands, however, will write data to the transparency channel only. They will not touch the color channel. So if you want both channels, color and transparency, to be affected, you need to use `SelectBGPic()` with `mode` set to `#SELMODE_COMBO`.

When you are finished with rendering to your BGPic and want your display to become the output device again, just call `EndSelect()`.

Only commands that output graphics directly can be used after `SelectBGPic()`. You may not call animated functions like `MoveBrush()` or `DisplayBrushFX()` while `SelectBGPic()` is active.

When `mode` is set to `#SELMODE_LAYERS`, `SelectBGPic()` can also come handy when you want to make multiple changes to the layers of the current BGPic without causing a refresh after each change. For example, you may want to insert 100 new layers at once. This would be pretty slow if you did it in the conventional way because Hollywood would refresh the display a hundred times. To avoid this, you can simply call `SelectBGPic()` and insert the 100 layers and Hollywood will not refresh the display before you call `EndSelect()`. Inside a `SelectBGPic()-EndSelect()` block, you can do as many changes as you like. They will not be drawn before `EndSelect()` is called. See below for an example.

INPUTS

`id` background picture which shall be used as output device

`mode` optional: rendering mode to use (see above); this can be either `#SELMODE_LAYERS`, `#SELMODE_NORMAL` or `#SELMODE_COMBO`; defaults to `#SELMODE_LAYERS` (V4.5)

`combomode`
optional: mode to use when `#SELMODE_COMBO` is active (see above); defaults to 0 (V5.0)

EXAMPLE

```
EnableLayers()
SelectBGPic(2)
```

```
TextOut(#CENTER, #CENTER, "Hello World")
Box(0, 0, 100, 100, #RED)
Box(#RIGHT, #BOTTOM, 100, 100, #BLUE)
EndSelect()
DisplayBGPic(2)
```

The above code selects background picture 2 as the current output device and adds three layers to it (one text and two rectangles). After that, the display is selected as the output device and then background picture 2 is displayed with its three layers.

```
SetFillStyle(#FILLCOLOR)
EnableLayers
SelectBGPic(1)    ; we assume that 1 is our current BGPic
; add 100 random layers
For Local k = 1 To 100
   Box(Rnd(540), Rnd(380), 100, 100, RGB(Rnd(255), Rnd(255), Rnd(255)))
Next
EndSelect       ; now the 100 layers are drawn in one go!
```

This code illustrates the case discussed above. You need to make lots of changes and you want to defer drawing for performance reasons. In our case, we want to add 100 layers to the current BGPic. So we encapsulate this code by a `SelectBGPic()`-EndSelect block. Hollywood will silently add the 100 layers and will draw them in one go when EndSelect() is called. This is much faster than adding them without `SelectBGPic()` because in that case every call to Box() would cause a refresh.

20.16 Vector BGPics

When you load a vector image using `LoadBGPic()` or `@BGPIC`, you will get a special type of BGPic: a vector BGPic. When loading normal images like PNG, JPEG, etc. you will always get a raster BGPic. You can find out the type of a BGPic by querying the `#ATTRTYPE` attribute using `GetAttribute()`.

The advantage of a vector BGPic is that you can scale and/or transform it without any quality losses. For example, when the user resizes a display, its BGPic can be adapted to the new size without quality sacrifices. So it is possible to create scripts which are infinitely scalable. All you have to do is stick to vector BGPics, vector brushes, and vector text (i.e. use TrueType fonts).

Besides vector BGPics generated from vector image formats, there are also some other types of vector BGPics in Hollywood. For example, the `CreateGradientBGPic()` and `CreateTexturedBGPic()` functions will also create vector BGPics that can be infinitely scaled.

21 Brush library

21.1 Overview

Brushes are the most flexible image type in Hollywood. You can create a brush by either loading an image file from disk using `LoadBrush()` or creating image data in memory using `CreateBrush()`. Hollywood's brush library contains a multitude of functions that allow you to transform brushes or process them using a wide variety of image filters. You can also draw to a brush by selecting it as the current output device. This is done by using the `SelectBrush()` function. See Section 21.64 [SelectBrush], page 303, for details. You can also draw to a brush's mask or alpha channel by using the `SelectMask()` or `SelectAlphaChannel()` functions, respectively.

Here is a short code snippet which loads a brush from an image file and draws it to the center of the display:

```
LoadBrush(1, "test.jpg")
DisplayBrush(1, #CENTER, #CENTER)
```

Most other image types in Hollywood can be converted to brushes and vice versa. That's why brushes are the most flexible image type Hollywood offers. For hardware-accelerated drawing, Hollywood also supports hardware brushes. See Section 21.37 [Hardware brush information], page 280, for details.

Normally, brushes contain raster pixel data, but Hollywood also supports special vector brushes which consist of vector path data instead and can thus be freely transformed. See Section 21.80 [Vector brush information], page 316, for details.

21.2 ArcDistortBrush

NAME

ArcDistortBrush – apply arc distortion to brush (V5.0)

SYNOPSIS

`ArcDistortBrush(id, angle1[, angle2, rtop, rbottom, smooth])`

FUNCTION

This command can be used to apply arc distortion to the brush specified in `id`. The `angle1` argument specifies the angle over which the brush should be arc'ed. The optional arguments can be used to control further parameters for the arc distortion. `angle2` can be used to rotate the brush around the circle, and the `rtop` and `rbottom` values can be used to adjust the top and bottom radii settings. Finally, the optional argument `smooth` can be used to enable antialiased pixel interpolation which leads to a smoother appearance but takes longer to calculate.

INPUTS

`id`	brush that shall be distorted
`angle1`	angle over which to bend the brush
`angle2`	optional: angle for rotating the brush around the circle (defaults to 0)

`rtop` optional: top edge of brush will be set to this radius (defaults to an automatically calculated value that tries to keep the aspect-ratio as good as possible)

`rbottom` optional: bottom edge of brush will be set to this radius (defaults to an automatically calculated value that tries to keep the aspect-ratio as good as possible)

`smooth` optional: whether or not anti-aliased distortion shall be used (defaults to `False`)

21.3 BarrelDistortBrush

NAME

BarrelDistortBrush – apply barrel distortion to brush (V5.0)

SYNOPSIS

```
BarrelDistortBrush(id, ...)
BarrelDistortBrush(id, A, B, C, D[, X, Y])
BarrelDistortBrush(id, Ax, Bx, Cx, Dx, Ay, By, Cy, Dy[, X, Y])
```

FUNCTION

This command can be used to apply barrel distortion to the brush specified in `id`. You can use this function in two different ways: The first way requires you to pass at least three coefficients (`A`, `B`, `C`) that define the barrel distortion. Optionally, you can specify a fourth coefficient (`D`) and a center point for the radial distortion (`X` and `Y`). The center point has to passed in pixels whereas the coefficients must be specified as floating point values. If all coefficients add up to 1.0, there will be no change in the picture.

The second way of using this function is to provide separate coefficients for the x and y axis. In that case, you have to pass 8 coefficients (4 for every axis). As in the first variant, you can optionally specify a center point.

Finally, the optional argument `smooth` can be used to enable antialiased pixel interpolation which leads to a smoother appearance but takes longer to calculate.

INPUTS

`id` brush that shall be distorted

`...` coefficients for the barrel distortion (see above)

`X` optional: x coordinate of center point (defaults to half of brush width)

`Y` optional: y coordinate of center point (defaults to half of brush height)

`smooth` optional: whether or not anti-aliased distortion shall be used (defaults to `False`)

21.4 BGPicToBrush

NAME

BGPicToBrush – convert a background picture to a brush

SYNOPSIS

```
BGPicToBrush(bgpicid, brushid)
```

FUNCTION

This function makes a copy of the background picture specified by `bgpicid` and converts it to a brush that will be accessible with the number `brushid` then. Everything will be cloned so the brush is independent of background picture (you could free it after this operation for example and the brush would still be usable!).

INPUTS

`bgpicid` background picture to clone

`brushid` id for the new brush

EXAMPLE

```
BGPicToBrush(1,5)
DisplayBrush(5,#CENTER,#CENTER)
```

The above code copies the background picture 1 to brush 5 and displays this brush then.

21.5 BlurBrush

NAME

BlurBrush – apply Gaussian blur to brush (V5.0)

SYNOPSIS

```
BlurBrush(id[, radius])
```

FUNCTION

This command applies a Gaussian blur to the specified brush. The optional argument `radius` can be used to specify the blur radius. The larger the radius you specify here, the longer this function needs to apply the blur effect. If you do not specify the optional argument, `BlurBrush()` will automatically choose a blur radius.

Note that this function cannot be used with palette brushes.

INPUTS

`id` brush to blur

`radius` optional: blur radius (defaults to 0 which means that the radius will be chosen automatically)

21.6 BRUSH

NAME

BRUSH – preload a brush for later use (V2.0)

SYNOPSIS

```
@BRUSH id, filename$[, table]
```

FUNCTION

This preprocessor command preloads the brush specified in `filename$` and assigns the identifier `id` to it.

Image formats that are supported on all platforms are PNG, JPEG, BMP, IFF ILBM, GIF, and image formats you have a plugin for. Depending on the platform Hollywood is running on, more image formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all image formats you have datatypes for as well. On Windows, `@BRUSH` can also load image formats supported by the Windows Imaging Component.

Starting with Hollywood 5.0, this function can also load vector formats like SVG if you have an appropriate plugin installed. Keep in mind, though, that when you load vector images using this command, the brush will be a special vector brush which does not support all features of the normal brushes. You can, however, convert vector brushes to raster brushes by using the `RasterizeBrush()` function. See Section 21.80 [Vector brushes], page 316, for more information on vector brushes.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall be made transparent in the brush.

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the image shall be loaded, too. Please note that not all pictures have an alpha channel and that not all picture formats are capable of storing alpha channel information. It is suggested that you use the PNG format if you need alpha channel data. This field defaults to `False`.

`Link:`
: Set this field to `False` if you do not want to have this brush linked to your executable/applet when you compile your script. This field defaults to `True` which means that the brush is linked to your executable/applet when Hollywood is in compile mode.

`X, Y, Width, Height:`
: These fields can be used to load only a part of the image into the brush. This is useful if you have one big image with many different small images in it and now you want to load the small images into single brushes. Using these fields you can specify a rectangle inside the image from which Hollywood will take the graphics data for the brush.

`Hardware:`
: If you set this tag to `True`, Hollywood will create this brush entirely in video memory for hardware-accelerated drawing in connection with a hardware double buffer. Hardware brushes are subject to several restrictions. See Section 21.37 [hardware brushes], page 280, for details. (V5.0)

`ScaleWidth, ScaleHeight:`
: These fields can be used to load a scaled version of the image. If the image driver supports scaled loading, this will give you some significant speed-up for example in case you just want to load a thumbnail-sized version of a large image. If the image driver does not support scaled loading, the full image will be loaded first before it is scaled. This is not much faster than manually scaling the image after loading. You can pass an absolute pixel value or a string containing a percent specification here. (V5.3)

`SmoothScale:`
: If `ScaleWidth` or `ScaleHeight` is set, you can use this item to specify whether or not Hollywood shall use anti-aliased scaling. Defaults to `False` which means no anti-aliasing. Note that anti-aliased scaling is much slower than normal scaling. (V5.3)

`Display:`
: If you specify the identifier of a display here, Hollywood will create a display-dependent hardware brush for you. Display-dependent hardware brushes can only be drawn to the display they belong to. This tag is only handled if the `Hardware` tag has been set to `True`. Also note that Hollywood's inbuilt display adapter does not support display-dependent hardware brushes, but plugins can install custom display adapters which support display-dependent hardware brushes. This tag defaults to the identifier of the currently active display. See Section 21.37 [hardware brushes], page 280, for details. (V6.0)

`Loader:`
: This tag allows you to specify one or more format loaders that should be asked to load this brush. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:`
: This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
: If this tag is set to `True`, the monochrome transparency of the image will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based image. If you want to load the alphachannel of an image, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
: If this tag is set to `True`, Hollywood will load the brush as a palette brush. This means that you can get and modify the brush's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette brushes also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`. (V9.0)

`TransparentPen:`
: If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the image file (if the image format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
: This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. A brush can only have one transparency setting!

If you want to load brushes manually, please use the `LoadBrush()` command.

INPUTS

`id` a value that is used to identify this brush later in the code

`filename$`
: the picture file you want to load

`table` optional argument specifying further options

EXAMPLE

```
@BRUSH 1, "MyBrush.png"
```

Load "MyBrush.png" as brush 1 with no transparency.

```
@BRUSH 1, "MyBrush.png", {Transparency = $FF0000}
```

Does the same like above but the brush is now transparent (transparency color is red=$FF0000).

```
@BRUSH 1, "Sprites.png", {X = 64, Y = 32, Width = 32, Height = 32}
```

Loads an image of 32x32 pixels from "Sprites.png" starting at X=64 and Y=32.

21.7 BrushToGray

NAME

BrushToGray – convert brush to gray (V1.5)

SYNOPSIS

`BrushToGray(id)`

FUNCTION

This function converts the brush specified by `id` to gray. If the brush is big, this can take some time.

Note that if `id` specifies a palette brush, `BrushToGray()` will just convert the palette colors to gray which makes this function really fast when used with palette brushes.

INPUTS

`id` identifier of the brush to convert

EXAMPLE

```
BrushToGray(1)
```

Convert brush 1 to gray.

21.8 BrushToMonochrome

NAME

BrushToMonochrome – convert brush to black and white (V5.0)

SYNOPSIS

```
BrushToMonochrome(id[, dither])
```

FUNCTION

This command can be used to map a brush to black and white colors. If the optional argument `dither` is set to `True`, dithering will be applied. Dithering is slower but generates better looking images.

Note that this function cannot be used with palette brushes.

INPUTS

`id` brush to convert to black & white

`dither` optional: whether or not dithering shall be used (defaults to `True`)

21.9 BrushToPenArray

NAME

BrushToPenArray – convert palette brush to pen array (V9.0)

SYNOPSIS

```
table = BrushToPenArray(id)
```

FUNCTION

This command copies all pens from the specified palette brush to a table and returns that table to you. The table can be seen as a matrix containing a number of rows that is identical to the brush's height where each row has a number of elements that is identical to the brush's width. The order of the pixel data in this table is as follows: Row after row in top-down format, i.e. the table starts with the first row of pixels.

Note that the rows won't be stored as subtables. The table returned by `BrushToPenArray()` will be one-dimensional and will contain exactly $height * $width elements, stored sequentially, row after row.

Please note that the table that you get from this function will usually eat lots of memory. Thus, you should set this table to `Nil` as soon as you no longer need it. Otherwise you will waste huge amounts of memory and it could even happen that your script runs out of memory altogether. So please keep in mind that you should always set pixel array tables to `Nil` as soon as you are done with them.

To convert a pen array back to a palette brush, you can use the `PenArrayToBrush()` function. See Section 21.48 [PenArrayToBrush], page 288, for details.

If you want to have RGB colors instead of pen values, you have to use the `BrushToRGBArray()` function instead. See Section 21.10 [BrushToRGBArray], page 256, for details.

INPUTS

`id` identifier of the palette brush to be converted to a pen array

RESULTS

`table` a table containing all pens from the source brush; do not forget to set this table to `Nil` when you are done with it!

21.10 BrushToRGBArray

NAME

BrushToRGBArray – convert brush to pixel array (V5.0)

SYNOPSIS

```
table = BrushToRGBArray(id[, invalpha])
```

FUNCTION

This command copies all pixels from the specified brush to a table and returns that table to you. The table can be seen as a matrix containing a number of rows that is identical to the brush's height where each row has a number of elements that is identical to the brush's width. The order of the pixel data in this table is as follows: Row after row in top-down format, i.e. the table starts with the first row of pixels. The single pixels are stored as ARGB values.

Note that the rows won't be stored as subtables. The table returned by `BrushToRGBArray()` will be one-dimensional and will contain exactly $height * $width elements, stored sequentially, row after row.

The optional argument `invalpha` can be used to tell `BrushToRGBArray()` that all alpha channel values shall be inverted. This means that a value of 0 means 100% visibility and a value of 255 means invisibility. Normally, it is just the other way round. Due to historical reasons, the Hollywood drawing library uses inverted alpha values, and this why they are also supported by `BrushToRGBArray()`, although they are not the default.

Please note that the table that you get from this function will usually eat lots of memory. Thus, you should set this table to `Nil` as soon as you no longer need it. Otherwise you will waste huge amounts of memory and it could even happen that your script runs out of memory altogether. So please keep in mind that you should always set pixel array tables to `Nil` as soon as you are done with them.

To convert a pixel array back to a brush, you can use the `RGBArrayToBrush()` function.

Note that for palette brushes, there is also the `BrushToPenArray()` function which will return the brush's pen values instead of RGB colors. See Section 21.9 [BrushToPenArray], page 255, for details.

INPUTS

`id` identifier of the brush to convert to RGB array

`invalpha` optional: whether to use inverted alpha values (defaults to `False` which means do not invert alpha values)

RESULTS

`table` a table containing all pixels from the source brush; do not forget to set this table to `Nil` when you are done with it!

21.11 ChangeBrushTransparency

NAME

ChangeBrushTransparency – change transparency mode of brush (V5.0)

SYNOPSIS

```
ChangeBrushTransparency(id, mode)
```

FUNCTION

This command can be used to change the transparency mode of a brush. Hollywood currently supports three different transparency modes:

`#NONE:` No transparency. The entire brush is visible.

`#MASK:` Monochrome transparency. Every pixel can either be visible or invisible.

`#ALPHACHANNEL:`
Gradual transparency. Every pixel can have 256 different levels of transparency. An alpha channel value of 0 means full transparency, whereas an alpha channel value of 255 means no transparency.

`ChangeBrushTransparency()` is especially useful for switching between the `#MASK` and `#ALPHACHANNEL` modes. For example, when you load a brush using `LoadBrush()` and you use the `Transparency` tag to make a color transparent, you will always get a brush that has a `#MASK` transparency mode. However, in some cases you might want the brush to use a mode of `#ALPHACHANNEL` instead; because you want to modify the values using `SelectAlphaChannel()`, for example. In that case, `ChangeBrushTransparency()` can be quite helpful.

Note that this function cannot be used with palette brushes.

INPUTS

`id` brush whose transparency mode you want to change

`mode` desired new transparency; can be `#NONE`, `#MASK`, or `#ALPHACHANNEL`

EXAMPLE

```
LoadBrush(1, "test.iff", {Transparency = #RED})
ChangeBrushTransparency(1, #ALPHACHANNEL)
```

The code above loads the image "test.iff" into brush 1, makes color red transparent, and then changes the transparency mode from `#MASK` to `#ALPHACHANNEL`.

21.12 CharcoalBrush

NAME

CharcoalBrush – apply charcoal drawing effect to brush (V5.0)

SYNOPSIS

```
CharcoalBrush(id, radius)
```

FUNCTION

This command applies a charcoal drawing effect to the specified brush. The `radius` argument specifies the charcoal radius. The larger the radius you specify here, the longer this function needs to calculate the resulting images.

Note that this function cannot be used with palette brushes.

INPUTS

`id`	brush to modify
`radius`	charcoal effect radius

21.13 ContrastBrush

NAME

ContrastBrush – enhance or reduce brush contrast (V5.0)

SYNOPSIS

```
ContrastBrush(id, inc[, repeat])
```

FUNCTION

This command can be used to enhance or reduce the color contrast in the specified brush. If the `inc` argument is set to `True`, the contrast is enhanced. If it is set to `False`, the contrast is reduced. The optional argument `repeat` can be used to apply the effect to the brush multiple times. This is useful if you want to create sharper contrasts.

Note that if `id` specifies a palette brush, `ContrastBrush()` will just apply the contrast to the palette colors which makes this function really fast when used with palette brushes.

INPUTS

`id`	brush to modify
`inc`	`True` to increase contrast, `False` to decrease contrast
`repeat`	optional: specifies how many times the contrast operation should be repeated (defaults to 1 which means run the effect just once)

21.14 ConvertToBrush

NAME

ConvertToBrush – convert object to brush (V2.5)

SYNOPSIS

```
[id] = ConvertToBrush(sourcetype, sourceid, dest[, t])
```

FUNCTION

This function allows you to create a new brush from an existing graphics object. This is useful, for example, to copy the image data from single anim or sprite frames to a brush. You could then modify them and convert them back into an animation or sprite. You can also access the graphics of layers and other image types with this function.

Having graphics as brushes is so convenient because brushes are the most flexible graphics type in Hollywood. Most of the image manipulating functions work only with brushes. That is why you will often want to convert your graphics data to the brush format.

The `sourcetype` argument specifies the type of the source object that shall be converted into a brush. It can be one of the following types:

`#ANIM` Create a new brush from a single frame from an anim object. By default, the first anim frame will be converted to a brush. You can change this by passing the `Frame` tag in the optional table argument (see below).

`#BGPIC` Create a new brush from a background picture.

`#BRUSH` Create a new brush from an other brush. This does the same as the `CopyBrush()` command.

`#ICON` Create a new brush from an image inside an icon. Since icons can contain multiple images, you can use the `Frame` tag of the optional table argument to specify the index of the image that should be converted to a brush. By default, the first image in the icon will be converted to a brush. You can also use the `Selected` tag of the optional table argument to specify whether or not the selected icon image should be converted to a brush. By default, the normal image will be converted to a brush. (V8.0)

This type requires you to pass the optional argument `par` which must be set to the index of icon image you wish to have converted into a brush. Icon image indices are counted from 1 until the number of images in the icon. Additionally, you can specify the optional argument `par2`: If you set this to `True`, the selected icon image will be converted to a brush, otherwise the normal image will be converted to a brush, which is also the default mode. (V8.0)

`#LAYER` Create a new brush from a layer (requires layers to be enabled!). If the layer is an anim layer, you can use the `Frame` tag in the optional table argument to specify which frame of the anim layer should be converted to a brush (see below). By default, the first frame will be converted.

`#SPRITE` Create a new brush from a single frame from a sprite object. By default, the first sprite frame will be converted to a brush. You can change this by passing the `Frame` tag in the optional table argument (see below).

`#TEXTOBJECT`
Create a new brush from a text object.

`#VECTORPATH`
Create a vector brush from one or more path object(s). If you use this type, the `sourceid` argument is unused. Instead, you need to pass a table argument in the `Path` tag in the optional table argument. This table must

contain information about the individual paths to be embedded inside the new vector brush. The table uses the same layout as the table you have to pass to the `PathToBrush()` function. See Section 56.26 [PathToBrush], page 1191, for details. (V7.0)

The optional table argument allows you to pass the following additional options:

`Frame:` If the source type specifies a graphics object that has multiple frames, you can use this tag to specify the frame that should be converted to a brush. Frames are counted from 1 until the number of frames. This tag defaults to 1.

`Selected:`
If the source type is `#ICON`, you can use this tag to specify whether the selected or normal image should be converted to a brush. Icon images have two states: normal and selected. If you set `Selected` to `True`, the selected image will be converted to a brush. Otherwise `ConvertToBrush()` will convert the normal image to a brush. Defaults to `False`. (V8.0)

`Path:` If the source type is `#VECTORPATH`, you must set this tag to a table which contains information about the individual paths to be embedded inside the new vector brush. The table uses the same layout as the table you have to pass to the `PathToBrush()` function. See Section 56.26 [PathToBrush], page 1191, for details. (V7.0)

`Vector:` By default, `ConvertToBrush()` will convert vector images to raster brushes. If you want to convert them to vector brushes, set this tag to `True`. This makes it possible to convert vector text objects or vector anim frames to vector brushes that can be scaled and rotated without any quality losses. Defaults to `False`. (V10.0)

INPUTS

`sourcetype`
type of the source object (see list above)

`sourceid` identifier of the source object

`dest` id for the brush to be created or `Nil` for auto id selection

`t` optional: table argument containing further options (see above)

RESULTS

`id` optional: handle to the new brush; will only be returned if you specified `Nil` in `dest`

EXAMPLE

```
ConvertToBrush(#SPRITE, 1, 10, {Frame = 5})
```

The code above creates a new brush with the id 10 from frame 5 of sprite number 1.

21.15 CopyBrush

NAME

CopyBrush – clone a brush (V1.5)

SYNOPSIS

```
[id] = CopyBrush(source, dest[, table])
```

FUNCTION

This function clones the brush specified by `source` and creates a copy of it in brush `dest`. The new brush is independent from the old brush so you could free the source brush after it has been cloned.

If you pass `Nil` as `dest`, `CopyBrush()` will return a handle to the new brush to you. Otherwise the new brush will use the identifier specified in `dest`.

Starting with Hollywood 5.0, this function accepts an optional table argument which accepts the following fields:

`Hardware:`
: If you set this tag to `True`, Hollywood will create this brush entirely in video memory for hardware-accelerated drawing in connection with a hardware double buffer. Hardware brushes are subject to several restrictions. See Section 21.37 [hardware brushes], page 280, for details. (V5.0)

`Display:`
: If you specify the identifier of a display here, Hollywood will create a display-dependent hardware brush for you. Display-dependent hardware brushes can only be drawn to the display they belong to. This tag is only handled if the `Hardware` tag has been set to `True`. Also note that Hollywood's inbuilt display adapter does not support display-dependent hardware brushes, but plugins can install custom display adapters which support display-dependent hardware brushes. This tag defaults to the identifier of the currently active display. See Section 21.37 [hardware brushes], page 280, for details. (V6.0)

`SmoothScale:`
: If you set this tag to `True` and the `Hardware` tag has also been set to `True`, Hollywood (or display adapters) will use bilinear interpolation when transforming the newly created brush. Normally, whether interpolation shall be used or not is set when calling a brush transformation command like `ScaleBrush()` or `RotateBrush()` but some display adapters need to know this information already at the time a hardware brush is created, and this is why this tag is here, though it's probably of not much use because it's only needed in rather special situations with display adapters like RebelSDL or hardware brushes on Android, because normally you can just specify whether interpolation shall be used or not in the transformation command directly. Note that `SmoothScale` is only supported when `Hardware` is set to `True`. (V8.0)

INPUTS

`source` source brush id

`dest` identifier of the brush to be created or `Nil` for auto id selection

`table` optional: table configuring further options (V5.0)

RESULTS

`id` optional: handle to the new brush; will only be returned if you specified `Nil` in `dest`

EXAMPLE

```
CopyBrush(1, 10)
FreeBrush(1)
```

The above code creates a new brush 10 which contains the same graphics data as brush 1. Then it frees brush 1 because it is no longer needed.

21.16 CreateBorderBrush

NAME

CreateBorderBrush – make border brush from brush (V5.0)

SYNOPSIS

```
[id] = CreateBorderBrush(id, src, color[, size])
```

FUNCTION

This command creates a border from the brush specified in `src` and copies that border to a new brush that is specified in `id`. If `id` is set to `Nil`, `CreateBorderBrush()` will automatically choose an identifier and return it to you. If `id` is not `Nil`, there will be no return value. The `color` argument must be set to the color that the border shall be drawn in. This must be a color in ARGB notation so you can also use a transparency setting here. Finally, the optional argument `size` can be used to specify the drop shadow's size.

Note that the size argument does not specify absolute width or height values but a relative factor by which the source brush will be grown on each side. This means that the border brush's width will be the source brush's width plus two times `size`, and the same applies to the border's height.

INPUTS

`id` identifier for the new border brush or `Nil` for auto id selection

`src` the brush whose border shall be generated

`color` desired border color as an ARGB value

`size` optional: desired border size (defaults to 5)

RESULTS

`id` optional: identifier of the border brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
bordersize = 20
CreateBrush(1, 320, 240, #RED)
CreateBorderBrush(2, 1, #BLACK, bordersize)
DisplayBrush(2, 0, 0)
```

```
DisplayBrush(1, bordersize, bordersize)
```

The code above creates a border for a 320x240 red rectangle and displays it beneath it.

21.17 CreateBrush

NAME

CreateBrush – create a blank brush (V1.5)

SYNOPSIS

```
[id] = CreateBrush(id, width, height[[, color], table])
```

FUNCTION

This function creates a new brush in the specified dimensions and initializes it to the specified color. If no color is specified, the brush is initialized to black. If you specify `Nil` in the `id` argument, `CreateBrush()` will automatically choose an identifier for this brush and return it to you.

Starting with Hollywood 4.5, there is an optional table argument which you can use to create a mask or an alpha channel for this brush. The following tags are recognized by the optional table:

`Mask:` Set this tag to `True` if `CreateBrush()` should attach a mask to the new brush. If this is `True`, `AlphaChannel` must be `False`. Defaults to `False`.

`AlphaChannel:`
Set this tag to `True` if `CreateBrush()` should attach an alpha channel to the new brush. If this is set to `True`, `Mask` must be set to `False`. Defaults to `False`.

`Clear:` This tag is only handled if either `AlphaChannel` or `Mask` was set to `True`. If that is the case, `Clear` specifies whether or not the mask or alpha channel should be cleared (i.e. fully transparent) or not (i.e. opaque). This defaults to `False` which means that by default, the new mask or alpha channel will be opaque.

`Hardware:`
If you set this tag to `True`, Hollywood will create this brush entirely in video memory for hardware-accelerated drawing in connection with a hardware double buffer. Hardware brushes are subject to several restrictions. See Section 21.37 [hardware brushes], page 280, for details. (V6.0)

`Display:` If you specify the identifier of a display here, Hollywood will create a display-dependent hardware brush for you. Display-dependent hardware brushes can only be drawn to the display they belong to. This tag is only handled if the `Hardware` tag has been set to `True`. Also note that Hollywood's inbuilt display adapter does not support display-dependent hardware brushes, but plugins can install custom display adapters which support display-dependent hardware brushes. This tag defaults to the identifier of the currently active display. See Section 21.37 [hardware brushes], page 280, for details. (V6.0)

`SmoothScale:`
: If you set this tag to `True` and the `Hardware` tag has also been set to `True`, Hollywood (or display adapters) will use bilinear interpolation when transforming this brush. Normally, whether interpolation shall be used or not is set when calling a brush transformation command like `ScaleBrush()` or `RotateBrush()` but some display adapters need to know this information already at the time a hardware brush is created, and this is why this tag is here, though it's probably of not much use because it's only needed in rather special situations with display adapters like RebelSDL or hardware brushes on Android, because normally you can just specify whether interpolation shall be used or not in the transformation command directly. Note that `SmoothScale` is only supported when `Hardware` is set to `True`. (V8.0)

`Palette:`
: If this tag is set to the identifier of a palette, Hollywood will create a palette brush for you. Palettes can be created using functions like `CreatePalette()` or `LoadPalette()`. Alternatively, you can also set this tag to one of Hollywood's inbuilt palettes, e.g. `#PALETTE_AGA`. See Section 44.36 [SetStandardPalette], page 918, for a list of inbuilt palettes. (V9.0)

`FillPen:`
: If the `Palette` tag is set (see above), you can use this tag to set the pen that should be used for filling the brush's background. Note that the `color` parameter that is passed to `CreateBrush()` is ignored if `Palette` is `True`. That's why this tag is here to allow you to specify a pen that will be used when initializing the brush's pixels. Defaults to 0. (V9.0)

`TransparentPen:`
: If `Palette` is set to `True`, this tag can be used to specify a pen that should be made transparent in the new brush. Defaults to `#NOPEN` which means that there should be no transparent pen. (V9.0)

`Depth:`
: This tag allows you to set the desired brush depth. If this is less than or equal to 8, `CreateBrush()` will create a palette brush. You can also specify the `Palette` tag together with the `Depth` tag. If the specified palette has more colors than the specified depth, those colors will be discarded. If it has less colors, the unused pens will be set to black. By default, `CreateBrush()` will create 24-bit or 32-bit brushes, depending on whether the `AlphaChannel` tag is set to `True` or `False`. (V10.0)

`Callback:`
: If you set this tag to a callback function, `CreateBrush()` will create a custom-drawn brush for you. In comparison to normal brushes, custom-drawn brushes are backed by a callback function that Hollywood will call whenever the dimensions of the brush change or a transformation is applied. This allows you to create brushes which dynamically adapt themselves to new resolutions and transformations. This is very similar to what vector brushes do except that custom-drawn brushes allow you to assume full control over the process because your callback does all the drawing. Your callback will also be called immediately by `CreateBrush()` to do the initial drawing of the brush.

Custom-drawn brushes can be very useful to implement your own vector brush type. Since custom-drawn brushes get the chance to redraw themselves whenever their resolution or transformation changes, you can use them to create brushes which can be scaled or transformed without any losses in quality because your callback redraws the graphics whenever the brush dimensions change instead of just scaling its pixels using conventional lossy scaling. This is especially useful when using the layerscale engine with custom graphics. If you use custom-drawn brushes for a layer, you can be sure that that layer will scale losslessly to all resolutions.

The callback function you specify in `Callback` will receive a message as parameter 1 with the following fields initialized:

`Action:` Initialized to `Draw`.

`ID:` Identifier of the brush to draw to. Note that this won't be the same identifier as the brush you created using `CreateBrush()`. Your callback needs to call `SelectBrush()` on the brush specified in `ID` and draw the desired graphics to the brush, taking the current transformation into account (see below). You can also call `SelectMask()` and `SelectAlphaChannel()` on the brush in case you need to adjust transparency setting.

`Width:` Current width of the brush.

`Height:` Current height of the brush.

`SX:` Specifies the amount of scaling on the x axis. If it is negative, the image is flipped on the y axis. This will never be 0.

`RX:` Specifies the amount of rotation on the x axis. This can be 0.

`RY:` Specifies the amount of rotation on the y axis. This can be 0.

`SY:` Specifies the amount of scaling on the y axis. If it is negative, the image is flipped on the x axis. This will never be 0.

`UserData:`
This will be set to the user data you passed in the `UserData` tag of the optional table argument supported by `CreateBrush()` (see below). If you haven't passed any user data in your call to `CreateBrush()`, this tag won't be set.

(V10.0)

`UserData:`
If you have set the `Callback` tag (see above), you can use this tag to store some user data that will be passed to your callback whenever Hollywood calls it. The user data can be of any type. (V10.0)

INPUTS

`id` id for the new brush or `Nil` for auto id selection

`width` width for the brush

`height` height for the brush

`color` optional: RGB color for background (defaults to `#BLACK`)

`table` optional: table for specifying further options (see above) (V4.5)

RESULTS

`id` optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateBrush(2, 320, 256, #BLUE)
```

The above code creates a new blue brush with the id 2 and the dimension of 320x256.

```
CreateBrush(2, 320, 256, #BLUE, {AlphaChannel = True, Clear = True})
```

The code above creates a new blue brush with id 2 in a size of 320x256. The new brush will also get an alpha channel that will be set to 100% transparent. Thus, if you display the new brush, you will see nothing because the brush is currently fully transparent.

```
CreateBrush(1, 320, 240, 0, {AlphaChannel = True, Clear = True, Callback =
    Function(msg)
        SelectBrush(msg.id, #SELMODE_COMBO)
        SetFormStyle(#ANTIALIAS)
        SetFillStyle(#FILLCOLOR)
        Ellipse(0, 0, msg.width / 2, msg.height / 2, #RED)
        EndSelect
    EndFunction
})
ScaleBrush(1, 640, 480)
DisplayBrush(1, 0, 0)
```

The code above demonstrates how to create a custom-drawn brush. It creates a brush that draws an anti-aliased ellipse in its callback function. Since the callback function is invoked whenever the brush's dimensions change, the ellipse will scale like a true vector image and will be perfectly crisp in all resolutions.

21.18 CreateGradientBrush

NAME

CreateGradientBrush – create brush with gradient fill (V5.0)

SYNOPSIS

```
[id] = CreateGradientBrush(id, width, height, type, startcolor,
                           endcolor[, angle, table])
```

FUNCTION

This function can be used to create a new brush which is initialized to a gradient backfill. If you specify `Nil` in the `id` argument, this function will choose an identifier for this brush automatically and return it to you. The `width` and `height` arguments specify the desired

dimensions for the new brush. The `type` argument specifies the type of the gradient you want to use. The following gradient types are currently available: `#LINEAR`, `#RADIAL`, and `#CONICAL`. The `angle` argument allows you to specify a rotation angle (in degrees) for the gradient. The angle argument is only supported by gradients of type `#LINEAR` and `#CONICAL`. Radial gradients cannot be rotated.

The optional table argument can be used to specify advanced options. The following tags are currently recognized:

`CenterX, CenterY:`
: These two tags can be used to specify the center point of the gradient. As linear gradients do not have a center point, these two tags are only handled when you use gradients of type `#RADIAL` or `#CONICAL`. The center point must be specified as a floating point value that is between 0.0 (left/top corner) and 1.0 (right/bottom corner). If not specified, both tags default to 0.5 which means that the center point of the gradient is in the center of the image.

`Border:` This tag can be used to set the border size for gradients of type `#RADIAL`. For the other gradient types this tag is ignored. The border size of the radial gradient must be a floating point value between 0.0 and 1.0. Defaults to 0.0 which means no border.

`Balance:` This tag can be used to set the balance point for gradients of type `#CONICAL`. For the other gradient types this tag is ignored. The balance point of the conical gradient must be floating point value between 0.0 and 1.0. Defaults to 0.5. Note that this is only used when creating a two-color gradient. When creating a multi-color gradient using the `Colors` table, `Balance` is ignored because the `Colors` table allows you to individually balance the colors in the gradient using color stops.

`Colors:` This tag allows you to create gradients that contain multiple colors. This tag must be set to a table that contains a sequence of alternating color and stop values. The colors must be specified in RGB format. The stop value is a floating point value between 0.0 and 1.0 and defines the position where the corresponding color should be merged into the gradient. A position of 0.0 means the start position of the gradient, and a position of 1.0 means the end position. Please note that the stop positions must be sorted in ascending order, i.e. starting from 0.0 to 1.0. If you specify this tag, the colors specified in the `startcolor` and `endcolor` arguments are ignored, and Hollywood will only use the colors specified in this tag.

INPUTS

`id` id for the new brush or `Nil` for auto ID select

`width` desired width for the new brush

`height` desired height for the new brush

`type` type of the gradient; see above for available types

`startcolor`
: RGB value defining the start color

`endcolor` RGB value defining the end color

`angle` optional: rotation angle for the gradient (default: 0)

`table` optional: table argument specifying further options; see above for a description of available options

RESULTS

`id` optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

See Section 20.6 [CreateGradientBGPic], page 232.

21.19 CreateShadowBrush

NAME

CreateShadowBrush – make shadow brush from brush (V5.0)

SYNOPSIS

```
[id] = CreateShadowBrush(id, src, color[, size])
```

FUNCTION

This command creates a drop shadow from the brush specified in `src` and copies that shadow to a new brush that is specified in `id`. If `id` is set to `Nil`, `CreateShadowBrush()` will automatically choose an identifier and return it to you. If `id` is not `Nil`, there will be no return value. The `color` argument must be set to the color that the shadow shall be drawn in. In most cases this will be a plain `#BLACK` but combined with a transparency value because opaque shadows do not look very good. You can use the `ARGB()` function to combine a color and a transparency value into an ARGB color. Finally, the optional argument `size` can be used to specify the drop shadow's size.

Note that the size argument does not specify absolute width or height values but a relative factor by which the source brush will be grown on each side. This means that the drop shadow brush's width will be the source brush's width plus two times `size`, and the same applies to the drop shadow's height.

INPUTS

`id` identifier for the new drop shadow brush function or `Nil` forauto id selection

`src` the brush to convert into a shadow

`color` desired shadow color as an ARGB value

`size` optional: desired shadow size (defaults to 5)

RESULTS

`id` optional: identifier of the drop shadow brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
shadowsize = 20
CreateBrush(1, 320, 240, #RED)
```

```
CreateShadowBrush(2, 1, ARGB(40, #BLACK), shadowsize)
DisplayBrush(2, 0, 0)
DisplayBrush(1, shadowsize, shadowsize)
```

The code above creates a drop shadow for a 320x240 red rectangle and displays it beneath it.

21.20 CreateTexturedBrush

NAME

CreateTexturedBrush – create a textured brush (V5.0)

SYNOPSIS

```
[id] = CreateTexturedBrush(id, brushid, width, height[, x, y])
```

FUNCTION

This function will create a new brush for you and it will texture it with the brush specified by `brushid`. If you specify `Nil` in the `id` argument, this function will choose an identifier for the new brush automatically and return it to you. The `width` and `height` arguments specify the desired dimensions for the new brush. The optional `x` and `y` parameters allow you to specify an offset into the texture brush. Texturing will then start from this offset in the brush. The default for these arguments is 0/0 which means start at the top-left corner inside the texture brush.

INPUTS

`id`	id for the new brush or `Nil` for auto ID select
`brushid`	identifier of the brush to be used as the texture
`width`	desired width for the new brush
`height`	desired height for the new brush
`x`	optional: start x offset in the texture brush
`y`	optional: start y offset in the texture brush

RESULTS

`id`	optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

21.21 CropBrush

NAME

CropBrush – crop a brush (V2.0)

SYNOPSIS

```
CropBrush(id, x, y, width, height)
```

FUNCTION

This function crops the brush specified by `id` at the position specified by `x` and `y` to the new dimension specified by `width` and `height`. If the brush has a mask and/or an alpha channel, they will be cropped as well.

INPUTS

`id`	brush to crop
`x`	x-position where to start cropping
`y`	y-position where to start cropping
`width`	crop width
`height`	crop height

EXAMPLE

```
CreateBrush(1, 200, 200)
SelectBrush(1)
Circle(50, 50, 50, #RED)
EndSelect

CropBrush(1, 50, 50, 101, 101)
DisplayBrush(1, #CENTER, #CENTER)
```

Creates a new brush and draws a red circle in the center of it. After that, the empty area surrounding the circle will be cropped.

21.22 DeleteAlphaChannel

NAME

DeleteAlphaChannel – remove a brush's alpha channel (V2.0)

SYNOPSIS

```
DeleteAlphaChannel(id)
```

FUNCTION

This function kills the alpha channel of the brush specified by `id`. See Section 21.63 [SelectAlphaChannel], page 301, for more information on alpha channels in general.

INPUTS

`id`	brush whose alpha channel is to be deleted

21.23 DeleteMask

NAME

DeleteMask – remove a brush's mask (V2.0)

SYNOPSIS

```
DeleteMask(id)
```

FUNCTION

This function kills the mask of the brush specified by `id`, i.e. the brush will appear opaque then.

INPUTS

`id`	brush whose mask is to be deleted

21.24 DisplayBrush

NAME

DisplayBrush – display a brush

SYNOPSIS

```
DisplayBrush(id, x, y[, table])
```

FUNCTION

This function displays the brush specified by `id` at the coordinates specified by `x` and `y`.

If layers are enabled, this command will add a new layer of the type `#BRUSH` to the layer stack.

New in Hollywood 4.0: This command has an optional table argument now which allows you to specify one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`id`	identifier of the brush to display
`x`	x offset to display
`y`	y offset to display
`table`	optional: table specifying further options (V4.0)

EXAMPLE

```
DisplayBrush(1, #CENTER, #CENTER)
```

Displays brush 1 centered on the screen.

```
DisplayBrush(1, 0, 0, {Width = 640, Height = 480})
```

Displays brush 1 scaled to 640x480.

21.25 DisplayBrushFX

NAME

DisplayBrushFX – display a brush with transition effects

SYNOPSIS

```
[handle] = DisplayBrushFX(id, x, y[, table])
```

FUNCTION

This function is an extended version of the `DisplayBrush()` command. It displays the brush specified by `id` at the position specified by `x`,`y` and it uses one of the many Hollywood transition effects to display it. You need also specify the speed for the transition.

If layers are enabled, this command will add a new layer of the type `#BRUSH` to the layer stack.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The

optional table argument can be used to configure the transition effect. The following options are possible:

`Type:` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of all supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed:` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter:` Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async:` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `DisplayBrushFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

INPUTS

`id` identifier of the brush to display

`x` desired x position for the brush

`y` desired y position for the brush

`table` optional: configuration for the transition effect

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
DisplayBrushFX(1, 0, 0, #VLINES, 10)   ; old syntax

OR

DisplayBrushFX(1, 0, 0, {Type = #VLINES, Speed = 10})  ; new syntax
```

The above code displays brush 1 at 0:0 with a `#VLINES` transition at speed 10.

21.26 DisplayBrushPart

NAME

DisplayBrushPart – display a part of a brush

SYNOPSIS

```
DisplayBrushPart(id, srcx, srcy, destx, desty, width, height[, table])
```

FUNCTION

This function displays a tile of the brush specified by `id` on the screen. The tile is defined by `srcx` and `srcy` and its `width` and `height` and it is displayed on the display at the position specified by `destx` and `desty`.

If layers are enabled, this command will add a new layer of the type `#BRUSHPART` to the layer stack.

New in Hollywood 4.0: This command has an optional table argument now which allows you to specify one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`id`	identifier of the brush to use as source
`srcx`	left corner in the brush
`srcy`	top corner in the brush
`destx`	desired x position for the brush on the screen
`desty`	desired y position for the brush on the screen
`width`	width of the tile
`height`	height of the tile
`table`	optional: table specifying further options (V4.0)

EXAMPLE

```
DisplayBrushPart(1,0,0,50,50,100,100)
```

Display the first 100 pixels and rows from brush 2 on the screen at the position 50,50.

21.27 EdgeBrush

NAME

EdgeBrush – detect edges within a brush (V5.0)

SYNOPSIS

```
EdgeBrush(id[, radius])
```

FUNCTION

This command can be used to detect edges with a brush. The optional argument `radius` can be used to specify the search radius. The larger the radius you specify here, the longer this function needs to execute. If you do not specify the optional argument, `EdgeBrush()` will choose an appropriate radius automatically.

Note that this function cannot be used with palette brushes.

INPUTS

`id`	brush to modify
`radius`	optional: search radius (defaults to 0 which means that the radius will be chosen automatically)

21.28 EmbossBrush

NAME
EmbossBrush – emboss brush (V5.0)

SYNOPSIS
`EmbossBrush(id[, radius])`

FUNCTION
This command applies an emboss effect to the specified brush. The optional argument `radius` can be used to specify the emboss radius. The larger the radius you specify here, the longer this function needs to apply the effect. If you do not specify the optional argument, `EmbossBrush()` will automatically choose an emboss radius.

Note that this function cannot be used with palette brushes.

INPUTS

`id`	brush to emboss
`radius`	optional: emboss radius (defaults to 0 which means that the radius will be chosen automatically)

21.29 EndSelect

NAME
EndSelect – select display as output device (V1.5)

SYNOPSIS
`EndSelect()`

FUNCTION
This function cancels the offscreen-rendering mode by selecting the display as the output device again. All commands which output graphics will now render directly to the display again.

Note that `EndSelect()` cannot be used with `SelectDisplay()`.

Please note that the SelectXXX()/`EndSelect()` calls do not nest, i.e. `EndSelect()` will always select the display as the output device, even if it was not the output device when SelectXXX() was called.

```
SelectBrush(1)
Box(0, 0, 100, 100, #RED)   ; draw box on brush #1
SelectBrush(2)
Box(0, 0, 100, 100, #GREEN) ; draw box on brush #2
EndSelect                   ; selects display (not brush #1 !!!)
EndSelect                   ; Error! Display already selected!
```

INPUTS
none

EXAMPLE
See Section 20.15 [SelectBGPic], page 246.

See Section 21.64 [SelectBrush], page 303.
See Section 17.22 [SelectAnim], page 201.
See Section 34.39 [SelectLayer], page 676.

21.30 ExtendBrush

NAME

ExtendBrush – enlarge a brush (V10.0)

SYNOPSIS

```
ExtendBrush(id, left, top, right, bottom[, t])
```

FUNCTION

This function enlarges the brush specified by `id`. You can specify the desired extension for all sides of the brush by passing a pixel value in the `left`, `top`, `right` and `bottom` parameters. A value of 0 means no extension on that side. Note that the extension values mustn't be negative. If you want to crop a brush, use the `CropBrush()` function instead.

The optional table argument can be used to specify the following additional options:

`Clear:` If the brush uses transparency, you can set this tag to `True` if you want the newly allocated areas of the brush to be transparent as well. If you set this tag to `False`, the new areas will be opaque. Defaults to `False`.

`Color:` This tag can be used to specify the filling color for the newly allocated areas of the brush. This is only used for RGB brushes. For palette brushes, use the `Pen` tag instead (see below). Defaults to `#BLACK`.

`Pen:` This tag can be used to specify the filling pen for the newly allocated areas of the brush. This is only used for palette brushes. For RGB brushes, use the `Color` tag instead (see above). Defaults to 0.

INPUTS

`id` brush to enlarge

`left` number of pixels to add on the left side

`top` number of pixels to add on the top side

`right` number of pixels to add on the right side

`bottom` number of pixels to add on the bottom side

`t` optional: table specifying further options (see above)

EXAMPLE

```
CreateBrush(1, 200, 200)
ExtendBrush(1, 50, 50, 50, 50, {Color = #RED})
```

The code above creates a 200x200 pixel brush and then adds a red border of 50 pixels around it.

21.31 FlipBrush

NAME

FlipBrush – flip a brush (V1.5)

SYNOPSIS

```
FlipBrush(id, xflip)
```

FUNCTION

This function flips (mirrors) the brush specified by `id`. If `xflip` is set to `True`, the brush will be flipped in x-direction otherwise it will be flipped in y-direction.

INPUTS

`id`	brush to flip
`xflip`	`True` for horizontal (x) flip, `False` for vertical (y) flip

EXAMPLE

```
FlipBrush(1, TRUE)
```

The code above flips the brush horizontally.

21.32 FloodFill

NAME

FloodFill – flood fill a brush area with a color (V2.0)

SYNOPSIS

```
FloodFill(id, x, y, bordercolor, color[, t])
```

FUNCTION

This function can be used to flood fill a bordered area in the brush specified by `id` with a color. You need to pass a starting position in the `x` and `y` arguments. `FloodFill()` will then start out in all directions replacing all pixels with the specified color until it reaches the border color which you also have to specify. The starting position is usually an arbitrary point within the bounded area.

If the brush is a palette brush, `FloodFill()` will operate in pen mode instead of color mode. This means that both the `bordercolor` and the `color` arguments must be set to a pen index instead of an RGB color. In pen mode, `FloodFill()` will behave exactly as in RGB mode except that it will use pens instead of RGB colors.

Starting with Hollywood 9.0, you can also pass `#NOCOLOR` in the `bordercolor` argument. In that case, borderless flood filling will be used, which means that all neighbouring pixels matching the color (or pen) of the starting pixel will be filled.

Furthermore, Hollywood 9.0 introduces an optional table argument that allows you to specify the following options:

`AlphaChannel:`

If you set this tag to `True`, `FloodFill()` will operate on the brush's alpha channel instead of on its color channels. This means that you have to pass values in the range of 0 to 255 instead of RGB colors to `FloodFill()`.

`ColorSource:`
If this and the `AlphaChannel` tag is set to `True`, the area to be filled will be determined by the color channels whereas all output will be written to the alpha channel. This means that the border color passed to `FloodFill()` must be an RGB color (or `#NOCOLOR`) and the fill color must be an alpha value between 0 and 255. (V9.1)

INPUTS

`id` brush to use for flood fill

`x` start x-position for the fill operation

`y` start y-position for the fill operation

`bordercolor`
color (or pen) of the border or `#NOCOLOR` for borderless flood filling

`color` color (or pen) to use for filling

`t` optional: table argument containing further options (V9.0)

EXAMPLE

```
CreateBrush(1, 241, 201)
SelectBrush(1)
SetFillStyle(#FILLNONE)
Ellipse(0, 0, 120, 100, #RED)
EndSelect
FloodFill(1, 120, 100, #RED, #WHITE)
DisplayBrush(1, 0, 0)
```

Creates a red ellipse outline and then fills it with the color white using the `FloodFill()` command starting from the center of the ellipse in all directions.

21.33 FreeBrush

NAME

FreeBrush – free a brush

SYNOPSIS

```
FreeBrush(id)
```

FUNCTION

This function frees the memory of the brush specified by `id`. To reduce memory consumption, you should free brushes when you do not need them any longer.

INPUTS

`id` identifier of the brush

21.34 GammaBrush

NAME

GammaBrush – correct gamma values of brush (V5.0)

SYNOPSIS

```
GammaBrush(id, red, green, blue)
```

FUNCTION

This function can be used to gamma correct the color channels of the specified brush. For each color channel, you have to pass a floating point value that specifies the desired gamma correction. A value of 1.0 means no change, a value smaller than 1.0 darkens the channel, a value greater than 1.0 lightens the channel.

Note that if `id` specifies a palette brush, InvertBrush() will just apply the gamma correction to the palette colors which makes this function really fast when used with palette brushes.

INPUTS

`id`	brush to gamma correct
`red`	gamma correction for red channel
`green`	gamma correction for green channel
`blue`	gamma correction for blue channel

EXAMPLE

```
GammaBrush(1, 1.5, 1.0, 0.5)
```

The code above lightens the red channel and darkens the blue channel, while leaving the green color channel untouched.

21.35 GetBrushLink

NAME

GetBrushLink – get a link to a brush (V1.5)

SYNOPSIS

```
GetBrushLink(id, sourcetype, sourceid[, par])
```

FUNCTION

Hollywood 2.0 Note: Brush links are no longer supported. You can still use this function but it will not create links any more. It will simply create a full copy of the image data; in other words: `GetBrushLink()` just calls `ConvertToBrush()`.

This function creates a new brush for you which will link data from an other object. Therefore the new brush will be read-only. This means e.g. that you can display or move it, but you cannot change its data (e.g. by calling `ScaleBrush()` or `SelectBrush()`).

Your brush is fully dependent on the source object. If you free the source object, your brush will also be freed and is no longer available. Brush links require only little bytes of memory because the graphics data will be linked from the source object.

It is useful to use brush links when you have many objects with the same graphics data and you want to access them as separate brushes (e.g. for convenience reasons). Another good reason for brush links is that you can do a lot of more stuff with brushes than with other objects. For example you could retrieve a link to the first anim frame, then display it with `DisplayBrushFX()` and then start the anim with `PlayAnim()`. This would display the anim with a transition effect then.

Sourcetype can be one of the following types:

`#ANIM` Get brush links from single anim frames; this type requires the optional argument `par` which specifies the frame you want to have linked

`#BGPIC` Get brush link from a background picture

`#BRUSH` Get brush link from an other brush

`#LAYER` Get brush link from a layer (requires layers to be enabled!)

`#TEXTOBJECT`
Get brush link from a text object

INPUTS

`id` identifier for the brush to be created

`sourcetype`
type of the source object (see list above)

`sourceid` identifier of the source object

`par` optional: currently only required for type `#ANIM`

EXAMPLE

```
LoadAnim(1, "MyAnim.gif")
GetBrushLink(1, #ANIM, 1, 1)
DisplayBrushFX(1, #CENTER, #CENTER, #CROSSFADE)
PlayAnim(1, #CENTER, #CENTER)
```

The above code loads the animation "MyAnim.gif", gets a brush link to the first frame, crossfades this frame on to the display and then starts playing the anim. This is how you would display an anim with a transition effect.

21.36 GetBrushPen

NAME

GetBrushPen – get pen color from brush's palette (V9.0)

SYNOPSIS

```
color = GetBrushPen(id, pen)
```

FUNCTION

This function gets the color of the pen specified by `pen` from the palette of the brush specified by `id`. The color will be returned as an RGB color.

INPUTS

`id` identifier of brush to use

`pen` pen you want to get (starting from 0)

RESULTS

`color` color of the pen, specified as an RGB color

EXAMPLE

```
color = GetBrushPen(1, 0)
```

The code gets the color of the first pen of brush 1.

21.37 Hardware brushes

Hardware brushes are used for hardware-accelerated drawing in connection with a hardware double buffer. To create a hardware brush, simply set the `Hardware` tag to `True` in `LoadBrush()` or `@BRUSH`. Alternatively, you can also create a hardware brush from a normal brush by using `CopyBrush()` and setting the `Hardware` tag to `True` in this function. To find out whether or not a brush has been successfully created in hardware, query the `#ATTRHARDWARE` attribute using `GetAttribute()`. Note that this attribute can return `False` even if you set the `Hardware` tag to `True`, because not all systems support hardware brushes. If the system Hollywood is running on does not support hardware brushes, a software brush will be created instead.

The advantage of hardware brushes is that they are stored completely in video memory and thus can be drawn extremely quickly. However, hardware brushes can only be drawn to hardware-accelerated double buffers. Nothing else can be done with hardware brushes than drawing them to a hardware-accelerated double buffer. That is why almost all functions of the brush library will not work with hardware brushes. If you would like to modify a hardware brush, you first have to create a software brush which you can modify and then convert it to a hardware brush. You can create a hardware-accelerated double buffer by passing `True` as the first argument to the `BeginDoubleBuffer()` function. See Section 30.3 [BeginDoubleBuffer], page 590, for details.

Keep in mind that you should only draw to hardware-accelerated double buffers using hardware brushes. All other drawing commands will be much slower! Only by using hardware brushes can you get full hardware-accelerated drawing. Using normal drawing functions with a hardware double buffer can even be slower than using them on a software double buffer. This is especially the case with graphics that use an alpha channel, e.g. anti-aliased text or vector shapes, because for alpha channel drawing, Hollywood has to read from the destination device which will be very slow for hardware double buffers because reading from video memory is very slow. Thus, you should try to use hardware brushes whereever possible when you work with a hardware double buffer.

On some systems (e.g AmigaOS 4.1) the `ScaleBrush()`, `RotateBrush()`, and `TransformBrush()` functions as well as the standard draw tags for on-the-fly image manipulation (e.g. `ScaleX` and `ScaleY`) support hardware accelerated image scaling/transformation for hardware brushes. In that case, scaling and transforming brushes is extremely faster than in software mode, especially for antialiased transformations.

By default, hardware brushes are only supported on AmigaOS and Android. Since Hollywood 6.0, however, plugins that install a display adapter are also able to support hardware

brushes for their display adapter. In that case you can also use hardware brushes on systems other than AmigaOS and Android. For example, the GL Galore and RebelSDL plugins allow you to use hardware brushes and hardware-accelerated double buffers on Windows, macOS, and Linux. See Section 5.4 [Obtaining plugins], page 66, for details.

Hardware brushes can also be display-dependent. This means that they can only be drawn to the display that has been used to allocate them. This is often the case when using custom display adapters made available by plugins. Hollywood's inbuilt hardware brushes on AmigaOS and Android, however, are not display-dependent and can be drawn to any display that is currently open. To allocate a display-dependent hardware brush, you need to pass the identifier of the display that should own the hardware brush in the `Display` tag in `LoadBrush()`, `@BRUSH` or `CopyBrush()`. Note that all display-dependent hardware brushes are automatically freed by Hollywood when the display they belong to is closed.

21.38 InvertAlphaChannel

NAME

InvertAlphaChannel – invert alpha channel of a brush (V2.0)

SYNOPSIS

```
InvertAlphaChannel(id)
```

FUNCTION

This function inverts the alpha channel of the brush specified by `id`. This means that the transparency for each pixel is turned around. If a pixel was previously 80% transparent, it will only be 20% transparent after an inversion and pixels who were 20% transparent will become 80% transparent after the inversion.

INPUTS

`id`	brush whose alpha channel is to be inverted

21.39 InvertBrush

NAME

InvertBrush – invert colors of a brush (V1.5)

SYNOPSIS

```
InvertBrush(id)
```

FUNCTION

This function inverts the brush specified by `id`, which means that all colors are replaced with their complements (white will become black, blue will become yellow etc.).

Note that if `id` specifies a palette brush, `InvertBrush()` will just invert the palette colors which makes this function really fast when used with palette brushes.

INPUTS

`id`	brush to invert

EXAMPLE

```
InvertBrush(1)
```

The code above inverts the colors of brush 1.

21.40 InvertMask

NAME

InvertMask – invert mask of a brush (V2.0)

SYNOPSIS

```
InvertMask(id)
```

FUNCTION

This function inverts the mask of the brush specified by `id`. This means that all areas which were previously transparent, will become opaque and areas that were opaque previously, will become transparent.

INPUTS

`id`	brush whose mask is to be inverted

21.41 IsBrushEmpty

NAME

IsBrushEmpty – check if a brush has only invisible pixels (V8.0)

SYNOPSIS

```
r = IsBrushEmpty(id)
```

FUNCTION

This function can be used to check if the brush specified in `id` has only invisible pixels, in which case it can be considered "empty". If there are only invisible pixels in the brush, `True` is returned, `False` otherwise.

Obviously, a brush can only be "empty" if it uses some kind of transparency, either a mask or an alpha channel. If you call this function on a brush that has neither a mask nor an alpha channel attached, the return value will always be `False`.

INPUTS

`id`	brush to check

RESULTS

`r`	`True` if there are only invisible pixels in the brush, `False` otherwise

EXAMPLE

```
CreateBrush(1, 100, 100, #RED, {Mask = True, Clear = True})
Print(IsBrushEmpty(1))
```

The code above will print 1 because although the brush is filled with red pixels, none of them will be visible because the mask has all pixels set to invisible.

21.42 LoadBrush

NAME

LoadBrush – load a brush

SYNOPSIS

```
[id] = LoadBrush(id, filename$[, table])
```

FUNCTION

This function loads the brush specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadBrush()` will automatically choose an identifier and return it.

Image formats that are supported on all platforms are PNG, JPEG, BMP, IFF ILBM, GIF, and image formats you have a plugin for. Depending on the platform Hollywood is running on, more image formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all image formats you have datatypes for as well. On Windows, `LoadBrush()` can also load image formats supported by the Windows Imaging Component.

Starting with Hollywood 5.0, this function can also load vector formats like SVG if you have an appropriate plugin installed. Keep in mind, though, that when you load vector images using `LoadBrush()`, the brush will be a special vector brush which does not support all features of the normal brushes. You can, however, convert vector brushes to raster brushes by using the `RasterizeBrush()` function. See Section 21.80 [Vector brushes], page 316, for more information on vector brushes.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall be made transparent in the brush.

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the image shall be loaded, too. Please note that not all pictures have an alpha channel and that not all picture formats are capable of storing alpha channel information. It is suggested that you use the PNG format if you need alpha channel data. This field defaults to `False`.

`X, Y, Width, Height:`
: These fields can be used to load only a part of the image into the brush. This is useful if you have one big image with many different small images in it and now you want to load the small images into single brushes. Using these fields you can specify a rectangle inside the image from which Hollywood will take the graphics data for the brush.

`Hardware:`
: If you set this tag to `True`, Hollywood will create this brush entirely in video memory for hardware-accelerated drawing in connection with a hardware double buffer. Hardware brushes are subject to several restrictions. See Section 21.37 [hardware brushes], page 280, for details. (V5.0)

`ScaleWidth, ScaleHeight:`
: These fields can be used to load a scaled version of the image. If the image driver supports scaled loading, this will give you some significant speed-up for example in case you just want to load a thumbnail-sized version of a large image. If the image driver does not support scaled loading, the full image will be loaded first before it is scaled. This is not much faster than manually scaling the image after loading. You can pass an absolute pixel value or a string containing a percent specification here. (V5.3)

`SmoothScale:`
: If `ScaleWidth` or `ScaleHeight` is set, you can use this item to specify whether or not Hollywood shall use anti-aliased scaling. Defaults to `False` which means no anti-aliasing. Note that anti-aliased scaling is much slower than normal scaling. (V5.3)

`Display:`
: If you specify the identifier of a display here, Hollywood will create a display-dependent hardware brush for you. Display-dependent hardware brushes can only be drawn to the display they belong to. This tag is only handled if the `Hardware` tag has been set to `True`. Also note that Hollywood's inbuilt display adapter does not support display-dependent hardware brushes, but plugins can install custom display adapters which support display-dependent hardware brushes. This tag defaults to the identifier of the currently active display. See Section 21.37 [hardware brushes], page 280, for details. (V6.0)

`Loader:`
: This tag allows you to specify one or more format loaders that should be asked to load this brush. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:`
: This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
: If this tag is set to `True`, the monochrome transparency of the image will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based image. If you want to load the alphachannel of an image, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
: If this tag is set to `True`, Hollywood will load the brush as a palette brush. This means that you can get and modify the brush's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette brushes also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`. (V9.0)

`TransparentPen:`
: If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the image file (if the image format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
: This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. A brush can only have one transparency setting!

This command is also available from the preprocessor: Use `@BRUSH` to preload brushes!

INPUTS

`id` identifier for the brush or `Nil` for auto id selection

`filename$`
: file to load

`table` optional: transparency and crop options (see above) (V2.0)

RESULTS

`id` optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadBrush(2, "MyBrush.png", {Transparency = #RED})
```

This loads "MyBrush.png" as brush 2 with the color red being transparent.

21.43 Mask and alpha channel

Hollywood supports two kinds of transparency for its graphics objects: Mask transparency and alpha transparency. This section will explain the difference between the two.

Mask transparency knows only two settings per pixel: Visible and invisible. Alpha transparency, however, supports 256 different levels of transparency for every pixel. An alpha level of 0 means that the pixel is not visible, and a level of 255 means that the pixel is fully visible. A level of 128 thus means that a pixel is only 50% visible. Alpha transparency is very useful if you want to embed images that shall smoothly adapt to your background. For example, a brush with a shadow where the background shines through, or a brush with antialiased edges. For such purposes a mask is not enough.

Please note also that mask and alpha channel are mutually exclusive. That is, a brush cannot have a mask and an alpha channel but only one of the two.

21.44 MixBrush

NAME

MixBrush – mix two brushes (V1.5)

SYNOPSIS

```
MixBrush(brush1, brush2, level)
```

FUNCTION

This function mixes `brush2` into `brush1` at the specified `level`. The argument `level` specifies the mixing level which ranges from 0 to 255. Alpha channel and mask data of the second brush are also taken into account.

Starting with Hollywood 2.0, level can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`brush1` source brush

`brush2` brush to mix

`level` mixing level (0 to 255 or percent specification)

EXAMPLE

```
MixBrush(1, 2, 128)
```

The code above mixes brush 2 into brush 1 at a mix ratio of 50% (= 128).

21.45 ModulateBrush

NAME

ModulateBrush – change brightness, saturation, and hue of brush (V5.0)

SYNOPSIS

```
ModulateBrush(id, brightness, saturation, hue)
```

FUNCTION

This function can be used to change the brightness, saturation, and hue settings of a brush. For each setting, you need to pass a floating point value that describes the desired change. A value of 1.0 means no change, a value smaller than 1.0 reduces the brightness/saturation/hue, while a value greater than 1.0 enhances it.

Note that if `id` specifies a palette brush, `ModulateBrush()` will just modulate the palette colors which makes this function really fast when used with palette brushes.

INPUTS

`id` brush to modulate

`brightness`
desired brightness correction

`saturation`
desired saturation correction

`hue` desired hue correction

EXAMPLE

```
ModulateBrush(1, 1.0, 2.0, 1.0)
```

The code above increases the saturation while leaving brightness and hue untouched. The result is an image with emphasized colors, just like in a cartoon.

21.46 MoveBrush

NAME

MoveBrush – move a brush from a to b

SYNOPSIS

```
[handle] = MoveBrush(id, xa, ya, xb, yb[, table])
```

FUNCTION

This function moves (scrolls) the brush specified by `id` softly from the location specified by `xa`,`ya` to the location specified by `xb`,`yb`.

Further parameters can be specified in the optional table argument. The following parameters are recognized:

`Speed:` Defines the number of pixels that the brush will be moved per draw. Therefore a higher number means higher speed. You can also specify a constant for the speed argument (`#SLOWSPEED`, `#NORMALSPEED` or `#FASTSPEED`).

`FX:` Specifies a special effect that shall be applied to the move. The following effects are currently possible:

`#BOUNCE:` Bounces the object at move end

`#DAMPED:` Damps the object at move end

`#SMOOTHOUT:`
Decreases object move speed towards the move end

`#SINE:` Displays the object on a sine wave (*)

`#BIGSINE:`
Displays the object on a big sine wave (*)

`#LOWERCURVE:`
Moves the object on a curve below the move line (*)

`#UPPERCURVE:`
Moves the object on a curve above the move line (*)

Effects marked with an asterisk are only possible with horizontal moves, which means that `ya` and `yb` coordinates must be equal!

`Async:` You can use this field to create an asynchronous draw object for this move. If you pass `True` here `MoveBrush()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

INPUTS

`id`	identifier of the brush to move
`xa`	source x position
`ya`	source y position
`xb`	destination x position
`yb`	destination y position
`table`	optional: further configuration for this move

RESULTS

`handle`	optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
MoveBrush(1, 100, 50, 0, 50, {Speed = 5})
```

Moves the brush from 100:50 to 0:50 with speed 5.

```
MoveBrush(1, #RIGHTOUT, #BOTTOM, #LEFTOUT, #BOTTOM, {Speed = #NORMALSPEED})
```

Moves the brush from the outer right position to the outer left position with a normal speed.

21.47 OilPaintBrush

NAME

OilPaintBrush – apply oil paint effect to brush (V5.0)

SYNOPSIS

```
OilPaintBrush(id, radius)
```

FUNCTION

This command applies an oil painting effect to the specified brush. The `radius` argument specifies the oil paint radius. The larger the radius you specify here, the longer this function needs to calculate the resulting images.

Note that this function cannot be used with palette brushes.

INPUTS

`id`	brush to modify
`radius`	oil paint effect radius

21.48 PenArrayToBrush

NAME

PenArrayToBrush – convert pen array to palette brush (V9.0)

SYNOPSIS

```
[id] = PenArrayToBrush(id, table, width, height[, t])
```

FUNCTION

This command creates a new palette brush from the array of pens specified in `table`. The table can be seen as a matrix containing `height` number of rows where each row has `width` number of elements, stored sequentially. The order of the pen data in this table must be as follows: Row after row in top-down format, i.e. the table starts with the first row of pens. Each row must contain exactly `width` number of pens, and there must be exactly `height` number of rows. The table must be one-dimensional, i.e. it mustn't use subtables for the individual rows but just store the pen values sequentially.

The palette that the pens should use can be set in the optional table argument `t`. The following table elements are currently recognized:

`Palette:` Set this to the id of a palette that has been created by `CreatePalette()`, `LoadPalette()` or the `@PALETTE` preprocessor command. If you don't set this tag, Hollywood will use a default palette that has all colors initialized to black.

`TransparentPen:`
This tag can be used to specify a pen that should appear transparent. If no pen should be made transparent, set this tag to `#NOPEN`, which is also the default.

Please note that the table that you pass to this function will usually eat lots of memory. Thus, you should set this table to `Nil` as soon as you no longer need it. Otherwise you will waste huge amounts of memory and it could even happen that your script runs out of memory altogether. So please keep in mind that you should always set pixel array tables to `Nil` as soon as you are done with them.

To convert a palette brush to a pen array, you can use the `BrushToPenArray()` function. See Section 21.9 [BrushToPenArray], page 255, for details.

INPUTS

`id`	identifier for the new palette brush or `Nil` for auto id selection
`table`	table containing an array of pens that describe the contents of the new brush
`width`	number of elements in each row
`height`	number of rows in table
`t`	optional: table containing further options (see above)

RESULTS

`id`	optional: identifier of the new palette brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
pixels = {}
For Local y = 0 To 479
   For Local x = 0 To 639 Do pixels[y * 640 + x] = y \ 2
Next
```

```
PenArrayToBrush(1, pixels, 640, 480, {Palette = #PALETTE_AGA})
pixels = Nil    ; IMPORTANT: free memory!
DisplayBrush(1, 0, 0)
```

The code above creates a palette brush that contains the first 240 pens from the default palette `#PALETTE_AGA`. Each palette pen uses two rows. Important: Do not forget to set the pixel array to `Nil` when you no longer need it because otherwise it will stay in memory and pixel arrays will eat huge amounts of memory!

21.49 PerspectiveDistortBrush

NAME

PerspectiveDistortBrush – apply perspective distortion to brush (V5.0)

SYNOPSIS

`PerspectiveDistortBrush(id,cx1,cy1,cx2,cy2,cx3,cy3,cx4,cy4[,smooth])`

FUNCTION

This command can be used to apply perspective distortion to the brush specified in `id`. You have to pass 4 control points that describe a quadrangle into which the brush shall be mapped. The optional argument `smooth` can be used to enable antialiased pixel interpolation which leads to a smoother appearance but takes longer to calculate.

The control point mapping is as follows: The top-left corner of the brush is mapped to control point 1, the top-right corner is mapped to control point 2, the bottom-right corner to control point 3, and the bottom-left corner to control point 4.

INPUTS

`id`	brush that shall be distorted
`cx1`	x coordinate of control point 1
`cy1`	y coordinate of control point 1
`cx2`	x coordinate of control point 2
`cy2`	y coordinate of control point 2
`cx3`	x coordinate of control point 3
`cy3`	y coordinate of control point 3
`cx4`	x coordinate of control point 4
`cy4`	y coordinate of control point 4
`smooth`	optional: whether or not anti-aliased distortion shall be used (defaults to `False`)

EXAMPLE

```
PerspectiveDistortBrush(1, 100, 0, 400, 0, 500, 300, 0, 300)
```

The code above maps brush 1 into a trapezoid shape.

21.50 PixelateBrush

NAME

PixelateBrush – zoom pixel cells of a brush (V5.0)

SYNOPSIS

```
PixelateBrush(id, cellsize)
```

FUNCTION

This command can be used to enlarge the pixel cells of the specified brush. Every pixel in the brush will be zoomed to the size specified in the `cellsize` argument. Pixelization will start in the top-left corner of the brush.

INPUTS

`id`	brush to pixelate
`cellsize`	desired pixelization level; must be greater than 1

21.51 PolarDistortBrush

NAME

PolarDistortBrush – apply polar distortion to brush (V5.0)

SYNOPSIS

```
PolarDistortBrush(id[, rmax, rmin, cx, cy, start, end, smooth])
```

FUNCTION

This command can be used to apply polar distortion to the brush specified in `id`. The optional arguments can be used to control the parameters for the polar distortion. The `rmin` and `rmax` arguments specify the minimum and maximum radii to use. `cx` and `cy` can be used to specify the center point for the distortion. Both the radius values and the center point must be specified in pixels. `start` and `end` specify the start and end angles for the polar distortion. Finally, the optional argument `smooth` can be used to enable antialiased pixel interpolation which leads to a smoother appearance but takes longer to calculate.

INPUTS

`id`	brush that shall be distorted
`rmax`	optional: maximum radius (defaults to brush diagonal divided by 2)
`rmin`	optional: minimum radius (defaults to 0)
`cx`	optional: x coordinate of center point (defaults to half of brush width)
`cy`	optional: y coordinate of center point (defaults to half of brush height)
`start`	optional: start angle (defaults to -180)
`end`	optional: end angle (defaults to 180)
`smooth`	optional: whether or not anti-aliased distortion shall be used (defaults to `False`)

21.52 QuantizeBrush

NAME

QuantizeBrush – reduce number of colors in brush (V6.0)

SYNOPSIS

```
QuantizeBrush(id[, t])
```

DEPRECATED SYNTAX

```
QuantizeBrush(id[, colors, dither])
```

FUNCTION

This function can be used to reduce colors in a brush. This is useful to create a retro palette-based-display look for your brush.

Starting with Hollywood 9.0, this function uses a new syntax with an optional table argument to specify additional options. The following table tags are currently recognized:

`Colors:` This tag allows you to specify the desired number of colors for the brush. This must be a value between 1 and 256. Defaults to 256. Alternatively, you can also set the `Depth` table tag to specify the desired number of colors for the brush (see below).

`Dither:` This tag allows you to control whether or not dithering should be used. Set this to `True` to enable or to `False` to disable dithering. Defaults to `True`.

`Depth:` This tag allows you to specify the desired number of colors for the brush. This must be a value between 1 (= 2 colors) and 8 (= 256 colors). Defaults to 8. This tag is an alternative to the `Colors` tag (see above). (V9.0)

`Palette:` Set this tag to `True` if you want `QuantizeBrush()` to convert your brush into a palette brush. Note that by default `QuantizeBrush()` will not create palette brushes even though it effectively always reduces colors to a number that would fit into a palette. Still, it doesn't do so due to compatibility reasons because palette brushes weren't supported before Hollywood 9.0. So if you want `QuantizeBrush()` to create a palette brush for you, you must set this tag to `True`. Defaults to `False`. (V9.0)

`TransparentPen:`
If `Palette` has been set to `True` (see above) and the brush to be quantized has a mask, all invisible pixels will be set to the number of the pen specified here so that this pen will become the transparent pen. Defaults to 0 which means tha the first pen should be made transparent by default. (V9.0)

`TransparentColor:`
If `Palette` has been set to `True` and the brush to be quantized has a mask, the brush's transparent pen will be set to the color you specify here. This color must be specified as an RGB color. If this tag is not set, the transparent pen won't be set to any specific color. (V9.0)

INPUTS

`id` identifier of the brush to quantize

`t` optional: table containing further arguments (see above) (V9.0)

EXAMPLE

```
QuantizeBrush(1, 32)
```

Convert brush 1 to a 32-color brush with dithering enabled

21.53 RasterizeBrush

NAME

RasterizeBrush – convert vector brush to raster brush (V5.0)

SYNOPSIS

```
RasterizeBrush(id)
```

FUNCTION

This function will convert the vector brush specified in `id` to a raster brush. Raster brushes are the normal brush type in Hollywood and they are supported by all commands of the brush library. The downside, however, is that scaling, rotation, and transformation are only possible with quality sacrifices on raster brushes.

You can find out the type of a brush by checking the `#ATTRTYPE` attribute using `GetAttribute()`.

INPUTS

`id` vector brush to convert

21.54 ReadBrushPixel

NAME

ReadBrushPixel – read single pixel from brush (V5.0)

SYNOPSIS

```
color, trans = ReadBrushPixel(id, x, y)
```

FUNCTION

This command reads the color and transparency states of the specified pixel from the brush specified in `id`. The color is returned in RGB format whereas the format of the `trans` value depends on the type of transparency used by the brush. If the brush has a mask, `trans` will be either 0 (invisible) or 1 (visible). If the brush has an alpha channel, then `trans` will be in the range of 0 (invisible) to 255 (visible). If the brush does not have a transparency channel, -1 is returned in `trans`.

You can also read pixels from brushes by selecting the brush as the output device using `SelectBrush()` and then call the `ReadPixel()` function. Using `ReadBrushPixel()`, however, is faster for most cases because it allows you to access color and transparency channels at the same time and you can also avoid the overhead that is generated by calling `SelectBrush()` and `EndSelect()`.

Note that when using this function with a palette brush, `ReadBrushPixel()` won't return the RGB color but the pen at the specified position.

INPUTS

`id` identifier of the brush to use

`x` x offset

`y` y offset

RESULTS

`color` RGB color or pen at the specified location

`trans` transparency state at the specified location

EXAMPLE

```
color, trans = ReadBrushPixel(1, 100, 100)
```

Reads pixel states from position 100:100 in brush 1.

21.55 ReduceAlphaChannel

NAME

ReduceAlphaChannel – reduce alpha channel intensity (V6.0)

SYNOPSIS

```
ReduceAlphaChannel(id, ratio)
```

FUNCTION

This function can be used to reduce the intensity of the alpha channel associated with the specified brush. Every alpha pixel is multiplied by the ratio you pass in argument 2. This ratio must be between 0 and 255. A ratio of 255 means 1.0 or 100% whereas 0 means 0.0 or 0%. Thus, if you want to reduce the alpha transparency of all pixels by 50%, you would have to pass 128 in the `ratio` parameter.

`ratio` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id` brush whose alpha channel should be modified

`ratio` value between 0 and 255 that specifies the intensity of the reduction operation or a percent specification

21.56 RemapBrush

NAME

RemapBrush – remap brush colors (V9.0)

SYNOPSIS

```
RemapBrush(id, palid[, t])
```

FUNCTION

This function can be used to remap the colors of the brush specified by `id` to the colors of the palette specified by `palid`. The source brush can either be a normal or a palette brush. If it is a normal brush, `RemapBrush()` will also convert it to a palette brush while remapping so the resulting brush will always be a palette brush.

The optional table argument `t` can be used to specify additional options. The following table tags are currently recognized:

`Dither:` This tag allows you to control whether or not dithering should be used. Set this to `True` to enable or to `False` to disable dithering. Defaults to `True`.

Note that if the brush uses transparency, you have to use `SetTransparentPen()` on the palette first to define a pen that should be made transparent.

INPUTS

`id`	identifier of the brush to remap
`palid`	identifier of the palette whose colors the brush should be remapped to
`t`	optional: table containing further arguments

EXAMPLE

```
CreatePalette(1, {#BLACK, #WHITE}, {Depth = 1})
RemapBrush(1, 1, {Dither = True})
```

Convert brush 1 to a black & white palette brush. Remapping will be done with dithering enabled.

21.57 RemoveBrushPalette

NAME

RemoveBrushPalette – convert palette brush to RGB (V9.0)

SYNOPSIS

```
RemoveBrushPalette(id[, trans])
```

FUNCTION

This function can be used to convert the palette brush specified by `id` into an RGB brush. This means that all brush pixels will be converted to 32-bit RGB and the palette will be removed from the brush. The optional argument `trans` allows you to specify how the brush's transparency should be converted. This can be either `#MASK` or `#ALPHACHANNEL`. If you set it to `#MASK`, which is also the default, the brush's transparent pen will be mapped to a mask. If it is set to `#ALPHACHANNEL`, the brush's transparent pen will be mapped to an alpha channel.

INPUTS

`id`	identifier of the brush to convert
`trans`	optional: desired type of brush transparency; must be either `#MASK` or `#ALPHACHANNEL`; defaults to `#MASK`

21.58 ReplaceColors

NAME

ReplaceColors – replace colors in a brush (V1.5)

SYNOPSIS

```
ReplaceColors(id, colors)
```

FUNCTION

This function scans through a color array that you specify and replaces each color with another color which you also have to specify. The color array must be organized in the way: Search color 1, Replace color 1, Search color 2, Replace color 2, ...,.

Note that if you pass a palette brush in `id`, you need to pass pens instead of colors in the table argument.

INPUTS

`id`	identifier of the brush to use
`colors`	color table that describes which colors (or pens) to replace

EXAMPLE

```
ReplaceColors(1, {#BLACK, #WHITE, #RED, #GREEN})
```

The code changes all black pixels in brush 1 to white ones and all red pixels to green ones.

21.59 RGBArrayToBrush

NAME

RGBArrayToBrush – convert pixel array to brush (V5.0)

SYNOPSIS

```
[id] = RGBArrayToBrush(id, table, width, height[, transtype, invalpha])
```

FUNCTION

This command creates a new brush from the array of RGB pixels specified in `table`. The table can be regarded as a matrix containing `height` number of rows where each row has `width` number of elements. The order of the pixel data in this table must be as follows: Row after row in top-down format, i.e. the table starts with the first row of pixels. Every row must contain exactly `width` number of pixels, and there must be at least `height` number of rows. The single pixels must be passed in the RGB format with an optional alpha value. The `transtype` argument allows you to specify the transparency type the new brush should use. This can be either `#NONE` for no transparency, `#MASK` for monochrome transparency, and `#ALPHACHANNEL` for alpha channel transparency.

The optional argument `invalpha` can be used to tell `RGBArrayToBrush()` that all alpha channel values are inverted. This means that a value of 0 means 100% visibility and a value of 255 means invisibility. Normally, it is just the other way round. Due to historical reasons, the Hollywood drawing library uses inverted alpha values, and this why they are also supported by `RGBArrayToBrush()`, although they are not the default.

If `transtype` is set to `#NONE`, the pixels' alpha values are ignored altogether and `invalpha` does not have any effect either.

Please note that the table that you pass to this function will usually eat lots of memory. Thus, you should set this table to `Nil` as soon as you no longer need it. Otherwise you will waste huge amounts of memory and it could even happen that your script runs out

of memory altogether. So please keep in mind that you should always set pixel array tables to `Nil` as soon as you are done with them.

To convert a brush to a pixel array, you can use the `BrushToRGBArray()` function.

INPUTS

`id` identifier for the new brush or `Nil` for auto id selection

`table` table containing an array of RGB pixels that describe the contents of the new brush; if `transtype` is set to `#MASK` or `#ALPHACHANNEL` you must also specify alpha values in the highest 8 bits

`width` number of elements in each row

`height` number of rows in table

`transtype` optional: desired transparency setting for brush (defaults to `#NONE`)

`invalpha` optional: whether to use inverted alpha values (defaults to `False` which means do not invert alpha values)

RESULTS

`id` optional: identifier of the new brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
pixels = {}
col = #BLUE
stp = 256 / 480
For Local y = 0 To 479
   For Local x = 0 To 639 Do pixels[y * 640 + x] = col
   col = col - stp
Next

RGBArrayToBrush(1, pixels, 640, 480)
pixels = Nil    ; IMPORTANT: free memory!
DisplayBrush(1, 0, 0)
```

The code above creates a color gradient from red to `#BLACK` and converts it into a brush using `RGBArrayToBrush()`. Important: Do not forget to set the pixel array to `Nil` when you no longer need it because otherwise it will stay in memory and pixel arrays will eat huge amounts of memory!

21.60 RotateBrush

NAME

RotateBrush – rotate a brush (V1.5)

SYNOPSIS

```
RotateBrush(id, angle[, factorx, factory, smooth])
```

FUNCTION

This function rotates the brush specified by `id` by the specified angle (in degrees). A positive angle rotates anti-clockwise, a negative angle rotates clockwise.

Starting with Hollywood 2.5, this function can also scale the brush while rotating it (called a rot-zoom). This is done in one pass so the quality of the resulting image data is much better than if you would first call `ScaleBrush()` and then `RotateBrush()`. If you want to have the brush scaled with the rotation, simply pass two scaling factors as factorx and factory. These two factors are floating point numbers representing a zoom percentage (1 corresponds to 100%, 0.5 to 50%, 1.5 to 150% etc.)

Additionally, you can choose to have the scaled and/or rotated graphics interpolated by passing `True` in the `smooth` argument. The graphics will then be scaled/rotated using anti-alias.

Please note:

- If you rotate a brush for instance by a 45 degree angle and your brush does not have a mask, Hollywood will automatically create a mask for this brush because the rotate operation usually leads to some unused areas in the brush. If your brush has a mask, then Hollywood will rotate this mask also.
- You should not rotate a rotated brush again because this will lead to loss of data! You should always use the original brush when creating rotated versions of the brush, e.g. if you rotate a brush by 45 degrees and then rotate it back by -45 degrees, the resulting brush will not be of the same quality as the original one.
- Note that for vector brushes, `RotateBrush()` will always operate on the untransformed brush. This means that any previous transformations applied to the brush using `RotateBrush()`, `ScaleBrush()`, or `TransformBrush()` will be undone when calling `RotateBrush()`.

INPUTS

`id`	brush to rotate
`angle`	rotation angle in degrees
`factorx`	optional: scaling factor on the x-axis (defaults to 1 which means no x scaling) (V2.5)
`factory`	optional: scaling factor on the y-axis (defaults to 1 which means no y scaling) (V2.5)
`smooth`	optional: whether or not anti-aliased rotation shall be used (V2.5)

21.61 SaveBrush

NAME

SaveBrush – save brush to a file (V2.0)

SYNOPSIS

```
SaveBrush(id, f$[, t])
```

DEPRECATED SYNTAX

```
SaveBrush(id, f$[, transcolor, fmt, table])
```

FUNCTION

This function saves the brush specified by `id` to the file specified by `f$`. By default, the brush will be saved as a Windows bitmap (BMP) file. This can be changed by passing a different format identifier to `SaveBrush()` (see below for details).

`SaveBrush()` supports several optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `SaveBrush()`.

The following table fields are recognized by this function:

`Format:` Set this tag to the image format that should be used. This can either be one of the following constants or an image saver provided by a plugin:

`#IMGFMT_BMP:`
Windows bitmap. Hollywood's BMP saver supports RGB and palette images. `#IMGFMT_BMP` is the default format used by `SaveBrush()`.

`#IMGFMT_PNG:`
PNG format. Hollywood's PNG saver supports RGB and palette images. RGB images also can have an alpha channel, palette images can have a transparent pen. (V2.5)

`#IMGFMT_JPEG:`
JPEG format. Note that the JPEG format does not support alpha channels or palette-based graphics. The `Quality` field (see below) allows you to specify the quality level for the JPEG image (valid values are 0 to 100 where 100 is the best quality). (V4.0)

`#IMGFMT_GIF:`
GIF format. Because GIF images are always palette-based, RGB graphics have to be quantized before they can be exported as GIF. You can use the `Colors` and `Dither` tags (see below) to specify the number of palette entries to allocate for the image and whether or not dithering shall be applied. When using `#IMGFMT_GIF` with a palette brush, no quantizing will be done. `#IMGFMT_GIF` also supports palette images with a transparent pen. (V4.5)

`#IMGFMT_ILBM:`
IFF ILBM format. Hollywood's IFF ILBM saver supports RGB and palette images. Palette images can also have a transparent pen, alpha channels are unsupported for this output format. (V4.5)

Defaults to `#IMGFMT_BMP`.

`Dither:` Set to `True` to enable dithering. This field is only handled when the destination format is palette-based and the source data is in RGB format. Defaults to `False` which means no dithering.

`Depth:` Specifies the desired image depth. This is only handled when the format is palette-based and the source data is in RGB format. Valid values are between 1 (= 2 colors) and 8 (= 256 colors). Defaults to 8. (V9.0)

`Colors:` This is an alternative to the `Depth` tag. Instead of a bit depth, you can pass how many colors the image shall use here. Again, this is only handled when the format is palette-based and the source data is in RGB format. Valid values are between 1 and 256. Defaults to 256.

`Quality:` Here you can specify a value between 0 and 100 indicating the compression quality for lossy compression formats. A value of 100 means best quality, 0 means worst quality. This is only available for image formats that support lossy compression. Defaults to 90 which means pretty good quality.

`FillColor:`
When saving an RGB image that has transparent pixels, you can specify an RGB color that should be written to all transparent pixels here. This is probably of not much practical use. Defaults to `#NOCOLOR` which means that transparent pixels will be left as they are. (V9.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Here is an overview that shows which formats support which tags:

	BMP	PNG	JPEG	GIF	ILBM
Dither	No	No	No	Yes	No
Colors	No	No	No	Yes	No
Quality	No	No	Yes	No	No

INPUTS

`id` identifier of the brush to save

`f$` destination file

`t` optional: table specifying further options (see above) (V9.0)

EXAMPLE

```
SaveBrush(1, "test.jpg", {Format = #IMGFMT_JPEG, Quality = 80})
```

The code above saves brush 1 as "test.jpg" using a quality of 80%.

21.62 ScaleBrush

NAME

ScaleBrush – scale a brush

SYNOPSIS

```
ScaleBrush(id, width, height[, smooth])
```

FUNCTION

This command scales the brush specified by `id` to the specified dimensions. Optionally, you can choose to have the scaled graphics interpolated by passing `True` in the `smooth` argument. The graphics will then be scaled using anti-alias.

Please note: You should always do scale operations with the original brush. For instance, if you scale brush 1 to 12x8 and then scale it back to 640x480, you will get a messed image. Therefore you should always keep the original brush and scale only copies of it.

Note that for vector brushes, `ScaleBrush()` will always operate on the untransformed brush. This means that any previous transformations applied to the brush using `ScaleBrush()`, `TransformBrush()`, or `RotateBrush()` will be undone when calling `ScaleBrush()`.

New in V2.0: You can pass `#KEEPASPRAT` as either `width` or `height`. Hollywood will calculate the size then automatically by taking the aspect-ratio of the brush into account.

Starting with Hollywood 2.0, `width` and `height` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id`	identifier of the brush to use as source
`width`	desired new width for the brush
`height`	desired new height for the brush
`smooth`	optional: whether or not anti-aliased scaling shall be used (V2.5)

EXAMPLE

```
ScaleBrush(1,640,480)
```

Scales brush 1 to a resolution of 640x480.

21.63 SelectAlphaChannel

NAME

SelectAlphaChannel – select an alpha channel as output device (V2.0)

SYNOPSIS

```
SelectAlphaChannel(id[, type, frame])
```

FUNCTION

This function selects the alpha channel of the graphics object specified by `id` as the current output device. This means that all graphics data that are output by Hollywood will be drawn to this alpha channel. Alpha channels are no stand-alone objects in

Hollywood; they are always attached to an image object, for example a brush or an animation.

By default, `SelectAlphaChannel()` always works with the alpha channels of brushes. However, starting with Hollywood 4.5, you can also use it to draw to the alpha channel of animations and BGPics. To do this you have to specify `#ANIM` or `#BGPIC` in the optional `type` argument. If you use `#ANIM` in the `type` argument, you have to specify the frame of the anim that you want to draw to, too. See `SelectAnim()` for more information. If you specify `#BGPIC` in `type`, note that you can only modify the alpha channel of BGPics that are currently not associated with a display. Starting with Hollywood 4.7, you can also pass `#LAYER` as the `type` to modify the alpha channel of a layer. Note that if the layer is an anim layer, you will also have to specify the number of the frame to select.

Alpha channels can be used to give each pixel its own transparency setting. There are 256 different transparency levels available for each pixel. An alpha channel value of 0 means that the pixel is fully transparent and an alpha channel value of 255 means that the pixel is opaque. All Hollywood graphics functions will render with a static alpha channel intensity into the alpha channel of the graphics object. You can configure this intensity using the `SetAlphaIntensity()` command. Alternatively, you can set the alpha render mode to a vanilla copy mode. This is done by calling `SetAlphaIntensity()` with `#VANILLACOPY` as the argument. Then, all Hollywood graphics commands which output alpha channel pixels will copy the exact alpha channel data to your brush's alpha channel. This vanilla copy mode is a new feature of Hollywood 2.5.

The color argument that several Hollywood functions (like `Box()` or `Circle()`) expect, is superfluous when rendering to alpha channels. You only need to use `SetAlphaIntensity()` when rendering to alpha channels.

Alpha channels are usually used for nice background shine-through effects or for anti-aliasing jagged edges. If you do not need different transparency levels but only two choices, namely transparent pixels and opaque pixels, you should use `SelectMask()` instead because it can be drawn faster. Please note that graphics objects cannot have a mask and an alpha channel. Only one transparency setting is possible. Thus, if you use this command on a graphics object that has a mask, this mask will be deleted first.

If the graphics object you specify in id does not have an alpha channel yet, it will be automatically created when you call a command that wants to draw to the alpha channel. The alpha channel created will be opaque then, i.e. every pixel will have an alpha intensity of 255 which means that it is 0% transparent. To cancel alpha channel rendering mode and return to main display output, just call the `EndSelect()` function.

If you do not need an alpha channel any longer, you should use the command `DeleteAlphaChannel()` to remove it from the brush.

You cannot use brush links with this command because the graphics data of the brush specified by `id` will be changed. It is also forbidden to call commands which change the dimensions of the brush/anim that is currently used as output device, e.g. you may not call `ScaleBrush()` or `ScaleAnim()` to scale the brush/anim that is currently the output device. Furthermore, it is not allowed to call `SelectAlphaChannel()` for animations that are loaded from disk. Animations must always reside completely in memory if you want to draw to their frames using `SelectAlphaChannel()`.

Only commands that output static graphics can be used when `SelectAlphaChannel()` is active. You may not call animated functions like `MoveBrush()` or `DisplayBrushFX()` while `SelectAlphaChannel()` is active.

If you are using type `#LAYER` and the specified layer is a vector layer, `SelectAlphaChannel()` will rasterize the layer to a brush layer first. See Section 34.39 [SelectLayer], page 676, for details.

INPUTS

`id` graphics object whose alpha channel shall be used as output device

`type` optional: type of the graphics object specified in id; can be `#BRUSH`, `#ANIM`, `#BGPIC`, or `#LAYER` (defaults to `#BRUSH`) (V4.5)

`frame` optional: animation frame to use; only required if type is set to `#ANIM` or `#LAYER` in case the specified layer is an anim layer (V4.5)

EXAMPLE

```
CreateBrush(1, 256, 50, #RED)     ; create a brush of size 256x50
SelectAlphaChannel(1)             ; alphachannel becomes output device
For k = 255 To 0 Step -1
   SetAlphaIntensity(k)           ; set alpha intensity to k
   Line(255 - k, 0, 255 - k, 49) ; draw lines with various intensities
Next
EndSelect                         ; make display the output device again

DisplayBrush(1, #CENTER, #CENTER)
```

This code demonstrates the 256 different transparency levels by creating a brush with the width of 256 pixels and drawing 256 lines with different transparency settings into it. The result will be a red rectangle which smoothly merges with the background picture. Please note that you need a 24-bit screen for the full eye-candy. On 15-bit and 16-bit screens there are not enough colors to display all different levels.

21.64 SelectBrush

NAME

SelectBrush – select a brush as output device (V1.5)

SYNOPSIS

```
SelectBrush(id[, mode, combomode])
```

FUNCTION

This function selects the brush specified by `id` as the current output device. This means that all graphics data that are drawn by Hollywood will be rendered to your brush.

The optional `mode` argument defaults to `#SELMODE_NORMAL` which means that only the color channels of the brush will be altered when you draw to it. The transparency channel of the brush (can be either a mask or an alpha channel) will never be altered. You can change this behaviour by using `#SELMODE_COMBO` in the optional `mode` argument. If you use this mode, every Hollywood graphics command that is called after `SelectBrush()`

will draw into the color and transparency channel of the brush. If the brush does not have a transparency channel, `#SELMODE_COMBO` behaves the same as `#SELMODE_NORMAL`.

Starting with Hollywood 5.0 you can use the optional `combomode` argument to specify how `#SELMODE_COMBO` should behave. If `combomode` is set to 0, the color and transparency information of all pixels in the source image are copied to the destination image in any case - even if the pixels are invisible. This is the default behaviour. If `combomode` is set to 1, only the visible pixels are copied to the destination image. This means that if the alpha value of a pixel in the source image is 0, i.e. invisible, it will not be copied to the destination image. Hollywood 6.0 introduces the new combomode 2. If you pass 2 in `combomode`, Hollywood will blend color channels and alpha channel of the source image into the destination image's color and alpha channels. When you draw the destination image later, it will look as if the two images had been drawn on top of each other consecutively. Please note that the `combomode` argument is only supported together with `#SELMODE_COMBO`. It doesn't have any effect when used with the other modes.

An alternative way to draw into the transparency channels of a brush is to do this separately using `SelectMask()` or `SelectAlphaChannel()`. These two commands, however, will write data to the transparency channel only. They will not touch the color channel. So if you want both channels, color and transparency, to be affected, you need to use `SelectBrush()` with `mode` set to `#SELMODE_COMBO`.

When you are finished with rendering to your brush and want your display to become the output device again, just call `EndSelect()`.

While `SelectBrush()` is active, it is forbidden to call commands which change the dimensions of the brush that is currently used as the output device, e.g. you may not call `ScaleBrush()` to scale the brush that is currently the output device.

Only Hollywood commands that draw graphics directly can be used when `SelectBrush()` is active. You may not call animated functions like `MoveBrush()` or `DisplayBrushFX()` while `SelectBrush()` is active.

INPUTS

`id` brush which shall be used as output device

`mode` optional: rendering mode to use (see above); this can be either `#SELMODE_NORMAL` or `#SELMODE_COMBO`; defaults to `#SELMODE_NORMAL` (V4.5)

`combomode`
optional: mode to use when `#SELMODE_COMBO` is active (see above); defaults to 0 (V5.0)

EXAMPLE

```
CreateBrush(1, 320, 256)
SelectBrush(1)
SetFillStyle(#FILLCOLOR)
Box(0, 0, 320, 256, #RED)
EndSelect()
MoveBrush(1, #CENTER, #BOTTOMOUT, #CENTER, #TOPOUT, 10)
```

The above code creates a 320x256 brush, draws a red rectangle into it and then scrolls the rectangle on the screen. This is very abstract example. You can of course do a lot of

more with this command, just have a look at the examples supplied with the Hollywood distribution. They use `SelectBrush()` in more advanced contexts.

21.65 SelectMask

NAME

SelectMask – select a mask as output device (V2.0)

SYNOPSIS

```
SelectMask(id[, type, frame])
```

FUNCTION

This function selects the mask of the graphics object specified by `id` as the current output device. This means that all graphics data that are output by Hollywood will be drawn to this mask. Masks are no stand-alone objects in Hollywood; they are always connected to an image object, for example a brush or an animation.

By default, `SelectMask()` always works with the masks of brushes. However, starting with Hollywood 4.5, you can also use it to draw to the masks of animations and BGPics. To do this you have to specify `#ANIM` or `#BGPIC` in the optional `type` argument. If you specify `#ANIM` in `type`, you have to specify the frame of the animation that you want to draw to, too. See `SelectAnim()` for more information. If you specify `#BGPIC` in `type`, note that you can only modify the masks of BGPics that are currently not associated with a display. Starting with Hollywood 4.7, you can also pass `#LAYER` as the `type` to modify the mask of a layer. Note that if the layer is an anim layer, you will also have to specify the number of the frame to select.

Masks are used to control the transparency of a graphics object. They do not carry any color information. Every pixel in a mask can only have two different states: 1, which means that this pixel is visible and 0, which means that the pixel is invisible. Therefore you need to tell Hollywood whether or not the drawing commands should draw visible pixels (1) or invisible pixels (0) to the mask. This is done by using the `SetMaskMode()` command. The color argument that several Hollywood functions (like `Box()` or `Circle()`) expect, is superfluous when rendering to masks. You only need to use SetMaskMode().

If the graphics object you specify in id does not have a mask yet, it will be automatically created when you call a command that wants to draw to the mask. If a mask is created by `SelectMask()`, it will initially be fully opaque.

To cancel mask rendering mode and return to main display output, just call the `EndSelect()` function.

If you do not need a mask any longer, you can remove it from a brush by calling `SetBrushTransparency()` with the argument `#NOTRANSPARENCY`. Or simply use `DeleteMask()`.

You cannot use brush links with this command because the graphics data of the brush specified by `id` will be changed. It is also forbidden to call commands which change the dimensions of the brush/anim that is currently used as output device, e.g. you may not call `ScaleBrush()` or `ScaleAnim()` to scale the brush/anim that is currently the output device. Furthermore, it is not allowed to call `SelectMask()` for animations that

are loaded from disk. Animations must always reside completely in memory if you want to draw to their frames using `SelectMask()`.

Only commands that output graphics directly can be used after `SelectMask()`. You may not call animated functions like `MoveBrush()` or `DisplayBrushFX()` while `SelectMask()` is active.

Please note that graphics objects cannot have a mask and an alpha channel. Only one transparency setting is possible. Thus, if you use this command on an object that already has an alpha channel, this alpha channel will be deleted.

If you are using type `#LAYER` and the specified layer is a vector layer, `SelectMask()` will rasterize the layer to a brush layer first. See Section 34.39 [SelectLayer], page 676, for details.

INPUTS

`id`	graphics object whose mask shall be used as output device
`type`	optional: type of the graphics object specified in id; can be `#BRUSH`, `#ANIM`, `#BGPIC`, or `#LAYER` (defaults to `#BRUSH`) (V4.5)
`frame`	optional: animation frame to use; only required if type is set to `#ANIM` or if `#LAYER` is used on an anim layer (V4.5)

EXAMPLE

```
w = GetAttribute(#BRUSH, 1, #ATTRWIDTH)
h = GetAttribute(#BRUSH, 1, #ATTRHEIGHT)

SetFillStyle(#FILLCOLOR)
SelectMask(1)                ; select mask as output device
SetMaskMode(#MASKINVISIBLE) ; all calls will draw invisible pixels now
Cls                          ; clear all pixels
SetMaskMode(#MASKVISIBLE)   ; all calls will draw visible pixels now
Box(0, 0, w, h, 0, 20)      ; draw a rectangle with rounded edges
EndSelect                    ; select display as output device again
```

The code above renders a rectangle with rounded edges to the mask of brush 1. When you display brush 1 now, it will appear with rounded edges.

21.66 SepiaToneBrush

NAME

SepiaToneBrush – apply sepia-tone effect to brush (V5.0)

SYNOPSIS

`SepiaToneBrush(id, level)`

FUNCTION

This command can be used to apply a sepia-tone effect to the specified brush. The sepia-tone effect tries to simulate the look of old photographs. The second argument controls the intensity of the sepia-toning and can be any value between 0 and 255, or a percentage specification inside a string. Usually, a value around 204 is used (= 80%) for the best looks.

Note that this function cannot be used with palette brushes.

INPUTS

`id` brush to sepia-tone

`level` desired sepia-toning level (0 to 255, or a string containing a percentage specification)

EXAMPLE

```
SepiaToneBrush(1, "80%")
```

The code above applies a sepia-tone effect to brush 1 using an intensity of 80%.

21.67 SetAlphaIntensity

NAME

SetAlphaIntensity – define intensity for alpha rendering (V2.0)

SYNOPSIS

```
SetAlphaIntensity(level)
```

FUNCTION

This function allows you to specify the level of transparency all graphics functions shall use when they render into a brush's alpha channel (when `SelectAlphaChannel()` is active). The transparency level must be in the range of 0 to 255, where 0 means 100% transparency and 255 means no transparency. The intensity you specify here will be used by all graphics functions of Hollywood instead of a color. The default alpha intensity is 128.

Level can also be a string containing a percent specification, e.g. `"50%"`.

Please note that this is just the other way round from `SetLayerTransparency()` where 0 means no transparency and 255 means full transparency.

See Section 21.63 [SelectAlphaChannel], page 301, for more information on alpha channels in general.

New in V2.5: You can also specify the special constant `#VANILLACOPY` as the level argument. If you do this, Hollywood will enable the new vanilla copy mode. This means that all graphics commands which render alpha channel pixels will copy these pixels directly to your brush's alpha channel. For instance, if you select an alpha channel of a brush and then use `TextOut()` to draw anti-aliased text, Hollywood will render the exact alpha channel data of the anti-aliased text to your brush's alpha channel where you can process it further. If `#VANILLACOPY` is active and you draw graphics to the alpha channel that do not have any alpha data, Hollywood will write an alpha intensity of 255 (i.e. fully visible) into your alpha channel.

INPUTS

`level` desired transparency level (0 to 255 or percent specification) or: special constant `#VANILLACOPY` for special vanilla copy mode (V2.5)

EXAMPLE

See Section 21.63 [SelectAlphaChannel], page 301.

21.68 SetBrushDepth

NAME

SetBrushDepth – set brush palette depth (V9.0)

SYNOPSIS

```
SetBrushDepth(id, depth[, t])
```

FUNCTION

This function sets the depth of the palette of the brush specified by `id` to the depth specified in `depth`. `depth` must be a bit depth ranging from 1 (= 2 colors) to 8 (= 256 colors). See Section 44.1 [Palette overview], page 889, for details. Note that if the specified depth is less than that of the pixel data attached to the palette, the pixel data will be remapped to match the new depth.

Starting with Hollywood 10.0, `SetBrushDepth()` accepts an optional table argument which can contain the following tags:

`Remap:` If this tag is set to `False`, out-of-range pens will not be remapped to existing pens but instead they will simply be set to the pen specified in the `ClipPen` tag (see below), i.e. no remapping will take place. Note that `Remap` is only effective when reducing colors. If the new depth has more pens than the old depth, `Remap` won't do anything. (V10.0)

`ClipPen:` This is only used in case the `Remap` tag is set to `False` (see above). In that case, out-of-range pens will not be remapped to existing pens but will simply be set to the pen specified in the `ClipPen` tag, i.e. no remapping will take place. Note that `ClipPen` is only effective when reducing colors. If the new depth has more pens than the old depth, `ClipPen` won't do anything. (V10.0)

INPUTS

`id` identifier of brush to modify

`depth` desired new palette depth (ranging from 1 to 8)

`t` optional: table argument containing further options (see above) (V10.0)

EXAMPLE

```
SetBrushDepth(1, 8)
```

The code above changes the depth of brush 1's palette to 8 (= 256 colors).

21.69 SetBrushPalette

NAME

SetBrushPalette – change brush palette (V9.0)

SYNOPSIS

```
SetBrushPalette(id, palid[, t])
```

FUNCTION

This function replaces the palette of the brush specified by `id` with the palette specified by `palid`. The optional table argument `t` allows you to specify some further options. The following tags are currently supported by the optional table argument `t`:

`Remap:` If this is set to `True`, the pixels of the brush will be remapped to match the colors of the new palette as closely as possible. By default, there will be no remapping and the actual pixel data of the brush will remain untouched. If you want remapping, set this tag to `True` but be warned that remapping all pixels will of course take much more time than just setting a new palette without remapping. Defaults to `False`.

`Dither:` If the `Remap` tag (see above) has been set to `True`, you can use the `Dither` tag to specify whether or not dithering should be used. Defaults to `True` which means dithering should be used.

`CopyCycleTable:`
Palettes can have a table containing color cycling information. If you set this tag to `True`, this cycle table will be copied to the brush as well. Defaults to `False`.

INPUTS

`id`	identifier of brush to use
`palid`	identifier of palette to copy to brush
`t`	optional: table for specifying further options (see above)

21.70 SetBrushPen

NAME

SetBrushPen – change brush palette pen (V9.0)

SYNOPSIS

```
SetBrushPen(id, pen, color)
```

FUNCTION

This function sets the color of the pen specified by `pen` to the color specified by `color` in the palette of the brush specified by `id`.

INPUTS

`id`	identifier of brush
`pen`	pen you want to modify (starting from 0)
`color`	new color for the pen, must be specified as an RGB color

EXAMPLE

```
SetBrushPen(1, 0, #RED)
```

The code above sets pen 0 to red in the palette of brush 1.

21.71 SetBrushTransparency

NAME

SetBrushTransparency – define transparent color of a brush (V1.5)

SYNOPSIS

```
SetBrushTransparency(id, col)
```

FUNCTION

This function makes the color specified by `col` transparent in the brush with the number `id`. This is done by creating a mask for the brush. `SetBrushTransparency()` will scan through all pixels of the brush and mask out all pixels that have the specified color. The mask that is created by this function is not automatically updated when you call `SelectBrush()` to modify pixels of your brush. Hence, it is necessary to call `SetBrushTransparency()` again after a call to `SelectBrush()`, so that the mask can be updated, too.

You can also use this function to remove a mask from a brush. Just specify `#NOTRANSPARENCY` as the color.

Note that this function cannot be used with palette brushes. You can use `SetTransparentPen()` to change the transparent pen in a palette brush. See Section 44.37 [SetTransparentPen], page 920, for details.

INPUTS

`id`	source brush id
`col`	color to be displayed transparently or `#NOTRANSPARENCY` to kill the brush's mask

EXAMPLE

```
CreateBrush(1, 320, 256)
SelectBrush(1)
SetFillStyle(#FILLCOLOR)
Circle(0, 0, 100, #RED)
EndSelect()
SetBrushTransparency(1, #BLACK)
MoveBrush(1, #CENTER, #BOTTOMOUT, #CENTER, #TOPOUT, 10)
```

The above code creates a brush, draws a filled circle on it and then sets the background color black as transparent color. After that it scrolls the circle through the screen. It is important that you call `SetBrushTransparency()` when output is done, i.e. after calling `EndSelect()`.

21.72 SetBrushTransparentPen

NAME

SetBrushTransparentPen – set transparent pen of brush palette (V9.0)

SYNOPSIS

```
SetBrushTransparentPen(id, pen)
```

FUNCTION

This function sets the transparent pen of the palette of the brush specified by `id` to the pen specified in `pen`. Pens are counted from 0.

INPUTS

`id` identifier of brush to use

`pen` desired transparent pen (starting from 0)

EXAMPLE

```
SetBrushTransparentPen(1, 4)
```

The code makes pen 4 in the palette of brush 1 transparent.

21.73 SetMaskMode

NAME

SetMaskMode – define rendering mode for masks (V2.0)

SYNOPSIS

```
SetMaskMode(mode)
```

FUNCTION

This function can be used to define the rendering mode when `SelectMask()` is active. The argument mode can either be `#MASKVISIBLE` or `#MASKINVISIBLE`. The default is `#MASKVISIBLE`. If you select the visible mode, all graphics commands will draw visible pixels into the mask, otherwise invisible pixels will be drawn. Obviously, a mask does not carry any color information but only those two flags per pixel (visible or invisible).

As of Hollywood 4.0, the following new mask modes are supported:

`#MASKVANILLACOPY:`
Masking data of the source image will be copied exactly to the destination.

`#MASKAND:`
Masking data of the source image will be copied to the destination mask using a logical AND operation on each pixel.

`#MASKOR:` Masking data of the source image will be copied to the destination mask using a logical OR operation on each pixel.

`#MASKXOR:`
Masking data of the source image will be copied to the destination mask using a logical XOR operation on each pixel.

Below is a table summing up the different mask modes. Please note that the mask mode `#MASKVISIBLE`, `#MASKINVISIBLE`, and `#MASKVANILLACOPY` are independent of the desti-

nation mask data. Destination mask data is only taken into account by the `#MASKAND`, `#MASKOR`, and `#MASKXOR` modes.

Source	Dest	VISIBLE	INVISIBLE	VANILLA	AND	OR	XOR
0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1
1	0	1	0	1	0	1	1
1	1	1	0	1	1	1	0

See Section 21.65 [SelectMask], page 305, for more information on mask rendering.

INPUTS

`mode` mask rendering mode (see above)

EXAMPLE

See Section 21.65 [SelectMask], page 305.

21.74 SharpenBrush

NAME

SharpenBrush – apply sharpening filter to brush (V5.0)

SYNOPSIS

```
SharpenBrush(id[, radius])
```

FUNCTION

This command sharpens the appearance of the specified brush. The optional argument `radius` can be used to specify the sharpening radius. The larger the radius you specify here, the longer this function needs to apply the sharpening effect. If you do not specify the optional argument, this command will automatically choose an appropriate radius.

Note that this function cannot be used with palette brushes.

INPUTS

`id` brush to sharpen

`radius` optional: sharpening radius (defaults to 0 which means that the radius will be chosen automatically)

21.75 SolarizeBrush

NAME

SolarizeBrush – apply solarization effect to brush (V5.0)

SYNOPSIS

```
SolarizeBrush(id, level)
```

FUNCTION

This command can be used to apply a solarization effect to the specified brush. The solarization effect tries to simulate the look of photographic film exposed to light. The

second argument controls the intensity of the solarization effect and can be any value between 0 and 255, or a percentage specification inside a string.

Note that if `id` specifies a palette brush, `SolarizeBrush()` will just solarize the palette colors which makes this function really fast when used with palette brushes.

INPUTS

`id`	brush to solarize
`level`	desired solarization level (0 to 255, or a string containing a percentage specification)

21.76 SwirlBrush

NAME

SwirlBrush – apply swirl effect to brush (V5.0)

SYNOPSIS

```
SwirlBrush(id, degrees)
```

FUNCTION

This command applies a swirl effect to the specified brush. The brush will be swirled around its center point by the specified amount of degrees. This can range from 0 for no swirling to 360 for the most dramatic swirling effect.

INPUTS

`id`	brush to swirl
`degrees`	desired swirl level (can be from 0 to 360)

21.77 TintBrush

NAME

TintBrush – tint brush (V1.5)

SYNOPSIS

```
TintBrush(id, color, level)
```

FUNCTION

This function tints the brush specified by `id` with the RGB color specified by `color` at a level which ranges from 0 to 255.

Starting with Hollywood 2.0, `level` can also be a string containing a percent specification, e.g. `"50%"`.

Note that if `id` specifies a palette brush, `TintBrush()` will just tint the palette colors which makes this function really fast when used with palette brushes.

INPUTS

`id`	identifier of the brush to tint
`color`	a RGB color to use for tinting

`level` tint level (0 to 255 or percent specification)

EXAMPLE

```
TintBrush(1, #RED, 128)
```

The code above gives brush number 1 a red look.

21.78 TransformBrush

NAME

TransformBrush – apply affine transformation to a brush (V4.5)

SYNOPSIS

```
TransformBrush(id, sx, rx, ry, sy[, smooth])
```

FUNCTION

This function can be used to apply affine transformation to a brush. You have to pass a 2x2 transformation matrix to this function that will define how each pixel in the brush will be transformed. This function is useful if you want to apply rotation and scaling at the same time. Of course, you could do this with calls to `ScaleBrush()` and then `RotateBrush()`, but this would lead to quality losses. If you do the transformation using `TransformBrush()` instead, everything will be done in a single run.

The 2x2 transformation matrix consists of four floating point factors:

`sx:` Specifies the amount of scaling on the x axis. This must not be zero. If it is negative, the image is flipped on the y axis.

`rx:` Specifies the amount of rotation on the x axis. This can be 0.

`ry:` Specifies the amount of rotation on the y axis. This can be 0.

`sy:` Specifies the amount of scaling on the y axis. This must not be zero. If it is negative, the image is flipped on the x axis.

The identity matrix is defined as

```
( 1  0 )
( 0  1 )
```

If you pass this matrix, then no transformation will be applied because there is no rotation and no scaling defined. I.e. if Hollywood applied this matrix to every pixel in your brush, the result would be just a copy of the brush. But of course, if `TransformBrush()` detects that you passed an identity matrix, it will return immediately and do nothing.

The optional argument `smooth` can be set to `True` if Hollywood shall use interpolation during the transformation. This yields results that look better but interpolation is quite slow.

Please note: You should always do transformation operations using the original brush. For instance, if you transform brush 1 to 12x8 pixels and then transform it back to 640x480, you will get a messed image. Therefore you should always keep the original brush and transform only copies of it.

Note that for vector brushes, `TransformBrush()` will always operate on the untransformed brush. This means that any previous transformations applied to the brush using `TransformBrush()`, `ScaleBrush()`, or `RotateBrush()` will be undone when calling `TransformBrush()`.

INPUTS

`id`	identifier of the brush to be transformed
`sx`	scale x factor; must never be 0
`rx`	rotate x factor
`ry`	rotate y factor
`sy`	scale y factor; must never be 0
`smooth`	optional: whether or not affine transformation should use interpolation

EXAMPLE

```
angle = Rad(45)    ; convert degrees to radians
TransformBrush(1, Cos(angle), Sin(angle), -Sin(angle), Cos(angle))
```

The code above rotates brush number 1 by 45 degrees using a 2x2 transformation matrix.

21.79 TrimBrush

NAME

TrimBrush – remove blank spaces from a brush (V4.6)

SYNOPSIS

```
left, top = TrimBrush(id)
```

FUNCTION

This command will remove all bordering blank spaces from the specified brush. If the specified brush does not have a mask or an alpha channel, this function does nothing. `TrimBrush()` will return the number of columns removed from the left side as well as the number of rows removed from the top side of the brush. You can calculate the right/bottom trim values on your own then.

INPUTS

`id`	brush to trim

RESULTS

`left`	number of columns removed from the left
`top`	number of rows removed from the top

EXAMPLE

```
CreateBrush(1, 640, 480, #BLACK, {Mask = True, Clear = True})
SelectBrush(1, #SELMODE_COMBO)
SetFillStyle(#FILLCOLOR)
Box(#CENTER, #CENTER, 100, 100, #RED)
EndSelect
```

```
TrimBrush(1)
```

The code above will create a 640x480 brush, draw a 100x100 red rectangle to it and then trim it down to 100x100 because all borders are blank.

21.80 Vector brushes

When you load a vector image using `LoadBrush()` or `@BRUSH`, you will get a special type of brush: a vector brush. When loading normal images like PNG, JPEG, etc. you will always get a raster brush. You can find out the type of a brush by querying the `#ATTRTYPE` attribute using `GetAttribute()`.

The advantage of a vector brush is that you can scale and/or transform it without any quality losses. For example, the `ScaleBrush()`, `RotateBrush()`, and `TransformBrush()` commands will produce high-quality images when used with vector brushes. Also, when layers and the layer scaling engine are enabled, vector brush layers will be automatically scaled without any quality losses. Therefore, if you only use vector brushes and TrueType text in your script, it can be scaled to any resolution and will still appear perfectly crisp.

The disadvantage of vector brushes is that they are not supported by all image manipulating functions. Of course, all the major functions like `DisplayBrush()`, `DisplayBrushPart()`, `ScaleBrush()`, `RotateBrush()`, etc. work fine with vector brushes but certain functions like `TintBrush()`, `GammaBrush()`, `SelectBrush()`, etc. can only handle raster brushes. So if you want to use one of these functions on a vector brush, you need to convert the brush to a raster brush first. This can be done by using the `RasterizeBrush()` function. Keep in mind, though, that as soon as you convert a vector brush to a raster brush, it will no longer be possible to scale and transform it without sacrifices in quality.

21.81 WaterRippleBrush

NAME

WaterRippleBrush – apply water ripple effect to brush (V5.0)

SYNOPSIS

```
WaterRippleBrush(id[, wavelength, amplitude, phase, cx, cy])
```

FUNCTION

This command can be used to apply a water ripple effect to the specified brush. The five optional arguments allow you to control the parameters of the water ripple effect. `Wavelength`, `amplitude`, and `phase` control the look of the ripples, whereas the `cx` and `cy` arguments can be used to specify the center point of the ripple. This point must be specified as a floating point value ranging from 0.0 (left/top) to 1.0 (right/bottom). The center of the brush is thus at position 0.5/0.5.

INPUTS

`id` brush to apply water ripples to

`wavelength`
optional: desired wavelength of the ripple (defaults to 32)

`amplitude`
optional: desired ripple amplitude (defaults to 1)

`phase`	optional: desired ripple phase (defaults to 0)
`cx`	optional: x center point of ripple (defaults to 0.5)
`cy`	optional: y center point of ripple (defaults to 0.5)

21.82 WriteBrushPixel

NAME

WriteBrushPixel – write single pixel to brush (V5.0)

SYNOPSIS

```
WriteBrushPixel(id, x, y, color[, trans])
```

FUNCTION

This command writes the specified RGB color to the brush passed in `id` at the position specified in the first two arguments. If the optional argument `trans` is specified and the brush has a mask or an alpha channel, then the value specified in the optional argument is written to the transparency channel of the brush. If the brush has a mask, `trans` may be set to 0 or 1, and if the brush has an alpha channel, then `trans` must be in the range of 0 to 255.

You can also write pixels to brushes by selecting the brush as the output device using `SelectBrush()` and then call the `Plot()` function. Using `WriteBrushPixel()`, however, is faster for most cases because it allows you to modify color and transparency channels at the same time and you can also avoid the overhead that is generated by calling `SelectBrush()` and `EndSelect()`.

Note that when `id` specifies a palette brush, `color` must not be an RGB color but a pen value.

INPUTS

`id`	identifier of the brush to use
`x`	x offset
`y`	y offset
`color`	RGB color or pen to write to brush
`trans`	optional: value to copy to the transparency channel

EXAMPLE

```
WriteBrushPixel(1, 100, 100, #RED)
```

Plots a red pixel at position 100:100 in brush 1.

22 Clipboard library

22.1 ClearClipboard

NAME
ClearClipboard – clear clipboard contents (V4.5)

SYNOPSIS
`ClearClipboard()`

FUNCTION
This function can be used to empty the clipboard. Any data that is currently stored in the clipboard will be deleted.

INPUTS
none

22.2 GetClipboard

NAME
GetClipboard – read clipboard contents (V4.5)

SYNOPSIS
`type[, data] = GetClipboard()`

FUNCTION
This function retrieves the data that is currently stored in the clipboard. `GetClipboard()` will return two values: The first return value indicates the format of the data in the clipboard, and the second return value then contains the format-specific data. Currently, Hollywood supports two different kinds of clipboard data: Text and images.

If there is currently text stored in the clipboard, `GetClipboard()` will return `#CLIPBOARD_TEXT` as the first return value, and a string containing the text in the clipboard as the second return value.

If there is currently an image stored in the clipboard, `GetClipboard()` will return `#CLIPBOARD_IMAGE` as the first return value, and the second return value will be a handle to a brush which will contain the image from the clipboard. Once you are done working with this brush, you should call `FreeBrush()` on it to free any memory allocated by it.

If there is neither text nor an image in the clipboard, `GetClipboard()` will return `#CLIPBOARD_UNKNOWN` to you. The second return value is unused in this case. If the clipboard is empty, then `#CLIPBOARD_EMPTY` will be returned.

To get notified whenever the contents of the clipboard change, you can install the `ClipboardChange` event handler using `InstallEventHandler()`.

If you just want to find out the format of the data currently in the clipboard without actually receiving a copy of this data, you can use the `PeekClipboard()` command. But keep in mind that the data on the clipboard can change at any time. So there is no guarantee that the data that was on the clipboard when you called `PeekClipboard()` will still be there when you call `GetClipboard()`.

INPUTS
none

RESULTS

`type` format of the data currently in the clipboard or `#CLIPBOARD_EMPTY` or `#CLIPBOARD_UNKNOWN`

`data` optional: if the first return value is not `#CLIPBOARD_EMPTY` or `#CLIPBOARD_UNKNOWN` then this return value contains the actual data retrieved from the clipboard; the data that is returned here depends on the format (see above)

EXAMPLE

```
SetClipboard(#CLIPBOARD_TEXT, "Hello clipboard!")
type, data = GetClipboard()
If type = #CLIPBOARD_TEXT
  NPrint(data)
Else
  NPrint("No text on the clipboard!")
EndIf
```

The code above puts the text "Hello clipboard!" on the clipboard and then retrieves the current clipboard contents. If no other program meddles with the clipboard between `SetClipboard()` and `GetClipboard()`, this code should print "Hello clipboard!" to the screen then.

22.3 PeekClipboard

NAME
PeekClipboard – examine clipboard contents (V4.5)

SYNOPSIS

```
type = PeekClipboard()
```

FUNCTION

This function peeks into the clipboard and returns the format of the data that is currently stored in the clipboard. Currently, Hollywood recognizes only text and image data in the clipboard. Thus, this function will return one of the following type specifiers:

`#CLIPBOARD_TEXT:`
Text is currently on the clipboard.

`#CLIPBOARD_IMAGE:`
Graphics data is currently on the clipboard.

`#CLIPBOARD_EMPTY:`
The clipboard is currently empty.

`#CLIPBOARD_UNKNOWN:`
Hollywood did not recognize the data currently stored on the clipboard.

To retrieve the data from the clipboard, you have to use `GetClipboard()` but keep in mind that the data on the clipboard can change at any time. So there is no guarantee

that the data that was on the clipboard when you called `PeekClipboard()` will still be there when you call `GetClipboard()`.

INPUTS

none

RESULTS

`type` value specifying the format of the data currently on the clipboard

22.4 SetClipboard

NAME

SetClipboard – change clipboard contents (V4.5)

SYNOPSIS

```
SetClipboard(type, data)
```

FUNCTION

This function can be used to put new data on the clipboard. The previous contents of the clipboard will be erased. Currently, you can put either text or an image on the clipboard with this function.

To put text on the clipboard, specify `#CLIPBOARD_TEXT` as `type` and pass a string containing the text to be put on the clipboard in the `data` argument.

To put an image on the clipboard, specify `#CLIPBOARD_IMAGE` as `type` and pass the identifier of a brush containing the image you wish to save to the clipboard in the `data` argument.

INPUTS

`type` format of the data specified in second argument

`data` depends on the format specified in argument 1; see above for more information

EXAMPLE

See Section 22.2 [GetClipboard], page 319.

23 Console library

23.1 AllocConsoleColor

NAME

AllocConsoleColor – allocate console color (V10.0)

SYNOPSIS

```
col = AllocConsoleColor(color)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function allocates the color specified by `color` for the current console and returns it. You can then make it the active color by passing it to functions like `SetConsoleColor()`. Color allocation is necessary because by default only a few predefined ANSI colors like `#BLACK`, `#WHITE`, `#RED`, etc. are available. If you want to use custom colors, you need to allocate them first.

When you're done with a color allocated by this function, call `FreeConsoleColor()` to a free the color. This is important to make sure that you don't run out of colors.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`color` color to allocate; this must be passed as an RGB color

RESULTS

`col` allocated color

EXAMPLE

```
EnableAdvancedConsole()
SetConsoleColor(1, AllocConsoleColor($FFA500))
ConsolePrint("Hello World")
RefreshConsole()
```

The code above prints the string "Hello World" in orange. Note that under normal circumstances you should free the console color when you're done with it. This part has been left out for readability reasons.

23.2 BeepConsole

NAME

BeepConsole – beep console (V10.0)

SYNOPSIS

```
BeepConsole()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function sounds the audible bell on the terminal. If that's not possible, it calls `FlashConsole()`.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

23.3 ClearConsole

NAME

ClearConsole – clear console (V10.0)

SYNOPSIS

```
ClearConsole()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function clears the console by copying the current background character set using `SetConsoleBackground()` to every cell of the window. `ClearConsole()` will also set the `Clear` flag from the `SetConsoleOptions()` command to `True` for the current window to ensure that the window is cleared on the next refresh. If you don't want that, use `EraseConsole()` instead. See Section 23.20 [EraseConsole], page 334, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

23.4 ClearConsoleStyle

NAME

ClearConsoleStyle – clear console style (V10.0)

SYNOPSIS

```
ClearConsoleStyle(style)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function clears the specified style(s) in the current console window. The `style` parameter can be one or more of the console style flags as described in the `SetConsoleStyle()` documentation. See Section 23.55 [SetConsoleStyle], page 358, for details. Since all style flags are bit masks you can combine multiple styles using the bitwise OR operator (`|`).

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`style` one or more style flags to clear

23.5 CloseConsole

NAME

CloseConsole – close console window (V10.0)

SYNOPSIS

```
CloseConsole()
```

PLATFORMS

Windows

FUNCTION

This closes a console window previously opened using `OpenConsole()`. This is only supported on Windows because only Windows distinguishes between non-console and console programs. See Section 23.44 [OpenConsole], page 349, for details.

INPUTS

none

23.6 ConsolePrint

NAME

ConsolePrint – print to console (V8.0)

SYNOPSIS

```
ConsolePrint(...)
```

FUNCTION

This function prints all arguments you specify to the console. You can specify as many arguments as you like and they may be of any type. If you pass multiple arguments to this function, they will be printed with a space to separate them.

`ConsolePrint()` will also automatically append a newline character to the end of its output. If you don't want that, use `ConsolePrintNR()` instead. See Section 23.7 [ConsolePrintNR], page 326, for details.

INPUTS

`...` at least one value to print to the console

EXAMPLE

```
ConsolePrint("The user entered", name$, "as his name and", age,
             "as his age!")
```

23.7 ConsolePrintNR

NAME

ConsolePrintNR – print to console without newline (V8.0)

SYNOPSIS

```
ConsolePrintNR(...)
```

FUNCTION

This does the same as `ConsolePrint()` but doesn't append a new line character to the string.

See Section 23.6 [ConsolePrint], page 325, for details.

INPUTS

`...`	at least one value to print to the console

EXAMPLE

```
ConsolePrintNR("Hello ")
ConsolePrintNR("World!")
ConsolePrintNR("\n")
```

This does the same as `ConsolePrint("Hello World!")`.

23.8 ConsolePrompt

NAME

ConsolePrompt – read user input from console (V8.0)

SYNOPSIS

```
s$ = ConsolePrompt(p$)
```

FUNCTION

This function can be used to prompt the user to enter a string in the console. `ConsolePrompt()` will present the string specified in `p$` as the prompt and halt the script's execution until the user has entered a string and confirmed his input using the RETURN key. The string will then be returned by this function.

INPUTS

`p$`	prompt to present to the user

RESULTS

`s$`	string entered by user

EXAMPLE

```
name$ = ConsolePrompt("What is your name? ")
age$ = ConsolePrompt("And your age? ")
home$ = ConsolePrompt("Where do you live? ")
ConsolePrint("Your name is", name$, "and you are", age$,
           "years old and live in", home$, "!")
```

The code above demonstrates the usage of the `ConsolePrompt()` function.

23.9 CopyConsoleWindow

NAME

CopyConsoleWindow – copy text from another console window (V10.0)

SYNOPSIS

```
CopyConsoleWindow(id[, overlay, srcx, srcy, dstx1, dsty1, dstx2, dsty2])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function copies the text from the console window specified by `id` to the currently active console window. If the `overlay` parameter is `True`, only non-blank characters are copied, i.e. all whitespace characters are ignored. If the optional arguments after `overlay` are left out, only those characters in the source window that intersect with the destination window are copied, so that the characters appear in the same physical position on the screen.

If you specify the optional arguments after `overlay`, the two windows needn't overlap. The arguments `srcx` and `srcy` specify the top left corner of the region to be copied. The arguments `dstx1`, `dsty1`, `dstx2`, and `dsty2` specify the region within the destination window to copy to. All positions must be given in characters.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`id`	id of the console window to copy from
`overlay`	optional: whether or not only non-blank characters should be skipped (defaults to `False`)
`srcx1`	optional: x offset in the source console window
`srcy1`	optional: y offset in the source console window
`dstx1`	optional: destination start x offset
`dsty1`	optional: destination start y offset
`dstx2`	optional: destination end x offset
`dsty2`	optional: destination end y offset

23.10 CreateConsoleWindow

NAME

CreateConsoleWindow – create a new console window (V10.0)

SYNOPSIS

```
[id] = CreateConsoleWindow(id, cols, rows, x, y[, parent, rel])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function creates a new window with the given number of columns and rows. The upper left corner of the window is at the coordinates specified by `x` and `y`. To create a new full-screen window, just set `cols`, `rows`, `x` and `y` all to 0. The new window will be given the identifier specified in `id`. If you pass `Nil` in `id`, `CreateConsoleWindow()` will automatically choose an identifier and return it.

If you set the `parent` argument to the identifier of an existing console window, the specified window will be set as the parent of the new console window so that the new console window will become a subwindow. You can pass -1 in `parent` to set the default console window as the parent. If you set `rel` to `True`, the `x` and `y` coordinates will be interpreted as relative to the parent window's origin instead of relative to the screen's origin.

Once you have created the console window, you can then use `SelectConsoleWindow()` to make it the active console window. See Section 23.49 [SelectConsoleWindow], page 353, for details.

Note that a console window is not the same as a window on your desktop. A console window merely is a certain area in the console that is regarded as an own window. This makes it easier to create textual user interfaces because you can divide your console screen into several windows and then all drawing is automatically clipped at the window edges.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`id`	id for the console window or `Nil` for auto id selection
`cols`	number of columns for the window
`rows`	number of rows for the window
`x`	left window offset
`y`	top window offset
`parent`	optional: identifier of a console window to be used as a parent (defaults to -1 which means the screen is the parent)
`rel`	optional: whether or not `x` and `y` specify relative offsets (defaults to `False`)

RESULTS

`id`	optional: identifier of the new console window; will only be returned if you pass `Nil` as argument 1 (see above)

EXAMPLE

```
EnableAdvancedConsole()
w, h = GetConsoleSize()
CreateConsoleWindow(1, 20, 20, (w - 20) / 2, (h - 20) / 2)
SelectConsoleWindow(1)
DrawConsoleBorder()
RefreshConsole()
```

The code above creates a new 20x20 window, draws a border around it and centers it on the screen.

23.11 DecomposeConsoleChr

NAME

DecomposeConsoleChr – decompose styled character (V10.0)

SYNOPSIS

```
ch, style, pen = DecomposeConsoleChr(c)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function can be used to extract the individual components of a styled character composed by `MakeConsoleChr()`. You have to pass the styled character in the `c` parameter and `DecomposeConsoleChr()` will return the Unicode codepoint of the character, the styling flags and the pen that should be used to draw the character. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`c`	styled character as composed by `MakeConsoleChr()`

RESULTS

`ch`	Unicode codepoint of character
`style`	console style flags
`pen`	pen that shall be used to draw the correct

23.12 DeleteConsoleChr

NAME

DeleteConsoleChr – delete console character (V10.0)

SYNOPSIS

```
DeleteConsoleChr()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function deletes the character under the cursor in the current window. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position isn't changed.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

23.13 DeleteConsoleLine

NAME
DeleteConsoleLine – delete console line (V10.0)

SYNOPSIS
`DeleteConsoleLine()`

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function deletes the line under the cursor in the current window. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

23.14 DisableAdvancedConsole

NAME
DisableAdvancedConsole – stop advanced console mode (V10.0)

SYNOPSIS
`DisableAdvancedConsole()`

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function ends the advanced console mode and puts the console back to normal mode. Ending advanced console mode will also clear the console and restore the original contents from before advanced mode was started on the console.

To start advanced console mode, call the `EnableAdvancedConsole()` function.

INPUTS
none

23.15 DrawConsoleBorder

NAME

DrawConsoleBorder – draw border to console (V10.0)

SYNOPSIS

```
DrawConsoleBorder([ls, rs, ts, bs, tl, tr, bl, br])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function draws a border around the edge of the current console window. All arguments must be character codes that specify the character that should be put at the respective position. The `ls` parameter specifies the character to be drawn at the left side of the border, `rs` specifies the right side of the border, `ts` the top side and `bs` the bottom side. The `tl` parameter specifies the character to be drawn in the top left corner of the border, `tr` specifies the top right corner of the border, `bl` the bottom left corner and `br` the bottom right corner. If a parameter is 0, `DrawConsoleBorder()` will use its default character for the specified border position.

Characters must be passed as numeric values, not as strings. For normal characters this value is simply the Unicode codepoint of the respective character, e.g. 65 for 'A'. You can, however, also pass a special character code composed by the `MakeConsoleChr()` function. This function allows you to merge text formatting styles into the character code and it also supports special character codes like arrows or border pieces. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`ls`	optional: character for left side of border (defaults to 0)
`rs`	optional: character for right side of border (defaults to 0)
`ts`	optional: character for top side of border (defaults to 0)
`bs`	optional: character for bottom side of border (defaults to 0)
`tl`	optional: character for top left corner of border (defaults to 0)
`tr`	optional: character for top right corner of border (defaults to 0)
`bl`	optional: character for bottom left corner of border (defaults to 0)
`br`	optional: character for bottom right corner of border (defaults to 0)

EXAMPLE

```
EnableAdvancedConsole()
DrawConsoleBorder()
RefreshConsole()
```

The code above draws a border using the default characters around the current console window.

23.16 DrawConsoleBox

NAME

DrawConsoleBox – draw box to console (V10.0)

SYNOPSIS

```
DrawConsoleBox([xc, yc])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function draws a box around the edge of the current console window. The `xc` and `yc` arguments must be character codes that specify the character that should be used for the horizontal and vertical frames. If a parameter is 0, `DrawConsoleBox()` will use its default character for the specified border position.

Characters must be passed as numeric values, not as strings. For normal characters this value is simply the Unicode codepoint of the respective character, e.g. 65 for 'A'. You can, however, also pass a special character code composed by the `MakeConsoleChr()` function. This function allows you to merge text formatting styles into the character code and it also supports special character codes like arrows or border pieces. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`xc`	optional: character for horizontal frame (defaults to 0)
`yc`	optional: character for vertical frame (defaults 0)

EXAMPLE

```
EnableAdvancedConsole()
DrawConsoleBox()
RefreshConsole()
```

The code above draws a box using the default characters around the current console window.

23.17 DrawConsoleHLine

NAME

DrawConsoleHLine – draw horizontal line to console (V10.0)

SYNOPSIS

```
DrawConsoleHLine([ch, n])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function draws a horizontal line using the character code specified in `ch`. By default, the line is extended until the end of the window. Alternatively, you can use the `n`

parameter to tell `DrawConsoleHLine()` how many characters should be drawn. Passing 0 in `ch` makes `DrawConsoleHLine()` use a default character. The cursor position won't be advanced by this function.

Characters must be passed as numeric values, not as strings. For normal characters this value is simply the Unicode codepoint of the respective character, e.g. 65 for 'A'. You can, however, also pass a special character code composed by the `MakeConsoleChr()` function. This function allows you to merge text formatting styles into the character code and it also supports special character codes like arrows or border pieces. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`ch`	optional: character to draw with (defaults to 0)
`n`	optional: number of characters to draw (defaults to number of columns)

23.18 DrawConsoleVLine

NAME

DrawConsoleVLine – draw vertical line to console (V10.0)

SYNOPSIS

```
DrawConsoleVLine([ch, n])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function draws a vertical line using the character code specified in `ch`. By default, the line is extended until the end of the window. Alternatively, you can use the `n` parameter to tell `DrawConsoleVLine()` how many characters should be drawn. Passing 0 in `ch` makes `DrawConsoleVLine()` use a default character. The cursor position won't be advanced by this function.

Characters must be passed as numeric values, not as strings. For normal characters this value is simply the Unicode codepoint of the respective character, e.g. 65 for 'A'. You can, however, also pass a special character code composed by the `MakeConsoleChr()` function. This function allows you to merge text formatting styles into the character code and it also supports special character codes like arrows or border pieces. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`ch`	optional: character to draw with (defaults to 0)
`n`	optional: number of characters to draw (defaults to number of rows)

23.19 EnableAdvancedConsole

NAME
EnableAdvancedConsole – put console into advanced mode (V10.0)

SYNOPSIS
`EnableAdvancedConsole()`

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function will clear the console screen and put the console into advanced mode. Most functions of the console library require the console to be in advanced mode because things like free cursor positioning, colors, input handling, etc. are not available in standard console mode. Thus, `EnableAdvancedConsole()` is typically the first function to call when you want to have advanced control over the console.

To go back to normal console mode, call the `DisableAdvancedConsole()` function.

Please also read the chapter on Hollywood and the console to learn more about using Hollywood in console mode. See Section 3.1 [Console mode], page 31, for details.

INPUTS
none

EXAMPLE
```
EnableAdvancedConsole()
w, h = GetConsoleSize()
s$ = "Hello World!"
SetConsoleCursor((w - StrLen(s$)) / 2, h / 2)
ConsolePrintNR(s$)
RefreshConsole()
```

The code above prints the string `"Hello World"` centered in the console.

23.20 EraseConsole

NAME
EraseConsole – erase console (V10.0)

SYNOPSIS
`EraseConsole()`

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function clears the console by copying the current background character set using `SetConsoleBackground()` to every cell of the window. In contrast to `ClearConsole()` `EraseConsole()` won't set the `Clear` flag from the `SetConsoleOptions()` command to `True` for the current window. If you want that, use `ClearConsole()` instead. See Section 23.3 [ClearConsole], page 324, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

23.21 FlashConsole

NAME
FlashConsole – flash console (V10.0)

SYNOPSIS
`FlashConsole()`

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function flashes the screen, by inverting the foreground and background of every cell, pausing, and then restoring the original attributes.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

23.22 FormatConsoleLine

NAME
FormatConsoleLine – set style and color for several characters (V10.0)

SYNOPSIS
`FormatConsoleLine(n, style[, pen, fgcolor, bgcolor])`

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function sets the specified style and color for the next `n` characters on the current line without changing the existing text, or altering the existing style or color settings. Passing -1 for `n` means to apply the format to all characters until the end of the current line.

All console styles supported by the `SetConsoleStyle()` function can be passed in the `style` parameter. See Section 23.55 [SetConsoleStyle], page 358, for details. If you don't want to modify any style settings, you can pass the special style `#CONSOLESTYLE_NONE` for `style`. In that case, `FormatConsoleLine()` won't apply any styles.

The optional parameters `pen`, `fgcolor` and `bgcolor` can be used to change the color of the characters. They can be used in the same way as with `SetConsoleColor()`. See Section 23.52 [SetConsoleColor], page 355, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`n`	number of characters to format or -1 for all character in the current line
`style`	formatting style to apply or `#CONSOLESTYLE_NONE` to ignore the argument
`pen`	optional: color pen to use or 0 to reset text color to default (defaults to 0)
`fgcolor`	optional: desired foreground color for the pen; this must be either an RGB color from the list above or a color allocated by `AllocConsoleColor()` (defaults to `#NOCOLOR`)
`bgcolor`	optional: desired background color for the pen; this must be either an RGB color from the list above or a color allocated by `AllocConsoleColor()` (defaults to `#NOCOLOR`)

23.23 FreeConsoleColor

NAME

FreeConsoleColor – free console color (V10.0)

SYNOPSIS

```
FreeConsoleColor(color)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function frees the console color allocated by `AllocConsoleColor()`. It is important to call this function when you no longer need a color to make sure that you don't run out of colors.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`color`	color to free; this must have been allocated by `AllocConsoleColor()`

23.24 FreeConsoleWindow

NAME

FreeConsoleWindow – free console window (V10.0)

SYNOPSIS

```
FreeConsoleWindow(id)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function frees the console window specified by `id`. This must have been allocated by `CreateConsoleWindow()` before. See Section 23.10 [CreateConsoleWindow], page 327, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`id` id of the console window to free

23.25 GetAllocConsoleColor

NAME

GetAllocConsoleColor – get allocated color (V10.0)

SYNOPSIS

```
color = GetAllocConsoleColor(c)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function returns the RGB color of a console color allocated using `AllocConsoleColor()`. You have to pass the color allocated by `AllocConsoleColor()` in the `c` parameter.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`c` allocated color to get

RESULTS

`color` allocated color as an RGB color

23.26 GetConsoleBackground

NAME

GetConsoleBackground – get console background (V10.0)

SYNOPSIS

```
ch = GetConsoleBackground()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function returns the character that is currently used for background filling. This character can be set using `SetConsoleBackground()`. Note that the character can also be a special character composed by the `MakeConsoleChr()` function. To decompose such

characters into their constituents, you can call the `DecomposeConsoleChr()` function. See Section 23.11 [DecomposeConsoleChr], page 329, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

RESULTS

`ch` character used for filling

23.27 GetConsoleChr

NAME

GetConsoleChr – get console character (V10.0)

SYNOPSIS

```
ch = GetConsoleChr()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function gets the character at the current cursor position and returns it. Note that the character can also be a special character composed by the `MakeConsoleChr()` function. To decompose such characters into their constituents, you can call the `DecomposeConsoleChr()` function. See Section 23.11 [DecomposeConsoleChr], page 329, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

RESULTS

`ch` character at current cursor position

23.28 GetConsoleColor

NAME

GetConsoleColor – get console color (V10.0)

SYNOPSIS

```
pen, fgcolor, bgcolor = GetConsoleColor([cursor])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function returns the active color of the current console window. If the optional argument `cursor` is set to `True`, the color at the current cursor position will be returned. Otherwise the color of the current window will be returned.

`GetConsoleColor()` will return three values: The currently active pen and the current foreground and background colors.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`cursor`	optional: `True` to get the color at the cursor position, `False` to get the color of the current window (defaults to `False`)

RESULTS

`pen`	current color pen
`fgcolor`	current foreground color as an RGB color
`bgcolor`	current background color as an RGB color

23.29 GetConsoleControlChr

NAME

GetConsoleControlChr – get a standard control character (V10.0)

SYNOPSIS

```
ch = GetConsoleControlChr(ctrl)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function returns the character code of a standard console control character. The following control characters are currently supported and can be passed in the `ctrl` parameter:

`#CONSOLECHR_KILL:`
: Returns the KILL character.

`#CONSOLECHR_ERASE:`
: Returns the ERASE character.

`#CONSOLECHR_WORD:`
: Returns the DELETEWORD character.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`ctrl`	control character to get; must be one from the list above

RESULTS

`ch`	the requested control character in the format of the current terminal

23.30 GetConsoleCursor

NAME
GetConsoleCursor – get console cursor position (V10.0)

SYNOPSIS
```
x, y = GetConsoleCursor()
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function returns the cursor position of the current window. The position returned is relative to the upper left corner of the window, which is (0,0).

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

RESULTS

`x`	current x position
`y`	current y position

23.31 GetConsoleOrigin

NAME
GetConsoleOrigin – get console origin (V10.0)

SYNOPSIS
```
x, y = GetConsoleOrigin()
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function returns the x and y coordinates of the origin of the current console window.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

RESULTS

`x`	origin x position
`y`	origin y position

23.32 GetConsoleSize

NAME
GetConsoleSize – get console dimensions (V10.0)

SYNOPSIS
```
cols, rows = GetConsoleSize()
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function returns the size of the current window. The size is returned as the number of columns and rows available in the current window. A typical terminal size is 80x24, i.e. 24 rows with 80 characters per row.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

RESULTS

`cols`	number of columns in window
`rows`	number of rows in window

23.33 GetConsoleStr

NAME
GetConsoleStr – read string from console (V10.0)

SYNOPSIS
```
s$ = GetConsoleStr()
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function reads all characters from the current cursor position to the end of the line and returns them as a string.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

RESULTS

`s$`	string read from console

23.34 GetConsoleStyle

NAME
GetConsoleStyle – get console style (V10.0)

SYNOPSIS
```
style = GetConsoleStyle([cursor])
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function returns the style of the current console window. If the optional argument `cursor` is set to `True`, the style at the current cursor position will be returned. Otherwise the style for the current window will be returned.

The `style` return value will be a bitmask of style flags. See Section 23.55 [SetConsoleStyle], page 358, for all available console styles.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`cursor`	optional: `True` to get the style at the cursor position, `False` to get the style for the current window (defaults to `False`)

RESULTS

`style`	style flags

23.35 GetConsoleWindow

NAME
GetConsoleWindow – return active console window (V10.0)

SYNOPSIS
```
id = GetConsoleWindow()
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function returns the currently active console window. If no console window has been made active using `SelectConsoleWindow()`, -1 will be returned.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

RESULTS

`id`	id of the active console window or -1 if the default screen is active

23.36 HaveConsole

NAME
HaveConsole – check if there is a console (V10.0)

SYNOPSIS
```
ok = HaveConsole()
```

FUNCTION
This function returns `True` if the program has an attached console. This is only of use when using the non-console version of Hollywood on Windows. In that case, there is initially no console but it has to be manually opened using `OpenConsole()`. On all other platforms and in the console version of Hollywood on Windows, there'll always be a console so this function will always return `True`. See Section 23.44 [OpenConsole], page 349, for details.

INPUTS
none

RESULTS
`ok` `True` if a console is available, `False` otherwise

23.37 HideConsoleCursor

NAME
HideConsoleCursor – hide console cursor (V10.0)

SYNOPSIS
```
HideConsoleCursor()
```

PLATFORMS
Linux, macOS, Windows

FUNCTION
This function hides the console cursor.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS
none

23.38 InitConsoleColor

NAME
InitConsoleColor – init console color (V10.0)

SYNOPSIS
```
InitConsoleColor(pen[, fgcolor, bgcolor])
```

PLATFORMS
Linux, macOS, Windows

FUNCTION

This function can be used to set the foreground color of the console pen specified by `pen` to `fgcolor` and the background color to `bgcolor`. The pen could then be set as the text pen using `SetConsoleColor()`. See Section 23.52 [SetConsoleColor], page 355, for details.

The specified pen must be greater than 0. The number of available pens depends on the console. Typically, 256 pens are available which means that the highest pen number you can use is 255 but for the best compatibility you should use lower pen numbers because not all consoles might have 256 pens.

The `fgcolor` and `bgcolor` arguments can be either RGB values or colors allocated using `AllocConsoleColor()`. Note that only a few colors are available by default and can be passed in `fgcolor` or `bgcolor` without allocation. These are: `#BLACK`, `#WHITE`, `#RED`, `#GREEN`, `#BLUE`, `#YELLOW`, `#AQUA` (cyan), and `#FUCHSIA` (magenta). If you want to use other colors, you need to allocate them first using `AllocConsoleColor()`. See Section 23.1 [AllocConsoleColor], page 323, for details.

Note that terminals might choose to use a darkened version of the colors you specify here so don't be surprised if your color appears as grey even though you specified white. Many terminals are configured to treat grey as white. If you want to avoid this, allocate custom colors using `AllocConsoleColor()`.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`pen` color pen to use

`fgcolor` optional: desired foreground color; this must be either an RGB color from the list above or a color allocated by `AllocConsoleColor()` (defaults to `#NOCOLOR`)

`bgcolor` optional: desired background color; this must be either an RGB color from the list above or a color allocated by `AllocConsoleColor()` (defaults to `#NOCOLOR`)

EXAMPLE

```
EnableAdvancedConsole()
SetConsoleStyle(#CONSOLESTYLE_BOLD)
InitConsoleColor(1, #BLACK, #WHITE)
SetConsoleColor(1)
ConsolePrint("Hello World")
RefreshConsole()
```

The code above prints the string "Hello World" in black on white background.

23.39 InsertConsoleChr

NAME

InsertConsoleChr – insert console character (V10.0)

SYNOPSIS

`InsertConsoleChr(ch)`

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function inserts the character specified by `ch` into the current console window at the current cursor position. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor is not advanced.

The character must be passed as a numeric value, not a string. For normal characters `ch` simply specifies the Unicode codepoint of the respective character, e.g. 65 for 'A'. The `ch` argument, however, can also be a special character code composed by the `MakeConsoleChr()` function. This function allows you to merge text formatting styles into the character code and it also supports special character codes like arrows or border bits. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`ch` character to insert

EXAMPLE

```
EnableAdvancedConsole()
InsertConsoleChr('A')
RefreshConsole()
```

The code above adds the character 'A' to the console.

23.40 InsertConsoleLine

NAME

InsertConsoleLine – insert console line (V10.0)

SYNOPSIS

`InsertConsoleLine()`

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function inserts a blank line above the current line. The bottom line is lost.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

23.41 InsertConsoleStr

NAME

InsertConsoleStr – insert console string (V10.0)

SYNOPSIS

```
InsertConsoleStr(s$)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function inserts the string specified by `s$` into the current console window at the current cursor position. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor is not advanced.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`s$` string to insert

EXAMPLE

```
EnableAdvancedConsole()
InsertConsoleStr("Hello World!")
RefreshConsole()
```

The code above prints the string "Hello World!" to the console.

23.42 MakeConsoleChr

NAME

MakeConsoleChr – merge style and color into character code (V10.0)

SYNOPSIS

```
ch = MakeConsoleChr(c[, style, pen])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function can be used to create a formatted character code that can be passed to all functions that accepted a numeric character code, e.g. `InsertConsoleChr()`. `MakeConsoleChr()` takes the character passed in `c` and applies the style settings passed in `style` and the color settings of the pen passed in `pen` to it.

The character must be passed by its numeric Unicode codepoint, not as a string. So for the character 'A' you would have to pass the value 65 in `c`. Alternatively, you can also pass one of the following special characters in `c`:

`#CONSOLECHR_BLOCK:`
Solid block

`#CONSOLECHR_BOARD:`
: Board of squares

`#CONSOLECHR_BTEE:`
: Bottom "T"

`#CONSOLECHR_BULLET:`
: Bullet

`#CONSOLECHR_CKBOARD:`
: Checkerboard

`#CONSOLECHR_DARROW:`
: Down arrow

`#CONSOLECHR_DEGREE:`
: Degree symbol

`#CONSOLECHR_DIAMOND:`
: Diamond

`#CONSOLECHR_GEQUAL:`
: Greater than or equal

`#CONSOLECHR_HLINE:`
: Horizontal line

`#CONSOLECHR_LANTERN:`
: Lantern symbol

`#CONSOLECHR_LARROW:`
: Left arrow

`#CONSOLECHR_LEQUAL:`
: Less than or equal

`#CONSOLECHR_LLCORNER:`
: Lower left box corner

`#CONSOLECHR_LRCORNER:`
: Lower right box corner

`#CONSOLECHR_LTEE:`
: Left "T"

`#CONSOLECHR_NEQUAL:`
: Not equal

`#CONSOLECHR_PI:`
: Pi

`#CONSOLECHR_PLMINUS:`
: Plus/minus sign

`#CONSOLECHR_PLUS:`
: Plus sign, cross, or four-corner piece

`#CONSOLECHR_RARROW:`
: Right arrow

`#CONSOLECHR_RTEE:`
: Right "T"

`#CONSOLECHR_S1:`
: Scan line 1

`#CONSOLECHR_S3:`
: Scan line 3

`#CONSOLECHR_S7:`
: Scan line 7

`#CONSOLECHR_S9:`
: Scan line 9

`#CONSOLECHR_STERLING:`
: Pounds sterling symbol

`#CONSOLECHR_TTEE:`
: Top "T"

`#CONSOLECHR_UARROW:`
: Up arrow

`#CONSOLECHR_ULCORNER:`
: Upper left box corner

`#CONSOLECHR_URCORNER:`
: Upper right box corner

`#CONSOLECHR_VLINE:`
: Vertical line

The `style` parameter supports all console styles offered by the `SetConsoleStyle()` function. See Section 23.55 [SetConsoleStyle], page 358, for details. If you don't want to set any style flags, you can pass the special style `#CONSOLESTYLE_NONE` for `style`. In that case, `MakeConsoleChr()` won't apply any styles.

The optional parameter `pen` can be used to define the pen that should be used to draw the character. Fore- and background colors of that pen can be initialized using the `InitConsoleColor()` function. See Section 23.38 [InitConsoleColor], page 343, for details.

To decompose a character that contains style or color formatting into its constituents, use the `DecomposeConsoleChr()` function. See Section 23.11 [DecomposeConsoleChr], page 329, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`c` character as a Unicode codepoint

`style` formatting style to apply or `#CONSOLESTYLE_NONE` to ignore the argument

`pen` optional: color pen to use or -1 to skip setting a pen (defaults to -1)

RESULTS

`ch` formatted character code

23.43 MoveConsoleWindow

NAME

MoveConsoleWindow – move console window (V10.0)

SYNOPSIS

```
MoveConsoleWindow(id, x, y)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function moves the console window specified by `id` to the position specified by `x` and `y`. This position must be in characters and it must not be off-screen.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`id` id of the console window to move

`x` desired x position for the window

`y` desired y position for the window

23.44 OpenConsole

NAME

OpenConsole – open console window (V10.0)

SYNOPSIS

```
OpenConsole()
```

PLATFORMS

Windows

FUNCTION

Windows distinguishes between non-console and console programs. That's why Hollywood can compile two different types of programs on Windows: Non-console programs and console programs. The difference between non-console programs and console programs is that console programs will automatically open a console when they are started whereas non-console programs won't do that. However, it's possible to manually open a console in non-console programs by calling this function.

On all other platforms there's no such distinction between console and non-console programs which is why this function is only supported on Windows and only if you use the non-console version of Hollywood.

To close the console opened by this function, simply call the `CloseConsole()` function.

Please also read the chapter on Hollywood and the console to learn more about using Hollywood in console mode. See Section 3.1 [Console mode], page 31, for details.

INPUTS

none

23.45 ReadConsoleKey

NAME

ReadConsoleKey – read console key (V10.0)

SYNOPSIS

```
key = ReadConsoleKey()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function reads a character from the current console window. By default, it will wait until the user presses a key. If you want it to return immediately in case the user hasn't pressed a key, you need to set the `Delay` option to `False` in `SetConsoleOptions()`. In that case, `ReadConsoleKey()` will return `#CONSOLEKEY_NONE` in case no key has been pressed.

If `ReadConsoleKey()` is set to wait until a key is pressed, its behaviour is also influenced by the `CBreak` setting of `SetConsoleOptions()`. If `CBreak` is set to `True`, which is also the default, one key press is enough to make `ReadConsoleKey()` stop blocking and return. If `CBreak` is set to `False`, however, it will block until a newline occurs.

By default, the key that the user presses is echoed in the console. If you don't want that, set the `Echo` tag in `SetConsoleOptions()` to `False`.

Finally, `ReadConsoleKey()` can also read function keys like F1, F2, cursor keys, ESC and so on. If you want `ReadConsoleKey()` to support function keys, you must put the console into keypad mode by setting the `Keypad` tag in `SetConsoleOptions()` to `True`. If you have done that, `ReadConsoleKey()` can also return the following function keys:

```
#CONSOLEKEY_ENTER
#CONSOLEKEY_UP
#CONSOLEKEY_DOWN
#CONSOLEKEY_RIGHT
#CONSOLEKEY_LEFT
#CONSOLEKEY_BACKSPACE
#CONSOLEKEY_DEL
#CONSOLEKEY_F1
#CONSOLEKEY_F2
#CONSOLEKEY_F3
#CONSOLEKEY_F4
#CONSOLEKEY_F5
#CONSOLEKEY_F6
```

```
#CONSOLEKEY_F7
#CONSOLEKEY_F8
#CONSOLEKEY_F9
#CONSOLEKEY_F10
#CONSOLEKEY_F11
#CONSOLEKEY_F12
#CONSOLEKEY_F13
#CONSOLEKEY_F14
#CONSOLEKEY_F15
#CONSOLEKEY_F16
#CONSOLEKEY_INSERT
#CONSOLEKEY_HOME
#CONSOLEKEY_END
#CONSOLEKEY_PRINT
#CONSOLEKEY_PAGEUP
#CONSOLEKEY_PAGEDOWN
#CONSOLEKEY_IC
#CONSOLEKEY_EIC
```

As you can see, many `SetConsoleOptions()` options influence the behaviour of `ReadConsoleKey()`. See Section 23.54 [SetConsoleOptions], page 357, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

RESULTS

`key` key that has been pressed or `#CONSOLEKEY_NONE` if no key has been pressed and the console is in `NoDelay` mode

23.46 ReadConsoleStr

NAME

ReadConsoleStr – read console string (V10.0)

SYNOPSIS

```
s$ = ReadConsoleStr()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function reads a string from the current console window and returns it. In contrast to `ReadConsoleKey()`, `ReadConsoleStr()` isn't compatible with the `Delay` option of `SetConsoleOptions()`. It will only work correctly if `Delay` is set to `True`, which is also the default. This implies that `ReadConsoleStr()` is always meant to block until there's some input on the console.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

RESULTS

`s$` string read from the console

23.47 RefreshConsole

NAME

RefreshConsole – refresh console (V10.0)

SYNOPSIS

```
RefreshConsole()
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function draws the changes in the current console window to the screen. To optimize drawing, the console typically won't refresh automatically when changes are made to its contents. Instead, `RefreshConsole()` must usually be called manually to make the console redraw itself. If you want Hollywood to automatically refresh the console, you can set the `Immediate` flag to `True` in `SetConsoleOptions()` but this can lead to flickery drawing.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

none

23.48 ScrollConsole

NAME

ScrollConsole – scroll console lines (V10.0)

SYNOPSIS

```
ScrollConsole(n)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function scrolls the console window up or down by the specified delta `n`. A positive value in `n` scrolls the console up `n` lines while a negative `n` scrolls the console down `n` lines.

Before you can use this function you must set the `Scroll` tag to `True` in `SetConsoleOptions`. See Section 23.54 [SetConsoleOptions], page 357, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`n` number of lines to scroll up or down; positive values scroll up, negative values scroll down

23.49 SelectConsoleWindow

NAME

SelectConsoleWindow – make console window active (V10.0)

SYNOPSIS

```
SelectConsoleWindow(id)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function makes the console window specified by `id` the active one. All commands of the console library will then target this console window. The `id` parameter must be the identifier of a console window allocated by `CreateConsoleWindow()`. See Section 23.10 [CreateConsoleWindow], page 327, for details. The special value -1 can be passed in `id` to make the default console screen the active window.

Make sure that you do not confuse `SelectConsoleWindow()` with the similarly named functions `SelectBrush()`, `SelectBGPic()`, `SelectAnim()`, `SelectMask()`, and `SelectAlphaChannel()`. All of these functions require you to call `EndSelect()` when you are done with them, but `SelectConsoleWindow()` does not have this requirement. In fact, it works in a completely different way so you must never call `EndSelect()` for `SelectConsoleWindow()`. If you want to return to the previously active console window, you must call `SelectConsoleWindow()` again. Calling `EndSelect()` to restore the previously active console window will definitely not work.

To find out the currently active console window, call the `GetConsoleWindow()` function. See Section 23.35 [GetConsoleWindow], page 342, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`id` id of the console window to make active

EXAMPLE

```
EnableAdvancedConsole()
w, h = GetConsoleSize()
CreateConsoleWindow(1, 20, 20, (w - 20) / 2, (h - 20) / 2)
SelectConsoleWindow(1)
DrawConsoleBorder()
RefreshConsole()
```

The code above creates a new 20x20 window, draws a border around it and centers it on the screen.

23.50 SetAllocConsoleColor

NAME

SetAllocConsoleColor – change allocated color (V10.0)

SYNOPSIS

```
SetAllocConsoleColor(c, color)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function changes a color allocated using `AllocConsoleColor()`. You have to pass the color allocated by `AllocConsoleColor()` in the `c` parameter and the new color in the `color` parameter (as an RGB color).

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`c`	allocated color to modify
`color`	new color as an RGB color

23.51 SetConsoleBackground

NAME

SetConsoleBackground – set console background (V10.0)

SYNOPSIS

```
SetConsoleBackground(ch)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function fills the background of the current console window with the character specified by `ch`. The character must be passed as a numeric value, not a string. For normal characters `ch` simply specifies the Unicode codepoint of the respective character, e.g. 65 for 'A'. The `ch` argument, however, can also be a special character code composed by the `MakeConsoleChr()` function. This function allows you to merge text formatting styles into the character code and it also supports special character codes like arrows or border bits. See Section 23.42 [MakeConsoleChr], page 346, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`ch`	character to use for filling

EXAMPLE

```
EnableAdvancedConsole()
SetConsoleBackground('=')
RefreshConsole()
```

The code above fills the console background with '=' characters.

23.52 SetConsoleColor

NAME

SetConsoleColor – set console color (V10.0)

SYNOPSIS

```
SetConsoleColor(pen[, fgcolor, bgcolor])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function can be used to set the specified pen as the text pen. If the `fgcolor` and `bgcolor` arguments are also specified, the pen will be initialized to the colors passed in `fgcolor` and `bgcolor` before making it the text pen. This is only possible if the pen number is greater than 0. Pen 0 is reserved for the default text color. Thus, if you pass 0 in `pen`, `fgcolor` and `bgcolor` will be ignored and the text color will be reset to the console's default text color.

The number of available pens depends on the console. Typically, 256 pens are available which means that the highest pen number you can use is 255 but for the best compatibility you should use lower pen numbers because not all consoles might have 256 pens.

The `fgcolor` and `bgcolor` arguments can be either RGB values or colors allocated using `AllocConsoleColor()`. Note that only a few colors are available by default and can be passed in `fgcolor` or `bgcolor` without allocation. These are: `#BLACK`, `#WHITE`, `#RED`, `#GREEN`, `#BLUE`, `#YELLOW`, `#AQUA` (cyan), and `#FUCHSIA` (magenta). If you want to use other colors, you need to allocate them first using `AllocConsoleColor()`. See Section 23.1 [AllocConsoleColor], page 323, for details.

Note that terminals might choose to use a darkened version of the colors you specify here so don't be surprised if your color appears as grey even though you specified white. Many terminals are configured to treat grey as white. If you want to avoid this, allocate custom colors using `AllocConsoleColor()`.

To initialize the foreground and background color of a pen without making it the active text pen, you can use the `InitConsoleColor()` function. See Section 23.38 [InitConsoleColor], page 343, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`pen` color pen to use or 0 to reset text color to default

`fgcolor` optional: desired foreground color for the pen; this must be either an RGB color from the list above or a color allocated by `AllocConsoleColor()` (defaults to `#NOCOLOR`)

`bgcolor` optional: desired background color for the pen; this must be either an RGB color from the list above or a color allocated by `AllocConsoleColor()` (defaults to `#NOCOLOR`)

EXAMPLE

```
EnableAdvancedConsole()
SetConsoleStyle(#CONSOLESTYLE_BOLD)
SetConsoleColor(1, #BLACK, #WHITE)
ConsolePrint("Hello World")
RefreshConsole()
```

The code above prints the string "Hello World" in black on white background.

23.53 SetConsoleCursor

NAME

SetConsoleCursor – set console cursor position (V10.0)

SYNOPSIS

```
SetConsoleCursor(x, y)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function moves the cursor of the current window to the location specified by `x` and `y`. This does not move the physical cursor of the terminal until `RefreshConsole()` is called. The position specified is relative to the upper left corner of the window, which is (0,0).

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`x` desired x position

`y` desired y position

EXAMPLE

```
EnableAdvancedConsole()
w, h = GetConsoleSize()
s$ = "Hello World!"
SetConsoleCursor((w - StrLen(s$)) / 2, h / 2)
ConsolePrintNR(s$)
RefreshConsole()
```

The code above prints the string "Hello World" centered in the console.

23.54 SetConsoleOptions

NAME

SetConsoleOptions – configure console settings (V10.0)

SYNOPSIS

`SetConsoleOptions(table)`

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function can be used to configure several settings that determine how the console should behave in advanced console mode. You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

`SetConsoleOptions()` accepts a single table argument that can contain the following tags:

`Delay:` If this tag is set to `False`, `ReadConsoleKey()` won't wait until a key is pressed but will return immediately if no key is pressed. Defaults to `True`.

`HalfDelay:`
When `Delay` is set to `True`, `HalfDelay` can be used to specify a time limit to be specified, in tenths of a second. This causes `ReadConsoleKey()` to block for that period before returning `#CONSOLEKEY_NONE` if no key has been received. If set, this value must be between 1 and 255.

`Echo:` If this tag is set to `False`, typed characters won't be echoed in the console. Defaults to `True`.

`Keypad:` If this tag is set to `True`, `ReadConsoleKey()` will also be able to read function keys like F1, F2, cursor keys, ESC, etc. Defaults to `False`.

`Scroll:` If this tag is set to `True`, the console will automatically scroll the console when writing past the end of the console window. Defaults to `False`.

`Clear:` If this tag is set to `True`, `RefreshConsole()` will clear the screen completely and redraw the entire screen. Defaults to `False`.

`Leave:` If this tag is set to `True`, the cursor will be left wherever a screen update happens to leave it. This can be useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled. Defaults to `False`.

`Immediate:`
If this tag is set to `True`, the console window will be refreshed every time a change is made to it. Defaults to `False`.

`CBreak:` When `Delay` is set to `True`, `CBreak` controls which characters can make `ReadConsoleKey()` stop blocking and return. If `CBreak` is set to `True`, one key press will be enough to make `ReadConsoleKey()` stop blocking and return. If `CBreak` is set to `False`, however, `ReadConsoleKey()` will block until a newline occurs.

`Newline:` If this is set to `True`, newlines are translated to carriage returns on input. If you don't want that, set `Newline` to `False`. Defaults to `True`.

INPUTS

`table` table containing one or more settings to modify (see above)

23.55 SetConsoleStyle

NAME

SetConsoleStyle – set console style (V10.0)

SYNOPSIS

```
SetConsoleStyle(style)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function sets the style of the current console window to the one passed in the `style` parameter. This can be one or more of the following flags:

`#CONSOLESTYLE_NORMAL:`
Reset text style to default. This cannot be combined with any other styles.

`#CONSOLESTYLE_BOLD:`
Make text bold.

`#CONSOLESTYLE_ITALIC:`
Make text italic.

`#CONSOLESTYLE_UNDERLINED:`
Underline text.

`#CONSOLESTYLE_STANDOUT:`
Highlight text.

`#CONSOLESTYLE_BLINK:`
Make text blink.

`#CONSOLESTYLE_REVERSE:`
Reverse video on text.

`#CONSOLESTYLE_DIM:`
Half bright effect for text. This is not supported everywhere.

`#CONSOLESTYLE_PROTECT:`
Protected mode for text. This is not supported everywhere.

`#CONSOLESTYLE_INVISIBLE:`
Invisible text. This is not supported everywhere.

`#CONSOLESTYLE_ALTCHARSET:`
Use the alternate character set.

Note that all style flags are bit masks so you can combine multiple styles using the bitwise OR operator (|).

To clear console one or more console styles, use the `ClearConsoleStyle()` function. See Section 23.4 [ClearConsoleStyle], page 324, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`style` one or more style flags to set (see above)

EXAMPLE

```
EnableAdvancedConsole()
SetConsoleStyle(#CONSOLESTYLE_BOLD|#CONSOLESTYLE_UNDERLINED)
ConsolePrint("Hello World!")
RefreshConsole()
```

The code above prints the text "Hello World!" in bold and underlined style.

23.56 SetConsoleTitle

NAME

SetConsoleTitle – set console title (V10.0)

SYNOPSIS

```
SetConsoleTitle(t$)
```

PLATFORMS

Windows

FUNCTION

This function changes the title of the console window to the text specified in `t$`. This is currently only supported on Windows.

INPUTS

`t$` new title for the console window

23.57 ShowConsoleCursor

NAME

ShowConsoleCursor – show console cursor (V10.0)

SYNOPSIS

```
ShowConsoleCursor([normal])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function can be used to show the console cursor. The optional `normal` argument can be used to set whether you'd like to have the normal or the block cursor. If `normal`

is set to `True`, the normal cursor will be shown. If it is set to `False`, the block cursor will be used. By default, `ShowConsoleCursor()` will show the normal cursor.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`normal` optional: whether to show the normal or the block cursor (defaults to `True`, which means show the normal cursor)

23.58 StartConsoleColorMode

NAME

StartConsoleColorMode – put console into color mode (V10.0)

SYNOPSIS

```
StartConsoleColorMode([defcolor])
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function puts the console in color mode. Normally, it's not necessary to call this function because all console library functions that operate in color mode will enable color mode automatically. There's one exception, though: If you don't want to use the default colors, you have to start color mode manually and set the `defcolor` argument to `False`. In that case color mode will be enabled but without using the default colors. Note that if you do that, it's important to call `StartConsoleColorMode()` before any other console library functions that might enable color mode so if you use this function the best idea is to call it right after `EnableAdvancedConsole()`.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`defcolor` optional: whether or not to use the default colors (defaults to `True`)

23.59 TouchConsoleWindow

NAME

TouchConsoleWindow – touch console window (V10.0)

SYNOPSIS

```
TouchConsoleWindow(id)
```

PLATFORMS

Linux, macOS, Windows

FUNCTION

This function marks the whole console window specified by `id` as needing refresh. By default, only the areas in the window that have changed are marked for refresh. If you want to force a refresh of the whole window, call `TouchConsoleWindow()` on it.

The `id` parameter must be the identifier of a console window allocated by `CreateConsoleWindow()`. See Section 23.10 [CreateConsoleWindow], page 327, for details.

You must enable advanced console mode using `EnableAdvancedConsole()` before you can use this function. See Section 23.19 [EnableAdvancedConsole], page 334, for details.

INPUTS

`id` id of the console window to use

24 Debug library

24.1 Assert

NAME

Assert – fail if given expression is false (V5.0)

SYNOPSIS

```
Assert(expr)
```

FUNCTION

This command checks whether or not the given expression is `False` (or `Nil`) and causes an error if this is the case. This is mainly useful for debugging purposes. You can also pass a table or a function in `expr`. In that case, `expr` will be considered `True`.

This call can be disabled by specifying the `-nodebug` console argument when running a script or applet. In that case, calling `Assert()` will do nothing at all. See Section 3.2 [Console arguments], page 33, for details.

INPUTS

`expr` expression to check

EXAMPLE

```
a = 5
b = 0
Assert(b <> 0)
c = a / b
```

In the above code, `Assert()` is used to make sure that we don't divide by zero. `Assert()` will prevent such an error by checking `b` against zero.

24.2 CloseResourceMonitor

NAME

CloseResourceMonitor – close Hollywood's resource monitor (V4.5)

SYNOPSIS

```
CloseResourceMonitor()
```

FUNCTION

This function will close Hollywood's inbuilt resource monitor. You can bring it up again by calling `OpenResourceMonitor()`.

See Section 24.7 [OpenResourceMonitor], page 366, for more information about Hollywood's resource monitor.

INPUTS

none

24.3 DebugOutput

NAME

DebugOutput – enable/disable debug output

SYNOPSIS

```
DebugOutput(enable)
```

FUNCTION

This function enables or disables debug output to the default debug device. If debug output is enabled, Hollywood will print information about any function it calls so you can easily track down problems.

This call can be disabled by specifying the `-nodebug` console argument when running a script or applet. In that case, calling `DebugOutput()` will do nothing at all. See Section 3.2 [Console arguments], page 33, for details.

INPUTS

`enable` enable flag; `True` to enable debug output, `False` to disable it

EXAMPLE

```
DebugOutput(TRUE)
```

The above code enables debug output.

24.4 DebugPrint

NAME

DebugPrint – print to debug device (V2.0)

FORMERLY KNOWN AS

DebugVal (V1.x) ; DebugStr (V1.x)

SYNOPSIS

```
DebugPrint(...)
```

FUNCTION

This function prints all arguments you specify to the current debug device. This is usually your console window. You can specify as many arguments as you like and they may be of any type. If you pass multiple arguments to this function, they will be printed with a space to separate them.

`DebugPrint()` will also automatically append a newline character to the end of its output. If you don't want that, use `DebugPrintNR()` instead. See Section 24.5 [DebugPrintNR], page 365, for details.

This function supersedes the DebugStr() and DebugVal() calls. They now simply point to this call.

This call can be disabled by specifying the `-nodebug` console argument when running a script or applet. In that case, calling `DebugPrint()` will do nothing at all. See Section 3.2 [Console arguments], page 33, for details.

Also note that when compiling an applet or executable, debugging will be automatically disabled unless you explicitly enable it by setting the `EnableDebug` tag to `True`

in `@OPTIONS`. So if you have compiled an applet or executable and you see that `DebugPrint()` doesn't do anything, the reason is probably that debugging is disabled.

INPUTS

`...` at least one value to print to the debug device

EXAMPLE

```
DebugPrint("The user entered", name$, "as his name and", age,
           "as his age!")
```

24.5 DebugPrintNR

NAME

DebugPrintNR – print to debug device without newline (V6.1)

SYNOPSIS

`DebugPrintNR(...)`

FUNCTION

This does the same as `DebugPrint()` but doesn't append a new line character to the string.

This call can be disabled by specifying the `-nodebug` console argument when running a script or applet. In that case, calling `DebugPrintNR()` will do nothing at all. See Section 3.2 [Console arguments], page 33, for details.

Also note that when compiling an applet or executable, debugging will be automatically disabled unless you explicitly enable it by setting the `EnableDebug` tag to `True` in `@OPTIONS`. So if you have compiled an applet or executable and you see that `DebugPrintNR()` doesn't do anything, the reason is probably that debugging is disabled.

See Section 24.4 [DebugPrint], page 364, for details.

INPUTS

`...` at least one value to print to the debug device

EXAMPLE

```
DebugPrintNR("Hello ")
DebugPrintNR("World!")
DebugPrintNR("\n")
```

This does the same as `DebugPrint("Hello World!")`.

24.6 DebugPrompt

NAME

DebugPrompt – read user input from debug device (V5.0)

SYNOPSIS

```
s$ = DebugPrompt(p$)
```

FUNCTION

This function can be used to prompt the user to enter a string in the current debug device. `DebugPrompt()` will present the string specified in `p$` as the prompt and halt the script's execution until the user has entered a string and confirmed his input using the RETURN key. The string will then be returned by this function.

This call can be disabled by specifying the `-nodebug` console argument when running a script or applet. In that case, calling `DebugPrompt()` will just return an empty string. See Section 3.2 [Console arguments], page 33, for details.

Also note that when compiling an applet or executable, debugging will be automatically disabled unless you explicitly enable it by setting the `EnableDebug` tag to `True` in `@OPTIONS`. So if you have compiled an applet or executable and you see that `DebugPrompt()` doesn't do anything, the reason is probably that debugging is disabled.

INPUTS

`p$`	prompt to present to the user

RESULTS

`s$`	string entered by user

EXAMPLE

```
name$ = DebugPrompt("What is your name? ")
age$ = DebugPrompt("And your age? ")
home$ = DebugPrompt("Where do you live? ")
DebugPrint("Your name is", name$, "and you are", age$,
           "years old and live in", home$, "!")
```

The code above demonstrates the usage of the `DebugPrompt()` function.

24.7 OpenResourceMonitor

NAME

OpenResourceMonitor – open Hollywood's resource monitor (V4.5)

SYNOPSIS

```
OpenResourceMonitor()
```

FUNCTION

This function will open Hollywood's inbuilt resource monitor. The resource monitor will display a list of all resources that Hollywood has currently in memory. The list is updated several times per second so it is always up to date.

The resource monitor is very useful to make sure the memory management of your script is correct. Although Hollywood features resource tracking and will automatically deallocate all resources when it terminates, it is still important for bigger projects to keep an eye on their resource management because otherwise, your program will consume more and more memory. If you do not keep an eye on your resources, it can often happen that the longer your program runs, the more memory it will consume, and that must never happen.

Hollywood's resource monitor conveniently allows you to keep an eye on your resources. If you have the resource monitor always open while you develop, you will easily notice

if there is a resource problem somewhere. For example, if you notice that brush or layer numbers are steadily increasing while your script runs, you should be alarmed and it is likely that there is something wrong with your code which you need to fix.

You can also enable the resource monitor directly at startup by using the `-resourcemonitor` console argument.

To close the resource monitor, simply close its window or just call the `CloseResourceMonitor()` function.

INPUTS

none

24.8 WARNING

NAME

WARNING – send a warning message to the debug device (V6.1)

SYNOPSIS

```
@WARNING msg$
```

FUNCTION

This preprocessor command will send the specified warning message to the debug device right before running the script. This allows you to conveniently store important information like to do lists, "fix me" parts, etc. alongside your source code and you will be reminded of them whenever you run your script.

This preprocessor command can be disabled by specifying the `-nodebug` console argument when running a script or applet. In that case, this preprocessor command will do nothing at all. See Section 3.2 [Console arguments], page 33, for details.

INPUTS

`msg$` error message to show

EXAMPLE

```
@WARNING "FIXME: support other image formats"
```

The code above will send the string "FIXME: support other image formats" to the debug device right before Hollywood runs the script.

25 Display library

25.1 Overview

A display is an area on the screen your script can draw to. Typically, a display is a window on your desktop screen but it can also be in full-screen mode and fill up the entire monitor space. In Hollywood, a display is always tied to a background picture (BGPic). The background picture is what will be initially shown to the user when your display becomes visible.

The background picture attached to a display must always be of the same size as the display. Thus, if you change the display size, e.g. by using `ChangeDisplaySize()` your background picture will automatically be scaled to fit the new dimensions because as already said before the display size is always the same as the current background picture size. If your window is resizeable, then the user may also adjust your display size. If he does, Hollywood will internally call `ChangeDisplaySize()` to adjust to the new dimensions.

If you choose to display a new background picture, e.g. by using the `DisplayBGPic()` command, and the dimensions of the new background picture differ from the dimensions of your current background picture, then your display will also be resized to fit the new dimensions.

At startup, Hollywood will display the background picture that has been assigned the identifier 1. If you haven't declared a background picture that uses the identifier 1 using the `@BGPIC` preprocessor command, Hollywood will create this background picture automatically for you and attach it to your display. The background picture will use the fill style and dimensions specified in the `@DISPLAY` preprocessor command for display 1 in your script.

Displays can also have transparent areas. Hollywood supports displays with alpha transparency (256 levels of transparency) and monochrome transparency.

Here is a minimal script which creates a 1024x768 display filled with the color red:

```
@DISPLAY {Width = 1024, Height = 768, Color = #RED}
WaitLeftMouse
End
```

Displays can also automatically scale their content using one of Hollywood's inbuilt scaling engines: Auto scaling or layer scaling. When enabling one of those scaling engines, the script will think it is running in its original resolution although the display is promoting it to an entirely different resolution using the selected scaling engine. See Section 25.18 [Scaling engines], page 401, for details.

Hollywood displays can also run in palette mode. Displays will switch to palette mode whenever a BGPic that uses a palette is displayed. Palette mode displays behave quite differently than normal true color displays so there are some important things to consider when using displays in palette mode. See Section 25.16 [Palette mode displays], page 400, for details.

Hollywood's display library also supports multiple monitors. You can specify the monitor a display should be opened on. See Section 25.14 [Multi-monitor support], page 397, for details.

25.2 ActivateDisplay

NAME

ActivateDisplay – activate a display (V4.5)

SYNOPSIS

```
ActivateDisplay(id[, nofront])
```

FUNCTION

This command can be used to activate the specified display. Activating a display just means that Hollywood tells the window manager of the host operating system to give the focus to this display. Activating a display does not make the display the current output device for Hollywood's graphics library. If you want to select a display, as the current output device, you have to use `SelectDisplay()` which will also activate the display if you do not explicitly forbid this.

See Section 25.20 [SelectDisplay], page 405, for more information on the difference between active displays and displays that are selected as the current output device.

Starting with Hollywood 5.0 there is a new optional argument called `nofront`. If you set this to `True`, the display will be activated but it will not be moved to the front of the windows' stacking order. This argument is only handled on AmigaOS compatible systems because active windows in the background are not supported on other operating systems.

INPUTS

`id`	identifier of the display that shall be activated
`nofront`	optional: `True` if display should not be brought to the front (defaults to `False` which means pop display to front) (V5.0)

25.3 BACKFILL

NAME

BACKFILL – configure backfill settings for script (V4.5)

SYNOPSIS

```
@BACKFILL table
```

FUNCTION

Important note: This preprocessor command is deprecated since Hollywood 6.0. As Hollywood 6.0 introduced support for multiple monitors, there could also be multiple backfills (one for every display as displays could be on separate monitors). That is why backfills should be set up using the `@DISPLAY` preprocessor command or the `CreateDisplay()` function now. You can still use this preprocessor command but it will affect the first display only.

This preprocessor command can be used to configure the backfill settings for your script. Backfills can be used to create a shielding window that covers the whole area not occupied by your main display. You can use a static color as a backfill, a gradient, an image, or a texture. Before Hollywood 4.5, backfills were configured using the `@DISPLAY` preprocessor command. Hollywood 4.5, however, introduced multiple displays which made it necessary

to move the backfill settings into its own preprocessor command because there can be only a single backfill per script.

You have to pass a table to this command. The following table tags are currently recognized:

`Type:` This field is obligatory. It can be `Color`, `Gradient`, `Texture` or `Picture`. The type must be passed as a string here.

`Color:` If you've specified `Color` as backfill type, pass the desired backfill color in this field.

`StartColor, EndColor:`
If you've specified `Gradient` as backfill type, use these two fields to define the start and end colors for the gradient.

`Brush:` If you've specified `Texture` or `Picture` as backfill type, specify the identifier of the brush to use as the source image here. If you want to pass the file name directly, use the `BrushFile` tag instead.

`X,Y:` If you've specified `Picture` as backfill type, you can use these two fields to position the picture on the screen. They both default to `#CENTER`.

`BrushFile:`
If you've specified `Texture` or `Picture` as backfill type, you can specify the file name of the brush to use as the source image here. The file specified here will be linked to the applet/executable on compilation unless you set `LinkBrushFile` to `False`. If you want to pass a brush identifier, use the `Brush` tag instead. (V4.0)

`LinkBrushFile:`
If `BrushFile` has been specified this tag can be used to declare whether or not the brush file shall be linked into the applet/executable on compilation. Defaults to `True` which means that the brush file will be linked. (V4.0)

`Transparency:`
If backfill type is `Picture` you can specify an RGB color here that shall be shown transparently. Defaults to `#NOTRANSPARENCY`. (V4.0)

`ScalePicture:`
If backfill type is `Picture` you can use this tag to define whether or not the picture shall be scaled to fit the backfill window's dimensions. Defaults to `False`. (V4.0)

`SmoothScale:`
Set this tag to `True` if you want to have interpolated scaling of the picture that should be used as a backfill image. This tag is only handled if `ScalePicture` has been set to `True`. Defaults to `False`. (V6.0)

Alternatively, backfill settings can also be configured from the command line. If you want to disable that, you should compile your scripts using the `-locksettings` console argument.

You might also want to specify the `HideTitleBar` tag in `@SCREEN`. If you specify `HideTitleBar`, the backfill will also shield the current screen's title bar (Amiga) or Finder's menu bar (macOS).

INPUTS

`table` table declaring the style of the script's backfill

EXAMPLE

```
@BACKFILL {Type = "Gradient", StartColor = #BLACK, EndColor = #BLUE}
```

This declaration will install a gradient from black to blue as the backfill.

25.4 ChangeDisplayMode

NAME

ChangeDisplayMode – switch between window and full screen mode (V4.5)

SYNOPSIS

```
ChangeDisplayMode(mode[, table])
```

DEPRECATED SYNTAX

```
ChangeDisplayMode(mode[, width, height, table])
```

FUNCTION

This function can be used to change the display mode to the mode specified in the `mode` parameter. This can be one of the following modes:

`#DISPMODE_FULLSCREEN:`
: Switch to full screen mode. Note that this will switch the monitor's resolution which might not be supported on all systems. Alternatively, you can also use `#DISPMODE_FULLSCREENSCALE` (see below) which will simply scale the display to the monitor's resolution. If you choose to use `#DISPMODE_FULLSCREEN`, you can pass the desired monitor resolution in the `Width` and `Height` tags of the optional table argument (see below). If you don't set `Width` and `Height`, the best monitor resolution for the display's current dimensions will be chosen automatically. If the display is already in full screen mode and you pass `#DISPMODE_FULLSCREEN` in the `mode` argument, `ChangeDisplayMode()` can be used to change the current monitor resolution to a different one.

`#DISPMODE_WINDOWED:`
: Switch to windowed mode. This mode can be used to switch a display back to windowed mode. Obviously, it only makes sense to use this mode on displays which are currently full screen.

`#DISPMODE_FULLSCREENSCALE:`
: Switch into scaled full screen mode. This mode will make the display full screen without changing the monitor's resolution. Instead, the display's graphics will be scaled to the monitor's current resolution. Thus, they will fill the whole screen even though the monitor didn't change its resolution. By default, Hollywood's auto scaling engine will be used for scaling but you can set the `LayerScale` tag in the optional table argument (see below) to `True` to use the layer scaling engine instead. See Section 25.18 [Scaling engines], page 401, for details. Note, however, that `#DISPMODE_FULLSCREENSCALE` can

become quite slow on platforms which don't support hardware-accelerated scaling. (V9.0)

`#DISPMODE_MODESWITCH:`
This switches between display modes. If the display is currently windowed, it will switch to full screen. If the display is currently full screen, it will switch to windowed. Note that the full screen mode chosed by `#DISPMODE_MODESWITCH` can be both, `#DISPMODE_FULLSCREEN` and `#DISPMODE_FULLSCREENSCALE`. `#DISPMODE_MODESWITCH` uses the same logic as the ALT+RETURN hotkey that switches a Hollywood display between full screen and windowed mode. See the documentation of the `ScaleSwitch` tag in the documentation of the `@DISPLAY` preprocessor command for details. See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

Starting with Hollywood 6.0 this function accepts an optional table argument which allows you to configure further options:

`Width:` If `mode` is `#DISPMODE_FULLSCREEN`, this tag can be used to specify the width of the resolution the monitor should be switched to. You can also pass the special constant `#NATIVE` here to indicate that Hollywood should use the monitor's native width.

`Height:` If `mode` is `#DISPMODE_FULLSCREEN`, this tag can be used to specify the height of the resolution the monitor should be switched to. You can also pass the special constant `#NATIVE` here to indicate that Hollywood should use the monitor's native height.

`LayerScale:`
If `mode` is `#DISPMODE_FULLSCREENSCALE`, this tag can be used to make Hollywood use the layer scaling engine instead of the auto scaling engine. See Section 25.18 [Scaling engines], page 401, for details. Defaults to `False`. (V9.0)

`KeepProportions:`
If `mode` is `#DISPMODE_FULLSCREENSCALE`, you can activate proportional scaling by setting this tag to `True`. Defaults to `False`. (V9.0)

`SmoothScale:`
If `mode` is `#DISPMODE_FULLSCREENSCALE`, you can activate interpolated scaling by setting this tag to `True`. Defaults to `False`. (V9.0)

`Monitor:` This tag allows you to specify the monitor that should be used. Monitors are counted from 1 which is the primary monitor. By default, the monitor currently associated with the active display is used.

`Backfill:`
This tag allows you to configure the backfill setting for this display. The table you have to specify here has to follow the same conventions as its counterpart that can be passed to the `Backfill` tag of the `@DISPLAY` preprocessor command. See Section 25.8 [DISPLAY], page 380, for details.

Note that starting with Hollywood 6.0 it is possible to have more than one display in full screen mode since Hollywood 6.0 introduces support for multiple monitors. This makes it possible to have several displays running in full screen mode on separate displays.

To find out whether or not the desired display mode can be handled by the current monitor, use the `GetDisplayModes()` function.

INPUTS

`mode` mode to switch into (see above)

`table` optional: table configuring further options (see above) (V6.0)

EXAMPLE

```
ChangeDisplayMode(#DISPMODE_FULLSCREEN, {Width = 1024, Height = 768})
NPrint("We are now in full screen mode. Press left mouse to return\n" ..
   "to windowed mode.")
WaitLeftMouse
ChangeDisplayMode(#DISPMODE_WINDOWED)
NPrint("We are back in windowed mode now.")
WaitLeftMouse
```

The code above goes into 1024x768 full screen mode, waits for left mouse button to be pressed and then returns to windowed mode again.

25.5 ChangeDisplaySize

NAME

ChangeDisplaySize – change the dimensions of the current display

SYNOPSIS

```
ChangeDisplaySize(width, height[, args])
```

FUNCTION

This function changes the dimensions of the currently active display to the specified dimensions. The background picture associated with the current display will be scaled to fit the new dimensions. Therefore you can also use this function for scaling background pictures.

New in V2.0: You can pass `#KEEPASPRAT` as either `width` or `height`. Hollywood will calculate the size then automatically by taking the aspect-ratio of the display into account.

Starting with Hollywood 2.0, `width` and `height` can also be a string containing a percent specification, e.g. `"50%"`.

New in Hollywood 4.0: You can pass a table in the optional third argument to specify further options. Currently, the table can contain the following fields:

`X:` Specifies the new x-position for the display. If you want the display to keep its current x-position, pass the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

`Y:` Specifies the new y-position for the display. If you want the display to keep its current y-position, pass the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

`Smooth:` Specifies whether or not the graphics shall be scaled using anti-alias interpolation. Defaults to `False`.

Starting with Hollywood 7.0 you can also pass the special constant `#NATIVE` in the `width` and `height` parameters. In that case, Hollywood will use the dimensions of the display's host device.

INPUTS

`width` desired new width for the display

`height` desired new height for the display

`args` optional: further configuration options (V4.0)

EXAMPLE

```
ChangeDisplaySize(320, 240)
```

This changes the display size to 320x240.

25.6 CloseDisplay

NAME

CloseDisplay – close a display (V4.5)

SYNOPSIS

```
CloseDisplay(id)
```

FUNCTION

This function closes a currently open display. Please note that it will not free the display. You can still call `SelectDisplay()` on it even if it is closed. You can also make it visible again later using `OpenDisplay()` if necessary.

If you do not want to close the display completely but only minimize it, you can use the `HideDisplay()` command to achieve this. If you want to dispose of a display completely, you have to use the `FreeDisplay()` command instead of `CloseDisplay()`.

INPUTS

`id` identifier of the display to close

25.7 CreateDisplay

NAME

CreateDisplay – create a new display (V4.5)

SYNOPSIS

```
[id] = CreateDisplay(id[, table])
```

FUNCTION

This function can be used to create a new display which you can open using `OpenDisplay()`. You have to pass an identifier for the new display or `Nil`. If you pass `Nil`, `CreateDisplay()` will return a handle to the new display which you can then use to refer to this display.

Furthermore, you should pass a table in the second argument to configure the style for the new display. Please note that every display must have a BGPic associated with it. Thus, it is advised that you always specify the `BGPic` tag in the optional table when creating a display. If you do not specify the `BGPic` tag, `CreateDisplay()` will create a new BGPic for the new display automatically. The newly created BGPic will be of the size specified in `Width` and `Height` and it will be filled according to the style specified in `FillStyle`. If you specify the `BGPic` tag, `Width`, `Height` and `FillStyle` are ignored.

Also note that the same BGPic cannot be associated with multiple displays. Each BGPic must only be associated with a single display. It is not possible to have BGPic 1 associated with display 1 and 2, for example. Simply make a copy of the BGPic using `CopyBGPic()` if you need to use a single BGPic with multiple displays.

The optional table argument recognizes the following tags:

`BGPic:` Specifies the BGPic that shall be attached to the new display. You need to specify either this tag or the `Width`, `Height` and `FillStyle` tags. See further notes above.

`Width, Height:`
See Section 25.8 [DISPLAY], page 380, for details. Ignored if `BGPic` tag is set.

`X, Y:` See Section 25.8 [DISPLAY], page 380, for details.

`Mode:` See Section 25.8 [DISPLAY], page 380, for details.

`Title:` See Section 25.8 [DISPLAY], page 380, for details.

`Borderless:`
See Section 25.8 [DISPLAY], page 380, for details.

`Sizeable:`
See Section 25.8 [DISPLAY], page 380, for details.

`Fixed:` See Section 25.8 [DISPLAY], page 380, for details.

`Backfill:`
See Section 25.8 [DISPLAY], page 380, for details.

`ScrWidth, ScrHeight:`
See Section 25.8 [DISPLAY], page 380, for details.

`ScrDepth:`
See Section 25.8 [DISPLAY], page 380, for details.

`NoHide:` See Section 25.8 [DISPLAY], page 380, for details.

`NoModeSwitch:`
See Section 25.8 [DISPLAY], page 380, for details.

`NoClose:` See Section 25.8 [DISPLAY], page 380, for details.

`Active:` See Section 25.8 [DISPLAY], page 380, for details.

`HidePointer:`
See Section 25.8 [DISPLAY], page 380, for details.

`UseQuartz:`
See Section 25.8 [DISPLAY], page 380, for details.

`ScaleMode:`
See Section 25.8 [DISPLAY], page 380, for details.

`ScaleWidth, ScaleHeight:`
See Section 25.8 [DISPLAY], page 380, for details.

`SmoothScale:`
See Section 25.8 [DISPLAY], page 380, for details.

`DragRegion:`
See Section 25.8 [DISPLAY], page 380, for details.

`SizeRegion:`
See Section 25.8 [DISPLAY], page 380, for details.

`Layers:` See Section 25.8 [DISPLAY], page 380, for details.

`FitScale:`
See Section 25.8 [DISPLAY], page 380, for details. (V4.7)

`KeepProportions:`
See Section 25.8 [DISPLAY], page 380, for details. (V4.7)

`FillStyle:`
See Section 25.8 [DISPLAY], page 380, for details.. Ignored if `BGPic` tag is set. Defaults to `#FILLCOLOR`. (V5.0)

`Color:` See Section 25.8 [DISPLAY], page 380, for details.

`TextureBrush:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`TextureX, TextureY:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientStyle:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientAngle:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientStartColor, GradientEndColor:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientCenterX, GradientCenterY:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientBalance:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientBorder:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`GradientColors:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.0)

`KeepScreenOn:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.1)

`PubScreen:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`HideFromTaskbar:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.3)

`HideOptionsMenu:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.3)

`Orientation:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.3)

`DisableBlanker:`
See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`Menu:` See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`Monitor:` See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`XServer:` See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`ScreenTitle:`
See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`ScreenName:`
See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`RememberPosition:`
See Section 25.8 [DISPLAY], page 380, for details. (V6.1)

`Maximized:`
See Section 25.8 [DISPLAY], page 380, for details. (V7.0)

`TrapRMB:` See Section 25.8 [DISPLAY], page 380, for details. (V7.0)

`NoScaleEngine:`
See Section 25.8 [DISPLAY], page 380, for details. (V7.0)

`NoLiveResize:`
See Section 25.8 [DISPLAY], page 380, for details. (V7.0)

`NativeUnits:`
See Section 25.8 [DISPLAY], page 380, for details. (V7.0)

`AlwaysOnTop:`
See Section 25.8 [DISPLAY], page 380, for details. (V7.1)

`NoCyclerMenu:`
See Section 25.8 [DISPLAY], page 380, for details. (V8.0)

`HideTitleBar:`
See Section 25.8 [DISPLAY], page 380, for details. (V8.0)

`Subtitle:`
See Section 25.8 [DISPLAY], page 380, for details. (V8.0)

`SingleMenu:`
See Section 25.8 [DISPLAY], page 380, for details. (V8.0)

`ScaleFactor:`
See Section 25.8 [DISPLAY], page 380, for details. (V8.0)

`ImmersiveMode:`
See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`Palette:` See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`FillPen:` See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`SoftwareRenderer:`
See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`VSync:` See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`ScaleSwitch:`
See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`UserTags:`
See Section 25.8 [DISPLAY], page 380, for details. (V10.0)

After the display has been successfully created, you can open it by calling `OpenDisplay()`, you can draw to it by using `SelectDisplay()` and you can close it using `CloseDisplay()`.

See Section 25.20 [SelectDisplay], page 405, for an in-depth discussion of using multiple displays in Hollywood.

This command is also available from the preprocessor: Use `@DISPLAY` to create displays at startup time!

INPUTS

`id` identifier for the display or `Nil` for auto id select

`table` optional: further configuration options for loading operation

RESULTS

`id` optional: identifier of the new display; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateDisplay(2, {BGPic = 2, Active = True})
OpenDisplay(2)
NPrint("Hello World")
```

The code above creates a new display and attaches BGPic number 2 to it. The display will inherit the size and graphics from BGPic. Then we open the display and print "Hello World" into it.

```
CreateDisplay(2, {Width = 800, Height = 600, Borderless = True,
   Color = #WHITE, Active = True})
OpenDisplay(2)
```

The code above creates and opens a 800x600 borderless display with white background. Because we did not specify a BGPic for this display, `CreateDisplay()` will create one automatically and attach it to the new display. You can get a handle to the automatically created BGPic by querying `#ATTRBGPIC` on the new display using `GetAttribute()`.

25.8 DISPLAY

NAME

DISPLAY – create new display (V2.0)

SYNOPSIS

```
@DISPLAY [id,] table
```

FUNCTION

This preprocessor command creates a new display using the attributes specified in the optional table argument. Specifying an identifier for the display is only needed if your script uses multiple displays. If you are using just a single display, you can leave out the identifier completely and Hollywood will then use the identifier 1 for the display. If you are using multiple displays, style guide suggests that you use the identifier 1 for your first/main display.

If you want to create displays dynamically at run time, you can use the function `CreateDisplay()` for this.

The style of the new display is configured by setting a number of tags in the optional table argument. The following style tags are currently supported by `@DISPLAY`:

`Title:` Use this field to set the title for this display. This title will be shown then in display's window bar (if the window has a border). Default title is `"Hollywood"`.

`X,Y:` These two allow you to define where on the host screen this display should open. Using absolute values here is rather counterproductive because you mostly do not know the size of the screen your window will be opened on (except you are using full screen mode). It is wiser to use Hollywood's special coordinate constants here. If you do not set these fields, the display will always be opened in the center of the screen. You can also move the display later by calling `MoveDisplay()`.

`BGPic:` Specifies the BGPic that shall be attached to the new display. Every display needs an associated BGPic. Thus, if you do not specify this tag, Hollywood will create a new BGPic for this display automatically using the fill style specified in the `FillStyle` tag (see below) and the size specified in `Width` and `Height`. An exception is made for the display that uses the identifier 1: For reasons of compatibility the display number 1 will automatically be associated with the BGPic number 1 if there is one. (V4.5)

`Width, Height:`
You only have to use these fields if you did not specify a background picture in `BGPic` (see above) and if you want your display size's dimensions to be different from the default size 640x480. If this applies, set these fields to the

desired dimensions and Hollywood will open such an initial display for you. These tags are ignored if you specify a BGPic. Starting with Hollywood 7.0 you can also set these tags to the special constant `#NATIVE`. In that case, Hollywood will use the dimensions of the display's host device.

`Desktop:` If you set this field to `True`, the initial background picture will be a copy of your desktop screen. This can be used for some nice effects with that screen. Hollywood will also automatically open a borderless window if this field is `True`.

`Mode:` This tag allows you to specify the mode the display should be opened in. You have to pass one of the following strings to this tag:

`Windowed` Open display in windowed mode.

`FullScreen`
: Open in full screen mode. This can change your monitor's resolution to the dimensions which fit best to your display's dimensions. If you don't want that, take a look at the `FullScreenScale` and `FakeFullScreen` modes below.

`FullScreenScale`
: This is a special full screen mode which won't change your monitor's resolution. Instead, Hollywood's display will be resized to fit your monitor's dimensions. Additionally, this full screen mode will activate the auto scaling engine so that your display is automatically scaled to fit your monitor's dimensions. `FullScreenScale` will use auto scaling by default. If you would like it to use layer scaling, you have to set `ScaleMode` to `#SCALEMODE_LAYER` as well. `FullScreenScale` is especially useful on mobile devices whose display hardware has a hard-coded resolution and doesn't support resolution changes in the same way as an external monitor connected to a desktop computer does. The downside of `FullScreenScale` is that it is slower because Hollywood has to scale all rendering operations to the monitor's dimensions. (V7.0)

`AutoFullScreen:`
: This will put the display into full screen mode using the auto scaling engine instead of changing the monitor's resolution but only when running Hollywood on systems that support GPU-accelerated scaling. On all other platforms a normal full screen mode will be used, i.e. Hollywood will change the monitor's resolution to fit the current display dimensions. Currently, GPU-accelerated scaling is supported on Windows, macOS, Android, and iOS which means that on those platforms no monitor resolution change will occur because Hollywood can simply scale the graphics to fit to the current monitor dimensions. On AmigaOS compatibles and Linux, however, there will still be a monitor resolution change with this mode because Hollywood doesn't support GPU-accelerated scaling on those platforms. (V9.1)

`LayerFullScreen:`
: This will put the display into full screen mode using the layer scaling engine instead of changing the monitor's resolution but only when running Hollywood on systems that support GPU-accelerated scaling. On all other platforms a normal full screen mode will be used, i.e. Hollywood will change the monitor's resolution to fit the current display dimensions. Currently, GPU-accelerated scaling is supported on Windows, macOS, Android, and iOS which means that on those platforms no monitor resolution change will occur because Hollywood can simply scale the graphics to fit to the current monitor dimensions. On AmigaOS compatibles and Linux, however, there will still be a monitor resolution change with this mode because Hollywood doesn't support GPU-accelerated scaling on those platforms. (V9.1)

`FakeFullScreen`
: Open in fake full screen mode. This means that Hollywood will not change the monitor's resolution but the backfill window will be configured to shield the desktop completely. Thus, the user gets the impression as if Hollywood was running full screen, although it is running on the desktop.

`ModeRequester`
: This will open a display mode requester allowing the user to choose the desired full screen mode for this display.

`Ask`
: This will open a requester asking the user to choose between windowed and full screen mode.

`SystemScale:`
: If you choose this display mode, the host system's scaling factor will automatically be applied to your display. This can be useful on systems with high-DPI monitors. For example, if your display normally opens in 640x480 pixels and you run it on a monitor that uses twice as many dots per inch (DPI), using `SystemScale` mode will automatically scale your script to 1280x960 pixels so that it doesn't look tiny just because the system uses a high-DPI monitor. Note that by default, using `SystemScale` will activate the auto scaling engine. If you want it to use layer scaling instead, just use the `ScaleMode` tag to change this to layer scaling. Note that `SystemScale` uses the same scale mode as `ScaleFactor` internally, so displays using `SystemScale` will behave as if `ScaleFactor` was specified. It is even possible to specify the `ScaleFactor` tag on top of `SystemScale`, in that case the value specified in `ScaleFactor` will be multiplied by host system's default scaling factor. See Section 25.18 [Scaling engines], page 401, for details. Note that on Windows you must also set the `DPIAware` tag to `True` in the `@OPTIONS` preproces-

sor command in order to use `SystemScale`. See Section 52.25 [OPTIONS], page 1088, for details. (V8.0)

By default, windowed mode will be used.

`Borderless:`
Set this field to `True` if this display shall be a borderless window. Defaults to `False`.

`Sizeable:`
Set this field to `True` if this display shall be resizeable by the user. If `Borderless` is also set to `True`, the size widget will be invisible in the bottom right window corner. Defaults to `False`.

`Fixed:` If you set this field to `True`, Hollywood will create a fixed, non-draggable window. This is especially useful on full screen displays. Defaults to `False`.

`Backfill:`
This tag can be used to configure the backfill settings for this display. Backfills can be used to create a shielding window that covers the whole area not occupied by your main display. They are only supported if `Mode` has been set to `FullScreen` or `FakeFullScreen`. You can use a static color as a backfill, a gradient, an image, or a texture. You have to pass a table in this tag. The following table tags are currently recognized:

`Type:` This field is obligatory. It can be `Color`, `Gradient`, `Texture` or `Picture`. The type must be passed as a string here.

`Color:` If you've specified `Color` as backfill type, pass the desired backfill color in this field.

`StartColor, EndColor:`
If you've specified `Gradient` as backfill type, use these two fields to define the start and end colors for the gradient.

`Brush:` If you've specified `Texture` or `Picture` as backfill type, specify the identifier of the brush to use as the source image here. If you want to pass the file name directly, use the `BrushFile` tag instead.

`X,Y:` If you've specified `Picture` as backfill type, you can use these two fields to position the picture on the screen. They both default to `#CENTER`.

`HideTitleBar:`
If you set this tag to `True`, the backfill will also shield the host screen's title bar (for example Finder's title bar on macOS or the Workbench title bar on AmigaOS compatibles). Note that you can also specify `HideTitleBar` outside the `Backfill` tag because on Android and iOS `HideTitleBar` can also be used without a backfill. When used without a backfill, `HideTitleBar` hides the device's status bar on Android and iOS.

`BrushFile:`
: If you've specified `Texture` or `Picture` as backfill type, you can specify the file name of the brush to use as the source image here. The file specified here will be linked to the applet/executable on compilation unless you set `LinkBrushFile` to `False`. If you want to pass a brush identifier, use the `Brush` tag instead. (V4.0)

`LinkBrushFile:`
: If `BrushFile` is specified this tag can be used to declare whether or not the brush file shall be linked into the applet/executable on compilation. Defaults to `True` which means that the brush file will be linked. (V4.0)

`Transparency:`
: If backfill type is `Picture` you can specify a RGB color here that shall be shown transparently. Defaults to `#NOTRANSPARENCY`. (V4.0)

`ScalePicture:`
: If backfill type is `Picture` you can use this tag to define whether or not the picture shall be scaled to fit the backfill window's dimensions. Defaults to `False`. (V4.0)

`SmoothScale:`
: Set this tag to `True` if you want to have interpolated scaling of the picture that should be used as a backfill image. This tag is only handled if `ScalePicture` has been set to `True`. Defaults to `False`. (V6.0)

`ScrWidth, ScrHeight:`
: If `Mode` has been set to `FullScreen`, these tags allows you to set the desired dimensions for the full screen mode. Defaults to what has been set when creating the display. Starting with Hollywood 7.0 you can also set these tags to the special constant `#NATIVE`. In that case, Hollywood will use the dimensions of the display's host device. (V3.0)

`ScrDepth:`
: If `Mode` has been set to `FullScreen`, this tag allows you to set the desired depth for the full screen mode. Defaults to what has been set when creating the display. (V3.0)

`HidePointer:`
: If you specify this field, the mouse pointer will automatically be hidden as soon as Hollywood enters full screen or fake full screen mode. This argument has the advantage over the `HidePointer()` command in that it only hides the mouse pointer in full screen mode. If Hollywood opens in windowed mode, the mouse pointer will remain visible because hiding the mouse pointer in windowed mode usually causes confusion with the user. Defaults to `False`. (V3.0)

`NoModeSwitch:`
: If you set this tag to `True`, it will not be possible to switch this display between windowed and full screen mode by pressing the CMD+RETURN (LALT+RETURN on Windows) hotkey. If `NoModeSwitch` is specified, this display will always remain in its initial display mode and no switches between windowed and full screen will be allowed. Defaults to `False`. (V3.0)

`NoHide:`
: Set this tag to `True` if you do not want this display to have an iconify widget. If you do not specify this tag, your display will always get an iconify widget. Defaults to `False`. (V4.5)

`ScaleMode:`
: By setting this argument, you can choose a scaling engine for this display. `ScaleMode` can be set to one of the following parameters: `#SCALEMODE_LAYER` (uses layer scale engine), `#SCALEMODE_AUTO` (uses auto scaling engine) or `#SCALEMODE_NONE` (uses no scaling engine). If `ScaleMode` is not specified, the display's scaling mode will be set to `#SCALEMODE_NONE`, i.e. no scaling is active. When specifying `ScaleMode`, you will usually also want to set either the `ScaleWidth` and `ScaleHeight` arguments or the `ScaleFactor` or the `SystemScale` display mode to define the scaling dimensions (see below). See Section 25.18 [Scaling engines], page 401, for more information on Hollywood's scaling engines. (V4.5)

`ScaleWidth, ScaleHeight:`
: These two can be used to specify the desired scaling dimensions if a scaling engine is active (check the documentation of `ScaleMode` above). You can pass the size either as a direct value or you can pass a percentage string (i.e. `ScaleWidth="200%"`). If you pass a percentage string, the scaling size is set relative to the original size (i.e. `ScaleWidth="200%"` means twice the original width). Starting with Hollywood 7.0 you can also set these tags to the special constant `#NATIVE`. In that case, Hollywood will use the dimensions of the display's host device. (V4.5)

`SmoothScale:`
: If `ScaleMode` is set, you can use this argument to specify whether or not Hollywood shall use anti-aliased scaling. Defaults to `False` which means no anti-aliasing. Note that anti-aliased scaling is much slower than normal scaling. (V4.5)

`Hidden:`
: Set this to `True` if you want this display to be initially hidden. If you set this to `True`, the display will not be shown until you call `OpenDisplay()` on it. You could also use this tag to run a Hollywood script that does not open a display at all but please keep in mind that some commands (e.g. `WaitLeftMouse()`) only work with a visible display. (V4.5)

`Active:`
: This tag allows you to specify the display that will be active on startup. Please note that only one display can be the active one so it is not valid to set `Active` to `True` for multiple displays. This will yield undefined results. If you do not specify `Active` for any of your displays, Hollywood will make the display number 1 the active one by default. (V4.5)

`DragRegion:`
: This tag allows you to define a custom drag region for this display. Custom drag regions are only supported for borderless displays, so you need to set `Borderless` to `True` too if you use this tag. You can define multiple drag regions with this tag; this is why you have to pass a table which contains a list of tables, each defining a single rectangular region, to this tag. Each table in the list must have the following tags specified: `Type`, `X`, `Y`, `Width`, and `Height`. `Type` currently must always be set to `#BOX` because currently, only rectangular regions are supported. This might sound pretty complicated, but in fact it is really easy. All you have to remember is to pass a list of tables to this tag. Even if you only want a single rectangular drag region, you have to pass a list. See below for an example. (V4.5)

`SizeRegion:`
: This tag allows you to define a custom size region for this display. Custom size regions are only supported for borderless displays, so you need to set `Borderless` to `True`, too if you use this tag. You can define multiple size regions with this tag; this is why you have to pass a table which contains a list of tables, each defining a single rectangular region, to this tag. Each table in the list must have the following tags specified: `Type`, `X`, `Y`, `Width`, and `Height`. `Type` currently must always be set to `#BOX` because currently, only rectangular regions are supported. This might sound pretty complicated, but in fact it is really easy. All you have to remember is to pass a list of tables to this tag. Even if you only want a single rectangular size region, you have to pass a list. See below for an example. (V4.5)

`Layers:`
: Set this tag to `True` if you want to enable layers for this display. If you set this tag to `True`, you do not have to call `EnableLayers()` for this display again. See Section 34.1 [Layers introduction], page 647, for more information on layers.. This tag is set to `False` by default. (V4.5)

`UseQuartz:`
: This tag is only supported if Hollywood is running on macOS. If you set this tag to `True`, this display will draw its graphics using the Quartz 2D API. If it is set to `False`, QuickDraw will be used. Note that this argument is only supported by the PowerPC version of Hollywood. The x86/x64 versions of Hollywood for macOS will always use Quartz 2D. Defaults to `False`. (V4.5)

`NoClose:`
: Set this tag to `True` if this display shall not have a close box in its window frame. Think twice before using this tag because it might confuse the user and you must provide a replacement for closing the display (e.g. by listening to the escape key etc.). Defaults to `False`. (V4.5)

`FitScale:`
: This tag is only handled when either `#SCALEMODE_AUTO` or `#SCALEMODE_LAYER` is active in `ScaleMode` (see above). In that case, setting `FitScale` to `True` will set the scaling resolution of the display to the current screen's resolution so that the script will always fill out the whole screen. This is basically the same as passing the current screen's dimensions in `ScaleWidth` and `ScaleHeight`. Note that using `FitScale` might distort the appearance of

your script in case the current screen resolution uses a different aspect-ratio than your script. To prevent distortion, you have to use `KeepProportions` (see below). Defaults to `False`. (V4.7)

`KeepProportions:`
This tag is only handled when either `#SCALEMODE_AUTO` or `#SCALEMODE_LAYER` is active in `ScaleMode` (see above). In that case, passing `True` here will not allow the user to distort the resolution of the current script by resizing the window to odd sizes. Instead, black borders will be used to pad the non-proportional window regions. The display itself will always keep its aspect-ratio. This is very useful for scripts that should not be distorted. (V4.7)

`FillStyle:`
This tag allows you to define a background fill style for this display. This tag is only handled if the `BGPic` tag is not specified. The default setting for this tag is `#FILLCOLOR`. See Section 27.14 [SetFillStyle], page 498, for information on all available fill styles. (V5.0)

`Color:` This tag is only handled if you did not specify a BGPic and fill style is set to `#FILLCOLOR`. In that case, you can use this tag to specify the color for the background picture that will be automatically created for this display.

`TextureBrush:`
If the `FillStyle` tag has been set to `#FILLTEXTURE`, you can use this tag to specify the identifier of the brush that shall be used for texturing. (V5.0)

`TextureX, TextureY:`
These tags control the start offset inside the texture brush and are only supported if `FillStyle` has been set to `#FILLTEXTURE`. See Section 27.14 [SetFillStyle], page 498, for details. (V5.0)

`GradientStyle:`
If the `FillStyle` tag has been set to `#FILLGRADIENT`, you can use this tag to specify the gradient type to use. This can be `#LINEAR`, `#RADIAL`, or `#CONICAL`. (V5.0)

`GradientAngle:`
Specifies the orientation of the gradient if filling style style is set to `#FILLGRADIENT`. The angle is expressed in degrees. Only possible for `#LINEAR` and `#CONICAL` gradients. (V5.0)

`GradientStartColor, GradientEndColor:`
Use these two to configure the colors of the gradient if filling style is set to `#FILLGRADIENT`. (V5.0)

`GradientCenterX, GradientCenterY:`
Sets the center point for gradients of type `#RADIAL` or `#CONICAL`. Must be a floating point value between 0.0 and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientBalance:`
: This tag controls the balance point for gradients of type `#CONICAL`. Must be a floating point value between 0.0 and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientBorder:`
: This tag controls the border size for gradients of type `#RADIAL`. Must be a floating point value between 0.0 and and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientColors:`
: This tag can be used to create a gradient between more than two colors. This has to be set to a table that contains sequences of alternating color and stop values. If this tag is used, the `GradientStartColor` and `GradientEndColor` tags are ignored. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`KeepScreenOn:`
: This tag is only supported if Hollywood is running on Android and iOS. If you set this tag to `True`, battery saving mode will be disabled on mobile devices. This means that the device's screen will never be dimmed or turned off to save energy. Useful for scripts that do not require user input. Defaults to `False`. (V5.1)

`PubScreen:`
: This tag can be used to specify the public screen this display should be opened on. You have to pass a string that contains the name of the public screen to use here. This is only supported on AmigaOS compatible operating systems. Please note that if you use multiple displays on multiple public screens, it is absolutely mandatory that the individual public screens use the same pixel format, i.e. it is not allowed to have one display on a 16-bit public screen while the other display is on a 32-bit public screen. The pixel format must be identical for all public screens. (V5.2)

`HideFromTaskbar:`
: This tag is only supported if Hollywood is running on Windows. If you set this tag to `True`, your display will not get an entry in the Windows taskbar. This is useful if your application has added an icon to the system tray and you want it to be accessible from the system tray only. Defaults to `False`. (V5.3)

`HideOptionsMenu:`
: This tag is only supported on Android devices. When the user opens the options menu on Android devices, Hollywood will allow the user to configure several display parameters like enabling or disabling autoscaling or layerscaling. If you do not want to give the user this possibility to change the display parameters via the app's options menu, set this tag to `True`. Defaults to `False`. (V5.3)

`Orientation:`
: This tag is only supported on mobile platforms. It allows you to specify a hard-coded orientation for your script. If you set this tag, Hollywood will not react to orientation changes when the user rotates the device. Instead, it will keep the orientation mode that you specified here. The following values are possible:

```
#ORIENTATION_NONE
#ORIENTATION_PORTRAIT
#ORIENTATION_LANDSCAPE
#ORIENTATION_PORTRAITREV
#ORIENTATION_LANDSCAPEREV
```

Defaults to `#ORIENTATION_NONE` which means that there is no fixed orientation and that Hollywood should dynamically adapt to orientation changes. (V5.3)

`NoHardwareScale:`
: This tag is only supported on Android. For performance reasons Hollywood will try to use hardware-accelerated scaling when autoscaling is enabled on Android devices by default. Some devices, however, do not implement hardware-accelerated scaling properly so if you experience strange behaviour when using autoscale mode, try to disable hardware-accelerated scaling using this tag and see if it helps. This tag is obsolete since Hollywood 8.0. Hollywood will always use hardware-accelerated scaling now. (V5.3)

`DisableBlanker:`
: Set this tag to `True` if you want to disable the screen blanker while this display is open. Defaults to `False`. (V6.0)

`Menu:`
: This tag can be used to attach a menu strip to this display. You have to pass the identifier of a menu strip that has been created using the `@MENU` preprocessor command or the `CreateMenu()` function to this tag. It is also possible to attach a single menu strip to multiple displays. See Section 39.8 [MENU], page 804, for details. (V6.0)

`Monitor:`
: This tag allows you to specify the monitor this display should be opened on. Monitors are counted from 1 to the number of monitors available to the system. Please note that if you set this tag, functions that accept display coordinates, e.g. `MoveDisplay()`, will interpret them as values relative to the origin of the monitor specified in the `Monitor` tag. This tag defaults to 1 which means that the display should open on the primary monitor. (V6.0)

`XServer:`
: This tag can be used to specify the X Server that should host this display. By default, Hollywood will use the X Server that has been specified in the `DISPLAY` environment variable. If you want Hollywood to use a different X Server for your display, use this tag. This tag is only available in the Linux version of Hollywood. (V6.0)

`ScreenTitle:`
: On AmigaOS, this tag can be used to set the text that should be shown in the screen's title bar whenever the display is active. By default, `"Workbench`

screen" will be shown. This tag is only available in the AmigaOS compatible versions of Hollywood. (V6.0)

`ScreenName:`
If this display is to be opened in full screen mode, you can set the desired public screen name of the display's own screen with this tag. By default, Hollywood will use "HOLLYWOOD.X" where "X" is replaced by a vacant index. This tag is only available in the AmigaOS compatible versions of Hollywood. (V6.0)

`RememberPosition:`
Set this to `True` if you want this display to remember its position and size. This is obviously only possible with windowed displays. It won't work for full screen displays. You also have to specify a unique identifier for your application using the `@APPIDENTIFIER` preprocessor command if you want to use this tag. The display also must use a numeric identifier, i.e. you cannot use this tag for displays which use automatic id selection. Note that this tag can be overridden by the `-overrideplacement` argument. If you start Hollywood or your compiled script using the `-overrideplacement` argument, any saved position or size information is ignored. See Section 3.2 [Console arguments], page 33, for details. (V6.1)

`Maximized:`
If you set this tag to `True`, the display will open in maximized mode. This is only possible if the display is sizeable. This tag is currently only supported on Windows. (V7.0)

`TrapRMB:` On AmigaOS, if this is set to `True`, Hollywood will deliver right mouse button events also when a menu strip is associated with this display. The downside is that menu access will only be possible via the screen's title bar then. This tag is only handled in case your display has a menu strip, otherwise it has no effect at all. `TrapRMB` defaults to `False` which means that when a menu strip is associated with a display, right mouse button events aren't generated. This tag is only available in the AmigaOS compatible versions of Hollywood. (V7.0)

`NoScaleEngine:`
This tag is only handled if `Mode` is set to `FullScreenScale` for your display. In that case Hollywood will not use any scaling engine but will simply open your display in the same dimensions as the monitor's resolution. Your script then needs to manually adapt to the monitor's resolution. This allows you to write scripts which can dynamically adapt to different resolutions without simply scaling their graphics. (V7.0)

`NoLiveResize:`
On many platforms Hollywood will use live resizing when the user is resizing a display. This means that the display's contents will be automatically scaled while the user is resizing the display. If you don't want this, you can set this tag to `True`. (V7.0)

`NativeUnits:`

If you set this tag to `True`, Hollywood will use the host system's native coordinate space and units instead of pixels. This currently only has an effect on macOS and iOS because both operating systems use custom units instead of pixels when running on a Retina device. By default, Hollywood will enforce the use of pixels on Retina Macs and iOS devices for cross-platform compatibility reasons but you may want to override this setting by using this tag. (V7.0)

`AlwaysOnTop:`

If you set this tag to `True`, the display will always stay on top of other windows. Use this tag with care because it can be quite annoying to the user. (V7.1)

`NoCyclerMenu:`

On Android, Hollywood will automatically add a cycler menu to the options menu in the app's action bar whenever more than one Hollywood display is open. You can then use this cycler menu to conveniently switch to other displays. If you don't want Hollywood to add such a cycler menu, set this tag to `False`. This is supported only on Android. Defaults to `False`. (V8.0)

`HideTitleBar:`

Set this to `True` to hide the status bar on iOS or the action bar on Android. By default, both the status bar and the action bar are always visible. Defaults to `False`. (V8.0)

`Subtitle:`

This tag allows you to set the display's subtitle. This is only supported on Android. The display subtitle is shown in the app's action bar below the display title set using the `Title` tag above. By default, there is no subtitle. (V8.0)

`SingleMenu:`

This tag allows you to place menu items in the root level of the action bar's options menu on Android. Normally, this is not possible because on desktop systems menu items always have to be members of certain root groups (e.g. `"File"`, `"Edit"`, `"View"`, etc.) When using menu strips on Android, Hollywood will of course replicate the desktop menu behaviour by creating individual submenus for those root groups. This means that the user has to tap at least twice to select a menu item because there won't be any menu items in the root level, they will always be in submenus instead. If you don't want Hollywood to create those submenus but just place all items in the root level, set this tag to `True`. This is especially useful if there are only a few menu items and it doesn't make sense to place them in submenus. This tag is only available on Android. Defaults to `False`. (V8.0)

`ScaleFactor:`

If a scaling engine has been activated using the `ScaleMode` tag (see above), this tag can be used to apply a global scaling factor to your display. The scaling factor must be specified as a fractional number indicating the desired scaling coefficient, e.g. a value of 0.5 shrinks everything to half of its

size whereas a value of 2.0 scales everything to twice its size. Note that setting `ScaleFactor` will make the script behave slightly different than setting `ScaleWidth` and `ScaleHeight` (see above). The latter will enforce a fixed display size which will never be changed unless the user manually uses the mouse to change the display size. Setting `ScaleFactor`, however, will apply the scale factor to all new BGPics and display sizes so the display size may change if the BGPic size changes or the script changes the display size. Thus, using `ScaleFactor` is perfect for scaling a script for a high dpi display because it makes sure that the script behaves exactly the same but just appears larger (or smaller if you want!). You can also set the `Mode` tag to `SystemScale` to automatically apply the host system's scaling factor to your display (see above). See Section 25.18 [Scaling engines], page 401, for details. (V8.0)

`ImmersiveMode:`
: On Android, this tag allows you to put the display into immersive mode. Immersive mode is a special full screen mode that hides most system UI components (like the status bar) to give your application as much screen space as possible. If you want your display to use immersive mode, you have to set `ImmersiveMode` to one of the following tags. All those modes only differ in the way the user can bring back the system bars and whether or not your script is notified about it.

  `#IMMERSIVE_NORMAL:`
  : Normal immersive mode. Users can bring back the system bars by swiping from any edge where there is a hidden system bar. You will be notified about system bar visibility changes via the `ShowSystemBars` and `HideSystemBars` event handlers.

  `#IMMERSIVE_LEANBACK:`
  : Lean back immersive mode. In this mode, system bars can be brought back by tapping anywhere on the screen. You will be notified about system bar visibility changes via the `ShowSystemBars` and `HideSystemBars` event handlers.

  `#IMMERSIVE_STICKY:`
  : Sticky immersive mode. This is the same as `#IMMERSIVE_NORMAL` except that there will be no notification when the system bar visibility changes. Instead, the raw swipe events will be forwarded to you even if they caused the system bars to reappear.

  Note that both `#IMMERSIVE_NORMAL` and `#IMMERSIVE_LEANBACK` will notify your script about system bar visibility changes using the `ShowSystemBars` and `HideSystemBars` event handlers. You can listen to those events using `InstallEventHandler()`. See Section 29.13 [InstallEventHandler], page 553, for details. (V9.0)

`Palette:` If this tag is set to the identifier of a palette, Hollywood will create a palette display for you. Palettes can be created using functions like `CreatePalette()` or `LoadPalette()`. Alternatively, you can also set

this tag to one of Hollywood's inbuilt palettes, e.g. `#PALETTE_AGA`. See Section 44.36 [SetStandardPalette], page 918, for a list of inbuilt palettes. (V9.0)

`FillPen:` If the `Palette` tag is set (see above), you can use the `FillPen` tag to set the pen that should be used for filling the display's background. (V9.0)

`SoftwareRenderer:`
On Windows systems, you can set this tag to `True` to disable Hollywood's GPU-accelerated Direct2D renderer on Windows systems. Hollywood will use its CPU-based renderer for maximum compatibility then. This is mostly useful for testing purposes. Normally, there shouldn't be any reason for setting this tag to `True`. (V9.0)

`VSync:` On Windows systems, this tag can be set to `True` to force Hollywood's renderer to throttle refresh to the monitor's refresh rate. This means that you'll no longer have to use functions like `VWait()` to throttle drawing. However, do note that if you set this to `True`, you must make sure to draw in full frames only otherwise drawing will become extremely slow. Full frame drawing can be achieved e.g. by either using a double buffer or by using `BeginRefresh()` and `EndRefresh()`. Also note that `VSync` is currently only supported on Windows and only if Hollywood uses its Direct2D backend. Direct2D is not available before Windows Vista SP2. (V9.0)

`ScaleSwitch:`
When switching a display between windowed and full screen mode by pressing the CMD+RETURN (LALT+RETURN on Windows) hotkey or passing the `#DISPMODE_MODESWITCH` mode to `ChangeDisplayMode()`, Hollywood will not change the monitor's screen mode on systems where hardware-accelerated scaling is available. On those systems, Hollywood will just simulate full screen mode by scaling the display to the monitor's current resolution. Only on older systems or platforms that don't support hardware-accelerated scaling will Hollywood switch the monitor to a new resolution. The reason why switching the monitor's resolution is no longer done by default is that it often takes considerable time for the monitor to do so and not all display devices support it (e.g. laptop screens often don't support it). If you want to force Hollywood to always change the monitor's resolution when going full screen and never simulate full screen mode by scaling, just set this tag to `True`. Defaults to `False`. (V9.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to display adapter plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to display adapter plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Many of the tags from above - especially the ones used to configure the appearance of the display - are also available from the command line to give the user some flexibility in controlling the appearance of the script. For example, it is possible to change the

display mode of your script (windowed, full screen, etc.) or window style (borderless, sizeable, etc.) from the command line as well. If you do not want that, you have to use the `-locksettings` argument when compiling your scripts. This will forbid any user changes to the settings defined by you in this preprocessor command.

INPUTS

`table` a table containing one or more of the fields listed above

EXAMPLE

```
@DISPLAY {Title = "My App", X = #LEFT, Y = #TOP, Width = 320,
    Height = 240, Color = #WHITE}
```

The above declaration opens a 320x240 sized display with a white colored background. The display is positioned at top-left edge of the host screen. The window title will be "My App".

```
@DISPLAY {Width = 640, Height = 480, Borderless = True,
DragRegion = {
{Type = #BOX, X=0, Y=460, Width=640, Height=20}, ; bottom drag bar
{Type = #BOX, X=0, Y=0, Width=20, Height=480} ; left drag bar
}
}
```

The code above will open a borderless 640x480 display. The drag bar of the display will not be in the top region of the display but it will be put into the bottom and the left regions of the display by specifying a custom drag region using the `DragRegion` tag. Note that the `DragRegion` tag always requires you to pass a list of rectangular regions, even if you are only using a single region. See above for more information.

25.9 FreeDisplay

NAME

FreeDisplay – free a display (V4.5)

SYNOPSIS

```
FreeDisplay(id)
```

FUNCTION

This function closes the specified display and frees all of its resources. When `FreeDisplay()` returns, the specified display will be no longer available.

Please note that you cannot call this on the current display. You must select an other display using `SelectDisplay()` first before you can call `FreeDisplay()`. This is because Hollywood always requires a current display. Freeing the current display would lead to a non-existent current display and this is not supported by Hollywood. There always must be a display that is the current one. Even if all displays are closed and there is no visible display at all, Hollywood will internally still have a current display that can handle calls that draw graphics.

INPUTS

`id` identifier of the display to free; must not be the current display

25.10 GetDisplayModes

NAME

GetDisplayModes – return available display modes (V5.0)

SYNOPSIS

```
t = GetDisplayModes([monitor])
```

FUNCTION

This function can be used to find out all display modes supported by the specified monitor. `GetDisplayModes()` will then store all supported modes in a table and return it to you. The table returned by this function is a collection of a number of sub-tables that all have `Width` and `Height` elements initialized.

This function is useful for finding out if a specific display mode is actually supported by this system before trying to switch to this mode using the `ChangeDisplayMode()` command.

INPUTS

`monitor` optional: monitor whose display modes shall be queried (defaults to 1 which means query the primary monitor) (V6.0)

RESULTS

`t` a collection of all available display modes

EXAMPLE

```
t = GetDisplayModes()
For Local k = 0 To ListItems(t) - 1
   DebugPrint("Mode", k + 1, "Width:", t[k].Width, "Height:", t[k].Height)
Next
```

The code above queries the display mode database and then prints all available modes.

25.11 GetMonitors

NAME

GetMonitors – return information about available monitors (V6.0)

SYNOPSIS

```
t = GetMonitors()
```

FUNCTION

This function can be used to obtain information about all monitors currently available to the system. A table will be returned that contains a number of subtables describing each individual monitor's dimensions and the position of this monitor on the desktop screen.

Please note that the way the monitor positions on the desktop screen are described is platform-dependent. See Section 25.14 [Multi-monitor support], page 397, for details.

INPUTS
none

RESULTS

`t` a collection of all available monitors and their desktop alignment

EXAMPLE

```
t = GetMonitors()
For Local k = 0 To ListItems(t) - 1
   DebugPrint("Monitor", k + 1, "X:", t[k].X, "Y:", t[k].Y,
        "Width:", t[k].Width, "Height:", t[k].Height)
Next
```

The code above queries the operating system's monitor database and then prints information about all available monitors.

25.12 HideDisplay

NAME
HideDisplay – minimize the current display (V3.0)

SYNOPSIS

```
HideDisplay([toback])
```

FUNCTION

This function minimizes the current display. The script execution will continue while the display is minimized. You can use the `ShowDisplay()` command to bring a minimized display back to the front.

If you want to close the display instead of minimizing it, use the `CloseDisplay()` command instead.

Starting with Hollywood 8.0, there is an optional argument named `toback`, which is only supported on AmigaOS and compatible systems. If you set it to `True`, the display won't be minimized, but instead it will be hidden by moving it all the way to the bottom of the current screen's window stack, i.e. all other windows will appear in front of it then.

INPUTS

`toback` optional: whether to hide the window by moving it to the bottom of the window stack instead of minimizing it; this is only supported on AmigaOS and compatible systems (defaults to `False`) (V8.0)

EXAMPLE

```
HideDisplay()
Wait(100)
ShowDisplay()
```

This code hides the display, waits two seconds, and pops up the display again.

25.13 MoveDisplay

NAME

MoveDisplay – move the display to a new position on the host screen (V2.0)

SYNOPSIS

```
MoveDisplay(x, y)
```

FUNCTION

This function moves the display to the new position specified by `x` and `y`. You can also use Hollywood's special coordinate constants here. To find out which x- and y-positions are valid, you can query the `#ATTRMAXWIDTH` and `#ATTRMAXHEIGHT` attributes with `GetAttribute()`.

The initial position of the display can be set with the `@DISPLAY` preprocessor command.

INPUTS

`x`	new x-position for the display
`y`	new y-position for the display

EXAMPLE

```
MoveDisplay(#LEFT, #TOP)
```

This moves the display to the top-left of the screen

25.14 Multi-monitor support

Starting with Hollywood 6.0 multiple monitor systems are supported by Hollywood. You can specify on which monitor a display should open by setting the `Monitor` tag in the `@DISPLAY` preprocessor command or in the `CreateDisplay()` or `OpenDisplay()` functions. To get information about all available monitors, you can use the `GetMonitors()` function.

Please note that the way extended desktop coordinates are handled with multiple monitors is platform-dependent. For example, on Windows the primary monitor's area in the extended desktop will always start at offset 0:0. If there is a monitor to the left or top of the primary monitor, its offset will use negative coordinates. On Linux, however, negative coordinate space isn't used at all. Thus, it can happen on Linux that the primary monitor's area in the extended desktop doesn't start at offset 0:0 but at a higher offset if there is a monitor to the left or top of the primary monitor. Hollywood doesn't level these platform dependencies for you because they are an integral part of the window manager's coordinate system and it could become very confusing in some parts for the programmer if Hollywood tried to introduce its own coordinate abstraction layer on top of the window manager's coordinate system. Normally, you won't have to deal with the absolute offsets anyway, since functions like `MoveDisplay()` from the display library work relative to the display's monitor offset anyway.

25.15 OpenDisplay

NAME

OpenDisplay – open a display (V4.5)

SYNOPSIS

```
OpenDisplay(id[, table])
```

FUNCTION

This function will open a display previously created using `CreateDisplay()` and make it visible. Starting with Hollywood 6.0 this function accepts an optional table argument which can be used to configure some advanced options. The following tags are currently recognized by the optional table argument:

`Mode:` This tag allows you to specify the mode the display should be opened in. You have to pass one of the following strings to this tag:

`Windowed` Open display in windowed mode.

`FullScreen`
: Open in full screen mode. This can change your monitor's resolution to the dimensions which fit best to your display's dimensions. If you don't want that, take a look at the `FullScreenScale` and `FakeFullScreen` modes below.

`FullScreenScale`
: This is a special full screen mode which won't change your monitor's resolution. Instead, Hollywood's display will be resized to fit your monitor's dimensions. Additionally, this full screen mode will activate the auto scaling engine so that your display is automatically scaled to fit your monitor's dimensions. `FullScreenScale` will use auto scaling by default. If you would like it to use layer scaling, you have to set `ScaleMode` to `#SCALEMODE_LAYER` as well. `FullScreenScale` is especially useful on mobile devices whose display hardware has a hard-coded resolution and doesn't support resolution changes in the same way as an external monitor connected to a desktop computer does. The downside of `FullScreenScale` is that it is slower because Hollywood has to scale all rendering operations to the monitor's dimensions. (V7.0)

`FakeFullScreen`
: Open in fake full screen mode. This means that Hollywood will not change the monitor's resolution but the backfill window will be configured to shield the desktop completely. Thus, the user gets the impression as if Hollywood was running full screen, although it is running on the desktop.

`ModeRequester`
: This will open a display mode requester allowing the user to choose the desired full screen mode for this display.

`Ask` This will open a requester asking the user to choose between windowed and full screen mode.

By default, `OpenDisplay()` will use the mode that was specified when creating the display.

`ScrWidth, ScrHeight:`
: If `Mode` has been set to `FullScreen`, these tags allow you to set the desired dimensions for the full screen mode. Defaults to what has been set when creating the display. Starting with Hollywood 7.0 you can also set these tags to the special constant `#NATIVE`. In that case, Hollywood will use the dimensions of the display's host device.

`ScrDepth:`
: If `Mode` has been set to `FullScreen`, this tag allows you to set the desired depth for the full screen mode. Defaults to what has been set when creating the display.

`Backfill:`
: This tag allows you to configure the backfill setting for this display. The table you have to specify here has to follow the same conventions as its counterpart that can be passed to the `Backfill` tag of the `@DISPLAY` preprocessor command. See Section 25.8 [DISPLAY], page 380, for details.

`NoSelect:`
: This tag allows you to specify whether or not the newly opened display shall be selected as the current output device. `NoSelect` defaults to `False` which means that by default, `OpenDisplay()` will open the specified display and select it as the current output device. If you do not want this behaviour, pass `True` in `NoSelect`. In that case, you need to manually call `SelectDisplay()` before you can draw into the display.

`Monitor:`
: This tag allows you to specify the monitor this display should be opened on. Monitors are counted from 1 to the number of monitors available to the system. Please note that if you set this tag, functions that accept display coordinates, e.g. `MoveDisplay()`, will interpret them as values relative to the origin of the monitor specified in the `Monitor` tag. This tag defaults to what has been set when creating the display.

`XServer:`
: This tag can be used to specify the X Server that should open this display. By default, Hollywood will use the X Server that has been specified when creating the display. This tag is only available in the Linux version of Hollywood.

`PubScreen:`
: This tag can be used to specify the public screen this display should be opened on. You have to pass a string that contains the name of the public screen to use here. By default, Hollywood will use the public screen specified when creating the display. This tag is only supported on AmigaOS compatible operating systems.

`ScreenName:`
: If this display is to be opened in full screen mode, you can set the desired public screen name of the display's own screen with this tag. By default, Hollywood will use the screen name specified when creating the display. This tag is only available in the AmigaOS compatible versions of Hollywood.

INPUTS

`id`	identifier of the display to open
`table`	optional: table argument containing further options (see above) (V6.0)

EXAMPLE

See Section 25.7 [CreateDisplay], page 375.

25.16 Palette displays

When showing a palette BGPic using `DisplayBGPic()` or assigning a palette BGPic to a display using the `@DISPLAY` preprocessor command or `SetDisplayAttributes()`, Hollywood will put the display into palette mode. This means that the only colors available for drawing on that display are the colors that are part of the palette. This is very similar to old graphics hardware from the 1980s and early 1990s which was only capable of showing a fixed amount of colors, ranging from 2 to 256 colors, on screen.

An advantage of putting a display into palette mode is that by changing the color of a pen in the palette, all pixels which use that pen will change their color instantly as well. This makes it possible to easily write code that fades palette colors or cycles them. Color cycling effects were a very popular way of animating graphics like waterfalls or rain in the 1980s and early 1990s. By using a Hollywood display in palette mode, these effects can be easily replicated in Hollywood.

Another advantage of palette mode displays is that the memory consumption is much lower than when using true colour displays. In 32-bit true colour mode, a single pixel will require 4 bytes of memory whereas in palette mode, a single pixel will just require 1 byte of memory. Thus, a 1920x1080 image will require about 8 megabytes of memory in 32-bit mode but only 2 megabytes of memory in palette mode.

However, great care needs to be taken when using displays in palette mode because otherwise drawing can become really slow. This is because by default, all graphics that you draw to a palette mode display will be remapped to the display's palette so that their appearance is as close to the original as possible. Precisely, remapping means that Hollywood has to scan all pixels of the source graphics and find the closest match for each pixel in the display's palette. This is of course a very time-consuming operation and will make drawing very slow. That's why the fastest way of drawing to a palette mode display is to make sure all graphics use the same palette. Then no remapping needs to be done and the graphics data can just be copied to the display without any expensive pixel adaptation algorithm.

To disable remapping in a palette mode display, you have to pass `#PALETTEMODE_PEN` to `SetPaletteMode()`. This will enable pen-based drawing and whenever you draw palette graphics to the display, Hollywood will assume that they use the same palette as the display. Thus, they can just be copied to the display as they are.

Furthermore, when the palette mode has been set to `#PALETTEMODE_PEN`, single-color drawing functions like `Box()` or `Circle()` will ignore the RGB color that is passed to them. Instead, they will simply draw the shapes using the pen that has been set using `SetDrawPen()`. Shadow and border effects that might be active will also ignore the standard shadow and border color set using `SetFormStyle()` or `SetFontStyle()` and will draw the shadow and

border using the pen that has been set using `SetShadowPen()` and `SetBorderPen()`, respectively. Also, antialiasing will be disabled when `#PALETTEMODE_PEN` is active because in most cases palettes don't have enough colors for satisfactorily anti-aliasing edges.

Note, however, that even if `#PALETTEMODE_PEN` is active, RGB graphics, of course, still have to be remapped because it's obviously impossible to draw RGB graphics to a palette mode display in any other way. Thus, drawing 32-bit true color graphics to palette displays should be avoided because it will always be slow because remapping needs to be done for those graphics and there is no way around this.

The way graphics data is remapped to a display's palette can be configured by calling `SetDitherMode()`. This allows you to enable or disable dithering and you can also specify the dithering algorithm to use.

Palette mode displays support all features of normal displays. It is possible to use layers, sprites, transition effects, clipping regions, videos, and double buffers with palette mode displays as well. Keep in mind, however, to pay attention to the remarks mentioned above or drawing might become very slow.

See Section 44.1 [Palette overview], page 889, to learn more about palettes.

25.17 RefreshDisplay

NAME

RefreshDisplay – force display refresh (V9.0)

SYNOPSIS

```
RefreshDisplay([t])
```

FUNCTION

This command can be used to force the current display to refresh itself. It is normally not needed to call this function as display refresh is handled by Hollywood automatically. This function is only here for debugging purposes. The optional table argument can be used to pass additional options but this is currently just for internal use and not publicly documented.

INPUTS

`t`	optional: table containing further options

25.18 Scaling engines

Starting with Hollywood 4.0, there are two scaling engines available which you can use to force your script to run in a different resolution than it was designed for. For example, you wrote a little game in 320x240 so that it runs fast enough on classic Amigas. On modern Amigas, however, 320x240 will appear as a tiny window. Thus, you can now use Hollywood's scaling engines to make your script appear in 640x480 or 800x600 without changing a single line of your code! All you have to do is activate one of Hollywood's scaling engines!

When a scaling engine is active, your script will still think that it is running in its original resolution. This means, for instance, that calls like

```
width = GetAttribute(#DISPLAY, 0, #ATTRWIDTH)
```

```
height = GetAttribute(#DISPLAY, 0, #ATTRHEIGHT)
```

will still return 320x240 even though your script is now running in a completely different resolution. It is obvious that such a behaviour is necessary for consistency i.e. to ensure that the script runs flawlessly in the new resolution. Your script will not notice that a scaling engine is active at all! The scaling engine will be installed completely transparently on top of your script.

Scaling engines can be activated either from the command line by using one of the scaling engine arguments (`-autoscale` or `-layerscale`) or it can be activated from within your script by either specifying `ScaleMode` in a `@DISPLAY` preprocessor command or by calling `SetDisplayAttributes()`.

Hollywood offers two scaling engines:

1. Auto scaling engine: You can activate this scaling engine by specifying the `-autoscale` argument or using `#SCALEMODE_AUTO` with `@DISPLAY` or `SetDisplayAttributes()`. This scaling engine is a lowlevel scaling engine which will simply scale all of Hollywood's graphical output to the new dimensions. Because of this very nature, the auto scaling engine will work with all Hollywood scripts without exceptions. The drawback of the auto scaling engine is a) that it can get quite expensive on the CPU in case the host system doesn't support hardware-accelerated scaling b) that it scales vector graphics in bitmap mode which means that, for instance, true type fonts, graphics primitives, or vector brushes will deteriorate in quality, and c) that drawing is slower because the whole display has to be refreshed even if only a single pixel has changed. You can improve the performance of the auto scaling engine by using a double buffer for drawing for encapsulating all your drawing commands inside a `BeginRefresh()` and `EndRefresh()` section. See Section 30.4 [BeginRefresh], page 591, for details. Another option to greatly improve the performance of the auto scaling engine is to use a plugin which supports hardware-accelerated scaling, e.g. the GL Galore or RebelSDL plugins. Plugins which support hardware-accelerated scaling can apply auto scaling in almost no time. So if Hollywood's inbuilt auto scaling performance is too poor for your requirements, you might want to use a plugin which supports hardware-accelerated scaling. See Section 5.4 [Obtaining plugins], page 66, for details.
2. Layer scaling engine: The layer scaling engine is a more sophisticated, high-level scaling engine which you can activate by specifying the `-layerscale` argument or using `#SCALEMODE_LAYER` with `@DISPLAY` or `SetDisplayAttributes()`. This scaling engine a) is often faster than the auto scaling engine because layers have only to be scaled once, b) offers a higher output quality for vector graphics (i.e. graphics primitives, vector brushes, true type text) which can be scaled without loss of quality, and c) draws faster because only parts of the display need to be refreshed. The drawback of the layer scaling engine is that it works only when Hollywood is in layer mode and it needs more memory. Also, you cannot call `DisableLayers()` when the layer scaling engine is active. Thus, if you want to use the layer scaling engine, your whole script has to run with enabled layers.

From these descriptions it might sound like option (2) is the way to go, but this is not necessarily true. In fact, the auto scaling engine is pretty sufficient in most cases. The layer scaling engine is only important for projects like presentations that shall be promoted to a UltraHD resolution or similar. In that case, it is important that the vector graphics

are property scaled in vector mode so that a crisp result is achieved. Under normal circumstances, however, using the auto scaling engine should do everything you want. And don't be discouraged by the fact that it is slower than the layer scaling engine - on modern systems you probably won't notice this slowdown at all!

When activating a scaling engine, you also have to specify the target scaling resolution. This can be done in two different ways: You can either set the target scaling resolution using the `-scalewidth` and `-scaleheight` console arguments or their counterparts `ScaleWidth` and `ScaleHeight` if you are using `@DISPLAY` or `SetDisplayAttributes()`, or you can set a global scaling coefficient using the `-scalefactor` or `-systemscale` console arguments or their runtime counterparts accepted by `@DISPLAY` or `SetDisplayAttributes()`.

Note that there is a difference in behaviour between setting the target resolution using `ScaleWidth` and `ScaleHeight`, and between setting the scaling resolution using `ScaleFactor` and `SystemScale`. If you use `ScaleWidth` and `ScaleHeight`, the size passed to these tags will be rigidly enforced and your display will always keep this size, even if the script requests to change it by calling `ChangeDisplaySize()` or by showing a background picture that is of a different size using `DisplayBGPic()`. The only way the display size can be changed when `ScaleWidth` and `ScaleHeight` were used to set the scaling resolution is that the user resizes the window. Otherwise the display will never change its dimensions.

When using `ScaleFactor` or `SystemScale`, on the other hand, a scaling coefficient is applied to the current display size. This means that the target scaling size is not static as it is the case with `ScaleWidth` and `ScaleHeight` (see above), but it is dynamic because it is relative to the current display size, e.g. if a scaling coefficient of 2.0 is applied and a display is first 640x480 pixels in size and later 800x600 pixels in size, the display will first be promoted to 1280x960 pixels and then to 1600x1200 pixels. This is what makes `ScaleFactor` perfect for scaling a script for a high dpi display because it makes sure that the script behaves exactly the same but just appears larger (or smaller if you want!).

There is also the difference that, in case a scaling engine is active, `SizeWindow` events are only delivered to your display if you use either `ScaleFactor` or `SystemScale` to set the target scaling resolution. If you use `ScaleWidth` and `ScaleHeight`, no `SizeWindow` events will be sent to your display because window size changes are handled by the scaling engine.

If you activate a scaling engine but specify neither `-scalewidth`, `-scaleheight`, nor `-scalefactor` or `-systemscale` the scaling engines won't be activated until the user resizes the window. As soon as he does this, the new window size will be set as the target scaling size.

Finally, you can specify the `-smoothscale` argument to enable interpolated anti-alias scaling for bitmap graphics which looks better but can be slower if no hardware-accelerated scaling is available.

25.19 SCREEN

NAME

SCREEN – configure screen mode for script (V4.5)

SYNOPSIS

```
@SCREEN table
```

FUNCTION

Important note: This preprocessor command is deprecated since Hollywood 6.0. As Hollywood 6.0 introduced support for multiple monitors, there can also be multiple displays in full screen mode on separate monitors. That is why a single `@SCREEN` preprocessor command is no longer sufficient. Instead, display mode parameters should now be configured using the `@DISPLAY` preprocessor command or the `CreateDisplay()` function. You can still use this preprocessor command but it will affect the first display only.

This preprocessor command can be used to configure the initial screen mode for your script. By default, all Hollywood scripts will open in a window. If you want your script to open in full screen mode by default, you can use this preprocessor command to achieve this.

Before Hollywood 4.5, screen mode was configured using the `@DISPLAY` preprocessor command. Hollywood 4.5, however, introduced multiple displays which made it necessary to move the screen mode settings into its own preprocessor command because it is impossible to have multiple displays running in full screen mode.

You have to pass a table to this command. The following table tags are currently recognized:

`Mode:` Defines which display mode your script should start in. This can be either `Windowed`, `FullScreen`, `FullScreenScale`, `FakeFullScreen`, or `Ask`. See Section 3.2 [Console arguments], page 33, for information on what the different modes mean. If you specify `Ask`, Hollywood will ask the user if the script should be run in full screen or windowed mode. Defaults to `Windowed`.

`HideTitleBar:` This field can be used to hide the screen's title bar. It is only effective when Hollywood opens on its own screen or when you use a backfill. Defaults to `False`.

`Desktop:` If you set this field to `True`, the initial background picture will be a copy of your desktop screen. This can be used for some nice effects with that screen. Hollywood will also automatically open a borderless window if this field is `True`. Note that setting this attribute puts Hollywood in a special mode and you must not open any other displays.

`Width, Height:` If `Mode` is set to `FullScreen`, you can use these two to specify the dimensions for the screen that Hollywood should open. If you pass 0 in here, Hollywood will use the dimensions of the desktop screen. If you do not specify them at all, Hollywood will automatically determine a screen size that fits.

`Depth:` This field can be used to specify the color depth of the screen that Hollywood should open. Thus, it can only be used when Hollywood opens in full screen mode. If you do not specify this field, Hollywood will use the color depth of the desktop screen for the new screen. Normally, you should not use this field and leave the choice to Hollywood to detect an appropriate depth for the current system.

Alternatively, screen mode settings can also be configured from the command line. If you want to disable this, you should compile your scripts using the `-locksettings` console argument.

You can also switch between full screen and windowed mode at run time using the `ChangeDisplayMode()` command.

INPUTS

`table` table declaring the initial screen mode for the script

EXAMPLE

```
@SCREEN {Mode = "FullScreen", Width = 800, Height = 600}
```

This declaration will set up 800x600 as the initial screen mode for the script.

25.20 SelectDisplay

NAME

SelectDisplay – select a display as output device (V4.5)

SYNOPSIS

```
SelectDisplay(id[, noactivate])
```

FUNCTION

This function makes the display specified by `id` the current display. The current display is the display that all commands of Hollywood's graphics library draw to. Furthermore, several other functions refer to the current display; for example, `WaitLeftMouse()` will wait for the left mouse button to be clicked in the currently active display, and `SetPointer()` will change the mouse pointer of the currently active display.

The optional argument `noactivate` specifies whether or not the display shall also be activated. This argument defaults to `False` which means that by default, the specified display is made the current display and it is also activated. Make sure you understand the difference between the current display and the active display: The current display is not necessarily the active display and vice versa. The active display has nothing to do with Hollywood itself. It just means that the window manager of the host OS will highlight the display and make it the active one. Active windows will also get the keyboard focus and on some operating systems active windows are always put to the front. The current display on the other hand is the display that Hollywood commands use to draw their graphics to. Thus, you could also turn an inactive display into the current display. Hence, you should keep in mind that this function makes the specified display the current display. Optionally, it activates the display, too. If you only want to activate a display without making it the current one, `ActivateDisplay()` is the proper function for you.

Also note that Hollywood always requires a current display. It is not possible to create a Hollywood script that runs without a current display. You can, however, create the illusion of a Hollywood script without a current display. By using `CloseDisplay()` you can close all displays (or use the `Hidden` tag in `@DISPLAY`). It will then seem as if there was no current display but that is a wrong assumption. There is still a current display; it is just not visible. Even if all displays are closed, you will still be able to call functions like

`DisplayBrush()`. They will still draw their graphics to the display, even if it is currently not visible. That is why you can also call `SelectDisplay()` on closed displays.

Make sure that you do not confuse `SelectDisplay()` with the similarly named functions `SelectBrush()`, `SelectBGPic()`, `SelectAnim()`, `SelectMask()`, and `SelectAlphaChannel()`. All of these functions require you to call `EndSelect()` when you are done with them, but `SelectDisplay()` does not have this requirement. In fact, it works in a completely different way so you must never call `EndSelect()` for `SelectDisplay()`. If you want to return to the previously current display, you must call `SelectDisplay()` again. Calling `EndSelect()` to restore the previously current display will definitely not work.

If you call `SelectDisplay()` when `SelectBrush()` (or one of its related commands) is active, Hollywood will internally call `EndSelect()` first to finish the `SelectBrush()` operation. After that, `SelectDisplay()` will switch the current display.

Double-buffering modes are not cancelled by `SelectDisplay()`. You can safely take the focus away from a double buffered display using `SelectDisplay()`, make some changes in another display, and then return to the double-buffered display. Double-buffered modes are fully preserved by `SelectDisplay()`.

Also note that `OpenDisplay()` will always automatically select the specified display as the current one, except when you specify use the optional argument to tell it not to do so. Thus, when using `OpenDisplay()` you normally do not have to call `SelectDisplay()` at all. `SelectDisplay()` is only needed when selecting a display as the current one that is already open.

INPUTS

`id` identifier of the display to select as the current display

`noactivate`
optional: set this to `True` if the display shall not be activated by this function (defaults to `False`)

25.21 SetDisplayAttributes

NAME

SetDisplayAttributes – change attributes of current display (V4.5)

SYNOPSIS

```
SetDisplayAttributes(table)
```

FUNCTION

This function can be used to change one or several window attributes of the current display with a single function call. It is a very powerful function that gives you the utmost flexibility for dealing with your displays. Almost all attributes of the display can be changed on the fly using this function. For example, you can make a display sizeable or borderless with this function.

You have to pass a table containing a collection of attributes that you want to modify to this function.

Here is a list of display attributes that you can modify using this function:

`BGPic:` Allows you to exchange the BGPic currently associated with this display. This is basically the same as calling `DisplayBGPic()`. The advantage of exchanging BGPics with `SetDisplayAttributes()` is that you can change other display attributes at the same time. To exchange the BGPic for this display, simply pass the identifier of the new BGPic in this tag.

`Width, Height:`
These allow you to change the dimensions of the current display. This is the same as calling `ChangeDisplaySize()`. You can specify either a direct value of a string containing a percentage specification (e.g `"200%"`). Starting with Hollywood 7.0 you can also set these tags to the special constant `#NATIVE`. In that case, Hollywood will use the dimensions of the display's host device.

`X, Y:` Allows you to change the position of the current display. This is the same as calling `MoveDisplay()`. You can either specify a direct value or one of Hollywood's special coordinate constants.

`Title:` See Section 25.8 [DISPLAY], page 380, for details.

`Borderless:`
See Section 25.8 [DISPLAY], page 380, for details.

`Sizeable:`
See Section 25.8 [DISPLAY], page 380, for details.

`Fixed:` See Section 25.8 [DISPLAY], page 380, for details.

`NoHide:` See Section 25.8 [DISPLAY], page 380, for details.

`NoModeSwitch:`
See Section 25.8 [DISPLAY], page 380, for details.

`NoClose:` See Section 25.8 [DISPLAY], page 380, for details.

`HidePointer:`
See Section 25.8 [DISPLAY], page 380, for details.

`ScaleMode:`
See Section 25.8 [DISPLAY], page 380, for details.

`ScaleWidth, ScaleHeight:`
See Section 25.8 [DISPLAY], page 380, for details.

`SmoothScale:`
See Section 25.8 [DISPLAY], page 380, for details.

`DragRegion:`
See Section 25.8 [DISPLAY], page 380, for details.

`SizeRegion:`
See Section 25.8 [DISPLAY], page 380, for details.

`PubScreen:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`FillStyle:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`Color:` See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`TextureBrush:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`TextureX, TextureY:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientStyle:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientAngle:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientStartColor, GradientEndColor:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientCenterX, GradientCenterY:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientBalance:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientBorder:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`GradientColors:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.2)

`HideFromTaskbar:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.3)

`HideOptionsMenu:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.3)

`Orientation:`
See Section 25.8 [DISPLAY], page 380, for details. (V5.3)

`Menu:` This tag can be used to attach a menu strip to the current display. You have to pass the identifier of a menu strip that has been created using the `@MENU` preprocessor command or the `CreateMenu()` function to this tag. If you pass the special value -1 here, the currently attached menu strip will be removed from the display. See Section 39.8 [MENU], page 804, for details. (V6.0)

`Monitor:` See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`XServer:` See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`ScreenTitle:`
See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`ScreenName:`
See Section 25.8 [DISPLAY], page 380, for details. (V6.0)

`RememberPosition:`
See Section 25.8 [DISPLAY], page 380, for details. (V7.0)

`HideTitleBar:`
See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`ImmersiveMode:`
See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

INPUTS

`table` a table containing a collection of attributes that shall be modified in the current BGPic; see the list above

EXAMPLE

```
SetDisplayAttributes({Borderless = True, Sizeable = True})
```

The code above makes the current display borderless and sizeable.

25.22 SetSubtitle

NAME

SetSubtitle – set subtitle of current display (V8.0)

SYNOPSIS

```
SetSubtitle(subtitle$)
```

PLATFORMS

Android only

FUNCTION

This function changes the subtitle of the currently active display to the new subtitle specified in `subtitle$`. The subtitle is shown in the action bar below the display's title, which can be set using `SetTitle()`. To remove the subtitle, pass an empty string in `subtitle$`. By default, there is no subtitle at all.

Alternatively, you can also set the subtitle in the `@DISPLAY` preprocessor command or when creating the display using `CreateDisplay()`.

INPUTS

`subtitle$`
new subtitle for display

EXAMPLE

```
SetSubtitle("Written by Andreas Falkenhahn")
```

The above code changes the subtitle to "Written by Andreas Falkenhahn".

25.23 SetTitle

NAME

SetTitle – change the title of the current display

SYNOPSIS

```
SetTitle(title$)
```

FUNCTION

This function changes the title of the currently active display to the new title specified in `title$`. This is only useful for changing the title while the script is running. If you want to give your application a global title, just use the `@DISPLAY` preprocessor command.

INPUTS

`title$` new window title

EXAMPLE

```
SetTitle("My cool program")
```

The above code changes the window title to "My cool program".

25.24 ShowDisplay

NAME

ShowDisplay – show minimized display (V3.0)

SYNOPSIS

```
ShowDisplay()
```

FUNCTION

This function unminimizes the current display and brings it back to the front. The display must have been minimized previously either using the `HideDisplay()` command or by the user using the window's minimize button.

Note that you cannot use this function to show a display that is closed. If the display is closed, you need to use `OpenDisplay()`.

INPUTS

none

EXAMPLE

See Section 25.12 [HideDisplay], page 396.

26 DOS library

26.1 CanonizePath

NAME

CanonizePath – convert path into canonical format (V9.0)

SYNOPSIS

```
p$ = CanonizePath(path$)
```

FUNCTION

This function can be used to turn the path specified in `path$` into a canonical one. Canonizing a path involves the following operations:

- shortcuts like `".."` or `"."` will be resolved to fully qualified paths
- relative paths will be converted to fully qualified paths
- on platforms with case-insensitive file systems, the spelling of all path components will be adapted to the spelling as it is stored in the file system
- slashes and backslashes will be adapted to the host operating system's convention
- any assigns in the path will be resolved on AmigaOS and compatibles

Note that the path passed to `CanonizePath()` needn't exist. If it doesn't exist, `CanonizePath()` will try to resolve as many components in the path as possible. Note, however, that `CanonizePath()` doesn't perform path validation. If you pass a path that is invalid because of syntactical errors, the result is undefined.

INPUTS

`path$` path to canonize

RESULTS

`p$` fully qualified path in the host system's canonical format

EXAMPLE

```
Print(CanonizePath("../image.jpg"))
```

The code above will print the fully qualified path of the file `image.jpg`.

26.2 ChangeDirectory

NAME

ChangeDirectory – change the current directory

SYNOPSIS

```
ChangeDirectory(dir$[, t])
```

FUNCTION

This function changes the directory to the one specified in `dir$`.

Starting with Hollywood 10.0, this function accepts an optional table argument which supports the following tags:

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing

the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`dir$` directory to make the current directory

`t` optional: table argument containing further options (see above) (V10.0)

EXAMPLE

```
ChangeDirectory("Data")
OpenFile(1, "Highscores.txt")
CloseFile(1)
```

The above code changes the current directory to "Data" and opens the file "Highscores.txt" inside the "Data" directory.

26.3 CloseDirectory

NAME

CloseDirectory – close an open directory (V4.0)

SYNOPSIS

```
CloseDirectory(id)
```

FUNCTION

This function closes a directory previously opened using `OpenDirectory()`, `@DIRECTORY`, or `MonitorDirectory()`. You should always close directories as soon as you are finished with them. This ensures that the file system does not unnecessarily keep directories locked.

INPUTS

`id` identifier of the directory to close

EXAMPLE

See Section 26.47 [OpenDirectory], page 458.

26.4 CloseFile

NAME

CloseFile – close an open file

SYNOPSIS

```
CloseFile(id)
```

FUNCTION
Closes the file with the identifier `id` which was opened with `OpenFile()`.

INPUTS

`id` number that the file was opened with

EXAMPLE
See Section 26.48 [OpenFile], page 459.

26.5 CompressFile

NAME
CompressFile – compress a file (V4.0)

SYNOPSIS

```
size = CompressFile(src$, dst$)
```

FUNCTION
This function compresses file `src$` and saves the packed data to `dst$`. The return value specifies the size of the compressed file. Hollywood uses zlib for data compression.

To decompress files packed by `CompressFile()` use the Hollywood function `DecompressFile()`.

INPUTS

`src$` file to compress

`dst$` output file

RESULTS

`size` size of the compressed file

EXAMPLE

```
CompressFile("image.bmp", "image.pak")
```

The code above compresses file image.bmp to image.pak.

26.6 CopyFile

NAME
CopyFile – copy file or directory (V2.0)

SYNOPSIS

```
CopyFile(src$, dst$[, t])
```

DEPRECATED SYNTAX

```
CopyFile(src$, dst$[, newname$, func, userdata, pattern$, matchdir])
```

FUNCTION
This function copies the file or directory specified in `src$` to the directory specified in `dst$`. Note that by default, existing files will be overwritten without asking. You can

customize this behaviour by specifying a callback function (see below). Also note that you have to specify a directory, not a file, in `dst$`.

This function is powerful. It will fully recurse into all subdirectories and copy the file attributes, date stamps and comments as well. If the destination directory does not exist, it will be created for you (even if it contains subdirectories that do not exist yet). All path specifications can be local to the current directory or qualified. You can also copy files to the current directory by specifying `""` as `dst$`.

`CopyFile()` supports many optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `CopyFile()`.

The following table fields are recognized by this function:

`NewName:` If you would like to change the name of the file while copying it, set this field to the desired new name for the file. Obviously, setting this field only makes sense when you specify a file in `src$`.

`Pattern:` You can pass a filter pattern in this table field. In that case, `CopyFile()` will only copy the files that match the specified pattern. For example, passing `*.jpg` in `Pattern` will only copy files that use the `.jpg` file extension. Of course, using a filter pattern makes only sense if you pass a directory in `src$`. Note that for historical reasons, the pattern specified in `Pattern` will also be matched against all subdirectories that are to be copied. If you don't want that, set the `MatchDir` table tag to `False` (see below). The pattern specified in `Pattern` must adhere to the pattern rules as described in the documentation of the `MatchPattern()` function. See Section 26.42 [MatchPattern], page 449, for details. (V5.0)

`MatchDir:`
This table field specifies whether or not the filter pattern specified in `Pattern` should also be matched against subdirectories. If this is set to `True`, `CopyFile()` will only recurse into subdirectories that match the specified filter pattern. If it is set to `False`, `CopyFile()` will recurse into all subdirectories. For compatibility reasons, `MatchDir` defaults to `True`, but most of the time you will want to pass `False` here because it usually does not make sense to match a file pattern against a directory name. For example, it does not make sense to match the `*.jpg` example from above against directories as well. (V5.0)

`BufferSize:`
This table field can be used to set the buffer size that should be used for copying files. The value passed here must be specified in bytes. The default is 16384, i.e. 16 kilobytes. (V9.0)

`FailOnError:`
By default, `CopyFile()` will fail when an error occurs. You can change this behaviour by setting `FailOnError` to `False`. In that case, `CopyFile()` won't fail on an error but instead your callback function, if there is one, will be notified using the `#COPYFILE_FAILED` message and your callback must tell

`CopyFile()` how to proceed (retry, continue, abort). See below to learn how to set up a callback function for `CopyFile()`. `FailOnError` defaults to `True`. (V9.0)

`Force:` If this tag is set to `True`, write- or delete-protected files will automatically be overwritten without asking the callback function first. Note that if there is no callback function and `Force` is set to `False` (the default), `CopyFile()` will fail if it can't overwrite a file because that file is write- or delete-protected. Defaults to `False`. (V9.0)

`Async:` If this is set to `True`, `CopyFile()` will operate in asynchronous mode. This means that it will return immediately, passing an asynchronous operation handle to you. You can then use this asynchronous operation handle to finish the operation by repeatedly calling `ContinueAsyncOperation()` until it returns `True`. This is very useful in case your script needs to do something else while the operation is in progress, e.g. displaying a status animation or something similar. By putting `CopyFile()` into asynchronous mode, it is easily possible for your script to do something else while the operation is being processed. See Section 19.4 [ContinueAsyncOperation], page 224, for details. Defaults to `False`. (V9.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to open the source file or directory. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file or directory adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`DstAdapter:`
This tag allows you to specify one or more filesystem adapters that should be asked to handle all operations on the side of the copy target. The filesystem adapter specified here will be responsible for creating directories and setting file and directory attributes, for example. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to `default`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`DstUserTags:`
This tag can be used to specify additional data that should be passed to filesystem adapters specified in `DstAdapter`. See above for details. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Callback:`
This table field can be used to pass a callback function which will be called by `CopyFile()` on various occasions, e.g. to allow you to update a progress

bar for example. The callback function also gets called if a destination file already exists or is write/delete protected. The callback function receives one argument: A table that contains more information.

The following callback types are available:

`#COPYFILE_OVERWRITE:`

`CopyFile()` will run this callback to ask if a file can be overwritten. Your callback function has to return `True` if the file shall be overwritten or `False` if it shall be skipped. To abort the copy operation completely, return -1. The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#COPYFILE_OVERWRITE`

`Source:` Contains the fully qualified path of the file to copy.

`Destination:`
Contains the fully qualified path of the file that does already exist.

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

`#COPYFILE_UNPROTECT:`

The callback function of type `#COPYFILE_UNPROTECT` will be called if the file that should be overwritten is write- or delete-protected. This callback function needs to return `True` if it is okay to unprotect the file or `False` if it shall be skipped. If you return -1, the copy operation will be completely aborted.

Starting with Hollywood 9.0, your callback function can also return -2 to indicate that copying should still be attempted even though the file is write- or delete-protected. This, however, will typically lead to an error because write- or delete-protected files can't be overwritten without unprotecting them first. However, if you return -2 you can catch the ensuing error in your `#COPYFILE_FAILED` callback if you have set `FailOnError` to `False` (see above).

The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#COPYFILE_UNPROTECT`

`Destination:`
Contains the write or delete protected destination file to be unprotected.

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

`#COPYFILE_STATUS:`
This callback function is run from time to time so that you can update a status bar or something similar. The callback function of type `#COPYFILE_STATUS` should normally return `False`. If it returns `True`, the copy operation will be aborted. The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#COPYFILE_STATUS`

`Source:` Contains the fully qualified path of the file that is currently being copied (source).

`Destination:`
Contains the fully qualified path of the file that is currently being written (destination).

`Copied:` Contains the number of bytes that were already copied.

`Filesize:`
Contains the filesize of the source file.

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

`#COPYFILE_FAILED:`
This callback can only be called if the `FailOnError` tag has been set to `False` (see above). In that case, the callback function of type `#COPYFILE_FAILED` will be called whenever a copy operation has failed. It has to return `True` to abort the copy operation, `False` to continue even though an error has occurred or -1 to retry the copy operation that has just failed. The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#COPYFILE_FAILED`

`Source:` Contains the fully qualified path of the file that is currently being copied (source).

`Destination:`
Contains the fully qualified path of the file that is currently being written (destination).

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

(V9.0)

`UserData:`
This field can be used to pass an arbitrary value to your callback function. The value you specify here will be passed to your callback function whenever

it is called. This is useful if you want to avoid working with global variables. Using the `UserData` tag you can easily pass data to your callback function. You can specify a value of any type in `UserData`. Numbers, strings, tables, and even functions can be passed as user data. Your callback will receive this data in the `UserData` field in the table that is passed to it. (V5.1)

INPUTS

`src$` source file or directory to copy

`dst$` destination directory

`t` optional: table containing additional options (see above) (V9.0)

EXAMPLE

```
CopyFile("image.png", "TestDir")
```

Copy the file "image.png" from the current directory to "TestDir".

```
CopyFile("Images", "Images_Bak")
```

Create a backup of the "Images" directory in the "Images_Bak" directory (the new directory will be created by this function automatically). All files including subdirectories will be copied to the new location.

```
CopyFile("Hollywood_Sources/WarpOS", "HW_BAK", {Pattern = "*.c;*.h",
  MatchDir = False})
```

Copies all source code and header files from Hollywood_Sources/WarpOS to HW_BAK.

```
Function p_CopyCallback(msg)
  Switch msg.action
  Case #COPYFILE_STATUS:
    DebugPrint("Now copying", FilePart(msg.source), "to",
               PathPart(msg.destination))
  Case #COPYFILE_OVERWRITE:
    Return(SystemRequest("Hollywood", FilePart(msg.destination) ..
      " does already exist!\nDo you want me to overwrite it?",
      "Yes|No"))
  Case #COPYFILE_UNPROTECT:
    Return(SystemRequest("Hollywood", FilePart(msg.destination) ..
      " is write/delete protected!\nDo you want me to unprotect it?",
      "Yes|No"))
  EndSwitch
  Return(False)
EndFunction
CopyFile("Images", "Copy_of_Images", {Callback = p_CopyCallback})
```

Demonstrates the use of a callback function.

26.7 CountDirectoryEntries

NAME

CountDirectoryEntries – count entries in directory (V8.0)

SYNOPSIS

```
n, ... = CountDirectoryEntries(id[, what, recursive])
```

FUNCTION

This function can be used to count all entries in the directory specified by `id`. This directory must have been opened using `OpenDirectory()` or `@DIRECTORY` before.

The optional argument `what` can be used to specify what kind of entries should be counted. The following entry types are currently supported:

`#COUNTFILES:`
: Count all files in the directory. This is the default.

`#COUNTDIRECTORIES:`
: Count all directories in the directory.

`#COUNTBOTH:`
: Count both, files and directories.

`#COUNTSEPARATE:`
: In that mode, files and directories will be counted separately. This means that two values will be returned: The first return value contains the number of files counted, the second return value the number of directories counted. (V9.0)

Starting with Hollywood 9.0, there is a new optional argument named `recursive`. If this is set to `True`, `CountDirectoryEntries()` will recurse into all subdirectories and include those in the count as well.

Note that `CountDirectoryEntries()` will iterate through all entries in the directory so it must not be used during an iteration using `NextDirectoryEntry()`. Doing so will automatically rewind any existing directory iterations.

INPUTS

`id` identifier of the directory whose entries should be counted

`what` optional: what should be counted (see above) (defaults to `#COUNTFILES`)

`recursive`
: optional: whether or not counting should recurse into subdirectories as well (defaults to `False`) (V9.0)

RESULTS

`n` number of entries of desired type in directory

`...` optional: additional return values depending on the current count mode (see above)

EXAMPLE

```
OpenDirectory(1, "data")
NPrint(CountDirectoryEntries(1))
```

The code above prints the number of files in the directory `data`.

26.8 CRC32

NAME
CRC32 – calculate 32-bit checksum of a file (V2.0)

SYNOPSIS
```
chk = CRC32(f$)
```

FUNCTION
This function computes the 32-bit cyclic redundancy checksum (CRC32) for a given file. This checksum allows you to identify your files.

If you want to compute the CRC32 checksum of a string, use the `CRC32Str()` function instead.

INPUTS

`f$` source file

RESULTS

`chk` 32-bit CRC32 of `f$`

26.9 DecompressFile

NAME
DecompressFile – decompress a file (V4.0)

SYNOPSIS
```
size = DecompressFile(src$, dst$)
```

FUNCTION
This function decompresses file `src$` and saves the unpacked data to `dst$`. The file must have been packed by the `CompressFile()` function. The return value specifies the size of the uncompressed file.

INPUTS

`src$` file to decompress

`dst$` output file

RESULTS

`size` size of the decompressed file

EXAMPLE
```
DecompressFile("image.pak", "image.bmp")
```
The code above decompresses file image.pak to image.bmp.

26.10 DefineVirtualFile

NAME

DefineVirtualFile – define a virtual file inside a real file (V4.0)

SYNOPSIS

```
virtfile$ = DefineVirtualFile(file$, offset, size, name$)
```

FUNCTION

This function allows you to define a virtual file inside another file which can be useful in several situations. Imagine you are working on a game and you want to store all data of the game in one huge resource file. Now you need to load some data from this huge resource file and that is when `DefineVirtualFile()` comes into play.

As parameter 1 you pass the name of the file that shall be the source of the virtual file. Parameters 2 and 3 then define the location of the virtual file inside `file$`. The virtual file to be created will be located inside `file$` from file position offset to file position offset+size. Parameter 4 finally specifies the file name for the virtual file. The only thing that is important here is the file extension because it gives Hollywood a hint of the virtual file's type. Thus, you should make sure that you pass the correct file extension. The name does not matter, but the file extension should be passed because not all files can be easily identified by their header.

`DefineVirtualFile()` returns a string describing the virtual file. You can pass this string to all Hollywood functions which accept a file name. Of course, only read access is supported by virtual files. Attempting to write to virtual files will not work.

INPUTS

`file$` source file from which data of the virtual file shall be taken

`offset` start position of the virtual file inside `file$`

`size` length in bytes of the virtual file inside `file$`

`name$` the name and file extension of the virtual file (see above)

RESULTS

`virtfile$`
string describing the virtual file

EXAMPLE

```
vf$ = DefineVirtualFile("hugeresource.dat", 100000, 32768, "image.png")
LoadBrush(1, vf$, {LoadAlpha = True})
```

The code above defines a virtual file inside "hugeresource.dat". The virtual file is of the size of 32768 bytes and starts at position 100000 inside "hugeresource.dat". The virtual file is a PNG image. After describing the virtual file, the image will be loaded with a simple call to LoadBrush().

26.11 DefineVirtualFileFromString

NAME

DefineVirtualFileFromString – define a virtual file from a string source (V5.0)

SYNOPSIS

```
virtfile$ = DefineVirtualFileFromString(data$, name$[, writable])
```

FUNCTION

This function allows you to define a virtual file from a string source. A virtual file is a file that exists only in memory but you can still pass it to all Hollywood functions and they will act as if the file was really present on a physical drive. `DefineVirtualFileFromString()` takes two mandatory arguments: In the first argument you have to provide the data that shall constitute your virtual file's contents. Argument two specifies the name of the virtual file. The only thing that is important here is the file extension because it gives Hollywood a hint of the virtual file's type. Thus, you should make sure that you pass the correct file extension. The name does not matter, but the file extension should be passed because not all files can be easily identified by looking at their header bytes.

Starting with Hollywood version 6.1 `DefineVirtualFileFromString()` also supports the creation of virtual files that can be written to. If you want the virtual file to be writable, you have to set the `writable` parameter to `True`. In that case, `DefineVirtualFileFromString()` will create a writable virtual file for you. The writable file will be initialized with the contents passed in `data$`. If you pass an empty string in `data$`, an empty new writable virtual file will be created.

`DefineVirtualFileFromString()` returns a string describing the virtual file. You can pass this string to all Hollywood functions which accept a file name.

Please note that the file's contents are not limited to text only. You can also pass binary data inside `data$` because Hollywood strings can contain special control characters and the `NULL` character as well. Thus, it is perfectly possible to create virtual files containing binary data with this function.

When you are finished dealing with the virtual file, you should free the virtual file by calling the `UndefineVirtualStringFile()` function. Doing this is important because it will free any memory occupied by the virtual file.

INPUTS

`data$` source string that constitutes the virtual file's contents

`name$` the name and file extension of the virtual file (see above)

`writable` optional: `True` if this virtual file should be writable, `False` otherwise (defaults to `False`) (V6.1)

RESULTS

`virtfile$`
string describing the virtual file

EXAMPLE

```
vf$ = DefineVirtualFileFromString("This is a virtual file test.",
                                  "test.txt")
OpenFile(1, vf$)
While Not Eof(1) Do Print(Chr(ReadChr(1)))
CloseFile(1)
UndefineVirtualStringFile(vf$)
```

The code above creates a virtual text file and then reads from this virtual file using the Hollywood DOS library.

```
data$ = DownloadFile("http://www.airsoftsoftwair.de/images/" ..
                     "products/hollywood/47_shot1.jpg")
vf$ = DefineVirtualFileFromString(data$, "image.jpg")
LoadBrush(1, vf$)
DisplayBrush(1, 0, 0)
UndefineVirtualStringFile(vf$)
data$ = Nil
```

The code above downloads a JPEG image to a string and loads the image directly into Hollywood without having to save it to an external file first.

```
vf$ = DefineVirtualFileFromString("", "test.txt", True)
OpenFile(1, vf$, #MODE_WRITE)
WriteLine(1, "A virtual file test!")
CloseFile(1)
CopyFile(vf$, GetSystemInfo().UserHome)
UndefineVirtualStringFile(vf$)
```

The code above writes a string to a virtual file and then copies this virtual file to the user's home directory.

26.12 DeleteFile

NAME

DeleteFile – delete a file or directory

SYNOPSIS

`DeleteFile(file$[, t])`

DEPRECATED SYNTAX

`DeleteFile(file$[, callback, userdata, pattern$, matchdir])`

FUNCTION

Deletes the file or directory specified in `file$`. Please note that this function will recursively delete whole directories by default. It does not check if the specified directory is empty or not! If you specify a directory, it will be deleted with all subdirectories and all files in it unless explicitly told not to do so. So be very careful with this function!

`DeleteFile()` supports several optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `DeleteFile()`.

The following table fields are recognized by this function:

`Recursive:`

By default, `DeleteFile()` will recurse into all subdirectories and delete them if `file$` specifies a directory. If you don't want that, set this tag to `False`. (V9.0)

`Force:` If this tag is set to `True`, write- or delete-protected files will automatically be deleted without asking the callback function first. Note that if there is no callback function and `Force` is set to `False` (the default), `DeleteFile()` will just skip all write- or delete-protected files. Defaults to `False`. (V9.0)

`MustExist:`
By default, `DeleteFile()` will silently fail if you specify a file or directory that does not exist in `file$`. No error will be generated in this case. If you want `DeleteFile()` to show an error instead, set this tag to `True`. (V9.0)

`Pattern:` You can pass a filter pattern in this table field. In that case, `DeleteFile()` will only delete the files that match the specified pattern. For example, passing `*.jpg` in `Pattern` will only delete files that use the `.jpg` file extension. Of course, using a filter pattern makes only sense if you pass a directory in `file$`. Note that for historical reasons, the pattern specified in `Pattern` will also be matched against all subdirectories that are to be deleted. If you don't want that, set the `MatchDir` table tag to `False` (see below). The pattern specified in `Pattern` must adhere to the pattern rules as described in the documentation of the `MatchPattern()` function. See Section 26.42 [MatchPattern], page 449, for details. (V5.0)

`MatchDir:`
This table field specifies whether or not the filter pattern specified in `Pattern` should also be matched against subdirectories. If this is set to `True`, `DeleteFile()` will only recurse into subdirectories that match the specified filter pattern. If it is set to `False`, `DeleteFile()` will recurse into all subdirectories. For compatibility reasons, `MatchDir` defaults to `True`, but most of the time you will want to pass `False` here because it usually does not make sense to match a file pattern against a directory name. For example, it does not make sense to match the `*.jpg` example from above against directories as well. (V5.0)

`FailOnError:`
By default, `DeleteFile()` will fail if a file or directory can't be deleted. You can change this behaviour by setting `FailOnError` to `False`. In that case, `DeleteFile()` won't fail if a file or directory can't be deleted but instead your callback function, if there is one, will be notified using the `#DELETEFILE_FAILED` message and your callback must tell `DeleteFile()` how to proceed (retry, continue, abort). See below to learn how to set up a callback function for `DeleteFile()`. Note that `FailOnError` isn't used when `file$` is just a single file. It is only used when deleting complete directories or multiple files using patterns. `FailOnError` defaults to `True`. (V9.0)

`Async:` If this is set to `True`, `DeleteFile()` will operate in asynchronous mode. This means that it will return immediately, passing an asynchronous operation handle to you. You can then use this asynchronous operation handle to finish the operation by repeatedly calling `ContinueAsyncOperation()` until it returns `True`. This is very useful in case your script needs to do something else while the operation is in progress, e.g. displaying a status animation or something similar. By putting `DeleteFile()` into asynchronous mode, it

is easily possible for your script to do something else while the operation is being processed. See Section 19.4 [ContinueAsyncOperation], page 224, for details. Defaults to `False`. (V9.0)

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`

This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Callback:`

For fine-tuned control of the delete operation, you can specify a callback function that will be called on various occasions. For example, `DeleteFile()` will call it from time to time so that you can update a progress bar. It will also be called when a file is delete-protected to ask you how to proceed. If there is no callback function, `DeleteFile()` will silently skip delete-protected files. The callback function receives one argument: A table that contains more information.

The following callback types are available:

`#DELETEFILE_UNPROTECT:`

The callback function of type `#DELETEFILE_UNPROTECT` will be called if a file that should be deleted is delete-protected. This callback function needs to return `True`, if it is okay to unprotect the file or `False` if it shall not be unprotected. If you return -1, the delete operation will be completely aborted.

`Action:` `#DELETEFILE_UNPROTECT`

`File:` Contains the delete protected file that is to be unprotected. (fully qualified path)

`UserData:`

Contains the value you specified in the `UserData` table field (see below).

(V2.0)

`#DELETEFILE_STATUS:`

This callback will be run whenever a file is deleted. This is useful for updating a status bar, for example. The callback function of type `#DELETEFILE_STATUS` should normally return `False`. If it returns `True`, the delete operation will be aborted.

`Action:` `#DELETEFILE_STATUS`

`File:` Contains the fully qualified path of the file that is to be deleted next

`UserData:`
Contains the value you specified in the `UserData` table field (see below).

(V2.0)

`#DELETEFILE_FAILED:`
This callback can only be called if the `FailOnError` tag has been set to `False` (see above). In that case, the callback function of type `#DELETEFILE_FAILED` will be called whenever a delete operation has failed. It has to return `True` to abort the delete operation, `False` to continue even though an error has occurred or -1 to retry the delete operation that has just failed. The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#DELETEFILE_FAILED`

`File:` Contains the fully qualified path of the file that could not be deleted.

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

(V9.0)

`UserData:`
This field can be used to pass an arbitrary value to your callback function. The value you specify here will be passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `UserData` tag you can easily pass data to your callback function. You can specify a value of any type in `UserData`. Numbers, strings, tables, and even functions can be passed as user data. Your callback will receive this data in the `UserData` field in the table that is passed to it. (V3.1)

INPUTS

`file$` Filename or directory to delete

`t` optional: table containing further options (see above) (V9.0)

EXAMPLE

```
DeleteFile("FooBar")
```

Deletes the file (or directory) "FooBar" from the current directory.

```
Function p_DeleteCallback(msg)
  Switch msg.action
  Case #DELETEFILE_STATUS:
    DebugPrint("Now deleting", FilePart(msg.file))
  Case #DELETEFILE_UNPROTECT:
    Return(SystemRequest("Hollywood", FilePart(msg.file) ..
      " is delete protected!\nDo you want me to unprotect it?",
```

```
        "Yes|No"))
    EndSwitch
    Return(False)
EndFunction
DeleteFile("TestDir", {Callback = p_DeleteCallback})
```

Demonstrates the usage of a callback function. It will delete the directory "TestDir" from the current directory and print out information about the file that is currently being deleted.

26.13 DIRECTORY

NAME

DIRECTORY – link whole directory into applet or executable (V8.0)

SYNOPSIS

```
@DIRECTORY id, dir$[, table]
```

FUNCTION

This preprocessor command can be used to link the whole directory specified in `dir$` into your applet or executable when compiling your script. This makes it possible to conveniently link a lot of files into your applet or executable when compiling your script because you only need to add one additional line to your script instead of individual lines for each file.

Note that if you use `@DIRECTORY` you have to use the `GetDirectoryEntry()` function to access individual files and subdirectories stored in the directory that you have linked to your applet or executable. See below for an example. In case you are just running your script using the Hollywood interpreter, `GetDirectoryEntry()` will simply return the string you passed to it so that the script will work identically no matter if you're running it as a script using the Hollywood interpreter or if you've compiled it as an applet or executable. See Section 26.29 [GetDirectoryEntry], page 441, for details.

This preprocessor command also accepts an optional table argument that can be used to configure further options. The following tags are currently supported by the optional table argument:

`Recursive:`
: If this tag is set to `True`, `@DIRECTORY` will link all files in subdirectories of `dir$` as well. This is the default. Set it to `False` if you don't want `@DIRECTORY` to recurse into subdirectories as well.

`Link:`
: Set this field to `False` if you do not want to have this directory linked to your executable/applet when you compile your script. This field defaults to `True` which means that the directory will be linked to your executable/applet when Hollywood is in compile mode.

Note that `@DIRECTORY` doesn't only link all files and subdirectories inside `dir$` into your applet or executable, it will also create a directory object which can then be used with all functions that support directory objects, e.g. `NextDirectoryEntry()`

and `RewindDirectory()`. It is even possible to iterate over all files and subdirectories linked by `@DIRECTORY` to your script. See below for an example.

Finally, please note that only file/directory names, sizes, and the files' actual content will be linked to your applet or executable. File attributes like protection flags, date stamps, and comments won't be linked so if you try to query them, you'll get some default values instead.

If you want to open directories at runtime, please use the `OpenDirectory()` command.

INPUTS

`id`	a value that is used to identify this directory later in the code
`dir$`	the directory you want to have linked to your applet or executable
`table`	optional: a table containing further options (see above)

EXAMPLE

```
@DIRECTORY 1, "data"
LoadBrush(1, GetDirectoryEntry("data/title.png"))
```

The code above shows how to link all files and subdirectories inside the `data` directory to your applet or executable and then load the file `title.png` from this directory into brush 1. Note that in case the script hasn't been compiled as an applet or executable, `LoadBrush()` will simply load the file from `data/title.png`. In case the script has been compiled as an applet or executable, however, the file `title.png` is loaded directly from the applet or executable because it has been linked to it.

```
@DIRECTORY 1, "data"

Function p_DumpDirs(d$, indent)
    Local handle
    If d$ <> ""
        handle = OpenDirectory(Nil, GetDirectoryFile(1, d$))
    Else
        handle = 1
    EndIf
    Local e = NextDirectoryEntry(handle)
    While e <> Nil
        If e.Type = #DOSTYPE_DIRECTORY Then e.size = 0
        NPrint(RepeatStr(" ", indent) .. IIf(e.type = #DOSTYPE_FILE,
            "File:", "Directory:") .. " " .. e.name .. " " .. e.size
            .. " " .. HexStr(e.flags) .. " " .. e.time)
        If e.Type = #DOSTYPE_DIRECTORY Then p_DumpDirs(FullPath(d$,
            e.name), indent + 4)
        e = NextDirectoryEntry(handle)
    Wend
    If GetType(handle) = #LIGHTUSERDATA Then CloseDirectory(handle)
EndFunction

p_DumpDirs("", 0)
```

The code above shows how to recursively print all files and directories in a directory that has been linked to the applet or executable.

26.14 DirectoryItems

NAME

DirectoryItems – iterate over all items in a directory (V7.0)

SYNOPSIS

```
f = DirectoryItems(d$)
```

FUNCTION

This function can be used together with the generic `For` statement to traverse all files and sub-directories in a directory. It returns an iterator function which will return two values for each directory item: The first return value will be the name of the file or directory, the second return value will be a table with additional information about the directory item. Once all directory items have been returned, the iterator function will return `Nil` to break the generic `For` statement.

See Section 11.4 [Generic For statement], page 127, for details.

The table that is returned by `DirectoryItems()` as the second return value when used in a generic `For` loop will have the following fields initialized:

`Type:` This will be `#DOSTYPE_FILE` if the entry is a file or `#DOSTYPE_DIRECTORY` if the entry is a directory.

`Size:` This field will only be present if the entry is a file. In that case, this field will receive the size of the file in bytes.

`Flags:` This field will receive a combination of protection flags of the file or directory. See Section 26.50 [Protection flags], page 461, for details.

`Time:` This field will receive a string containing the time the file or directory was last modified. The string will always be in the format dd-mmm-yyyy hh:mm:ss. E.g.: 08-Nov-2004 14:32:13.

`LastAccessTime:`
This field will receive a string containing the time the file or directory was last accessed. This attribute is not supported on AmigaOS.

`CreationTime:`
This field will receive a string containing the time the file or directory was created. This attribute is only supported on Windows.

`Comment:` This field will contain the comment of a file. This is only supported by the Amiga versions.

Note that you can also manually traverse all files and sub-directories inside a directory by using the `OpenDirectory()`, `NextDirectoryEntry()` and `CloseDirectory()` functions. Using `DirectoryItems()`, however, is often more convenient.

INPUTS

`d$` directory to traverse

RESULTS

`f` iterator function for generic for loop

EXAMPLE

```
Function p_TraverseDir(d$, indent)
   For s$,t In DirectoryItems(d$)
      DebugPrint(RepeatStr(" ", indent) .. s$, t.time)
      If t.type = #DOSTYPE_DIRECTORY
        p_TraverseDir(FullPath(d$, s$), indent + 8)
      EndIf
   Next
EndFunction

p_TraverseDir("images", 0)
```

The function `p_TraverseDir()` can be used recursively print all files and sub-directories in the given directory. The example call prints the contents of a directory named "images" that must be stored relative to the script's path.

26.15 Eof

NAME

Eof – returns whether end of file has been reached

SYNOPSIS

```
result = Eof(id)
```

FUNCTION

Returns `True` in case the end of the file specified by `id` has been reached. Otherwise returns `False`.

INPUTS

`id` identifier of a file

RESULTS

`result` `True` if end of file was reached or `False`

EXAMPLE

See Section 26.48 [OpenFile], page 459.

26.16 Execute

NAME

Execute – synchronously execute a program

SYNOPSIS

```
Execute(file$[, args$, t])
```

DEPRECATED SYNTAX

```
Execute(cmdline$[, resetkeys])
```

FUNCTION

This function executes the program specified by `file$` synchronously and passes the arguments specified in `args$` to it. If you need to execute a program asynchronously, you have to use the `Run()` function. See Section 26.63 [Run], page 470, for details.

If supported by the operating system, this command can also be used to view data files like documents or images using their default viewer. In that case, `file$` can also be a non-executable file like a JPEG image or an MP3 file.

On Android `file$` has to be either a data file like a JPEG image or a package name like `com.airsoftsoftwair.hollywood` if you want this function to start another app.

Note that due to historical reasons, there are some pitfalls when using this function. Before Hollywood 9.0 this command expected program and arguments combined in just a single `cmdline$` string. In that case, extra care has to be taken when dealing with spaces (see below for details). Starting with Hollywood 9.0, there is a new syntax which allows you to pass program and arguments as two separate arguments which makes things much easier. However, to maintain compatibility with previous versions this new syntax can only be used if you explicitly pass a string in the second argument. So if you want to use the new syntax, make sure to pass a string in the second argument. If the program you want to start doesn't need any arguments, just pass an empty string (`""`) just to signal Hollywood that you want to use the new syntax.

If you don't pass a string in the second argument, the old syntax will be used which means that you need to be very careful when passing program paths that contain spaces since the very first space in `cmdline$` is interpreted as the separator of program and arguments. If you want to start a program whose path specification uses spaces, you need to use double quotes around this path specification or it won't work. You can easily avoid these complications by simply passing a string in the second argument, even if it is empty (see above for details).

Starting with Hollywood 9.0, it is possible to specify the program and its arguments in two separate arguments, which makes things much more convenient. Also, there is a new optional table argument now that can be used to specify further options.

The following options are currently supported by the optional table argument:

`Directory:`
: This table argument allows you to set the current directory for the program that is to be started. (V9.0)

`ResetKeys:`
: This table argument is only interesting for advanced users. If this is set to `False`, `Execute()` won't reset all internal key states after executing the program. By default, all key states will be reset when `Execute()` returns because programs started using `Execute()` often assume the keyboard focus and Hollywood might be unable to reset its internal state flags because the new program started via `Execute()` takes over keyboard focus. That's why by default `Execute()` will reset all internal key state flags when it returns. Disabling this behaviour can make sense if you use `Execute()` to start programs that don't have a GUI and don't take away the keyboard focus. Defaults to `True`. (V5.1)

`ForceExe:`

If this tag is set to `True`, `Execute()` will always treat the file passed in `file$` as an executable. This is only useful on Linux and macOS because on those platforms files that have an extension will be treated as data files so Hollywood will try to launch the corresponding viewer for the data file instead. Thus, trying to use `Execute()` on an executable named "test.exe" will not work on Linux and macOS because of the *.exe extension. By setting `ForceExe` to `True`, however, you can make it work. Defaults to `False`. (V9.0)

`Verb:` On Windows, this can be set to a string telling `Execute()` what to do with the file. This can be one of the following verbs:

`edit`	Opens the specified file in an editor.
`explore`	Opens the specified folder in Explorer. When using this verb, you must pass a folder instead of a file to `Execute()`.
`find`	Opens the search dialog for the specified folder. When using this verb, you must pass a folder instead of a file to `Execute()`.
`open`	Opens the specified file.
`print`	Prints the specified file.
`runas`	Launches the specified file in administrator mode.

Note that the `Verb` tag is only supported on Windows. (V9.1)

INPUTS

`file$`	the program (or data file) to be started
`args$`	optional: arguments to pass to the program; note that you must pass this parameter to signal Hollywood to use the new syntax; you can do so by just passing an empty string (`""`); see above for a detailed discussion (V9.0)
`t`	optional: table containing further arguments (see above) (V9.0)

EXAMPLE

```
Execute("Sys:Prefs/Locale")
```

On AmigaOS systems the above code executes the locale preferences. Your script's execution will be halted until the user closes the locale preferences (synchronous execution).

```
Execute("Echo", ">Ram:Test \"Hello World\"")
```

On AmigaOS systems the above code writes "Hello World" to "Ram:Test".

```
Execute("\"C:\\Program Files (x86)\\Hollywood\\ide.exe\"")
```

The code above runs the Hollywood IDE on Windows systems. Note that we've embedded the program specification inside double quotes. This is absolutely necessary because the first space in the string passed to `Execute()` is normally interpreted as the separator between program and arguments. If we didn't use double quotes in the code above, `Execute()` would try to start the program "C:\Program" and pass the arguments "Files (x86)\Hollywood\ide.exe" to it which we obviously don't want. Note that since Hollywood 9.0, it is now much easier to deal with spaces in paths. You just need to use

the new syntax which takes the program and its arguments in two separate arguments. With Hollywood 9.0, you could simply use this code:

```
Execute("C:\\Program Files (x86)\\Hollywood\\ide.exe", "")
```

Note that passing the empty string in the second argument is absolutely necessary here to signal Hollywood that you want to use the new syntax. See above for a detailed discussion on this.

26.17 Exists

NAME

Exists – check if the specified file exists

SYNOPSIS

```
result = Exists(filename$)
```

FUNCTION

Checks if the file specified by `filename$` exists and returns `True` to the variable result if it does. Otherwise result receives the value of `False`.

INPUTS

`filename$`
: file to check

RESULTS

`result` `True` if the specified file exists, `False` otherwise

EXAMPLE

```
result = Exists("test.hws")
Print(result)
```

This tests whether the file "test.hws" exists in the current directory and returns `True` or `False`.

26.18 FILE

NAME

FILE – open a file for later use (V2.0)

SYNOPSIS

```
@FILE id, filename$[, table]
```

FUNCTION

This preprocessor command can be used to open a file so you can use it later. The file will not be loaded completely into memory, it will just be opened as if you called `OpenFile()`. The file will always be opened in read-only mode. You cannot use this preprocessor command to write to files.

The innovative feature of the `@FILE` preprocessor command is that when you compile your script, the file will be linked to it and you can still access it in the same way as if

it were a normal file on your harddisk, i.e. you can use the normal functions of the DOS library on the file.

The third argument is optional. It is a table that can be used to set further options for the operation. The following fields of the table can be used:

`Link:` Set this field to `False` if you do not want to have this file linked to your executable/applet when you compile your script. This field defaults to `True` which means that the file is linked to your executable/applet when Hollywood is in compile mode.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

If you want to open the file manually, please use the `OpenFile()` command.

INPUTS

`id` a value that is used to identify this file later in the code

`filename$`
the file you want to have opened

`table` optional: a table containing further options

EXAMPLE

```
@FILE 1, "Highscore.txt"
```

The declaration above opens the file "Highscore.txt" for further processing in the script.

26.19 FileAttributes

NAME

FileAttributes – get attributes of a file (V6.0)

SYNOPSIS

```
t = FileAttributes(id)
```

FUNCTION

This function returns a table that contains the attributes of a file that has been opened using `OpenFile()`. This includes information such as the file time, the full path of the file, protection flags, and more, depending on the host file system.

On return, the table will have the following fields initialized:

`Path:` This field will contain a string with the full path to this file.

`Size:` This field will be set to the size of the file in bytes.

`Flags:` This field will be set to a combination of protection flags of the file. See Section 26.50 [Protection flags], page 461, for details.

`Time:` This field will receive a string containing the time the file or directory was last modified. The string will always be in the format dd-mmm-yyyy hh:mm:ss. E.g.: 08-Nov-2004 14:32:13.

`LastAccessTime:`
This field will receive a string containing the time the file or directory was last accessed. This attribute is not supported on AmigaOS.

`CreationTime:`
This field will receive a string containing the time the file or directory was created. This attribute is only supported on Windows.

`Comment:` This field will contain the comment of a file. This is only supported by the Amiga versions.

`Streaming:`
This field will be set to `True` if the file is being streamed from a remote source instead of being read from a physical drive.

`NoSeek:` This field will be set to `True` if this file cannot be seeked. This could happen if the file is being streamed from a remote source that only allows sequential reads without any seeking capabilities.

If you want to query the attributes of a file that is not currently open, use `GetFileAttributes()` instead. See Section 26.31 [GetFileAttributes], page 442, for details.

INPUTS

`id` identifier of the file to query

RESULTS

`t` a table initialized as shown above

EXAMPLE

```
OpenFile(1, "test.txt")
t = FileAttributes(1)
Print(t.time)
If t.flags & #FILEATTR_READ_USR
  Print("#FILEATTR_READ_USR is set.")
Else
  Print("#FILEATTR_READ_USR is not set.")
EndIf
```

The code above examines the file "test.txt" and prints the time it was last modified to the screen. Additionally, it checks if the protection flag `#FILEATTR_READ_USR` is set.

26.20 FileLength

NAME

FileLength – return size of an open file (V3.0)

SYNOPSIS

```
size = FileLength(id)
```

FUNCTION

This function returns the current size of the file specified by `id`. The size returned by this function will be up to date with all operations done on this file. For example, you could write to the file and then `FileLength()` would return the new size of the file.

Please note that `FileLength()` can also return -1 if it does not know the file's size. This can happen in case the file is read from a streamed source through a file adapter, for example.

INPUTS

`id`	identifier of the file to query

RESULTS

`size`	current size of this file

EXAMPLE

```
OpenFile(1, "test.txt", #MODE_WRITE)
NPrint(FileLength(1))
WriteLine(1, "Hello World.")
NPrint(FileLength(1))
CloseFile(1)
```

The code above opens file "test.txt" for writing and calls `FileLength()` twice. The first call will return 0 because the file is empty at that point but the second call will return 13 because some characters have been written to the file now.

26.21 FileLines

NAME

FileLines – return a line-based iterator function (V5.0)

SYNOPSIS

```
f = FileLines(file$)
```

FUNCTION

This function can be used in conjunction with the generic `For` statement to traverse over all lines of a file. It will return an iterator function which will return the next line of the file specified in `file$`. When the end of the file is reached, the iterator function will return `Nil` to break the generic `For` statement.

See Section 11.4 [Generic For statement], page 127, for details.

INPUTS

`file$`	source filename

RESULTS

`f` line-based iterator function

EXAMPLE

```
For s$ In FileLines("Highscores.txt") Do DebugPrint(s$)
```

This will print all lines of the file "Highscores.txt".

26.22 FilePart

NAME

FilePart – return the file component of a path

SYNOPSIS

```
file$ = FilePart(path$)
```

FUNCTION

This function extracts the filename from a path specified by path$ and returns it.

INPUTS

`path$` source path

RESULTS

`file$` file part

EXAMPLE

```
f$ = FilePart("Data/Gfx/Test.jpg")
Print(f$)
```

The above code prints "Test.jpg" to the screen.

26.23 FilePos

NAME

FilePos – return file cursor position (V2.0)

SYNOPSIS

```
pos = FilePos(id)
```

FUNCTION

This function returns the file cursor position of the file specified by `id`. The cursor starts at 0 (beginning of the file) and ends at the length of the file. You can use this function to find out where you are in a file because all read and write operations will start at this cursor position. You can use `Seek()` to modify the file cursor position.

INPUTS

`id` identifier of the file to query

RESULTS

`pos` cursor position of this file

EXAMPLE

```
OpenFile(1, "test.txt", #MODE_READ)
Seek(1, 1024)
Print(FilePos(1))
CloseFile(1)
```

This prints 1024.

26.24 FileSize

NAME

FileSize – return the size of a specified file

SYNOPSIS

```
size = FileSize(file$)
```

FUNCTION

Returns the size of file `file$`. If the file does not exist, -1 is returned.

Please note that `FileSize()` can also return -1 if it does not know the file's size. This can happen in case the file is read from a streamed source through a file adapter, for example.

INPUTS

`file$` source filename

RESULTS

`size` size of the specified file in bytes

EXAMPLE

```
result = FileSize("test.jpg")
Print("The file test.jpg takes up", result, "bytes!")
```

This will print the size of the file "test.jpg".

26.25 FileToString

NAME

FileToString – read whole file into a string (V5.0)

SYNOPSIS

```
s$, len = FileToString(file$[, t])
```

FUNCTION

This command is a convenience function which simply reads the specified file into memory and returns it as a string. The second return value contains the file length in bytes. Note that since Hollywood strings can also contain binary data, you can also use this function to read non-text files into strings.

Starting with Hollywood 10.0, `FileToString()` accepts an optional table argument that allows you to pass additional arguments to the function. The following tags are currently supported by the optional table argument:

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to open the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`file$` file to read into string

`t` optional: table containing additional options (see above) (V10.0)

RESULTS

`s$` contents of the specified file as a string

`len` length of the file in bytes

26.26 FlushFile

NAME

FlushFile – flush all pending buffers (V2.5)

SYNOPSIS

```
FlushFile(id)
```

FUNCTION

This function flushes any pending buffers on the file specified by `id` and re-adjusts the file cursor. It is important that you call this function if you switch between buffered and unbuffered IO on the same file. If you do not use `SetIOMode()` at all, you do not have to worry about flushing buffers either because everything will be done automatically by the file system if you only use buffered IO.

INPUTS

`id` identifier of the file to be flushed

26.27 FullPath

NAME

FullPath – combine directory and file into a path (V2.0)

SYNOPSIS

```
path$ = FullPath(dir$, file$[, ...])
```

FUNCTION

This function combines `dir$` and `file$` into a path specification.

Starting with Hollywood 9.0, this function accepts an unlimited number of arguments, allowing you to combine an unlimited number of path constituents into a single path.

INPUTS

`dir$` source directory

`file$` source file

`...` optional: additional items to be appended to the path (V9.0)

RESULTS

`path$` path specification

EXAMPLE

```
path$ = FullPath("/home/andreas", "image.jpg")
```

`path$` receives the string "/home/andreas/image.jpg".

```
path$ = FullPath("/home", "andreas", "image.jpg")
```

This does the same as the first example but passes three instead of two arguments to `FullPath()`.

26.28 GetCurrentDirectory

NAME

GetCurrentDirectory – return full path of current directory (V4.5)

SYNOPSIS

```
dir$ = GetCurrentDirectory([t])
```

FUNCTION

This function simply returns a fully qualified path to the current directory. You can change the current directory using `ChangeDirectory()`.

Starting with Hollywood 10.0, this function accepts an optional table argument which supports the following tags:

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:` This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`t` optional: table argument containing further options (see above) (V10.0)

RESULTS

`dir$` path to the current directory

26.29 GetDirectoryEntry

NAME

GetDirectoryEntry – get entry from linked directory (V8.0)

SYNOPSIS

```
e$ = GetDirectoryEntry(id, entry$)
```

FUNCTION

This function can be used to get the entry specified by `entry$` from the directory specified by `id`. This directory must have been opened by using the `@DIRECTORY` preprocessor command before.

When called from an applet or executable which has all the files in the directory specified by `id` linked to it, `GetDirectoryEntry()` will return a special handle which can then be passed to all Hollywood functions that accept a file or directory name, e.g. `LoadBrush()`.

If this function is called when just running a script with the Hollywood interpreter, i.e. not from a stand-alone applet or executable, `GetDirectoryEntry()` will simply return the string that has been passed to it in `entry$`, which makes it possible for scripts using `@DIRECTORY` to behave exactly the same, no matter whether they have been compiled as applets or executables, or if they are run as scripts using the Hollywood interpreter. If they are run using the Hollywood interpreter, they will just load the data from a real file then, whereas the data will be loaded directly from the applet or executable in case `GetDirectoryEntry()` is called from a stand-alone applet or executable.

INPUTS

`id` identifier of the directory to query

`entry$` entry of the directory you want to receive

RESULTS

`e$` special handle to this entry that can be passed to all Hollywood functions accepting files or directories as their parameters

EXAMPLE

See Section 26.13 [DIRECTORY], page 427.

26.30 GetEnv

NAME

GetEnv – read environment variable (V5.0)

SYNOPSIS

```
s$, ok = GetEnv(var$)
```

FUNCTION

This command can be used to read the contents of the environment variable specified in `var$`. If the specified environment variable could not be found, an empty string is returned and the second return value is set to `False`. If the environment variable could be found, the second return value will be `True`.

INPUTS

`var$` environment variable to examine

RESULTS

`s$` contents of specified environment variable

`ok` `True` or `False` depending whether or not the specified environment variable could be found

26.31 GetFileAttributes

NAME

GetFileAttributes – get attributes of a file or directory (V3.0)

SYNOPSIS

```
t = GetFileAttributes(f$[, table])
```

DEPRECATED SYNTAX

```
t = GetFileAttributes(f$[, adapter$])
```

FUNCTION

This function returns a table that contains the attributes of a file or directory. This includes information such as the file time, the full path of the file, protection flags, and more, depending on the host file system. Pass the name of a file or a directory to this command. You can specify an empty string (`""`) to get information of the current directory.

This function accepts an optional table argument which can be used to pass additional parameters. The following table elements are currently recognized:

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

On return, the table will have the following fields initialized:

`Type:` This will be `#DOSTYPE_FILE` if `f$` is a file or `#DOSTYPE_DIRECTORY` if `f$` is a directory.

`Path:` This field will contain a string with the full path to this file or directory.

`Size:` This field will only be present if `f$` is a file. In that case, this field will receive the size of the file in bytes.

`Flags:` This field will receive a combination of protection flags of the file or directory. See Section 26.50 [Protection flags], page 461, for details.

`Time:` This field will receive a string containing the time the file or directory was last modified. The string will always be in the format dd-mmm-yyyy hh:mm:ss. E.g.: 08-Nov-2004 14:32:13.

`LastAccessTime:`
This field will receive a string containing the time the file or directory was last accessed. This attribute is not supported on AmigaOS.

`CreationTime:`
This field will receive a string containing the time the file or directory was created. This attribute is only supported on Windows.

`Comment:` This field will contain the comment of a file. This is only supported by the Amiga versions.

`Virtual:` This field will be set to `True` if the file you passed to this function is a virtual file, i.e. a file linked to your applet/executable or a file created using `DefineVirtualFile()`. (V5.2)

If you want to query the attributes of a file that you have opened using `OpenFile()`, use `FileAttributes()` instead. See Section 26.19 [FileAttributes], page 434, for details.

INPUTS

`f$` name of file or directory to be examined

`table` optional: table containing further options (see above) (V10.0)

RESULTS

`t` a table initialized as shown above

EXAMPLE

```
t = GetFileAttributes("test.txt")
Print(t.time)
If t.flags & #FILEATTR_READ_USR
  Print("#FILEATTR_READ_USR is set.")
Else
  Print("#FILEATTR_READ_USR is not set.")
EndIf
```

The code above examines the file "test.txt" and prints the time it was last modified to the screen. Additionally, it checks if the protection flag `#FILEATTR_READ_USR` is set.

26.32 GetProgramDirectory

NAME
GetProgramDirectory – return program directory (V9.0)

SYNOPSIS
```
dir$ = GetProgramDirectory()
```

FUNCTION
This function returns the program's directory. Note that using this function only makes sense in compiled programs because when running scripts, `GetProgramDirectory()` will return the directory of the Hollywood interpreter because there is no other program at this time.

INPUTS
none

RESULTS
`dir$` path to the program directory

26.33 GetStartDirectory

NAME
GetStartDirectory – return initial directory (V9.0)

SYNOPSIS
```
dir$ = GetStartDirectory()
```

FUNCTION
This function returns the directory that was the current directory when the Hollywood script was started. This is only of use when running Hollywood scripts because in that case Hollywood will always change the current directory to the script's directory (unless you use the `-nochdir` console argument) so it's not easily possible to find out the directory that was current before Hollywood changed it to the script's directory.

For compiled programs, the start directory will obviously always be identical to the program directory that you can get using `GetProgramDirectory()`. See Section 26.32 [GetProgramDirectory], page 444, for details.

INPUTS
none

RESULTS
`dir$` path to the initial directory

26.34 GetTempFileName

NAME
GetTempFileName – return name for a temporary file (V3.0)

SYNOPSIS

```
f$ = GetTempFileName()
```

FUNCTION

This function can be used to obtain a file that you can use temporarily. This is useful in case you temporarily need to store some information in a file which you will delete later. Hollywood will delete all temporary files automatically when it terminates but you can also do that manually using `DeleteFile()`.

It is preferable to use this function if you need to work with temporary files because each operating system stores its temporary files in a different place. By using this function you can be sure that your temporary files end up in the correct folder.

Please note that this function will not only return a file name but it will also create an empty file for you. This is done to avoid any possible race conditions with other applications which might want to store their own temporary file under the very same name. This is not possible if the file already exists so this is why `GetTempFileName()` will create an empty file for you.

INPUTS

none

RESULTS

`f$` file name that you can use for temporary operations

EXAMPLE

```
f$ = GetTempFileName()
OpenFile(1, f$, #MODE_WRITE)
WriteLine(1, "My temporary file")
CloseFile(1)
```

The code above will obtain the name of a temporary file and then write some text into it. The file will be automatically deleted when Hollywood terminates.

26.35 GetVolumeInfo

NAME

GetVolumeInfo – get space information about a volume

SYNOPSIS

```
space = GetVolumeInfo(vol$, type)
```

FUNCTION

This function queries the volume specified by `vol$` for the information specified by `type`. The following constants are possible for `type`:

`#FREESPACE:`
Returns the free space of the volume

`#USEDSPACE:`
Returns the used space of the volume

INPUTS

`vol$` name of a DOS volume

`type` one of the constants as listed above

RESULTS

`info` free/used space of the volume

EXAMPLE

```
space = GetVolumeInfo("SYS:",#FREESPACE)
Print(space, "bytes are free on SYS:!")
```

The above code returns the free space on your SYS: volume on AmigaOS systems.

26.36 GetVolumeName

NAME

GetVolumeName – get a volume name

SYNOPSIS

```
name$ = GetVolumeName(vol$)
```

FUNCTION

This function tries to get the name of the volume specified by `vol$`. If it is successful, the volume's name is returned to `name$`.

INPUTS

`vol$` a DOS volume descriptor

RESULTS

`name$` name of the volume

EXAMPLE

```
n$=GetVolumeName("df0:")
Print(n$)
```

The above code prints the name of the volume in drive df0: (if there is any).

26.37 HaveVolume

NAME

HaveVolume – check if a volume exists in the system (V8.0)

SYNOPSIS

```
r = HaveVolume(vol$)
```

FUNCTION

This function can be used to check if the volume specified by `vol$` is currently available. This is especially useful on AmigaOS and compatible systems because it will suppress the "Please insert volume XXX into any drive" system requester that usually pops up on AmigaOS systems when trying to access non-existent volumes. By checking the existence of the volume using this command first, you can easily get rid of the annoying requester on AmigaOS.

INPUTS

`vol$` name of a DOS volume whose presence should be checked

RESULTS

`r` `True` if volume exists, `False` otherwise

EXAMPLE

```
Print(HaveVolume("FOOBAR:"))
```

The code above should return 0 because the specified typically doesn't exist. On AmigaOS, there will be no system requester asking for volume "FOOBAR:" if you use this code.

26.38 IsAbsolutePath

NAME

IsAbsolutePath – check if path is absolute (V9.0)

SYNOPSIS

```
result = IsAbsolutePath(p$)
```

FUNCTION

This function checks if the path specified by `p$` is an absolute path and returns `True` if it is or `False` if it isn't.

Note that this expects the specified path to be in the host's canonical format. Thus, a path like "/home" will be absolute on Linux (among others) but not on AmigaOS where "/home" just refers to a directory named "home" in the parent directory. To convert a path to the host format, you can use `MakeHostPath()`.

INPUTS

`p$` path to check

RESULTS

`result` `True` if path is absolute, `False` otherwise

26.39 IsDirectory

NAME

IsDirectory – check for file or directory (V2.0)

SYNOPSIS

```
r = IsDirectory(f$)
```

FUNCTION

This function checks if `f$` is a file or directory. If it is a directory, this function returns `True`, otherwise `False`.

INPUTS

`f$` file system object

RESULTS

`r` `True` if `f$` is a directory, `False` otherwise

EXAMPLE

```
r = IsDirectory("S:")
```

This returns `True`.

26.40 MakeDirectory

NAME

MakeDirectory – make a new directory (V1.5)

SYNOPSIS

```
MakeDirectory(dir$[, t])
```

FUNCTION

This function creates the new directory specified by `dir$`. Note that this function won't fail the directory specified by `dir$` already exists.

This function can also create more than one directory if required. `MakeDirectory()` will scan `dir$` recursively and create every directory that does not exist yet (Hollywood 1.9 and up).

Starting with Hollywood 10.0, this function accepts an optional table argument which supports the following tags:

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:` This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`dir$` directory to create

`t` optional: table containing further options (see above) (V10.0)

EXAMPLE

```
MakeDirectory("Test")
```

The code above creates the new directory `"Test"` in the current directory.

```
MakeDirectory("A/B/C/D/E")
```

The code above creates five new directories inside the current directory.

26.41 MakeHostPath

NAME

MakeHostPath – convert Hollywood path to host path format (V9.0)

SYNOPSIS

```
p$ = MakeHostPath(path$)
```

FUNCTION

This function can be used to convert a platform-independent Hollywood path to a path in the host system's canonical path format. The path that should be converted has to be passed in `path$`.

To ensure cross-platform compatibility, Hollywood paths may contain several constituents that the underlying host operating system doesn't understand. For example, it is possible to use `..` on AmigaOS and compatibles to indicate the parent directory even though AmigaOS doesn't understand this. It's also possible to use normal slashes in paths on Windows even though that operating system normally uses backslashes. Conversely, it's also possible to use backslashes on all other systems although they use slashes, and so on.

`MakeHostPath()` will make sure that the path it returns is fully compliant with the host operating system's requirements. However, you normally don't have to use this function as all Hollywood functions can deal with platform-independent Hollywood paths. It might only be necessary to call this function when passing paths to external programs which don't understand Hollywood's platform-independent path format.

INPUTS

`path$` path to convert

RESULTS

`p$` converted path in the host's canonical format

EXAMPLE

```
Print(MakeHostPath("../image.jpg"))
```

On AmigaOS, this will print `"/image.jpg"` since AmigaOS doesn't understand the `..` token. On Windows, this will print `"..\image.jpg"` since Windows uses backslashes instead of slashes. On all other platforms the source string will be returned because no changes are necessary.

26.42 MatchPattern

NAME

MatchPattern – check for a pattern match with a string (V2.0)

SYNOPSIS

```
bool = MatchPattern(src$, pattern$)
```

FUNCTION

This function checks if the string specified in `src$` matches the pattern specified in `pattern$`. If it does, `True` will be returned, else `False`. `MatchPattern()` will compare

`pattern$` to `src$` character by character and abort as soon as it finds a difference. If it does not find a difference, it will return `True`.

The pattern specified in `pattern$` is a string that can contain normal characters and wildcards. A wildcard is a special character that can be used to match more than one character in the source string. The following wildcards are currently supported:

`*`	Matches all characters.
`?`	Matches just a single character.
`#`	Matches all numbers.
`[]`	Matches one or several characters or a range of characters if delimited using a hyphen. For example, [a] matches only `a`, whereas [af] matches `a` and `f` and [a-f] matches all characters in the range of `a` to `f`. You can use the '!' prefix to negate the result, i.e. [!a] matches every character except `a`.

You can also combine multiple patterns in a single string by separating them using a semicolon.

If you need more sophisticated pattern matching, have a look at the `PatternFindStr()` function. See Section 51.44 [PatternFindStr], page 1050, for details.

INPUTS

`src$`	source string
`pattern$`	pattern to compare string with

RESULTS

`bool`	`True` if string matches the pattern or `False`

EXAMPLE

```
r = MatchPattern("Pictures/JPG/Pic1.jpg", "*.jpg")
```

Returns `True` because the string matches the pattern.

```
r = MatchPattern("Pictures/JPG/Pic1.gif", "*.jpg;*.gif")
```

Returns `True` because the string matches the pattern.

```
r = MatchPattern("Hollywood 2.a", "Hollywood #.#")
```

Returns `False` because `a` does not match the numeric wildcard (#).

26.43 MD5

NAME

MD5 – calculate MD5 checksum of file (V5.0)

SYNOPSIS

```
sum$ = MD5(f$)
```

FUNCTION

This function calculates the MD5 checksum of the file specified in `f$` and returns it. The 128-bit checksum is returned as a string containing 16 hex digits.

If you want to compute the MD5 checksum of a string, use the `MD5Str()` function instead.

INPUTS

`f$` file whose checksum you want to have calculated

RESULTS

`sum$` MD5 checksum of file

26.44 MonitorDirectory

NAME

MonitorDirectory – monitor changes in a directory (V8.0)

SYNOPSIS

```
[id] = MonitorDirectory(id, dir$[, table])
```

FUNCTION

This function can be used to monitor changes in the directory specified by `dir$`. In order to monitor directory changes, `MonitorDirectory()` will create a new directory object and assign the specified `id` to it. If you pass `Nil` in `id`, `MonitorDirectory()` will automatically choose an identifier and return it.

Whenever something in the directory specified by `dir$` changes, `MonitorDirectory()` will send a `DirectoryChanged` event to your script. In order to handle this event, you need to install an event handler for it first using the `InstallEventHandler()` function. See Section 29.13 [InstallEventHandler], page 553, for details.

`MonitorDirectory()` also accepts an optional table argument which allows you to configure some further options. The following tags are currently recognized by the optional table argument:

`All:` If this tag is set to `True`, `MonitorDirectory()` will forward all directory change notifications from the operating system to your script. Think twice before using this because, depending on the operating system and file system, you might get several messages for just a single change because of file system internals. By default, `MonitorDirectory()` will try to filter such duplicate notifications for you so that you don't get several messages for just a single change. If you don't want `MonitorDirectory()` to apply this filter, i.e. if you want all notifications, set this tag to `True`. Defaults to `False`.

`UserData:`
This tag can be set to a value of an arbitrary type. `MonitorDirectory()` will store it in the `MonitorUserData` field of the message that is sent by `InstallEventHandler()`. This is useful for avoiding global variables.

`ReportChanges:`
If this tag is set to `True`, your event callback will also be notified about what exactly has changed. Your event callback will receive two new parameters: `Type` informing you about the type of change, i.e. whether a file or directory has been added, removed, or changed, and `Name` will contain the name of the file or directory that has been changed. Note that the `All` table tag (see above) will be ignored when setting `ReportChanges` to `True`. (V9.0)

Note that the directory object created by this function must only be used for monitoring directory changes. It is not possible to pass it to other directory functions like `NextDirectoryEntry()` or `RewindDirectory()`. An exception is the `CloseDirectory()` function: You should call `CloseDirectory()` as soon as you are finished monitoring the directory. This ensures that no resources are wasted and no unnecessary messages are posted to your script.

Also note that some file systems do not support monitoring of directories. This can happen especially on network volumes or network file systems. In that case, `MonitorDirectory()` can fail.

INPUTS

`id`	id for the directory or `Nil` for auto id selection
`dir$`	name of the directory to monitor
`table`	optional: table containing further parameters (see above)

RESULTS

`id`	optional: identifier of the directory; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
InstallEventHandler({DirectoryChanged = Function(msg)
        NPrint(msg.action, msg.id, msg.directory)
    EndFunction})
MonitorDirectory(1, "Data")
Repeat
    WaitEvent
Forever
```

The code above monitors all changes in the "Data" directory and prints a message whenever something changes in that directory.

26.45 MoveFile

NAME

MoveFile – move file or directory (V7.1)

SYNOPSIS

```
MoveFile(src$, dst$[, t])
```

DEPRECATED SYNTAX

```
MoveFile(src$, dst$[, func, userdata])
```

FUNCTION

This function moves the file or directory specified in `src$` to the file or directory specified in `dst$`. Note that `dst$` must not exist or `MoveFile()` will fail. Also, `src$` must not be a volume's root directory because this obviously cannot be moved anywhere.

Moving files (or directories) on the same volume is really quick and takes almost no time. When moving files from one volume to another, `MoveFile()` first has to copy the files

and in a second step, delete them from the original volume. This process is much slower than moving files around on the same volume. That is why you can specify a callback function which monitors the progress of this operation.

`MoveFile()` supports several optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `MoveFile()`.

The following table fields are recognized by this function:

`Force:` If this tag is set to `True`, write- or delete-protected files will automatically be deleted without asking the callback function first. Note that if there is no callback function and `Force` is set to `False` (the default), `MoveFile()` will just skip all write- or delete-protected files instead of deleting them. Note that `Force` is only used when `MoveFile()` actually needs to delete files, i.e. when moving files from one volume to another. Moving files on the same volume doesn't involve any deleting. Defaults to `False`. (V9.0)

`BufferSize:`
This table field can be used to set the buffer size that should be used when copying files. The value passed here must be specified in bytes. The default is 16384, i.e. 16 kilobytes. Note that `BufferSize` is only used when `MoveFile()` actually needs to copy files, i.e. when moving files from one volume to another. Moving files on the same volume doesn't involve copying. (V9.0)

`FailOnError:`
By default, `MoveFile()` will fail if a file or directory can't be copied or deleted. You can change this behaviour by setting `FailOnError` to `False`. In that case, `MoveFile()` won't fail if a file or directory can't be copied or deleted but instead your callback function, if there is one, will be notified using the `#MOVEFILE_COPYFAILED` and `#MOVEFILE_DELETEFAILED` messages and your callback must tell `MoveFile()` how to proceed (retry, continue, abort). See below to learn how to set up a callback function for `MoveFile()`. Note that `FailOnError` is only used when `MoveFile()` actually needs to copy and delete files, i.e. when moving files from one volume to another. Moving files on the same volume doesn't involve any copying or deleting. `FailOnError` defaults to `True`. (V9.0)

`Async:` If this is set to `True`, `MoveFile()` will operate in asynchronous mode. This means that it will return immediately, passing an asynchronous operation handle to you. You can then use this asynchronous operation handle to finish the operation by repeatedly calling `ContinueAsyncOperation()` until it returns `True`. This is very useful in case your script needs to do something else while the operation is in progress, e.g. displaying a status animation or something similar. By putting `MoveFile()` into asynchronous mode, it is easily possible for your script to do something else while the operation is being processed. See Section 19.4 [ContinueAsyncOperation], page 224, for details. Defaults to `False`. (V9.0)

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`

This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Callback:`

For fine-tuned control of the move operation, you can specify a callback function that will be called on various occasions. For example, `MoveFile()` will call it from time to time so that you can update a progress bar. It will also be called when a file is delete-protected to ask you how to proceed. If there is no callback function, `MoveFile()` will silently skip delete-protected files. The callback function receives one argument: A table that contains more information.

Note that the callback function will only be called when moving files across volumes. Moving files on the same volume can be done instantly and won't result in any callback invocation. Also note that if files are moved across volumes and you do not specify a callback function, files that are delete-protected won't be deleted but will just be copied to the new location without deleting the old file.

The following callback types are available:

`#MOVEFILE_UNPROTECT:`

The callback function of type `#MOVEFILE_UNPROTECT` will be called when `MoveFile()` needs to delete a file which is delete-protected. The parameter table for this callback type will contain the following fields:

`Action:` `#MOVEFILE_UNPROTECT`

`File:` Contains the fully qualified path to the file that is delete-protected.

`UserData:`

Contains the value you passed in the `UserData` table field (see below).

This callback function needs to return `True` if it is okay to unprotect the file or `False` if it shall not be unprotected. If you return -1, the move operation will be completely aborted.

`#MOVEFILE_DELETE:`

This callback will be run whenever a file is deleted. The callback function of type `#MOVEFILE_DELETE` should normally re-

turn `False`. If it returns `True`, `MoveFile()` will abort the entire operation.

`Action:` `#MOVEFILE_DELETE`

`File:` Contains the fully qualified path of the file that is to be deleted next.

`UserData:`
Contains the value you specified in the `UserData` table field (see below).

`#MOVEFILE_COPY:`
This callback will be called while `MoveFile()` is copying files. The callback function of type `#MOVEFILE_COPY` should normally return `False`. If it returns `True`, `MoveFile()` will abort the entire operation.

`Action:` `#MOVEFILE_COPY`

`Source:` Contains the fully qualified path of the file that is currently being copied (source).

`Destination:`
Contains the fully qualified path of the file that is currently being copied (destination).

`Copied:` Contains the number of bytes that have already been copied.

`Filesize:`
Contains the filesize of the source file.

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

`#MOVEFILE_DELETEFAILED:`
This callback can only be called if the `FailOnError` tag has been set to `False` (see above). In that case, the callback function of type `#MOVEFILE_DELETEFAILED` will be called whenever a delete operation has failed. It has to return `True` to abort the entire operation, `False` to continue even though an error has occurred or -1 to retry the delete operation that has just failed. The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#MOVEFILE_DELETEFAILED`

`File:` Contains the fully qualified path of the file that could not be deleted.

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

(V9.0)

`#MOVEFILE_COPYFAILED:`
This callback can only be called if the `FailOnError` tag has been set to `False` (see above). In that case, the callback function of type `#MOVEFILE_COPYFAILED` will be called whenever a copy operation has failed. It has to return `True` to abort the entire operation, `False` to continue even though an error has occurred or -1 to retry the copy operation that has just failed. The following fields will be set in the table parameter that is passed to your callback function:

`Action:` `#MOVEFILE_COPYFAILED`

`Source:` Contains the fully qualified path of the file that is currently being copied (source).

`Destination:`
Contains the fully qualified path of the file that is currently being written (destination).

`UserData:`
Contains the value you passed in the `UserData` table field (see below).

(V9.0)

`UserData:`
This field can be used to pass an arbitrary value to your callback function. The value you specify here will be passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `UserData` tag you can easily pass data to your callback function. You can specify a value of any type in `UserData`. Numbers, strings, tables, and even functions can be passed as user data. Your callback will receive this data in the `UserData` field in the table that is passed to it.

INPUTS

`src$` source file or directory to move

`dst$` destination file or directory; must not exist

`t` optional: table containing additional options (see above) (V9.0)

EXAMPLE

```
MoveFile("image.png", "images/image.png")
```

Moves the file "image.png" to the subdirectory "images" while keeping its name.

26.46 NextDirectoryEntry

NAME

NextDirectoryEntry – get next entry from an open directory (V4.0)

SYNOPSIS

```
t = NextDirectoryEntry(id)
```

FUNCTION

This function gets the next entry from a directory previously opened using `OpenDirectory()` or `@DIRECTORY`. The function will return a table that contains detailed information about the entry just retrieved. If there are no more entries in the specified directory, this function will return `Nil`. Normally, this function is called in a loop until it returns `Nil`. That way you can scan the whole contents of a directory.

The table that is returned by this function will have the following fields initialized:

`Name:` This field will contain the name of entry.

`Type:` This will be `#DOSTYPE_FILE` if the entry is a file or `#DOSTYPE_DIRECTORY` if the entry is a directory.

`Size:` This field will only be present if the entry is a file. In that case, this field will receive the size of the file in bytes.

`Flags:` This field will receive a combination of protection flags of the file or directory. See Section 26.50 [Protection flags], page 461, for details.

`Time:` This field will receive a string containing the time the file or directory was last modified. The string will always be in the format dd-mmm-yyyy hh:mm:ss. E.g.: 08-Nov-2004 14:32:13.

`LastAccessTime:`
This field will receive a string containing the time the file or directory was last accessed. This attribute is not supported on AmigaOS.

`CreationTime:`
This field will receive a string containing the time the file or directory was created. This attribute is only supported on Windows.

`Comment:` This field will contain the comment of a file. This is only supported by the Amiga versions.

To rewind a directory iteration, use the `RewindDirectory()` function. See Section 26.62 [RewindDirectory], page 469, for details.

INPUTS

`id` identifier of the directory to query

RESULTS

`t` a table initialized as shown above

EXAMPLE

See Section 26.47 [OpenDirectory], page 458.

26.47 OpenDirectory

NAME

OpenDirectory – open a directory for examination (V4.0)

SYNOPSIS

```
[id] = OpenDirectory(id, dir$[, table])
```

FUNCTION

This function opens the directory specified in `dir$` and assigns the specified `id` to it. If you pass `Nil` in `id`, `OpenDirectory()` will automatically choose an identifier and return it. The directory can then subsequently be examined by using the `NextDirectoryEntry()` function which gives you low-level access to the directory which is especially useful for large directories or if you need additional information like sizes/attributes for the individual directory entries. You can get these very fast using a loop as presented in the example below.

Starting with Hollywood 6.0 this function accepts an optional table argument which can be used to pass additional parameters. The following table elements are currently recognized:

`Adapter:` This tag allows you to specify one or more directory adapters that should be asked to open the specified directory. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to directory adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

You should call `CloseDirectory()` as soon as you are finished with the directory. This ensures that the directory does not stay locked by the file system longer than needed.

This command is also available from the preprocessor: Use `@DIRECTORY` to link whole directories to your applet or executable.

INPUTS

`id`	id for the directory or `Nil` for auto id selection
`dir$`	name of the directory to open
`table`	optional: table containing further parameters (V6.0)

RESULTS

`id`	optional: identifier of the directory; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenDirectory(1, "Data")
e = NextDirectoryEntry(1)
While e <> Nil
```

```
  NPrint(IIf(e.type = #DOSTYPE_FILE, "File:", "Directory:"), e.name)
  e = NextDirectoryEntry(1)
Wend
CloseDirectory(1)
```

The code above opens directory "Data" and prints all files and directories present in that directory.

26.48 OpenFile

NAME

OpenFile – open a file for reading and writing

SYNOPSIS

```
[id] = OpenFile(id, filename$[, mode, table])
```

FUNCTION

This function attempts to open the file specified by `filename$` and assigns `id` to it. If you pass `Nil` in `id`, `OpenFile()` will automatically choose an identifier and return it. If the file does not exist, this function will fail unless you use the `mode` argument to open a file for writing. In that case, `OpenFile()` will create the file for you.

All read and write operations will start at the current file cursor position. You can manually set the file cursor by using the `Seek()` function but it is also increased if you use other functions which read from or write to the file.

Starting with Hollywood 2.0 you can use the optional argument `mode` to open the file in read (default) or write mode or in shared mode, which means that you can read from the file and you can also write to it. If a file is opened in read mode, all write operations will fail. If a file is opened in write mode, all read operations will fail.

Starting with Hollywood 6.0 this function accepts an optional table argument which can be used to pass additional parameters. The following table elements are currently recognized:

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Encoding:`

In case the file is a text file, you can set this tag to the charset used by the file. Hollywood will then handle charset conversions automatically when reading from or writing to the file using functions like `ReadLine()`, `ReadString()`, `WriteLine()` or `WriteString()`. By default, Hollywood expects text files to be in the UTF-8 charset because that's Hollywood's default charset. If you want to read from or write to a file using the ISO 8859-1 encoding instead, just set `Encoding` to `#ENCODING_ISO8859_1` and Hollywood will handle all conversions to and from ISO 8859-1 automatically. See Section 54.30 [SetDefaultEncoding], page 1138, for a list of available charsets. (V9.0)

`WriteBOM:`
: Set this tag to `True` if you want `OpenFile()` to add the UTF-8 BOM (byte order mark) to the beginning of the file. Obviously, `OpenFile()` will only do this if the file has been opened in write mode (`#MODE_WRITE`) and the file's encoding has been set to `#ENCODING_UTF8`. (V9.0)

`UserTags:`
: This tag can be used to specify additional data that should be passed to file adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Although Hollywood will automatically close all open files when it quits, it is strongly advised that you close an open file when you are done with it using the `CloseFile()` function so that it becomes available to the operating system again.

Starting with Hollywood 9.0, `filename$` may also be one of the special constants `#STDIN`, `#STDOUT`, and `#STDERR`. This is useful for advanced programmers who want to access the `stdin`, `stdout`, and `stderr` file streams associated with each program.

This command is also available from the preprocessor: Use `@FILE` to preopen files.

INPUTS

`id` identifier of the file or `Nil` for auto id selection

`filename$`
: name of the file to open

`mode` mode to open the file; can be `#MODE_READ`, `#MODE_WRITE` or `#MODE_READWRITE` (defaults to `#MODE_READ`) (V2.0)

`table` optional: table containing further parameters (V6.0)

RESULTS

`id` optional: identifier of the file; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenFile(1, "Highscores.txt")
While Not Eof(1) Do NPrint(ReadLine(1))
CloseFile(1)
```

This code opens the file "Highscores.txt" as file 1 and prints all of its lines to the screen.

26.49 PathPart

NAME

PathPart – return the path component of a path

SYNOPSIS

```
p$ = PathPart(path$)
```

FUNCTION

This function extracts the pathname from a path specified by `path$` and returns it. The returned path part will always end with a `"/"` or a `":"` so that you can immediately add a filename to it.

INPUTS

`path$` source path

RESULTS

`p$` path part

EXAMPLE

```
p$ = PathPart("Data/Gfx/Test.jpg")
Print(p$)
```

The above code prints "Data/Gfx" to the screen.

26.50 Protection flags

The functions `GetFileAttributes()`, `FileAttributes()` and `SetFileAttributes()` allow you to get and set the protection flags of a file or directory. The flags are returned and set in a single table item called `flags`. This item contains bitmask which is a combination of the active flags.

As protection flags are dependent on the host file system, not all of the flags listed below are supported on all platforms. See the brackets for information on which platform supports which attributes.

The following flags are recognized by Hollywood:

```
#FILEATTR_READ_USR        [AmigaOS, macOS, Linux, iOS, Android]
#FILEATTR_WRITE_USR       [AmigaOS, macOS, Linux, iOS, Android]
#FILEATTR_DELETE_USR      [AmigaOS]
#FILEATTR_EXECUTE_USR     [AmigaOS, macOS, Linux, iOS, Android]
#FILEATTR_READ_GRP        [macOS, Linux, iOS, Android]
#FILEATTR_WRITE_GRP       [macOS, Linux, iOS, Android]
#FILEATTR_EXECUTE_GRP     [macOS, Linux, iOS, Android]
#FILEATTR_READ_OTH        [macOS, Linux, iOS, Android]
#FILEATTR_WRITE_OTH       [macOS, Linux, iOS, Android]
#FILEATTR_EXECUTE_OTH     [macOS, Linux, iOS, Android]
#FILEATTR_PURE            [AmigaOS]
#FILEATTR_ARCHIVE         [AmigaOS, Windows]
#FILEATTR_SCRIPT          [AmigaOS]
#FILEATTR_HIDDEN          [AmigaOS, Windows]
#FILEATTR_SYSTEM          [Windows]
#FILEATTR_READONLY        [Windows]
```

To set some of these attributes for a file, simply combine them using the bitwise Or operator. For example:

```
t = {}
t.flags = #FILEATTR_READ_USR | #FILEATTR_WRITE_USR
```

```
SetFileAttributes("test.txt", t)
```

The code above will give read and write permission to the file "test.txt". But please note, that this code would not work correctly under Windows because Windows does not know these two attributes.

To check if a flag is set, use the bitwise And operator. For example:

```
t = GetFileAttributes("test.txt")
If (t.flags & #FILEATTR_READ_USR)
   Print("#FILEATTR_READ_USR is set.")
EndIf
```

There is another flag named `#FILEATTR_NORMAL`. This flag has a special meaning and can only be used with `SetFileAttributes()` and it cannot be combinated with other flags. When you pass `#FILEATTR_NORMAL` to `SetFileAttributes()`, the file's protection flags will be reset to the operating system defaults, which differ from platform to platform.

26.51 ReadByte

NAME

ReadByte – read byte from file (V7.0)

SYNOPSIS

```
b = ReadByte(id[, flags])
```

FUNCTION

This function reads a single byte from the file specified by `id` and returns it. Reading starts from the current file cursor position which you can modify using the `Seek()` command. After reading, `ReadByte()` will advance the file cursor by one byte.

The `flags` parameter may be set to one of the following flags:

`#IO_UNSIGNED:`
: The return value will be unsigned and will range from 0 to 255. This is the default.

`#IO_SIGNED:`
: The return value will be signed and will range from -128 to 127.

INPUTS

`id`	file to read data from
`flags`	optional: additional flags (see above) (defaults to `#IO_UNSIGNED`) (V9.0)

RESULTS

`b`	byte read from file

26.52 ReadBytes

NAME

ReadBytes – read bytes from file (V7.0)

SYNOPSIS

```
data$ = ReadBytes(id[, len])
```

FUNCTION

This function reads `len` bytes from the file specified by `id`. If the `len` argument is omitted, `ReadBytes()` will read all bytes from the current file cursor position until the file end. `ReadBytes()` will advance the file cursor position by the number of bytes read.

This function is useful for reading binary data from a file. Since Hollywood strings can store binary data as well as text, `ReadBytes()` can just copy all the bytes it has read from the file in a Hollywood string and return it.

INPUTS

`id`	file to read data from
`len`	optional: number of bytes to read (defaults to 0 which means read until file end)

RESULTS

`data$`	data read from file

EXAMPLE

See Section 26.74 [WriteBytes], page 479.

26.53 ReadChr

NAME

ReadChr – read a character from the specified file

SYNOPSIS

```
chr = ReadChr(id[, encoding])
```

FUNCTION

This reads a single character from the file specified by `id` and returns its code point value. Note that depending on the character encoding, this might read up to 4 bytes from the file since in UTF-8, characters can use up to 4 bytes. The file cursor position is incremented by the number of bytes read.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

If you want to read a single byte from a file, use `ReadByte()` instead. See Section 26.51 [ReadByte], page 462, for details.

INPUTS

`id`	identifier of file to use
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`chr`	next character from file stream

EXAMPLE

```
OpenFile(1, "test", #MODE_READWRITE)
WriteLine(1, "Hello People! How are you?")
Seek(1, 0)
test = ReadChr(1)
CloseFile(1)
test$ = Chr(test)
Print(test$)
```

The above code will print "H" to the screen.

26.54 ReadDirectory

NAME

ReadDirectory – read a directory into a string array

SYNOPSIS

```
fcount, dcount = ReadDirectory(dir$, files$, dirs$[, sort])
```

FUNCTION

This function examines the directory specified by `dir$` and puts all filenames found in the directory tree to the string array specified in `files$` and all directory names to the string array specified in `dirs$`. After the last item, this function will insert an empty string into the array, so you know how many files/directories were found.

By default, all file and directory entries will be automatically sorted by this function. If you do not want this behaviour, you can set the optional argument `sort` to `False`.

Starting with Hollywood 2.0 this function returns two values: The first return value indicates how many files were in the directory and the second one indicates how many subdirectories were in the directory.

INPUTS

`dir$`	directory to examine
`files$`	string array to put the filenames to
`dirs$`	string array to put the directory names to
`sort`	optional: whether or not file and directory names shall be sorted (V4.5)

RESULTS

`fcount`	number of files in `dir$` (V2.0)
`dcount`	number of subdirectories in `dir$` (V2.0)

EXAMPLE

```
f$ = {}
d$ = {}
ReadDirectory("Data", f$, d$)
```

The above code reads the contents of the "Data" directory into the string arrays `f$` and `d$`.

26.55 ReadFloat

NAME

ReadFloat – read a float from a file (V2.0)

SYNOPSIS

```
float = ReadFloat(id[, width, le])
```

FUNCTION

This function reads a signed float value from the file specified by `id` and returns it. Reading starts from the current file cursor position which you can modify using the `Seek()` command. A float value takes up 8 bytes which is enough to store really big integers and floats with many decimal places.

Starting with Hollywood 6.0 there is an optional argument which allows you to specify the byte width of the floating point number. This can be 8 for a double-precision floating point number or 4 for a single-precision floating point number. By default, `ReadFloat()` reads double-precision floats.

By default, this function expects the data to be stored in big endian format (most significant byte first). Starting with Hollywood 6.0 you can use the optional argument `le` to explicitly request this function to use the little endian format instead.

INPUTS

`id`	file to read data from
`width`	optional: byte width of the float (defaults to 8) (V6.0)
`le`	optional: `True` to read bytes in little endian order, `False` for big endian order (defaults to `False`) (V6.0)

RESULTS

`float`	float value

26.56 ReadFunction

NAME

ReadFunction – read a function from a file (V4.0)

SYNOPSIS

```
func = ReadFunction(id)
```

FUNCTION

This function reads a Hollywood function from the file specified by `id` and returns it. Reading starts from the current file cursor position which you can modify using the `Seek()` command.

The function must have been written to the file by the `WriteFunction()` command.

INPUTS

`id`	file to read from

RESULTS

`func`	the function read from the file

EXAMPLE

See Section 26.77 [WriteFunction], page 481.

26.57 ReadInt

NAME

ReadInt – read an integer from a file (V2.0)

SYNOPSIS

```
int = ReadInt(id[, flags])
```

FUNCTION

This function reads an integer from the file specified by `id` and returns it. Reading starts from the current file cursor position which you can modify using the `Seek()` command. By default, `ReadInt()` will read a 32-bit integer, advancing the file cursor by 4 bytes.

The `flags` parameter may be a combination of the following flags:

`#IO_SIGNED:`
: The return value will be signed and will range from -2147483648 to 2147483647 (if `#IO_FAKE64` isn't set). This is the default.

`#IO_UNSIGNED:`
: The return value will be unsigned and will range from 0 to 4294967295. This cannot be combined with `#IO_FAKE64`.

`#IO_LITTLEENDIAN:`
: By default, this function expects the data to be stored in big endian format (most significant byte first). You can set this flag to request this function to use the little endian format instead.

`#IO_FAKE64:`
: Use 64-bit integers. This is called "fake 64" because Hollywood can't use the full 64-bit integer range because its numeric type is a 64-bit floating point value which can't represent exactly the same range as a true 64-bit integer value. Still, Hollywood's fake 64-bit integers should be large enough for almost anything. Using `#IO_FAKE64` you can read integers in the range of -9007199254740992 to 9007199254740992. Note that `#IO_UNSIGNED` can't be used with `#IO_FAKE64`. Hollywood's fake 64-bit integers will always be signed. (V9.0)

INPUTS

`id`	file to read data from
`flags`	optional: additional flags (see above) (defaults to `#IO_SIGNED`) (V9.0)

RESULTS

`int`	integer value

26.58 ReadLine

NAME

ReadLine – read a line from the specified file

SYNOPSIS

```
string$ = ReadLine(id[, lf])
```

FUNCTION

This command reads characters from the file specified by `id` until a line feed character occurs. Neither line feed nor carriage return characters are included in the destination string. This function also terminates when it reaches the end-of-file mark. The string read is returned.

Starting with Hollywood 9.0, there is an optional `lf` argument. If this is set to `True`, `ReadLine()` will also include newline characters (i.e. line feed and carriage return) if present in the source file. Note that no platform adaptation will take place in that case. If `lf` is set to `True`, ReadString() will return the exact newline characters that are in the source file, i.e. you might get line feed, carriage return and line feed or just carriage return, depending on the file's contents.

INPUTS

`id` identifier an open file

`lf` optional: whether or not to include newline characters in the string (defaults to `False`) (V9.0)

RESULTS

`string$` receives the line read

EXAMPLE

See Section 26.48 [OpenFile], page 459.

26.59 ReadShort

NAME

ReadShort – read 16-bit integer from a file (V2.0)

SYNOPSIS

```
short = ReadShort(id[, flags])
```

FUNCTION

This function reads a 16-bit integer from the file specified by `id` and returns it. Reading starts from the current file cursor position which you can modify using the `Seek()` command. Since `ReadShort()` reads a 16-bit integer from the file, the file cursor will be advanced by 2 bytes.

The `flags` parameter may be a combination of the following flags:

`#IO_UNSIGNED:`
: The return value will be unsigned and will range from 0 to 65535. This is the default.

`#IO_SIGNED:`
: The return value will be signed and will range from -32768 to 32767.

`#IO_LITTLEENDIAN:`
: By default, this function expects the data to be stored in big endian format (most significant byte first). You can set this flag to request this function to use the little endian format instead.

INPUTS

`id`	file to read data from
`flags`	optional: additional flags (see above) (defaults to `#IO_UNSIGNED`) (V9.0)

RESULTS

`short`	short value read from file

26.60 ReadString

NAME

ReadString – read string from file

SYNOPSIS

```
s$ = ReadString(id[, length, encoding])
```

FUNCTION

This function reads a string from the file specified by `id`. The optional `length` argument allows you to specify the number of characters to read from the file. If it is omitted, all characters from the current file cursor position until the end will be read and returned. The file cursor will be advanced by the number of bytes read from the file. This is not necessarily the same as the character count passed in `length` because in UTF-8 a single character may use up to 4 bytes.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

This function is used for reading text from files. If you need to read binary data from a file, use the `ReadBytes()` function instead. See Section 26.52 [ReadBytes], page 462, for details.

INPUTS

`id`	file to read data from
`length`	optional: characters to read or 0 to read all characters until the end of the file (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`s$`	string that contains the characters read

26.61 Rename

NAME

Rename – rename a file or directory (V2.0)

SYNOPSIS

```
Rename(oldname$, newname$[, t])
```

FUNCTION

This function renames a file or a directory. `oldname$` is the name of the file or directory to be renamed and can include a path specification. `newname$` is just the desired new name for the file/directory and must not contain any path specification.

Starting with Hollywood 10.0, this function accepts an optional table argument which supports the following tags:

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:` This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`oldname$`	file or directory to rename
`newname$`	new name for the file/directory
`t`	optional: table argument containing further options (see above) (V10.0)

EXAMPLE

```
Rename("image1.png", "image2.png")
```

Renames the file "image1.png" to "image2.png".

26.62 RewindDirectory

NAME

RewindDirectory – rewind directory iteration (V8.0)

SYNOPSIS

```
RewindDirectory(id)
```

FUNCTION

This function can be used to rewind the iteration of a directory opened by the `OpenDirectory()` function or the `@DIRECTORY` preprocessor command. This directory must be passed to `RewindDirectory()` in the `id` argument and the iteration must have been started using the `NextDirectoryEntry()` function before. After this function

returns, a call to `NextDirectoryEntry()` will return the first entry in the directory again.

See Section 26.46 [NextDirectoryEntry], page 456, for details.

INPUTS

`id` identifier of the directory to rewind

26.63 Run

NAME

Run – asynchronously execute a program

SYNOPSIS

```
Run(file$[, args$, t])
```

DEPRECATED SYNTAX

```
Run(cmdline$[, resetkeys, userdata])
```

FUNCTION

This function executes the program specified by `file$` asynchronously and passes the arguments specified in `args$` to it. If you need to execute a program synchronously, you have to use the `Execute()` function. See Section 26.16 [Execute], page 430, for details.

If supported by the operating system, this command can also be used to view data files like documents or images using their default viewer. In that case, `file$` can also be a non-executable file like a JPEG image or an MP3 file.

On Android `file$` has to be either a data file like a JPEG image or a package name like `com.airsoftsoftwair.hollywood` if you want this function to start another app.

If you want to be informed when the program started using `Run()` is terminated, you can install a listener for the `RunFinished` event handler using `InstallEventHandler()`. See Section 29.13 [InstallEventHandler], page 553, for details.

There is also a listener called `RunOutput` which can be installed using `InstallEventHandler()`. The `RunOutput` listener will redirect the program's output to your program which is useful when writing GUIs for console programs, for example. See Section 29.13 [InstallEventHandler], page 553, for details.

Note that due to historical reasons, there are some pitfalls when using this function. Before Hollywood 9.0 this command expected program and arguments combined in just a single `cmdline$` string. In that case, extra care has to be taken when dealing with spaces (see below for details). Starting with Hollywood 9.0, there is a new syntax which allows you to pass program and arguments as two separate arguments which makes things much easier. However, to maintain compatibility with previous versions this new syntax can only be used if you explicitly pass a string in the second argument. So if you want to use the new syntax, make sure to pass a string in the second argument. If the program you want to start doesn't need any arguments, just pass an empty string (`""`) just to signal Hollywood that you want to use the new syntax.

If you don't pass a string in the second argument, the old syntax will be used which means that you need to be very careful when passing program paths that contain spaces

since the very first space in `cmdline$` is interpreted as the separator of program and arguments. If you want to start a program whose path specification uses spaces, you need to use double quotes around this path specification or it won't work. You can easily avoid these complications by simply passing a string in the second argument, even if it is empty (see above for details).

Starting with Hollywood 9.0, it is possible to specify the program and its arguments in two separate arguments, which makes things much more convenient. Also, there is a new optional table argument now that can be used to specify further options.

The following options are currently supported by the optional table argument:

`Directory:`
: This table argument allows you to set the current directory for the program that is to be started. (V9.0)

`ResetKeys:`
: This table argument is only interesting for advanced users. If this is set to `False`, `Run()` won't reset all internal key states after executing the program. By default, all key states will be reset when `Run()` returns because programs started using `Run()` often assume the keyboard focus and Hollywood might be unable to reset its internal state flags because the new program started via `Run()` takes over keyboard focus. That's why by default `Run()` will reset all internal key state flags when it returns. Disabling this behaviour can make sense if you use `Run()` to start programs that don't have a GUI and don't take away the keyboard focus. Defaults to `True`. (V5.1)

`RawMode:`
: This tag is only used when the `RunOutput` event handler is active. By default, the `RunOutput` event handler expects programs to output text only. This is why `RunOutput` will make sure to pass only properly UTF-8 encoded text to your callback function. If you don't want `RunOutput` to format the text as UTF-8, you need to set the `RawMode` argument to `True` when calling `Run()`. In that case, `RunOutput` won't do any preformatting and will just forward the program's raw output to you. This means that your event handler callback has to be ready to process binary data as well. Defaults to `False`. (V9.0)

`IgnoreHandlers:`
: If event handlers for `RunFinished` or `RunOutput` are installed, those handlers will automatically trigger whenever `Run()` is called. If you only want those event handlers to trigger for certain calls to `Run()`, you can use this tag to tell `Run()` which event handlers to ignore. This must be set to a string containing the event handlers that should be ignored. Multiple event handlers must be separated by a vertical bar character. For example, setting `IgnoreHandlers` to `RunFinished|RunOutput` would tell `Run()` to not throw events for both event handlers, `RunFinished` and `RunOutput`. (V9.0)

`ReturnCode:`
: If you have a `RunFinished` event handler installed, you can set this tag to `True` to indicate that your event handler should also receive the program's return code when it terminates. Note that when setting this tag to `True` on AmigaOS 4 and MorphOS, Hollywood can't be quit before the program started using `Run()` has terminated. Defaults to `False`. (V9.0)

`ForceExe:`

If this tag is set to `True`, `Run()` will always treat the file passed in `file$` as an executable. This is only useful on Linux and macOS because on those platforms files that have an extension will be treated as data files so Hollywood will try to launch the corresponding viewer for the data file instead. Thus, trying to use `Run()` on an executable named `"test.exe"` will not work on Linux and macOS because of the *.exe extension. By setting `ForceExe` to `True`, however, you can make it work. Defaults to `False`. (V9.0)

`UserData:`

This argument can be used to specify user data that should be passed to the `RunFinished` and `RunOutput` event handlers that can be installed via `InstallEventHandler()`. See Section 29.13 [InstallEventHandler], page 553, for details. The user data you specify here can be of any type. (V6.1)

`Verb:` On Windows, this can be set to a string telling `Run()` what to do with the file. This can be one of the following verbs:

`edit`	Opens the specified file in an editor.
`explore`	Opens the specified folder in Explorer. When using this verb, you must pass a folder instead of a file to `Run()`.
`find`	Opens the search dialog for the specified folder. When using this verb, you must pass a folder instead of a file to `Run()`.
`open`	Opens the specified file.
`print`	Prints the specified file.
`runas`	Launches the specified file in administrator mode.

Note that the `Verb` tag is only supported on Windows. (V9.1)

INPUTS

`file$`	the program (or data file) to be started
`args$`	optional: arguments to pass to the program; note that you must pass this parameter to signal Hollywood to use the new syntax; you can do so by just passing an empty string (`""`); see above for a detailed discussion (V9.0)
`t`	optional: table containing further arguments (see above) (V9.0)

EXAMPLE

```
Run("Sys:Prefs/Locale")
```

The above code executes the locale preferences on AmigaOS based systems. Your script's execution will go on immediately after executing the locale program (asynchronous execution).

```
Run("\"C:\\Program Files (x86)\\Hollywood\\ide.exe\"")
```

The code above runs the Hollywood IDE on Windows systems. Note that we've embedded the program specification inside double quotes. This is absolutely necessary because

the first space in the string passed to `Run()` is normally interpreted as the separator between program and arguments. If we didn't use double quotes in the code above, `Run()` would try to start the program "C:\Program" and pass the arguments "Files (x86)\Hollywood\ide.exe" to it which we obviously don't want. Note that since Hollywood 9.0, it is now much easier to deal with spaces in paths. You just need to use the new syntax which takes the program and its arguments in two separate arguments. With Hollywood 9.0, you could simply use this code:

```
Run("C:\\Program Files (x86)\\Hollywood\\ide.exe", "")
```

Note that passing the empty string in the second argument is absolutely necessary here to signal Hollywood that you want to use the new syntax. See above for a detailed discussion on this.

26.64 Seek

NAME

Seek – set file cursor to a new position

SYNOPSIS

```
Seek(id, newpos[, mode])
```

FUNCTION

This function sets the file cursor (from which all read/write operations start) to `newpos`. The beginning of the file is at position 0. If you want to seek to the end-of-file, set `newpos` to the special constant `#EOF`.

To find out the cursor position of a specific file, you can use the `FilePos()` command.

Starting with Hollywood 6.0 you can use the optional `mode` argument to set the seek mode which should be used. This can be one of the following mode constants:

`#SEEK_BEGINNING:`
: The specified seeking position is relative to the beginning of the file. Negative positions are not allowed. This is the default seek mode.

`#SEEK_CURRENT:`
: The specified seeking position is relative to the current position of the file cursor. You may also pass negative positions here to seek backwards from the current file cursor position.

`#SEEK_END:`
: The specified seeking position is relative to the file's ending. You may only pass 0 or negative positions here. To seek to the end of the file, simply pass 0.

INPUTS

`id`	number specifying the file to use
`newpos`	position to set the file cursor to
`mode`	optional: seek mode to use (defaults to `#SEEK_BEGINNING`) (V6.0)

EXAMPLE
See Section 26.53 [ReadChr], page 463.

26.65 SetEnv

NAME
SetEnv – write environment variable (V5.0)

SYNOPSIS
`SetEnv(var$, s$)`

FUNCTION
This command can be used to set the environment variable specified in `var$` to the value specified in `s$`. Please note that the environment variable will be local to your Hollywood script. You cannot modify global environment variables with this function.

INPUTS

`var$`	environment variable to set
`s$`	desired value for environment variable

26.66 SetFileAttributes

NAME
SetFileAttributes – set attributes of a file or directory (V3.0)

SYNOPSIS
`SetFileAttributes(f$, t)`

FUNCTION
This function can be used to change one or multiple attributes of a file or directory. This includes information such as the file time, protection flags, and more, depending on the host file system.

The file (or directory) whose attributes you want to change must be passed as parameter `f$`. The second parameter is a table which contains all attributes you want to modify. The following fields can be set in the table:

`Flags:` Use this field to change the protection flags of the file or directory. Set this field to a combination of protection flags or to `#FILEATTR_NORMAL` to reset all protection flags. See Section 26.50 [Protection flags], page 461, for details.

`Time:` Use this field to change the time stamp of the file or directory. This field can be used to change the time when the file was last changed. You need to set this field to a string in the format dd-mmm-yyyy hh:mm:ss. E.g.: 08-Nov-2004 14:32:13.

`LastAccessTime:`
Use this field to modify the time the file or directory was last accessed. The string you specify here must be in the format dd-mmm-yyyy hh:mm:ss. This attribute is not supported on AmigaOS.

`CreationTime:`
: Use this field to modify the creation time of the file or directory. The string you specify here must be in the format dd-mmm-yyyy hh:mm:ss. This attributed is only supported on Windows.

`Comment:` Use this field to change the comment of a file. This is only supported by the Amiga versions.

`Adapter:` This tag allows you to specify one or more filesystem adapters that should be asked to handle the operation. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
: This tag can be used to specify additional data that should be passed to filesystem adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`f$` name of file or directory whose attributes are to be changed

`t` a table containing the attributes to be set

EXAMPLE

```
t = {}
t.time = "15-Dec-2006 23:30:12"
t.flags = #FILEATTR_READ_USR | #FILEATTR_WRITE_USR
SetFileAttributes("test.txt", t)
```

The code above sets the time stamp of file "test.txt" to December 15th, 2006 at 11:30pm and 12 seconds. Additionally it sets the protection flags `#FILEATTR_READ_USR` and `#FILEATTR_WRITE_USR`.

26.67 SetFileEncoding

NAME

SetFileEncoding – set file charset (V9.0)

SYNOPSIS

```
SetFileEncoding(id, encoding)
```

FUNCTION

This function changes the charset used by the file specified by `id` to the charset specified by `encoding`. All subsequent read and write operations will then be done using the new charset.

This must only be used if the file is a text file. In that case, specifying the file's charset can be quite convenient because Hollywood will then handle all charset conversions automatically when reading from or writing to the file using functions like `ReadLine()`, `ReadString()`, `WriteLine()` or `WriteString()`. By default, Hollywood expects text

files to be in the UTF-8 charset because that's Hollywood's default charset. If you want to read from or write to a file using the ISO 8859-1 encoding instead, just pass `#ENCODING_ISO8859_1` in `encoding` and Hollywood will handle all conversions to and from ISO 8859-1 automatically.

See Section 54.30 [SetDefaultEncoding], page 1138, for a list of available charsets.

INPUTS

`id` identifier of file

`encoding` desired new charset to use

26.68 SetIOMode

NAME

SetIOMode – switch between buffered and unbuffered IO (V2.5)

SYNOPSIS

```
SetIOMode(mode)
```

FUNCTION

This function can be used to specify the IO mode the functions of the Hollywood DOS library shall use. By default, all DOS functions use buffered IO. This is especially efficient for small read and write operations. For some cases, however, buffered IO is not very convenient and you might want to use unbuffered IO instead. For example, when you write to the parallel device using the DOS library or you have opened a console window using `OpenFile()`. In those cases unbuffered IO is to be preferred because the data is passed directly to the file system.

The mode you set using this function is respected by all functions of the DOS library but please note that if you switch between buffered and unbuffered IO on the same file, you have to use `FlushFile()` to flush all pending buffers. If you forget to do this, you might end up with data at the wrong positions in your file.

This function is meant for advanced users. Normally, you do not have to care about the IO mode.

INPUTS

`mode` desired IO mode for the DOS library; this can be either `#IO_BUFFERED` or `#IO_UNBUFFERED` (by default, Hollywood will always use `#IO_BUFFERED`)

26.69 StringToFile

NAME

StringToFile – save string to file (V5.0)

SYNOPSIS

```
StringToFile(s$, file$[, t])
```

FUNCTION

This command is a convenience function which simply saves the string specified in by `s$` as the file specified by `file$`. Be warned that this function does not append the string to the file. If the file specified by `file$` already exists, it will be overwritten without any warning. Note that since Hollywood strings can also contain binary data, you can also use this function to write strings containing raw data to files.

Starting with Hollywood 10.0, `StringToFile()` accepts an optional table argument that allows you to pass additional arguments to the function. The following tags are currently supported by the optional table argument:

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`s$`	string to write to file
`file$`	destination file
`t`	optional: table containing further options (see above)

26.70 UndefineVirtualStringFile

NAME

UndefineVirtualStringFile – undefine a string source virtual file (V5.0)

SYNOPSIS

```
UndefineVirtualStringFile(virtfile$)
```

FUNCTION

This function can be used to undefine a virtual file created by the function `DefineVirtualFileFromString()`. It is important to call this function when you are done with a virtual file because it will release memory occupied by the virtual file.

See Section 26.11 [DefineVirtualFileFromString], page 421, for more information on virtual string files.

INPUTS

`virtfile$`
a virtual file created by DefineVirtualFileFromString()

EXAMPLE

See Section 26.11 [DefineVirtualFileFromString], page 421.

26.71 UnsetEnv

NAME

UnsetEnv – delete environment variable (V5.0)

SYNOPSIS

```
UnsetEnv(var$)
```

FUNCTION

This command can be used to delete the specified environment variable. Please note that you cannot delete global environment variables with this function. You may only delete environment variables that are local to your Hollywood script.

INPUTS

`var$` environment variable to delete

26.72 UseCarriageReturn

NAME

UseCarriageReturn – configure line break behaviour (V7.1)

SYNOPSIS

```
UseCarriageReturn(enable)
```

FUNCTION

This function allows you to set whether or not `WriteLine()` will write a carriage return character before the line feed character. MS-DOS and its successor Windows both use carriage return and line feed characters to indicate a line break whereas Unix, Amiga, and macOS just use a line feed character to force a line break.

If `enable` is set to `True`, `WriteLine()` will write a carriage return character before each line feed character, otherwise it will only output a line feed character. By default, `UseCarriageReturn()` is set to `True` on Windows systems and to `False` on all other systems.

INPUTS

`enable` `True` or `False` indicating whether `WriteLine()` should output a carriage return before a line feed character

26.73 WriteByte

NAME

WriteByte – write byte to file (V7.0)

SYNOPSIS

```
WriteByte(id, b[, flags])
```

FUNCTION

This function writes a single byte to the file specified by `id` at the current file cursor position which you can modify by using the `Seek()` command. `WriteByte()` will advance the file cursor position by one byte.

The `flags` parameter may be set to one of the following flags:

`#IO_UNSIGNED:`
: Write an unsigned byte to the file. This means that `b` must be between 0 and 255. This is the default.

`#IO_SIGNED:`
: Write a signed byte to the file. This means that `b` must be between -128 and 127.

INPUTS

`id`	file to write to
`b`	byte data to write to the file
`flags`	optional: additional flags (see above) (defaults to `#IO_UNSIGNED`) (V9.0)

26.74 WriteBytes

NAME

WriteBytes – write bytes to file (V7.0)

SYNOPSIS

```
WriteBytes(id, data$[, len])
```

FUNCTION

This function writes `len` bytes from the string `data$` to the file specified by `id`. If the optional argument `len` is omitted, the complete string will be written to the file. `WriteBytes()` will advance the file cursor position by the number of bytes written.

This function is useful for writing binary data to a file. The string specified by `data$` will be treated as raw binary data instead of text.

INPUTS

`id`	file to write to
`data$`	data to write to file
`len`	optional: number of bytes to write (defaults to 0 which means write the complete string)

EXAMPLE

```
size = FileSize("test")
OpenFile(1, "test")
OpenFile(2, "copy_of_test", #MODE_WRITE)
data$ = ReadBytes(1, size)
WriteBytes(2, data$, size)
CloseFile(2)
CloseFile(1)
```

The above code makes a copy of the file "test" and saves it as "copy_of_test".

26.75 WriteChr

NAME

WriteChr – write a character to a file

SYNOPSIS

```
WriteChr(id, chr[, encoding])
```

FUNCTION

Writes the character specified by `chr` to the file specified by `id` and increments the file cursor by the number of bytes written. The character to be written has to be passed to `WriteChr()` as a code point value. Note that depending on the encoding, this function might write up to 4 bytes to the file because in UTF-8, a single character may use up to 4 bytes.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

If you need to write a single byte to a file, use the `WriteByte()` function instead. See Section 26.73 [WriteByte], page 478, for details.

INPUTS

`id`	identifier specifying the file to use
`chr`	code point value of character to write to file
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

EXAMPLE

```
OpenFile(1, "Test", #MODE_WRITE)
WriteChr(1, 65)
CloseFile()
```

This code writes the character "A" to the file Test.

26.76 WriteFloat

NAME

WriteFloat – write a float to a file (V2.0)

SYNOPSIS

```
WriteFloat(id, float[, width, le])
```

FUNCTION

This function writes a signed float to the file specified by `id` at the current file cursor position which you can modify by using the `Seek()` command. A float will use 8 bytes of disk space which is enough to store really big integers and floats with many decimal places.

Starting with Hollywood 6.0 there is an optional argument which allows you to specify the byte width of the floating point number. This can be 8 for a double-precision floating

point number or 4 for a single-precision floating point number. By default, `WriteFloat()` writes double-precision floats.

By default, this function stores the value in big endian format (most significant byte first). Starting with Hollywood 6.0 you can use the optional argument `le` to explicitly request this function to use the little endian format instead.

INPUTS

`id`	file to write to
`float`	float value to write to the file
`width`	optional: byte width of the float to write (defaults to 8) (V6.0)
`le`	optional: `True` to write bytes in little endian order, `False` for big endian order (defaults to `False`) (V6.0)

26.77 WriteFunction

NAME

WriteFunction – write a function to a file (V4.0)

SYNOPSIS

```
WriteFunction(id, func[, txtmode, nobrk])
```

FUNCTION

This function writes the Hollywood function specified by `func` to the file specified by `id` at the current file cursor position which you can modify by using the `Seek()` command. The function will be written to the file as precompiled bytecode, i.e. it will not be human readable.

You can load saved functions into other projects by using the `ReadFunction()` command. The optional argument `txtmode` specifies whether or not the function shall be written to the file as binary data or as base64 encoded data. The latter is useful for embedding Hollywood functions in human readable text files, for instance XML files. In case you enable text mode, `WriteFunction()` will automatically insert a line break after every 72 characters for better readability. If you don't want that, set the optional argument `nobrk` to `True`. In that case, no line breaks will be inserted.

INPUTS

`id`	file to write to
`func`	function to write to the file
`txtmode`	optional: `True` to write the function in base64 notation or `False` to write plain binary data (defaults to `False`)
`nobrk`	optional: `True` if you don't want to have line breaks inserted when in text mode (defaults to `False`); this argument is ignored in binary mode (V6.1)

EXAMPLE

```
Function p_LittleTestFunc(a, b)
  Return(a+b)
```

```
EndFunction

OpenFile(1, "func.bin", #MODE_WRITE)
WriteFunction(1, p_LittleTestFunc)
CloseFile(1)

OpenFile(1, "func.bin", #MODE_READ)
p_MyAdd = ReadFunction(1)
CloseFile(1)

Print(p_MyAdd(5, 6))  ; prints 11
```

The code above writes the function p_LittleTestFunc() to file "func.bin". After that, it opens file "func.bin" again and reads the function back into Hollywood. The imported function will be stored in the variable p_MyAdd(). Finally, we will call the newly imported function p_MyAdd() and it will add the numbers 5 and 6 for us.

26.78 WriteInt

NAME

WriteInt – write an integer to a file (V2.0)

SYNOPSIS

`WriteInt(id, int[, flags])`

FUNCTION

This function writes an integer to the file specified by `id` at the current file cursor position which you can modify by using the `Seek()` command. By default, `WriteInt()` will write a 32-bit integer, advancing the file cursor by 4 bytes.

The `flags` parameter may be a combination of the following flags:

`#IO_SIGNED`:
: Use signed integers. This means that `int` must be in the range of -2147483648 to 2147483647 (in case `#IO_FAKE64` isn't set). This is the default.

`#IO_UNSIGNED`:
: Use unsigned integers. This means that `int` must be in the range of 0 to 4294967295. Note that `#IO_UNSIGNED` cannot be combined with `#IO_FAKE64`.

`#IO_LITTLEENDIAN`:
: By default, this function stores the value in big endian format (most significant byte first). You can set this flag to request this function to use the little endian format instead.

`#IO_FAKE64`:
: Use 64-bit integers. This is called "fake 64" because Hollywood can't use the full 64-bit integer range because its numeric type is a 64-bit floating point value which can't represent exactly the same range as a true 64-bit

integer value. Still, Hollywood's fake 64-bit integers should be large enough for almost anything. Using `#IO_FAKE64` you can write integers in the range of -9007199254740992 to 9007199254740992. Note that `#IO_UNSIGNED` can't be used with `#IO_FAKE64`. Hollywood's fake 64-bit integers will always be signed. (V9.0)

INPUTS

`id`	file to write to
`int`	integer value to write to the file
`flags`	optional: additional flags (see above) (defaults to `#IO_SIGNED`) (V9.0)

26.79 WriteLine

NAME

WriteLine – write a new line to a file

SYNOPSIS

```
WriteLine(id, line$)
```

FUNCTION

Writes the string specified by `line$` to the file described by `id` and increases the file cursor accordingly.

Note that on Windows systems `WriteLine()` appends both, carriage return and line feed, to `line$` whereas on all other systems only the line feed character is appended to `line$`. This behaviour can be changed by calling the `UseCarriageReturn()` command. See Section 26.72 [UseCarriageReturn], page 478, for details.

INPUTS

`id`	number specifying the file to use
`line$`	string to write to file

EXAMPLE

See Section 26.53 [ReadChr], page 463.

26.80 WriteShort

NAME

WriteShort – write 16-bit integer to a file (V2.0)

SYNOPSIS

```
WriteShort(id, short[, flags])
```

FUNCTION

This function writes a 16-bit integer to the file specified by `id` at the current file cursor position which you can modify by using the `Seek()` command. Since a short integer is 16-bit, the file cursor will be advanced by 2 bytes.

The `flags` parameter may be a combination of the following flags:

`#IO_UNSIGNED:`
: Use an unsigned integer. This means that `short` must be in the range of 0 to 65535. This is the default.

`#IO_SIGNED:`
: Use a signed integer. This means that `short` must be in the range of -32768 to 32767.

`#IO_LITTLEENDIAN:`
: By default, this function stores the value in big endian format (most significant byte first). You can set this flag to request this function to use the little endian format instead.

INPUTS

`id`	file to write to
`short`	short integer value to write to the file
`flags`	optional: additional flags (see above) (defaults to `#IO_UNSIGNED`) (V9.0)

26.81 WriteString

NAME

WriteString – write string to file

SYNOPSIS

```
WriteString(id, s$[, len, encoding])
```

FUNCTION

This function writes the string `s$` to the file specified by `id`. The optional argument `len` can be used to set the number of characters that should be written to the file. If `len` is omitted, the complete string is written. The file cursor position is advanced by the number of bytes written to the file. Note that this is not necessarily the same as `len` because in UTF-8 encoding a single character can use up to 4 bytes.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

This function is used for writing text to files. If you need to write binary data to a file, use the `WriteBytes()` function instead. See Section 26.74 [WriteBytes], page 479, for details.

INPUTS

`id`	file to write to
`s$`	string to write to the file
`len`	optional: number of characters to file or 0 to write the complete string (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

27 Draw library

27.1 Arc

NAME

Arc – draw a partial ellipse (V2.0)

SYNOPSIS

```
Arc(x, y, xradius, yradius, start, end[[, color], table])
```

FUNCTION

This function draws a partial ellipse at the position specified by `x` and `y` using the specified radii and color (RGB value) in the style configured using the `SetFormStyle()` and `SetFillStyle()` commands. The arguments `start` and `end` specify the start and end angles of the ellipse and must be specified in degrees. If you want to draw a closed ellipse, the `start` argument needs to be 0 and the `end` argument needs to be 360. Using the `Ellipse()` command is of course easier in this case.

The width of the partial ellipse will be `xradius` * 2 + 1 (center point) and the height will be `yradius` * 2 + 1 (center point).

If layers are enabled, this command will add a new layer of the type `#ARC` to the layer stack.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Starting with Hollywood 4.5 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the style of the arc. The following options are possible:

`Clockwise:`

> You can use this tag to specify whether or not the elliptic arc shall be drawn in clockwise direction. This tag defaults to `True` which means clockwise drawing. If you set it to `False`, `Arc()` will connect the angles in anti-clockwise direction. (V2.5)

Furthermore, the optional table argument can also contain one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Please note that due to historical reasons the position that has to be passed to this function in the first two arguments is really the top-left corner of the elliptical arc's bounding rectangle. This might be confusing since traditionally elliptical arcs are drawn relative to their center point. Due to a design mistake in Hollywood 1.0, however, Hollywood unfortunately deviates from this standard.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x` x offset for the ellipse

`y`	y offset for the ellipse
`xradius`	x radius of your ellipse
`yradius`	y radius of your ellipse
`start`	start angle in degrees (must be positive)
`end`	end angle in degrees (must be positive)
`color`	optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel
`table`	optional: table containing further arguments; can be any from the ones listed above and from the standard tags (V4.5)

EXAMPLE

```
Arc(0, 0, 183, 183, 45, 315, #WHITE)
Circle(164, 33, 16, #BLACK)
```

Draws a pac-man shape.

27.2 Box

NAME

Box – draw a rectangle

SYNOPSIS

```
Box(x, y, width, height[[, color], table])
```

FUNCTION

This function draws a rectangle at the position specified by `x` and `y` in the dimensions specified by `width` and `height` using the specified color (RGB value). The rectangle will be drawn in the form style specified using `SetFormStyle()` and will be filled according to the configuration selected with `SetFillStyle()`.

If layers are enabled, this command will add a new layer of the type `#BOX` to the layer stack.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Starting with Hollywood 4.5 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the style of the box. The following options are possible:

`RoundLevel:`
: You can specify this tag to create a rectangle with rounded corners. You need to pass a round level in percentage which specifies how round the corners will be (possible values are 0 to 100). Defaults to 0 which means no round corners. (V1.9)

`CornerA, CornerB, CornerC, CornerD:`
: These four tags allow you to fine-tune the corner rounding of the rectangle. You can specify a rounding level (0 to 100) for every corner of the rectangle

thus allowing you to create a rectangle where not all corners are rounded, or where the different corners use different rounding levels. These tags will override any setting specified in the `RoundLevel` tag. (V5.0)

Furthermore, the optional table argument can also contain one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x`	destination x offset
`y`	destination y offset
`width`	desired width
`height`	desired height
`color`	optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel
`table`	optional: table containing further arguments; can be any from the ones listed above and from the standard tags (V4.5)

EXAMPLE

```
Box(0, 0, 640, 480, #YELLOW)
```

The above code draws a yellow box as a border of the 640x480 display.

```
Box(0, 0, 300, 200, #RED, {RoundLevel = 25})
```

The code above draws a red box with corners rounded by 25%.

27.3 Circle

NAME

Circle – draw a circle

SYNOPSIS

```
Circle(x, y, radius[[, color], table])
```

FUNCTION

This function draws a circle at the position specified by `x` and `y` using the specified radius and color (RGB value). The circle will be drawn in the form style specified using `SetFormStyle()` and will be filled according to the configuration selected with `SetFillStyle()`.

The width and height of circle will be `radius` * 2 + 1 (center point).

If layers are enabled, this command will add a new layer of the type `#CIRCLE` to the layer stack.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Starting with Hollywood 4.5, this function accepts a new optional table argument that can be used to specify one or more of the standard tags for all of Hollywood's draw commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Please note that due to historical reasons the position that has to be passed to this function in the first two arguments is really the top-left corner of the circle's bounding rectangle. This might be confusing since traditionally circles are drawn relative to their center point. Due to a design mistake in Hollywood 1.0, however, Hollywood unfortunately deviates from this standard.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x`	x offset
`y`	y offset
`radius`	radius of your circle
`color`	optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel
`table`	optional: table containing further arguments; can be any from the ones listed above and from the standard tags (V4.5)

27.4 Cls

NAME

Cls – clear screen (V2.0)

SYNOPSIS

```
Cls([color])
```

FUNCTION

This function quickly clears the current output device. Its behaviour depends on what is currently selected as the output device.

If the output device is the display, the current background picture will be restored and all layers and sprites will be removed. The color argument is not used in this case.

If the output device is a brush, it will be cleared with the specified color.

If the output device is a background picture, i.e. `SelectBGPic()` with mode set to `#SELMODE_LAYERS` is active, all layers of that picture will be removed. The color argument is not used in this case.

If the output device is a mask, `Cls()` will clear the pixels using the mode that was installed by `SetMaskMode()`. The color argument is not used in this case.

If the output device is an alpha channel, `Cls()` will fill the pixels with the intensity specified using `SetAlphaIntensity()`. The color argument is not used in this case.

Note that when the current draw target is palette-based and the palette mode is set to `#PALETTEMODE_PEN`, this function will clear the screen using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`color` optional: RGB color (defaults to `#BLACK`); only required when used while `SelectBrush()` is active

27.5 Directional constants

The directional constants are used by the `SetFontStyle()` and `SetFormStyle()` commands to configure the direction of the shadow of the text or graphics object. There are eight directional constants available and they correspond to the points of the compass:

```
#SHDWNORTH
#SHDWNORTHEAST
#SHDWEAST
#SHDWSOUTHEAST
#SHDWSOUTH
#SHDWSOUTHWEST
#SHDWWEST
#SHDWNORTHWEST
```

27.6 Ellipse

NAME

Ellipse – draw an ellipse

SYNOPSIS

```
Ellipse(x, y, xradius, yradius[[, color], table])
```

FUNCTION

This function draws an ellipse at the position specified by `x` and `y` using the specified radii and color (RGB value). The ellipse will be drawn in the form style specified using `SetFormStyle()` and will be filled according to the configuration selected with `SetFillStyle()`.

The width of the ellipse will be `xradius` * 2 + 1 (center point) and the height will be `yradius` * 2 + 1 (center point).

If layers are enabled, this command will add a new layer of the type `#ELLIPSE` to the layer stack.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Starting with Hollywood 4.5, this function accepts a new optional table argument that can be used to specify one or more of the standard tags for all of Hollywood's draw commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Please note that due to historical reasons the position that has to be passed to this function in the first two arguments is really the top-left corner of the ellipse's bounding

rectangle. This might be confusing since traditionally ellipses are drawn relative to their center point. Due to a design mistake in Hollywood 1.0, however, Hollywood unfortunately deviates from this standard.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x` x offset

`y` y offset

`xradius` x radius of your ellipse

`yradius` y radius of your ellipse

`color` optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel

`table` optional: table containing further arguments; can be any from the ones listed above and from the standard tags (V4.5)

27.7 GetFillStyle

NAME

GetFillStyle – get current fill style (V7.1)

SYNOPSIS

```
style[, t] = GetFillStyle()
```

FUNCTION

This function returns the current fill style set using `SetFillStyle()`. The return value `style` is set to either `#FILLNONE`, `#FILLCOLOR`, `#FILLGRADIENT`, or `#FILLTEXTURE`. See Section 27.14 [SetFillStyle], page 498, for details.

If `style` is `#FILLNONE`, `GetFillStyle()` will also return a table with the following fields:

`Thickness:`
The outline thickness in pixels.

If `style` is `#FILLGRADIENT`, the return table will contain the following fields:

`Type:` Gradient type. This will be either `#LINEAR`, `#CONICAL`, or `#RADIAL`.

`StartColor:`
Gradient start color.

`EndColor:`
Gradient end color.

`Angle:` Gradient angle. Only supported for types `#LINEAR` and `#CONICAL`.

`CenterX, CenterY:`
Gradient center point. Only supported for types `#RADIAL` and `#CONICAL`.

`Balance:` Gradient balance point. Only supported for `#CONICAL`.

`Border:` Gradient border. Only supported for `#RADIAL`.

`Colors:` If the gradient uses more than two colors, this field will contain a table with all those colors and their stop points.

See Section 20.6 [CreateGradientBGPic], page 232, for more information on gradients.

If `style` is `#FILLTEXTURE`, the return table will contain the following fields:

`Brush:` Identifier of the brush used for texturing.

`X, Y:` Pixel offset inside the brush to use as a starting offset for texturing.

See Section 27.14 [SetFillStyle], page 498, for more information on fill styles.

INPUTS

none

RESULTS

`style` the current fill style

`t` optional: table containing additional style information (see above)

27.8 GetFormStyle

NAME

GetFormStyle – get current form style (V7.1)

SYNOPSIS

```
style[, t] = GetFormStyle()
```

FUNCTION

This function returns the current form style set using `SetFormStyle()`. The return value `style` is set to a combination of the flags `#ANTIALIAS`, `#SHADOW`, and `#BORDER`. See Section 27.15 [SetFormStyle], page 499, for details.

If `#SHADOW` is set, `GetFormStyle()` also returns a table as the second return value which contains the following fields:

`ShadowColor:`
The shadow color.

`ShadowSize:`
The distance of the shadow from the main shape in pixels.

`ShadowDir:`
The direction of the shadow. This will be one of the directional constants.

If `#BORDER` is set, the return table will contain the following fields:

`BorderColor:`
The color of the border.

`BorderSize:`
The thickness of the border in pixels

See Section 27.15 [SetFormStyle], page 499, for more information on form styles.

INPUTS
none

RESULTS

`style` a combination of form style flags

`t` optional: table containing additional style information (see above)

27.9 GetLineWidth

NAME
GetLineWidth – get current outline thickness (V7.1)

SYNOPSIS
```
t = GetLineWidth()
```

FUNCTION
This function returns the current outline thickness set by `SetLineWidth()` or `SetFillStyle()`. See Section 27.16 [SetLineWidth], page 501, for details.

INPUTS
none

RESULTS

`t` current outline thickness

27.10 Line

NAME
Line – draw a line

SYNOPSIS
```
Line(x1, y1, x2, y2[[, color], table])
```

FUNCTION
This function draws a line from the point defined by `x1` and `y1` to the point defined by `x2` and `y2` in the specified color (RGB value).

If you want to have anti-aliased lines, use the `SetFormStyle()` command to enable anti-aliased drawing.

If layers are enabled, this command will add a new layer of the type `#LINE` to the layer stack.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Starting with Hollywood 4.5 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The

optional table argument can be used to configure the style of the line. The following options are possible:

`Thickness:`
: This tag allows you to specify the thickness of the line you want to draw. The minimum acceptable value here is 1 which will draw a normal line. This is also the default if you do not specify this tag. (V2.5)

`Arrowhead:`
: This tag allows you to turn the line into an arrow. It can be set to one of the following tags:

  `#ARROWHEAD_NONE:`
  : No arrowhead. This is the default mode.

  `#ARROWHEAD_SINGLE:`
  : Add arrowhead to end of line.

  `#ARROWHEAD_DOUBLE:`
  : Add arrowhead to start and end of line.

  Defaults to `#ARROWHEAD_NONE`. (V9.1)

Furthermore, the optional table argument can also contain one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x1` source x offset

`y1` source y offset

`x2` destination x offset

`y2` destination y offset

`color` optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel

`table` optional: table containing further arguments; can be any from the ones listed above and from the standard tags (V4.5)

EXAMPLE

```
Line(0, 0, 639, 479, #WHITE)
Line(639, 0, 0, 479, #WHITE)
```

The above code draws a white cross on the display.

27.11 Plot

NAME

Plot – draw a pixel

SYNOPSIS

```
Plot(x, y[, color])
```

FUNCTION

This function draws a single pixel to the display in the specified color. `Plot()` works only with disabled layers. If you want to have a 1x1 sized layer, you can use `Box()` to achieve this result.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x`	x offset
`y`	y offset
`color`	optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel

EXAMPLE

```
Plot(#CENTER, #CENTER, #RED)
```

Plot a red pixel in the middle of your display.

27.12 Polygon

NAME

Polygon – draw a polygon (V1.9)

SYNOPSIS

```
Polygon(x, y, vertices, count[[, color], table])
```

FUNCTION

This function draws a polygon from a table of vertices. You have to pass a table of x,y points that are used to draw the polygon in the `vertices` parameter. The `count` parameter specifies how many vertices your polygon has. Additionally, you need to specify the color for the polygon (in RGB format). The polygon will be drawn in the form style specified using `SetFormStyle()` and will be filled according to the configuration selected with `SetFillStyle()`. Also remember that the last vertex should close your polygon, which means that it must be the same as the first vertex.

The vertices can also be negative and/or specified in floating point notation to achieve the best precision. Hollywood will not round them off before it does the final rasterization.

If layers are enabled, this command will add a new layer of the type `#POLYGON` to the layer stack.

New in Hollywood 2.0: Color can also be an ARGB value for alpha-blended drawing.

Starting with Hollywood 4.5, this function accepts a new optional table argument that can be used to specify one or more of the standard tags for all of Hollywood's draw commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`x`	x offset for the polygon on the display
`y`	y offset for the polygon on the display
`vertices`	a table of vertices that describe the polygon
`count`	number of vertices in the table
`color`	optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you render to a mask or alpha channel
`table`	optional: table containing further arguments; can be any from the ones listed above and from the standard tags (V4.5)

EXAMPLE

```
v = {}
v[0] = 0    ;X1
v[1] = 100  ;Y1
v[2] = 50   ;X2
v[3] = 0    ;Y2
v[4] = 100  ;X3
v[5] = 100  ;Y3
v[6] = 0    ;X4
v[7] = 100  ;Y4
Polygon(#CENTER, #CENTER, v, 4, #RED)
```

The above code draws a red triangle in the center of the display.

```
Polygon(#CENTER, #CENTER, v, 4, #RED, {Rotate = 45})
```

The above code draws a triangle rotated by 45 degrees.

27.13 ReadPixel

NAME

ReadPixel – read a pixel from output device (V1.5)

SYNOPSIS

```
col = ReadPixel(x, y)
```

FUNCTION

This function reads a pixel from the position specified by `x` and `y` from the current output device and returns its color.

Please note: The color returned by this function can slightly vary from the original color at this position. For example, if you have a brush which is completely white (i.e. all pixels have the color $FFFFFF). If you use `ReadPixel()` now to read the color of an arbitrary pixel from this brush, you will receive the color $FFFFFF only if Hollywood runs on a 24-bit or 32-bit screen. If Hollywood runs on a 16-bit screen, you will get the color $FCF8FC, on a 15-bit screen you would receive $F8F8F8. This is because those screens do not have 16.7 million colors but only 65536 (16-bit screens) or 32768 (15-bit screens) respectively. You can use the function `GetRealColor()` to find out which color represents the specified color on the current screen.

If the current output device is the mask of a brush, then `ReadPixel()` will return either 0 if the pixel is invisible or 1 if the pixel is visible.

If the current output device is an alpha channel of a brush, then `ReadPixel()` will return the transparency level of the requested pixel which ranges from 0 (full transparency) to 255 (no transparency).

INPUTS

`x`	x coordinate of the pixel
`y`	y coordinate of the pixel

RESULTS

`col`	color or transparency setting of the pixel at the specified position

EXAMPLE

```
CreateBrush(1, 320, 256)
SelectBrush(1)
SetFillStyle(#FILLCOLOR)
Box(0, 0, 320, 256, #GREEN)
a = ReadPixel(100, 100)
EndSelect()
```

The above code draws a green rectangle to brush 1 and reads the pixel at 100:100 from it. The variable a will receive the value of `#GREEN` because the whole brush is filled with green.

27.14 SetFillStyle

NAME

SetFillStyle – set filling style for draw commands (V1.5)

SYNOPSIS

```
SetFillStyle(style)
SetFillStyle(#FILLGRADIENT, type, startcol, endcol[, angle, t]) (V2.0)
SetFillStyle(#FILLTEXTURE, brush_id[, x, y]) (V2.0)
SetFillStyle(#FILLNONE[, thickness]) (V2.0)
```

FUNCTION

This function allows you to define the filling style for the `Arc()`, `Box()`, `Circle()`, `Ellipse()` and `Polygon()` commands. By default, the filling style is set to `#FILLNONE`, which means that just the outlines are drawn.

Currently the following styles are supported:

`#FILLCOLOR:`
: Fill objects with color

`#FILLNONE:`
: Do not fill object, just draw the outline. Starting with Hollywood 2.0, you can also specify a line thickness. The default for `thickness` is 1 which means an outline thickness of a single pixel. Note that `#FILLNONE` only supports the `#SHADOW` and `#BORDER` form styles when layers are enabled. Otherwise no shadow or border will be drawn.

`#FILLGRADIENT:`
: Fills objects with a gradient. You have to specify three more arguments which set the type of the gradient as well as its colors; additionally you can specify the optional argument `angle` which will rotate the gradient, and there is an optional table argument for even more options; See Section 20.6 [CreateGradientBGPic], page 232, for details. (V2.0)

`#FILLTEXTURE:`
: Fills objects with a texture. You have to specify an additional argument which specifies the identifier of the brush that shall be used for texturing. Please note that any transparency channels that the brush may have (mask or alpha channel) are currently not supported by texturing. The optional `x` and `y` parameters are new in Hollywood 4.6. They allow you to specify an offset into the texture brush. Texturing will then start from this offset in the brush. The default for these arguments is 0/0 which means start at the top-left corner inside the texture brush. (V2.0)

INPUTS

`style` a style id as listed above

`...` additional arguments depend on the chosen style

EXAMPLE

```
SetFillStyle(#FILLCOLOR)
Box(0, 0, 320, 256, #RED)
```

Draw a filled red rectangle at 0:0 with a dimension of 320x256.

27.15 SetFormStyle

NAME

SetFormStyle – set the form style for graphics primitives (V2.5)

SYNOPSIS

```
SetFormStyle(style[, t])
```

DEPRECATED SYNTAX

```
SetFormStyle(#SHADOW, color, distance, direction)
SetFormStyle(#BORDER, color, size)
```

FUNCTION

This function can be used to configure the drawing style for the commands of the graphics primitives library. This function affects the look of the `Arc()`, `Box()`, `Circle()`, `Ellipse()`, `Line()`, and `Polygon()` commands.

The style must be one of the following predefined constants:

`#NORMAL` This resets the form style to the default style.

`#ANTIALIAS`
: This sets the form style to anti-alias; the graphics primitives will be drawn with anti-aliasing.

`#SHADOW` This sets the form style to shadow. The forms will be drawn with a shadow with this style. The `color` argument specifies the shadow color. This color can be either in RGB or ARGB notation. Shadow transparency is fully supported. The `distance` argument specifies the distance of the shadow from the main form in pixels. The `direction` argument specifies the direction of the shadow. This must be one of the directional constants. Please note that `#SHADOW` is not supported for the fill style `#FILLNONE` if layers are disabled.

`#BORDER` This sets the form style to bordered. A border of the specified size will be drawn around the form with this style. The `color` argument specifies the color for the border. This color can either be in RGB or ARGB notation. Border transparency is fully supported. The `size` argument specifies the desired thickness of the border in pixels. Please note that `#BORDER` is not supported for the fill style `#FILLNONE` if layers are disabled. Before Hollywood 9.0, this style was called `#EDGE`.

To combine multiple form styles in a single call simply bit-or them with another, e.g. a call to `SetFormStyle(#SHADOW|#BORDER)` will enable the shadow and border form styles. Obviously, the style `#NORMAL` is mutually exclusive and cannot be combined with any other style.

Starting with Hollywood 9.0, `SetFormStyle()` uses a new syntax that accepts an optional table argument that supports the following tags:

`ShadowDir:`
: Specifies the direction of the shadow. This must be set to one of Hollywood's directional constants. This tag is only handled when the `#SHADOW` style has been set (see above). (V9.0)

`ShadowColor:`
: Specifies the color of the shadow. This must be an ARGB value that can contain a transparency setting. This tag is only handled when the `#SHADOW` style has been set (see above). (V9.0)

`ShadowSize:`
: Specifies the length of the shadow. This tag is only handled when the `#SHADOW` style has been set (see above). (V9.0)

`BorderColor:`
: Specifies the color of the border. This must be an ARGB value that can contain a transparency setting. This tag is only handled when the `#BORDER` style has been set (see above). (V9.0)

`BorderSize:`
: Specifies the size of the border. This tag is only handled when the `#BORDER` style has been set (see above). (V9.0)

Please note that the `Line()` command does not support the form styles `#SHADOW` and `#BORDER`. It only recognizes the style `#ANTIALIAS`.

INPUTS

`style` special style constant (see list above)

`t` optional: table argument containing further options (see above) (V9.0)

EXAMPLE

```
SetFormStyle(#ANTIALIAS)
```

The function call above enables anti-aliased forms.

```
SetFormStyle(#SHADOW, {ShadowColor = ARGB(128, $939393),
     ShadowSize = 16, ShadowDir = #SHDWSOUTHEAST})
```

The above code enables a half-transparent grey shadow which will be positioned 16 pixels to the south-east of the main form.

27.16 SetLineWidth

NAME

SetLineWidth – set width for outline drawing (V5.0)

SYNOPSIS

```
SetLineWidth(width)
```

FUNCTION

This command can be used to change the line thickness when the fill style is set to `#FILLNONE`. You can also change the line thickness by calling `SetFillStyle()` with fill style set to `#FILLNONE` but using `SetLineWidth()` makes the code more readable.

INPUTS

`width` desired line thickness for outline drawing

27.17 Standard draw tags

Starting with Hollywood 4.0, most of the drawing functions accept an optional table argument now, which lets you configure further options. For example, you can specify tags which will automatically scale or rotate the graphics before displaying it.

Many of the standard drawing tags only work when layers are enabled. Some, however, can also be used when layers are off. Note that the graphics data of the source object will never

be changed. If layers are on Hollywood will insert the new layer and immediately apply any transformation before the layer is made visible. If layers are off a copy of the original graphics will be made. The original graphics data will never be changed (in contrast to functions like `ScaleBrush()`, `RotateBrush()`, `TransformBrush()` etc. which modify the brushes data). Thus, you can be sure that you will always get the best quality when using the standard draw tags to apply transformations, because Hollywood will always use the original graphics data.

For every standard draw tag there is a default value that is used when the tag is not specified. You can modify this default setting using the new `SetDrawTagsDefault()` command which allows you to define new default values for each tag. This is very useful if you would like to permanently use a different default value for a certain tag; for example, if you permanently want to use an anchor point of 0.5/0.5 instead of 0.0/0.0, or if you want to change the default layer insert position from frontmost to backmost etc.

The following standard tags are currently defined:

`Width, Height:`
: If you specify these tags, the object will be scaled to the specified dimension and then displayed. (V4.0)

`Rotate:`
: This tag will rotate the graphics by the specified degrees and then display it. (V4.0)

`SmoothScale:`
: If this is set to `True`, scaling and/or rotation will be done using anti-aliased interpolation. This looks better but is a lot slower. (V4.0)

`ScaleX, ScaleY:`
: This is an alternative way of scaling the object. You have to pass a floating point value here that indicates a scaling factor. For example, 0.5 means half the size, 2.0 means twice the size. This is especially convenient if you would like to keep the proportions of the object that you want to scale. If you use the same factor for `ScaleX` and `ScaleY`, the proportions of the graphics will remain intact. Please note that `ScaleX/Y` and `Width/Height` are mutually exclusive. You must not mix both groups. Either use `ScaleX/Y`, or stick to `Width/Height`. (V4.5)

`Transform:`
: This tag allows you to specify a 2x2 transformation matrix. Transformation matrices are useful if you want to apply scaling and rotation at the same time, or if you want to mirror an object. You have to pass a table to `Transform`. The table must contain the four constituents of a 2x2 transformation matrix in the following order: sx, rx, ry, sy. See Section 21.78 [TransformBrush], page 314, for more information about transformation matrices. Please note that the `Transform` tag is mutually exclusive with the following tags: Width/Height/ScaleX/ScaleY/Rotate. You must not combine it with any of these tags. (V4.5)

`AnchorX, AnchorY:`
: You can use these two tags to specify the anchor point of the graphics object (sometimes the anchor point is referred to as the 'hot spot'). The anchor point

can be any point between 0.0/0.0 (top left corner of the graphics object) and 1.0/1.0 (bottom right corner of the graphics object). The center of the graphics object would be defined by anchor point of 0.5/0.5. All transformations (scaling, rotation etc.) will be applied relative to the anchor point. Also, the position of an object is always relative to the anchor point. For example, take a look at the following code:

```
DisplayBrush(1, 0, 0, {AnchorX = 0.5, AnchorY = 0.5})
```

This call will make the brush's center appear at 0:0 because the anchor point is set to 0.5/0.5. If you want the brush's top left corner to appear at 0:0, you will have to use an anchor point of 0.0/0.0 (which is also the default anchor point, so you do not have to specify it at all). Anchor points are most of the time only used when layers are enabled. However, you can also use `AnchorX/Y` if layers are off. See Section 34.40 [SetLayerAnchor], page 678, for some more information about the anchor point concept. (V4.5)

`Hidden:` Allows you to create a layer that is initially hidden. If you set this tag to `True`, the graphics will be inserted as a layer but nothing will be shown because the layer is kept hidden. This function is only available with enabled layers. Defaults to `False`. (V4.5)

`InsertPos:`
Allows you to specify the insert position for the new layer. The layer will be inserted at this position with all other layers being shifted down in the hierarchy. The first layer is at position 1. Specifying a position of 0 here will insert the layer as the last layer. This is also the default setting. You can also specify the name of a layer at whose position the new layer should be inserted. This is perfectly allowed. Of course, this function only works with enabled layers. (V4.5)

`Name:` This tag can be used to assign a name to the layer right at creation time. This is pretty much the same as calling the `SetLayerName()` function on the layer right after inserting it. Specifying the `Name` tag here just saves you some typing and makes the code more readable. This tag is only handled when layers are enabled. By default, no names are given to new layers. By default, they can only be referred to using IDs. (V4.5)

`Group:` If layers are enabled, you can use this tag to attach the layer to a group right at creation time. This is pretty much the same as calling the `GroupLayer()` function on the layer right after creating it. Specifying the `Group` tag here just saves you some typing and makes the code more readable. This tag is only handled when layers are enabled. See Section 34.16 [GroupLayer], page 660, for details. (V10.0)

`Transparency:`
You can use this tag to specify a global transparency setting for this graphics object. This can be a value ranging from 0 (no transparency) to 255 (full transparency), or alternatively, a string containing a percentage (e.g. `"50%"` for half shine through transparency). Defaults to 0 which means no global transparency. (V4.5)

`GlobalTransparency:`
: This is only supported when layers are enabled. If it is set to `True`, the layer's transparency setting set using the `Transparency` tag or using functions like `SetLayerTransparency()` will also be applied to the layer's shadow. For some reason Hollywood never did that by default and in order to maintain compatibility with scripts which expect the old behaviour the functionality has been added using a new tag. If you want to globally enable the new behaviour for all your layers, just call `SetDrawTagsDefault()` with `GlobalTransparency` set to `True`. (V9.1)

`Tint:`
: You can use this tag to specify a global color tinting setting for this graphics object. This can be a value ranging from 0 (no tinting) to 255 (full color tinting), or alternatively, a string containing a percentage (e.g. `"50%"` for medium color tinting). If you set this tag to anything else than 0, you must also provide a tint color in the `TintColor` tag (see below). Defaults to 0 which means no tinting. Starting with Hollywood 5.0 this tag is directly mapped to the `Tint` filter. Thus, specifying the `Tint` tag is the same as specifying a `Tint` filter in the `Filters` table below. (V4.5)

`TintColor:`
: Only required if you also specify `Tint`. In that case, you have to specify a RGB color color here that will be used for tinting. (V4.5)

`Shadow:`
: If you set this tag to `True`, the graphics object will be drawn with a drop shadow. You can configure the look of the shadow using the `ShadowDir`, `ShadowSize`, `ShadowColor`, and `ShadowRadius` tags. See below for more information. This tag is only supported when layers are turned on. See Section 34.47 [SetLayerShadow], page 688, for details. (V5.0)

`ShadowDir:`
: Specifies the direction of the shadow. This must be set to one of Hollywood's directional constants. This tag is only handled when `Shadow` is set to `True` (see above). (V5.0)

`ShadowColor:`
: Specifies the color of the shadow. This must be an ARGB value that can contain a transparency setting. This tag is only handled when `Shadow` is set to `True` (see above). (V5.0)

`ShadowPen:`
: When palette mode is set to `#PALETTEMODE_PEN` and the drawing target uses a palette, the shadow will be drawn using the pen specified here instead of the color specified in the `ShadowColor` tag from above. (V9.0)

`ShadowSize:`
: Specifies the size of the shadow. This tag is only handled when `Shadow` is set to `True` (see above). (V5.0)

`ShadowRadius:`
: Specifies the shadow radius. This tag is only handled when `Shadow` is set to `True` (see above). (V5.0)

`Border:` This tag can be used to draw the graphics object with a border frame. You can configure the look of the border using the `BorderSize` and `BorderColor` tags. Please see below for more information. This tag is only supported when layers are turned on. See Section 34.41 [SetLayerBorder], page 679, for details. (V5.0)

`BorderColor:`
Specifies the color of the border. This must be an ARGB value that can contain a transparency setting. This tag is only handled when `Border` is set to `True` (see above). (V5.0)

`BorderPen:`
When palette mode is set to `#PALETTEMODE_PEN` and the drawing target uses a palette, the border will be drawn using the pen specified here instead of the color specified in the `BorderColor` tag from above. (V9.0)

`BorderSize:`
Specifies the size of the border. This tag is only handled when `Border` is set to `True` (see above). (V5.0)

`Filters:` This tag can be used to apply filters to this graphics object. You have to pass a table to this tag that describes the desired configuration of the single filters. See Section 34.43 [SetLayerFilter], page 681, for more information on how this table needs to be organized. Note that although the documentation points to the layers library, the `Filters` tag actually also works when layers are disabled. (V5.0)

28 Error management library

28.1 ERROR

NAME

ERROR – abort compilation with an error message (V6.1)

SYNOPSIS

```
@ERROR msg$
```

FUNCTION

Whenever the preprocessor reaches this command, compilation will be immediately aborted and the specified error message will be shown. This is only useful for debugging purposes.

INPUTS

`msg$` error message to show

28.2 Error

NAME

Error – exit with custom error message (V2.0)

SYNOPSIS

```
Error(msg$)
```

FUNCTION

This function terminates the script and displays the message specified in `msg$`. This is useful if your script shall be aborted because an unexpected condition occurred.

INPUTS

`msg$` error message to show

EXAMPLE

```
If Exists("Game.dat") = False Then Error("Cannot read game data!")
```

The code above checks for the file "Game.dat" and exits with an error message if it does not exist.

28.3 Error codes

By default, Hollywood will always exit when an error occurs. You can change this behaviour by using the `ExitOnError()` and `GetLastError()` functions, or by installing a custom error handler using `RaiseOnError()`. Both ways will supply you with an error code that indicates the error that has just happened. Here is a list of all error codes currently defined by Hollywood (some error codes are no longer used but have to be kept for compatibility reasons):

`#ERR_NONE`

No error (0)

`#ERR_MEM` Out of memory! (1000)

`#ERR_UNIMPLCMD`
: Unimplemented command! (1001)

`#ERR_NORETVAL`
: No return value specified! (1002)

`#ERR_USERABORT`
: User abort! (1003)

`#ERR_SCREEN`
: Error opening screen! (1004)

`#ERR_WRITE`
: Could not write all characters to file! Check if there is enough free space! (1005)

`#ERR_UNTERMINTDSTR`
: Unterminated string! (1006)

`#ERR_UNKNOWNCOND`
: Unknown condition! (1007)

`#ERR_MISSINGSEPARTR`
: Multiple commands in one line have to be separated by a colon! (1008)

`#ERR_READ`
: Could not read all characters from file! Check if it is read protected! (1009)

`#ERR_WINDOW`
: Unable to open window! (1010)

`#ERR_ELSEWOIF`
: ELSE without IF! (1011)

`#ERR_ENDIFWOIF`
: ENDIF without IF! (1012)

`#ERR_IFWOENDIF`
: IF without ENDIF! (1013)

`#ERR_MISSINGPARAMTR`
: Not enough arguments! (1014)

`#ERR_FORWONEXT`
: FOR without NEXT! (1015)

`#ERR_NEXTWOFOR`
: NEXT without FOR! (1016)

`#ERR_WHILEWOWEND`
: WHILE without WEND! (1017)

`#ERR_SYNTAXERROR`
: General syntax error! (1018)

`#ERR_WRONGDTYPE`
: Wrong data type specified! (1019)

`#ERR_VARSYNTAX`
: Syntax error in variable name! (1020)

`#ERR_WENDWOWHILE`
: WEND without WHILE! (1021)

`#ERR_UNKNOWNCMD`
: Unknown command %s ! (1022)

`#ERR_MISSINGBRACKET`
: You specified too many arguments or forgot a close bracket! (1023)

`#ERR_VALUEEXPECTED`
: Value expected! (1024)

`#ERR_OPENLIB`
: Cannot open %s ! (1025)

`#ERR_VAREXPECTED`
: Variable expected! (1026)

`#ERR_LABINFOR`
: Labels within a For() loop are not allowed! (1027)

`#ERR_LABINIF`
: Labels inside If() conditions are not allowed! (1028)

`#ERR_LABINWHILE`
: Labels within a While() loop are not allowed! (1029)

`#ERR_WRONGOP`
: Wrong operator for this type! (1030)

`#ERR_GETDISKOBJ`
: Cannot open icon! (1031)

`#ERR_EVNTEXPCTED`
: You need to specify a Hollywood event! (1032)

`#ERR_EMPTYOBJ`
: Cannot create empty text objects! (1033)

`#ERR_EMPTYSCRIPT`
: Your script is empty! (1034)

`#ERR_COMMENTSTRUCT`
: Incoherent comment structure! (1035)

`#ERR_ALRDYDECLRD`
: This variable was already used and initialized! (1036)

`#ERR_WRONGFLOAT`
: Illegal float number format! (1037)

`#ERR_REQUIREFIELD`
: You need to specify an array field! (1038)

`#ERR_OUTOFRANGE`
: Specified array field is out of range! (1039)

`#ERR_RETWOGOSUB`
: Return without Gosub! (1040)

`#ERR_FINDARRAY`
: Requested object not found! (1041)

`#ERR_FINDCST`
: Constant not found! (1042)

`#ERR_LOCK`
: Error locking directory! (1043)

`#ERR_LOADPICTURE`
: Cannot load picture %s! Make sure that you have a datatype for this format! (1044)

`#ERR_READFILE`
: Cannot read file %s ! (1045)

`#ERR_NOTPROTRACKER`
: Module is not in Protracker format! (1046)

`#ERR_UNKNOWNSEQ`
: Unknown sequence character after backslash! (1047)

`#ERR_DIRLOCK`
: Error locking directory %s ! (1048)

`#ERR_KEYWORD`
: Unknown keyword! (1049)

`#ERR_KICKSTART`
: You need at least Kickstart 3.0! (1050)

`#ERR_FREEABGPIC`
: You cannot free the background picture that is currently displayed! (1051)

`#ERR_WRITEFILE`
: Cannot write to file %s ! (1052)

`#ERR_VERSION`
: This script requires at least Hollywood %s ! (1053)

`#ERR_NOFUNCTION`
: This command does not return anything! (1054)

`#ERR_WRONGUSAGE`
: Wrong usage/parameters for this command! Read the documentation! (1055)

`#ERR_SELECTBG`
: This command cannot be used when SelectBGPic() is active! (1056)

`#ERR_ARRAYDECLA`
: Array "%s[]" was not declared! (1057)

`#ERR_CMDASVAR`
: This variable name is already used by a command! (1058)

`#ERR_CONFIG`
: No script filename specified! (1059)

`#ERR_ARGS`
: Wrong arguments specified! (1060)

`#ERR_DOUBLEDECLA`
: Double declaration! Number already assigned previously! (1061)

`#ERR_EQUALEXPECTED`
: Equal sign expected! (1062)

`#ERR_OPENANIM`
: Cannot load animation! Make sure that you got at least version 40 of animation.datatype and realtime.library! Please note that MorphOS does not have a datatype for IFF ANIM files currently. So if you want to use IFF ANIM files, you need to install a datatype first, e.g. the IFF ANIM datatype of OS3.1! (1063)

`#ERR_OPENFONT`
: Undefined (1064)

`#ERR_OPENSOUND`
: Cannot load sample %s ! Make sure you have a datatype for this sample format! If you tried to load a 16-bit sample and get this error, you need to install a sound.datatype replacement because the OS 3.x datatype does only support 8-bit samples. You can get the sound.datatype replacement from http://www.stephan-rupprecht.de/ or from the Hollywood CD-ROM. You do NOT need the replacement on MorphOS 1.x because that already uses a new sound.datatype which supports 16-bit samples! (1065)

`#ERR_INTERNAL`
: Internal limit encountered! Contact the author... (1066)

`#ERR_PUBSCREEN`
: Cannot find the specified public screen! (1067)

`#ERR_BRUSHLINK`
: This command cannot handle linked brushes! (1068)

`#ERR_WRONGID`
: Please use only positive integers for your objects! (1069)

`#ERR_AHI` General AHI error! Check your AHI installation and settings! (1070)

`#ERR_VARLENGTH`
: Variable length is limited to 64 characters! (1071)

`#ERR_LABELDECLA`
: Cannot find label "%s" ! (1072)

`#ERR_LABELDOUBLE`
: Label "%s" was already defined! (1073)

`#ERR_NOKEYWORDS`
: Keywords are not allowed here! (1074)

`#ERR_NOCONSTANTS`
: Constants are not allowed here! (1075)

`#ERR_SEEK`
: Invalid seek position specified! (1076)

`#ERR_CSTDOUBLEDEF`
: Constant #%s was already declared! (1077)

`#ERR_NOLAYERS`
: This function requires enabled layers! (1078)

`#ERR_LAYERSUPPORT`
: This layer is not supported by GetBrushLink()! (1079)

`#ERR_UNKNOWNATTR`
: Unknown attribute specified! (1080)

`#ERR_LAYERRANGE`
: Specified layer is out of range! (1081)

`#ERR_SELECTBRUSH`
: This command cannot be used when SelectBrush() is active! (1082)

`#ERR_POINTERFORMAT`
: Pointer image must be in 4 colors and not wider than 16 pixels! (1083)

`#ERR_CREATEDIR`
: Error creating directory %s ! (1084)

`#ERR_DISPLAYSIZE`
: Unable to change display size to %s ! (1085)

`#ERR_DISPLAYDESKTOP`
: You cannot specify an initial BGPic together with DISPLAYDESKTOP! (1086)

`#ERR_GUIGFX`
: Cannot open guigfx.library version 20! Make sure that you have at least version 20 installed! (1087)

`#ERR_RENDER`
: Cannot open render.library version 30! Make sure that you have at least version 30 installed! (1088)

`#ERR_ZERODIVISION`
: Division by zero! (1089)

`#ERR_WARPOS`
: You need at least WarpUP v5.1 for this program! (1090)

`#ERR_UNKNOWN`
: Unknown error code! (1091)

`#ERR_STRINGCST`
: You cannot specify string constants here! (1092)

`#ERR_LABINFUNC`
: Labels are not allowed in functions! (1093)

`#ERR_ANIMFRAME`
: Specified animation frame is out of range! (1094)

`#ERR_REPEATWOUNTIL`
: REPEAT without UNTIL! (1095)

`#ERR_UNTILWOREPEAT`
: UNTIL without REPEAT! (1096)

`#ERR_FUNCWOENDFUNC`
: FUNCTION without ENDFUNCTION! (1097)

`#ERR_ENDFUNCWOFUNC`
: ENDFUNCTION without FUNCTION! (1098)

`#ERR_UNEXPECTEDSYM`
: Unexpected symbol! (1099)

`#ERR_FUNCARGS`
: Variable, closing bracket or "..." expected! (1100)

`#ERR_NOLOOP`
: No loop to break! (1101)

`#ERR_TOKENEXPECTED`
: "%s" expected! (1102)

`#ERR_BRACKETOPEN`
: Opening bracket expected! (1103)

`#ERR_BRACKETCLOSE`
: Closing bracket expected! (1104)

`#ERR_BRACEOPEN`
: Opening brace expected! (1105)

`#ERR_BRACECLOSE`
: Closing brace expected! (1106)

`#ERR_SQBRACKETOPEN`
: Opening square bracket expected! (1107)

`#ERR_SQBRACKETCLOSE`
: Closing square bracket expected! (1108)

`#ERR_NOCOMMA`
: Comma expected! (1109)

`#ERR_SYNTAXLEVELS`
: Too many syntax levels! (1110)

`#ERR_COMPLEXWHILE`
: WHILE condition too complex! (1111)

`#ERR_NOCHAR`
: ASCII code specification is out of range! (1112)

`#ERR_CHRCSTLEN`
: Character constant too long! (1113)

`#ERR_CHRCSTEMPTY`
: Empty character constant not allowed! (1114)

`#ERR_HEXPOINT`
: Decimal point used in hexadecimal value! (1115)

`#ERR_NUMCONCAT`
: A space is necessary between number concatenations! (1116)

`#ERR_MAXLOCALS`
: Too many local variables! (1117)

`#ERR_MAXUPVALS`
: Too many upvalues! (1118)

`#ERR_MAXPARAMS`
: Too many parameters! (1119)

`#ERR_CONITEMS`
: Too many items in a constructor! (1120)

`#ERR_MAXLINES`
: Too many lines in a chunk! (1121)

`#ERR_COMPLEXEXPR`
: Expression or function too complex! (1122)

`#ERR_CTRLSTRUCT`
: Control structure too long! (1123)

`#ERR_NOCOLON`
: Colon expected! (1124)

`#ERR_CASECST`
: Case expression must be constant! (1125)

`#ERR_SWCHWOENDSWCH`
: SWITCH without ENDSWITCH! (1126)

`#ERR_ENDSWCHWOSWCH`
: ENDSWITCH without SWITCH! (1127)

`#ERR_BLKWOENDBLK`
: BLOCK without ENDBLOCK! (1128)

`#ERR_ENDBLKWOBLK`
: ENDBLOCK without BLOCK! (1129)

`#ERR_NUMSTRCMP`
: Attempt to compare a number with a string! (1130)

#ERR_CONCAT
Wrong data types for concatenation! (1131)

#ERR_TABLEDECLA
Table %s not found! (1132)

#ERR_FUNCDECLA
Function %s not found! (1133)

#ERR_INTERNAL1
Internal limit reached! Error code %s. (1134)

#ERR_STACK
Stack overflow! (1135)

#ERR_MEMCODE
Code size overflow! (1136)

#ERR_MEMCST
Constant table overflow! (1137)

#ERR_NUMEXPECTED
Number expected in argument %d! (1138)

#ERR_STREXPECTED
String expected in argument %d! (1139)

#ERR_TABEXPECTED
Table expected in argument %d! (1140)

#ERR_READONLY
File was opened in read-only mode! (1141)

#ERR_WRITEONLY
File was opened in write-only mode! (1142)

#ERR_DELETEFILE
Could not delete file! (1143)

#ERR_EXAMINE
Could not examine %s! (1144)

#ERR_RENAME
Could not rename file! (1145)

#ERR_MEMRANGE
Specified offset is out of range! (1146)

#ERR_SELECTMASK
This command cannot be used when SelectMask() is active! (1147)

#ERR_MODIFYABG
Attempt to modify the active background picture! (1148)

#ERR_MODIFYABR
Attempt to modify the active brush! (1149)

`#ERR_FUNCJMP`
: You cannot use GOTO/GOSUB inside functions! (1150)

`#ERR_REVDWORD`
: You cannot use this reserved word here! (1151)

`#ERR_LOCKBMAP`
: Could not lock bitmap! (1152)

`#ERR_PALSCREEN`
: Hollywood does not run on palette screens! Please switch to a high or true color mode! (1153)

`#ERR_NEGCOORDS`
: Negative coordinates are not allowed here! (1154)

`#ERR_NOANMLAYER`
: Specified layer is not an anim layer! (1155)

`#ERR_BRUSHSIZE`
: Brush size does not match specified arguments! (1156)

`#ERR_ENDWITHWOWITH`
: ENDWITH without WITH! (1157)

`#ERR_WITHWOENDWITH`
: WITH without ENDWITH! (1158)

`#ERR_FIELDINIT`
: Table field %s was not initialized! (1159)

`#ERR_LABMAINBLK`
: Labels are only allowed in the main script block! (1160)

`#ERR_NAMEUSED`
: Layer name was already assigned! (1161)

`#ERR_LAYERSOFF`
: Layers must be turned off when using this function! (1162)

`#ERR_LAYERSON`
: Layers cannot be turned on/off while off-screen rendering is active! (1163)

`#ERR_NOLOOPCONT`
: No loop to continue! (1164)

`#ERR_LOOPRANGE`
: Loop number is out of range! (1165)

`#ERR_INTEXPECTED`
: Integer value expected! (1166)

`#ERR_SELECTALPHACHANNEL`
: This command cannot be used when SelectAlphaChannel() is active! (1167)

`#ERR_PIXELRANGE`
: Specified pixel is out of range! (1168)

`#ERR_DATATYPEALPHA`
: Your picture.datatype does not support alpha channel! (1169)

`#ERR_NOALPHA`
: Image "%s" does not have an alpha channel! (1170)

`#ERR_PIXELFORMAT`
: Unknown pixel format detected! Hollywood cannot run on this screen! (1171)

`#ERR_NOMASKBRUSH`
: This brush does not have a mask! (1172)

`#ERR_FOREVERWOREPEAT`
: FOREVER without REPEAT! (1173)

`#ERR_FINDBRUSH`
: Could not find brush %d! (1174)

`#ERR_FINDTEXTOBJECT`
: Could not find text object %d! (1175)

`#ERR_FINDANIM`
: Could not find anim %d! (1176)

`#ERR_FINDBGPIC`
: Could not find BGPic %d! (1177)

`#ERR_FINDSAMPLE`
: Could not find sample %d! (1178)

`#ERR_FINDFILE`
: Could not find file handle %d! (1179)

`#ERR_FINDMEMBLK`
: Could not find memory block %d! (1180)

`#ERR_FINDTIMER`
: Could not find timer %d! (1181)

`#ERR_FINDMOVE`
: Could not find move queue %d! (1182)

`#ERR_STRORNUM`
: String or number expected in argument %d! (1183)

`#ERR_PERCENTFORMAT`
: Invalid percent format in argument %d! (1184)

`#ERR_FUNCEXPECTED`
: Function expected in argument %d! (1185)

`#ERR_UNMPARENTHESES`
: Unmatched parentheses! (1186)

`#ERR_WRONGOPCST`
: This operator cannot be used here! (1187)

`#ERR_FINDBUTTON`
: Could not find button %d! (1188)

`#ERR_NUMTABLEARG`
: Number expected in table argument "%s"! (1189)

`#ERR_NUMCALLBACK`
: Callback function was expected to return a number! (1190)

`#ERR_BGPICBUTTON`
: A BGPic needs to be active while calling button functions! (1191)

`#ERR_WRONGHEX`
: Invalid hexadecimal specification! (1192)

`#ERR_TOOMANYARGS`
: Too many arguments for this function! (1193)

`#ERR_FINDINTERVAL`
: Could not find interval function %d! (1194)

`#ERR_FINDTIMEOUT`
: Could not find timeout function %d! (1195)

`#ERR_LOADSOUND`
: Error loading sample to sound card! (1196)

`#ERR_STRINGEXPECTED`
: String expected! (1197)

`#ERR_UNEXPECTEDEOF`
: Unexpected end of file! (1198)

`#ERR_VMMISMATCH`
: Virtual machine data type mismatch! (1199)

`#ERR_BADINTEGER`
: Bad integer in bytecode! (1200)

`#ERR_BADUPVALUES`
: Bad upvalues in bytecode! (1201)

`#ERR_BADCONSTANT`
: Bad constant type in bytecode! (1202)

`#ERR_BADBYTECODE`
: Bad bytecode! (1203)

`#ERR_BADSIGNATURE`
: Bad bytecode signature! (1204)

`#ERR_UNKNUMFMT`
: Unknown number format in bytecode! (1205)

`#ERR_INVNEXTKEY`
: Invalid key for next table item! (1206)

`#ERR_TABLEOVERFLOW`
: Table overflow! (1207)

`#ERR_TABLEINDEX`
: Table index is NaN! (1208)

`#ERR_APPLETVERSION`
: This applet requires at least Hollywood %s! (1209)

`#ERR_UNKNOWNSEC`
: Unknown section in applet! (1210)

`#ERR_NOAPPLET`
: %s is no Hollywood applet! (1211)

`#ERR_PLAYERCOMP`
: Compilation is not possible with HollywoodPlayer! (1212)

`#ERR_FILEEXIST`
: File %s does not exist! (1213)

`#ERR_MAGICKEY`
: Cannot locate magic key in player file! (1214)

`#ERR_FINDCLIPREGION`
: Could not find clip region %d! (1215)

`#ERR_FUNCREMOVED`
: This function is not supported any longer! (1216)

`#ERR_COORDSRANGE`
: Specified coordinates are out of range! (1217)

`#ERR_BADDIMENSIONS`
: Width/height values must be greater than 0! (1218)

`#ERR_FINDSPRITE`
: Could not find sprite %d! (1219)

`#ERR_SPRITEONSCREEN`
: Sprite %d is not on screen! (1220)

`#ERR_PREPROCSYM`
: Unknown preprocessor command @%s! (1221)

`#ERR_UNKNOWNTAG`
: Unknown tag "%s"! (1222)

`#ERR_MASKNALPHA`
: Mask and alpha channel are mutually exclusive! (1223)

`#ERR_NOSPRITES`
: Please remove all sprites first! (1224)

`#ERR_WRONGCLIPREG`
: Clip region does not fit into the output device's dimensions! (1225)

`#ERR_NOCLIPREG`
: Please remove clip region before enabling layers! (1226)

`#ERR_MODIFYSPRITE`
: Cannot modify a sprite that is on screen! (1227)

`#ERR_MODIFYSPRITE2`
: Cannot modify a linked sprite! (1228)

`#ERR_ENDDOUBLEBUFFER`
: Please end double buffering first! (1229)

`#ERR_DBTRANSWIN`
: Double buffering is currently not supported for transparent displays! (1230)

`#ERR_FINDMUSIC`
: Could not find music %d! (1231)

`#ERR_MUSNOTPLYNG`
: Music %d is not currently playing! (1232)

`#ERR_SEEKRANGE`
: Specified seek position is out of range! (1233)

`#ERR_MIXMUSMOD`
: Music and tracker modules cannot be played at the same time! (1234)

`#ERR_UNKNOWNMUSFMT`
: Unknown music format! (1235)

`#ERR_MUSFMTSUPPORT`
: Music format does not support this function! (1236)

`#ERR_TABLEORNIL`
: Table or `Nil` expected! (1237)

`#ERR_PROTMETATABLE`
: Cannot change a protected metatable! (1238)

`#ERR_ERRORCALLED`
: Undefined (1239)

`#ERR_ADDTASK`
: Error adding task to the system! (1240)

`#ERR_TASKSETUP`
: Error setting up task! (%s) (1241)

`#ERR_READRANGE`
: Cannot read beyond end of file! (1242)

`#ERR_BACKFILL`
: Wrong backfill configuration! (1243)

`#ERR_NODOUBLEBUFFER`
: Double buffering mode is not currently active! (1244)

`#ERR_STRTOOSHORT`
: Specified length exceeds string length! (1245)

`#ERR_CACHEERROR`
: An error occurred while processing the gfx cache! (1246)

`#ERR_STRTABLEARG`
: String expected in table argument "%s"! (1247)

`#ERR_APPLET`
: No applet filename specified! (1248)

`#ERR_KEYFILE`
: Keyfile error! (1249)

`#ERR_NOTADIR`
: %s is not a directory! (1250)

`#ERR_UNKTEXTFMT`
: Text format tag after square bracket not recognized! (1251)

`#ERR_TEXTSYNTAX`
: Syntax error in text format specification! (1252)

`#ERR_TEXTARG`
: Not enough arguments to this text format tag! (1253)

`#ERR_DEFFONT`
: Error opening default font! (1254)

`#ERR_ANTIALIAS`
: This font type does not support anti-aliased output! (1255)

`#ERR_CREATEPORT`
: Could not create message port! (1256)

`#ERR_NOREXX`
: ARexx server is not running! (1257)

`#ERR_REXXERR`
: Rexx interpreter returned an error! (%s) (1258)

`#ERR_STRCALLBACK`
: Callback function was expected to return a string! (1259)

`#ERR_PORTNOTAVAIL`
: There is already a port with the name %s! (1260)

`#ERR_BAD8SVX`
: Bad data in IFF 8SVX or IFF 16SV file! (1261)

`#ERR_CMPUNSUPPORTED`
: This sound file uses an unsupported compression format! (1262)

`#ERR_BADWAVE`
: Bad data in RIFF WAVE file! (1263)

`#ERR_MUSNOTPAUSED`
: This music is not in pause state! (1264)

`#ERR_CONFIG2`
: Undefined (1265)

`#ERR_EXETYPE`
: Unknown executable type specified! (1266)

`#ERR_OPENAUDIO`
: Cannot open audio device! (1267)

`#ERR_DATATYPESAVE`
: Cannot open specified datatype for saving! (1268)

`#ERR_DATATYPESAVE2`
: Datatype used for saving returned an error code! (1269)

`#ERR_LOADFRAME`
: Error loading animation frame! (1270)

`#ERR_LAYERSUPPORT2`
: This function cannot be used with layers enabled! (1271)

`#ERR_SHORTIF`
: Short IF statement must be on a single line! (1272)

`#ERR_SYSTOOOLD`
: Your Hollywood.sys version is too old! (1273)

`#ERR_KEYNOTFOUND`
: Key "%s" not found in system base! (1274)

`#ERR_FINDPORT`
: Port "%s" could not be found! (1275)

`#ERR_TOOSMALL2`
: The active screen is not large enough to hold a %s display! (1276)

`#ERR_SAVEPNG`
: Error saving PNG picture! (1277)

`#ERR_NOTIGER`
: Hollywood requires at least version 10.4 (Tiger) of macOS! (1278)

`#ERR_STREAMASSAMPLE`
: Cannot load audio stream as a sample! (1279)

`#ERR_AUDIOCONVERTER`
: Error creating an audio converter for this format! (1280)

`#ERR_RENDERCALLBACK`
: Error installing render callback on mixer bus! (1281)

`#ERR_SETFILEATTR`
: Error setting file attributes! (1282)

`#ERR_SETFILEDATE`
: Error setting file date! (1283)

`#ERR_SETFILECOMMENT`
: Error setting file comment! (1284)

`#ERR_INVALIDDATE`
: Invalid date format specification! (1285)

`#ERR_LOCK2`
: Error locking %s! (1286)

`#ERR_THREAD`
: Error setting up thread! (1287)

`#ERR_UNSUPPORTEDFEAT`
: This feature is currently not supported on this platform! (1288)

`#ERR_NOCHANNEL`
: Could not allocate audio channel for this sound! (1289)

`#ERR_CREATEEVENT`
: Error creating unnamed event object! (1290)

`#ERR_DSOUNDNOTIFY`
: Error obtaining sound notification interface! (1291)

`#ERR_DSOUNDNOTIPOS`
: Error setting sound buffer notification positions! (1292)

`#ERR_DSOUNDPLAY`
: Error starting sound buffer playback! (1293)

`#ERR_AFILEPROP`
: Error getting audio file properties! (1294)

`#ERR_DIRECTSHOW`
: Error setting up DirectShow environment! (#%d) (1295)

`#ERR_REGCLASS`
: Error registering window class! (1296)

`#ERR_TIMER`
: Error setting up timer function! (1297)

`#ERR_SEMAPHORE`
: Error allocating semaphore object! (1298)

`#ERR_8OR16BITONLY`
: Hollywood currently only supports 8 or 16 bit sounds! (1299)

`#ERR_DISPMINIMIZED`
: This function cannot be used with a minimized display! (1300)

`#ERR_COMMODITY`
: Error creating commodity object! (1301)

`#ERR_MSGPORT`
: Error setting up message port! (1302)

`#ERR_TEXTCONVERT`
: Error converting text to Unicode! (1303)

`#ERR_ATSUI`
: Error in text operation (ATSUI error)! (1304)

`#ERR_LFSYNTAX`
: Syntax error in link file database! (1305)

`#ERR_ZLIBIO`
: A zlib IO error occurred! (1306)

`#ERR_ZLIBSTREAM`
: A zlib stream error occurred! (1307)

`#ERR_ZLIBVERSION`
: Invalid zlib version detected! (1308)

`#ERR_ZLIBDATA`
: Invalid or incomplete deflate data (zlib)! (1309)

`#ERR_PAKFORMAT`
: Unknown compression format! (1310)

`#ERR_NOTXTLAYER`
: Specified layer is not a text layer! (1311)

`#ERR_DDAUTOSCALE`
: Autoscale cannot be used together with DisplayDesktop! (1312)

`#ERR_NODISLAYERS`
: Layers cannot be disabled when the layer scaling engine is used! (1313)

`#ERR_LOCKEDOBJ`
: Cannot modify object while it is locked! (1314)

`#ERR_WRITEJPEG`
: Error writing JPEG image! (1315)

`#ERR_DDRECVIDEO`
: Scripts using DisplayDesktop cannot be recorded! (1316)

`#ERR_WRONGCMDRECVIDEO`
: This command cannot be used while in video recording mode! (1317)

`#ERR_FINDDIR`
: Could not find directory handle %d! (1318)

`#ERR_MUSNOTPLYNG2`
: Music is not currently playing! (1319)

`#ERR_FINDPOINTER`
: Could not find pointer image %d! (1320)

`#ERR_POINTERIMG`
: Error creating pointer from image! (1321)

`#ERR_READFUNC`
: Cannot find Hollywood function at this offset! (1322)

`#ERR_BADBASE64`
: Invalid Base64 encoding! (1323)

`#ERR_NOHWFUNC`
: Specified function is not a user function! (1324)

`#ERR_SPRITEONSCREEN2`
: Sprite is not on screen! (1325)

`#ERR_FINDASYNCDRAW`
: Could not find async draw function %d! (1326)

`#ERR_FREECURPOINTER`
: Cannot free currently active pointer! (1327)

`#ERR_READTABLE`
: Cannot find Hollywood table at this offset! (1328)

`#ERR_LAYERSWITCH`
: Cannot switch layer mode while async draw is active! (1329)

`#ERR_VIDEOSTRATEGY`
: Unknown video strategy specified! (1330)

`#ERR_WRONGVSTRATEGY`
: Invalid video strategy configuration! (1331)

`#ERR_FINDFONT`
: Cannot find font %s on this system! (1332)

`#ERR_LINKFONT`
: Font %s cannot be linked because it is of a wrong type! (1333)

`#ERR_FINDFONT2`
: Could not find font %d! (1334)

`#ERR_FONTPATH`
: Font specification must not be a file! (1335)

`#ERR_FONTFORMAT`
: Font is in an unsupported format! (1336)

`#ERR_NOCOORDCST`
: You cannot use coordinate constants here! (1337)

`#ERR_ANIMDISK`
: This function cannot be used with disk-based animations! (1338)

`#ERR_SELECTANIM`
: This command cannot be used when SelectAnim() is active! (1339)

`#ERR_MODIFYAANIM`
: Attempt to modify the active anim! (1340)

`#ERR_FINDANIMSTREAM`
: Could not find anim stream %d! (1341)

`#ERR_NEEDMORPHOS2`
: This feature requires at least MorphOS 2.0! (1342)

`#ERR_SMODEALPHA`
: Screen doesn't support alpha transparent windows! (1343)

`#ERR_FINDDISPLAY`
: Could not find display %d! (1344)

`#ERR_MULTIBGPIC`
: Cannot use the a single BGPic for multiple displays! (1345)

`#ERR_FREEADISPLAY`
: Cannot free the active display! (1346)

`#ERR_CLOSEDDISPLAY`
: Cannot use this function while display is closed! (1347)

`#ERR_ADDAPPICON`
: Error adding app icon to Workbench! (1348)

`#ERR_SCREENSIZE`
: Screen size %s not supported by current monitor settings! (1349)

`#ERR_DIFFDEPTH`
: Cannot switch display mode because of different color resolution! (1350)

`#ERR_VIDRECMULTI`
: Cannot use multiple displays while in video recording mode! (1351)

`#ERR_VIDRECTRANS`
: Cannot use transparent displays while in video recording mode! (1352)

`#ERR_NEEDOS41`
: This feature requires at least AmigaOS 4.1! (1353)

`#ERR_SYSIMAGE`
: Error obtaining system image! (1354)

`#ERR_SYSBUTTON`
: Error creating system button! (1355)

`#ERR_OPENANIM2`
: Animation file "%s" is in an unknown/unsupported format! (1356)

`#ERR_OPENSOUND2`
: Sample file "%s" is in an unknown/unsupported format! (1357)

`#ERR_LOADPICTURE2`
: Image file "%s" is in an unknown/unsupported format! (1358)

`#ERR_SIGNAL`
: Error allocating signal! (1359)

`#ERR_ADDAPPWIN`
: Error adding app window to Workbench! (1360)

`#ERR_CLIPFORMAT`
: Unknown data format in clipboard! (1361)

`#ERR_SORTFUNC`
: Invalid order function for sorting! (1362)

`#ERR_INISYNTAX`
: Syntax error in configuration file! (1363)

`#ERR_CLIPOPEN`
: Failed to open clipboard! (1364)

`#ERR_CLIPREAD`
: Error reading from clipboard! (1365)

`#ERR_SCALEBGPIC`
: Cannot change size of a BGPic that is selected into a display! (1366)

`#ERR_SELECTBGPIC`
: You need to select the BGPic's display before modifying the BGPic! (1367)

`#ERR_CLIPWRITE`
: Error writing to clipboard! (1368)

`#ERR_FINDLAYER`
: Cannot find layer "%s" in current BGPic! (1369)

`#ERR_INVINSERT`
: Invalid insert position specified! (1370)

`#ERR_ALREADYASYNC`
: Specified layer already has an async draw object attached! (1371)

`#ERR_REMADLAYER`
: Cannot remove layer while it is used by an async draw object! (1372)

`#ERR_NAMETOOLONG`
: Specified name is too long! (1373)

`#ERR_GROUPNAMEUSED`
: Specified group name already assigned to a layer! (1374)

`#ERR_REGISTRYREAD`
: Error reading from registry key %s! (1375)

`#ERR_REGISTRYWRITE`
: Error writing to registry key %s! (1376)

`#ERR_SELECTBGPIC2`
: Cannot modify the graphics of a BGPic associated with a display! (1377)

`#ERR_MODIFYABGPIC`
: Attempt to modify the BGPic currently selected as output device! (1378)

`#ERR_ADFWRONGDISP`
: Asynchronous drawing object is not associated with current display! (1379)

`#ERR_ADFFREEDISP`
: Cannot free display before associated async draw objects have been freed! (1380)

`#ERR_SPRITELINK`
: Cannot create sprite link from sprite link! (1381)

`#ERR_WRONGSPRITESIZE`
: Specified sprites must have the same dimensions! (1382)

`#ERR_TRANSBRUSH`
: Cannot trim brush because it is fully transparent! (1383)

`#ERR_DINPUT`
: Error opening DirectInput! (1384)

`#ERR_JOYSTICK`
: Cannot acquire joystick! (1385)

`#ERR_FT2` Error initializing freetype2! (1386)

`#ERR_ICONDIMS`
: Specified image does not match required icon dimensions (%s)! (1387)

`#ERR_BRUSHTYPE`
: This operation is not supported by the specified brush type! (1388)

`#ERR_TFVBRUSH`
: Cannot insert a transformed vector brush as a layer! Use draw tags instead of transforming the brush directly! (1389)

`#ERR_BGPICTYPE`
: This operation is not supported by the specified BGPic type! (1390)

`#ERR_TFVBRUSHBGPIC`
: Cannot convert a transformed vector brush into a BGPic! (1391)

`#ERR_TFVBGPICBRUSH`
: Cannot convert a transformed vector BGPic into a brush! (1392)

`#ERR_FINDPATH`
: Could not find path %d! (1393)

`#ERR_EMPTYPATH`
: Cannot draw empty path! (1394)

`#ERR_VFONTTYPE`
: You must use the inbuilt font engine for vector text! (1395)

`#ERR_VFONT`
: Error setting up vector font! (1396)

`#ERR_CREATESHORTCUT`
: Error creating shortcut! (1397)

`#ERR_NOACCESS`
: Access denied! (1398)

`#ERR_BADPLATFORM`
: Compiling for architecture "%s" not supported by this version! (1399)

`#ERR_NEWHWPLUGIN`
: This plugin requires at least Hollywood %s! (1400)

`#ERR_PLUGINVER`
: Version %s is required at minimum! (1401)

`#ERR_PLUGINARCH`
: Plugin is incompatible with current platform! (%s) (1402)

`#ERR_IMAGEERROR`
: Error in image data in file %s! (1403)

`#ERR_RENDERADLAYER`
: Cannot render layer because it is attached to async draw object! (1404)

`#ERR_NOJOYATPORT`
: No joystick found at specified game port! (1405)

`#ERR_DEMO`
: This feature is not available in the demo version of Hollywood! (1406)

`#ERR_DEMO2`
: Demo version script size is limited to 800 lines and/or 32 kilobyte! (1407)

`#ERR_DEMO3`
: This demo version has expired! Please buy the full version! (1408)

`#ERR_FINDCLIENT`
: Could not find connection %d! (1409)

`#ERR_SOCKET`
: The following network error occurred: %s (1410)

`#ERR_OPENSOCKET`
: Could not initialize base socket interface! (1411)

`#ERR_FINDSERVER`
: Could not find server %d! (1412)

`#ERR_SOCKOPT`
: Error setting socket options! (1413)

`#ERR_PEERNAME`
: Error obtaining peer name! (1414)

`#ERR_HOSTNAME`
: Error obtaining host name! (1415)

`#ERR_UNKPROTOCOL`
: Unknown protocol in URL! (1416)

`#ERR_BADURL`
: Invalid URL specified! (1417)

`#ERR_HTTPERROR`
: HTTP error %d occurred! (1418)

`#ERR_HTTPTE`
: Unsupported HTTP transfer mode! (1419)

`#ERR_SENDDATA`
: An error occurred during data send! (1420)

`#ERR_FTPERROR`
: FTP error %d occurred! (1421)

`#ERR_RECVTIMEOUT`
: Receive timeout reached! (1422)

`#ERR_RECVCLOSED`
: Remote server has closed the connection! (1423)

`#ERR_RECVUNKNOWN`
: Unknown error occurred during data receive! (1424)

`#ERR_FILENOTFOUND`
: File %s not found on this server! (1425)

`#ERR_FTPAUTH`
: Access denied for specified user/password! (1426)

`#ERR_UPLOADFORBIDDEN`
: No permission to upload file to %s! (1427)

`#ERR_SOCKNAME`
: Error obtaining socket name! (1428)

`#ERR_FINDUDPOBJECT`
: Could not find UDP object %d! (1429)

`#ERR_BADIP`
: Invalid IP specified! (1430)

`#ERR_XDISPLAY`
: Error opening connection to X server! (1431)

`#ERR_CREATEGC`
: Error creating graphics context! (1432)

`#ERR_PIPE`
: Error creating pipe! (1433)

`#ERR_GTK` Error opening GTK! (1434)

`#ERR_NEEDCOMPOSITE`
: Compositing must be enabled for displays with alpha transparency! (1435)

`#ERR_NOARGBVISUAL`
: Error obtaining a visual info that can handle ARGB graphics! (1436)

`#ERR_XFIXES`
: The Xfixes extension is required for this feature! (1437)

`#ERR_XCURSOR`
: The Xcursor extension is required for this feature! (1438)

`#ERR_ALSAPCM`
: Error configuring ALSA PCM output stream! (#%d) (1439)

`#ERR_SETENV`
: Error setting environment variable! (1440)

`#ERR_UNSETENV`
: Error removing environment variable! (1441)

`#ERR_XF86VIDMODEEXT`
: Screen mode switching requires the XFree86-VidModeExtension! (1442)

`#ERR_NODISPMODES`
: No display modes found! (1443)

`#ERR_UNKNOWNFILTER`
: Filter "%s" not recognized! (1444)

`#ERR_NOFILTERNAME`
: Missing filter name in table field %s! (1445)

`#ERR_TABEXPECTED2`
: Subtable expected in table "%s"! (1446)

`#ERR_SMPRANGE`
: Specified sample value is out of range! (1447)

`#ERR_NOTENOUGHPIXELS`
: Table does not contain enough pixels for specified size! (1448)

`#ERR_FINDVIDEO`
: Could not find video %d! (1449)

`#ERR_LOADVIDEO`
: File "%s" not recognized as a video stream! (1450)

`#ERR_VIDNOTPLAYING`
: Cannot pause video because it is not playing! (1451)

`#ERR_VIDNOTPAUSED`
: Cannot resume video because it is not paused! (1452)

`#ERR_COLORSPACE`
: Error obtaining colorspace! (1453)

`#ERR_QUICKTIME`
: This function requires QuickTime to be installed! (1454)

`#ERR_VIDATTACHED`
: This functionality is not available while videos are attached to the display! (1455)

`#ERR_FGRABVIDSTATE`
: Cannot grab frame while video is playing or paused! (1456)

`#ERR_VIDEOFRAME`
: Specified video frame is out of range! (1457)

`#ERR_VIDEOTRANS`
: Videos cannot be played on top of transparent BGPics! (1458)

`#ERR_LOADPLUGIN`
: Error loading plugin "%s"! (1459)

`#ERR_VECGFXPLUGIN`
: This functionality requires a vectorgraphics plugin to be installed! (1460)

`#ERR_INVCAPIDX`
: Invalid capture index! (1461)

`#ERR_INVPATCAP`
: Invalid pattern capture! (1462)

`#ERR_MALFORMPAT1`
: Malformed pattern! (ends with "%%") (1463)

`#ERR_MALFORMPAT2`
: Malformed pattern! (missing "]") (1464)

`#ERR_UNBALANCEDPAT`
: Unbalanced pattern! (1465)

`#ERR_TOOMANYCAPTURES`
: Too many captures! (1466)

`#ERR_MISSINGOPBRACK`
: Missing "[" after "%%f" in pattern! (1467)

`#ERR_UNFINISHEDCAPTURE`
: Unfinished capture! (1468)

`#ERR_TFIMAGE`
: Error transforming image! (1469)

`#ERR_DRAWPATH`
: Error drawing path! (1470)

`#ERR_MOBILE`
: This command is not available in the mobile version of Hollywood! (1471)

`#ERR_DDMOBILE`
: Scripts using DisplayDesktop not supported on mobile devices! (1472)

`#ERR_MULDISMOBILE`
: Multiple displays not supported in the mobile version of Hollywood! (1473)

#ERR_TRANSBGMOBILE
: Transparent BGPics not supported in the mobile version of Hollywood! (1474)

#ERR_MODIFYPSMP
: Cannot modify a sample that is currently playing! (1475)

#ERR_TABCALLBACK
: Callback was expected to return a table! (1476)

#ERR_BADCALLBACKRET
: Invalid callback return value! (1477)

#ERR_NOCALLBACK
: This command must not be called from a callback function! (1478)

#ERR_LOWFREQ
: Specified pitch value is too low! (1479)

#ERR_FINDLAYERDATA
: Data item "%s" not found in specified layer! (1480)

#ERR_NODIRPATTERN
: Filter patterns can only be used on directories! (1481)

#ERR_SEEKFORMAT
: Source file format does not support seeking! (1482)

#ERR_PLUGINTYPE
: Plugin type not recognized! (%s) (1483)

#ERR_NOMUSICCB
: This command must only be called while in a music callback! (1484)

#ERR_NOFMBHANDLER
: You have to install a "FillMusicBuffer" event handler first! (1485)

#ERR_UNKNOWNIMGOUT
: Unknown image format specified! (1486)

#ERR_SAVEIMAGE
: Error saving image! (1487)

#ERR_UNKNOWNANMOUT
: Unknown anim format specified! (1488)

#ERR_SAVEANIM
: Error saving anim! (1489)

#ERR_UNKNOWNSMPOUT
: Unknown sample format specified! (1490)

#ERR_SAVESAMPLE
: Error saving sample! (1491)

#ERR_UDEXPECTED
: Userdata expected in argument %d! (1492)

`#ERR_ASSERTFAILED`
: Assertion failed! (1493)

`#ERR_REQUIREPLUGIN`
: This program requires %s! (1494)

`#ERR_NOABSPATH`
: Absolute path specifications are not allowed here! (1495)

`#ERR_FINDOBJECTDATA`
: Data item "%s" not found in specified object! (1496)

`#ERR_HWBRUSH`
: Hardware brushes cannot be used here! (1497)

`#ERR_HWBRUSHFUNC`
: This functionality is currently not supported for hardware brushes! (1498)

`#ERR_SAVERALPHA`
: Format saver does not support alpha channel! (1499)

`#ERR_VIDPAUSED`
: Video is paused. Use ResumeVideo() to resume playback! (1500)

`#ERR_VIDPLAYING`
: Video is already playing! (1501)

`#ERR_PERCENTFORMATSTR`
: Invalid percent format in table argument "%s"! (1502)

`#ERR_SCRPIXFMT`
: Incompatible screen pixel format detected! (1503)

`#ERR_SATFREEDISP`
: Cannot free display before attached satellites have been detached! (1504)

`#ERR_CREATEICON`
: Error creating icon from image! (1505)

`#ERR_GETSHORTCUT`
: Error retrieving full path from shortcut file! (1506)

`#ERR_UNKNOWNMIMETYPE`
: Unknown MIME type for extension *.%s! (1507)

`#ERR_NOMIMEVIEWER`
: Cannot find viewer for extension *.%s! (1508)

`#ERR_JAVA`
: Cannot attach thread to Java VM! (1509)

`#ERR_FINDACTIVITY`
: Cannot find activity "%s"! (1510)

`#ERR_BEGINREFRESH`
: Cannot call this command while in BeginRefresh() mode! (1511)

`#ERR_DBVIDEOLAYER`
: Video object is already in use as a layer on a BGPic! (1512)

`#ERR_VIDEOLAYER`
: This functionality is not supported for video layers! (1513)

`#ERR_VIDEOLAYERDRV`
: Video layers are only supported by Hollywood's platform independent video renderer! (1514)

`#ERR_BADLAYERTYPE`
: Specified layer type does not support this functionality! (1515)

`#ERR_VIDSTOPPED`
: Video is already stopped! (1516)

`#ERR_VIDLAYERFUNC`
: Use functions from layers library to change attributes of video layers! (1517)

`#ERR_SETADAPTER`
: Cannot set adapter! (1518)

`#ERR_DISPLAYADAPTERSUPPORT`
: This functionality is not available with this display adapter! (1519)

`#ERR_DLOPEN`
: Cannot load plugin: %s (1520)

`#ERR_PLUGINSYMBOL`
: Error loading plugin symbol: %s (1521)

`#ERR_BITMAP`
: Error allocating bitmap! (1522)

`#ERR_SATELLITE`
: This functionality is not available when using display satellites! (1523)

`#ERR_READVIDEOPIXELS`
: Error reading pixels from hardware bitmap! (1524)

`#ERR_HWDBFREEDISP`
: Cannot free display while hardware double buffering is active! (1525)

`#ERR_HWBMCLOSEDISP`
: Cannot allocate hardware bitmap while display has not been realized! (1526)

`#ERR_INCOMPATBRUSH`
: Hardware brush is incompatible with the current display! (1527)

`#ERR_ADDSYSEVENT`
: Error adding system event! (1528)

`#ERR_SEEKFILE`
: This file adapter does not support seeking! (1529)

`#ERR_CLOSEFILE`
: Error closing file handle! (1530)

`#ERR_FINDPLUGIN`
: Cannot find plugin %s! (1531)

`#ERR_PLUGINDOUBLET`
: Plugin %s has already been loaded! (1532)

`#ERR_APPLICATION`
: Error registering application! (1533)

`#ERR_NEEDAPPLICATION`
: This functionality is only available for system-registered applications! (1534)

`#ERR_FINDAPPLICATION`
: Cannot find application %s! (1535)

`#ERR_SENDMESSAGE`
: Error sending message! (1536)

`#ERR_FINDMENU`
: Could not find menu %d! (1537)

`#ERR_MENUCOMPLEXITY`
: Menu tree definition is too complex! (1538)

`#ERR_CREATEMENU`
: Error creating menu! (1539)

`#ERR_VISUALINFO`
: Error obtaining visual info! (1540)

`#ERR_SETMENU`
: Error setting menu strip! (1541)

`#ERR_MENUATTACHED`
: Cannot free menu while it is still attached to a display! (1542)

`#ERR_FINDMENUITEM`
: Cannot find menu item %s! (1543)

`#ERR_NOMENU`
: Specified display does not have a menu attached! (1544)

`#ERR_EMPTYMENUTREE`
: Empty menu trees are not allowed! (1545)

`#ERR_TAGEXPECTED`
: Tag expected! (1546)

`#ERR_FULLSCREEN`
: This functionality is not supported in full screen mode! (1547)

`#ERR_CREATEDOCKY`
: Error creating application docky! (1548)

`#ERR_UPDATEICON`
: Error updating dock icon! (1549)

`#ERR_DOUBLEMENU`
: Tree has already been defined for this menu! (1550)

`#ERR_CONTEXTMENU`
: Context menus must only contain a single tree! (1551)

`#ERR_VECTORBRUSH`
: This functionality is not available for vector brushes! (1552)

`#ERR_NOCONTEXTMENU`
: Application does not expose a context menu! (1553)

`#ERR_ACCELERATOR`
: Error creating accelerator table! (1554)

`#ERR_FINDMONITOR`
: Cannot find monitor %d! (1555)

`#ERR_MONITORFULLSCREEN`
: Monitor %d is already in fullscreen mode! (1556)

`#ERR_MONITORRANGE`
: Specified monitor is out of range! (1557)

`#ERR_GETMONITORINFO`
: Error obtaining monitor information! (1558)

`#ERR_SCREENMODE`
: Cannot find an appropriate screen mode for this display! (1559)

`#ERR_NOCOMPRESS`
: The Hollywood Player only supports compressed applets! (1560)

`#ERR_GRABSCREEN`
: Error grabbing screen pixels! (1561)

`#ERR_ALLOCCHANNEL`
: Error allocating audio channel! (1562)

`#ERR_REQUIRETAGFMT`
: Syntax error in tag format! (1563)

`#ERR_ALLOCALPHA`
: Error allocating alpha channel! (1564)

`#ERR_ALLOCMASK`
: Error allocating mask! (1565)

`#ERR_OLDAPPLET`
: This functionality is only available to applets compiled by Hollywood %s or higher! (1566)

`#ERR_MUSPAUSED`
: Music is paused. Use ResumeMusic() to resume playback! (1567)

`#ERR_MUSPLAYING`
: Music is already playing! (1568)

`#ERR_CONSOLEARG`
: Invalid parameter for console argument! (1569)

`#ERR_FILESIZE`
: Error determining file size! (1570)

`#ERR_STAT`
: Error examining file system object! (1571)

`#ERR_REQAUTH`
: This server requires user authentification! (1572)

`#ERR_MISSINGFIELD`
: Table field "%s" must be specified! (1573)

`#ERR_NOTRANSPARENCY`
: Image "%s" does not have a transparent pen! (1574)

`#ERR_LEGACYPTMOD`
: Legacy audio driver does not support playing multiple Protracker modules at once! (1575)

`#ERR_CHANNELRANGE`
: Specified audio channel is out of range! (1576)

`#ERR_FILEFORMAT`
: File format error! (1577)

`#ERR_LINKPLUGIN`
: Error linking plugin %s! (1578)

`#ERR_EXECUTE`
: Failed to execute program! (1579)

`#ERR_AMIGAGUIDE`
: Error opening AmigaGuide file %s! (1580)

`#ERR_COMPLEXPATTERN`
: Pattern to complex! (1581)

`#ERR_ESCREPLACE`
: Invalid use of escape character in replacement string! (1582)

`#ERR_INVREPLACE`
: Invalid replacement value! (1583)

`#ERR_BADENCODING`
: Encoding not recognized! (1584)

`#ERR_INVALIDUTF8`
: Invalid UTF-8 sequence encountered! (1585)

`#ERR_DIFFENCODING`
: Cannot include applet because it uses a different encoding than the current script! (1586)

`#ERR_DBLENCODING`
: Conflicting encodings specified! (1587)

#ERR_INVALIDUTF8ARG
: Invalid UTF-8 string in argument %d! (1588)

#ERR_CORETEXT
: Error drawing string using Core Text! (1589)

#ERR_COREFOUNDATION
: A Core Foundation allocation error has occurred! (1590)

#ERR_FRAMEGRABBER
: Error grabbing frame from video stream! (1591)

#ERR_FINDSELECTOR
: Cannot find selector %s! (1592)

#ERR_FIRSTPREPROC
: Conditional compile preprocessor commands must be first in line! (1593)

#ERR_ELSEIFAFTERELSE
: ELSEIF after ELSE! (1594)

#ERR_ELSETWICE
: ELSE used twice! (1595)

#ERR_NOBLOCKBREAK
: No block to break! (1596)

#ERR_NOFALLTHROUGH
: No block to fall through! (1597)

#ERR_TABEXPECTED3
: Table expected! (1598)

#ERR_EMPTYTABLE
: Table needs to have at least one item! (1599)

#ERR_MOVEFILE
: Error moving file! (1600)

#ERR_RADIOTOGGLEMENU
: Radio and toggle menu flags cannot be combined! (1601)

#ERR_RANDOMIZE
: Error generating random number! (1602)

#ERR_TRIALCOMPILE
: Compiling applets or executables isn't supported in the trial version! (1603)

#ERR_TRIALSAVEVID
: Video recording isn't supported in the trial version! (1604)

#ERR_TRIALLIMIT
: The trial version doesn't support scripts bigger than 16kb! (1605)

#ERR_TRIALINCLUDE
: Including files isn't supported in the trial version! (1606)

`#ERR_VIDEOINIT`
: Error initializing video device! (1607)

`#ERR_FINDICON`
: Could not find icon %d! (1608)

`#ERR_ICONPARMS`
: Selected image parameters don't match normal image parameters! (1609)

`#ERR_ICONSIZE`
: Icon size used twice! (1610)

`#ERR_LOADICON`
: Icon file \"%s\" is in an unknown/unsupported format! (1611)

`#ERR_ICONSTANDARD`
: There can be only one standard icon size! (1612)

`#ERR_ICONENTRY`
: Specified icon entry is out of range! (1613)

`#ERR_ICONVECTOR`
: Vector brushes must be the only icon entry! (1614)

`#ERR_MULTIDISPLAYS`
: Hollywood only supports a single display on this platform! (1615)

`#ERR_TEXTURE`
: Error creating texture! (1616)

`#ERR_SURFACE`
: Error creating surface! (1617)

`#ERR_RENDERER`
: Error creating renderer! (1618)

`#ERR_FINDSERIAL`
: Could not find serial connection %d! (1619)

`#ERR_INITSERIAL`
: Error initializing serial interface! (1620)

`#ERR_OPENSERIAL`
: Error opening serial port %s! (1621)

`#ERR_SERIALIO`
: Serial I/O error! (1622)

`#ERR_SENDTIMEOUT`
: Send timeout reached! (1623)

`#ERR_SENDUNKNOWN`
: Unknown error occurred during data send! (1624)

`#ERR_PLUGINSUPPORT`
: Plugin doesn't support this feature! (1625)

`#ERR_REWINDDIR`
: Error rewinding directory! (1626)

`#ERR_MONITORDIR`
: Error monitoring directory! (1627)

`#ERR_JAVAMETHOD`
: Java method \"%s\" not found! (1628)

`#ERR_NUMBEREXPECTED`
: Number expected! (1629)

`#ERR_CHANGEDIR`
: Error changing directory to %s! (1630)

`#ERR_FUNCTABLEARG`
: Function expected in table argument \"%s\"! (1631)

`#ERR_BADYIELD`
: Attempt to yield across metamethod/C-call boundary! (1632)

`#ERR_CYIELD`
: Cannot yield a C function! (1633)

`#ERR_THREADEXPECTED`
: Coroutine expected in argument %d! (1634)

`#ERR_YIELD`
: This error is for internal use only. (1635)

`#ERR_DEADRESUME`
: Cannot resume dead coroutine! (1636)

`#ERR_NONSUSPENDEDRESUME`
: Cannot resume non-suspended coroutine! (1637)

`#ERR_FORBIDMODAL`
: This command has been disabled by a plugin! (1638)

`#ERR_SERIALIZE`
: Error serializing item! (1639)

`#ERR_SERIALIZETYPE`
: This data type cannot be serialized! (1640)

`#ERR_DESERIALIZE`
: Error deserializing item! (1641)

`#ERR_FINDASYNCOBJ`
: Could not find async operation %d! (1642)

`#ERR_NOPALETTE`
: Image file \"%s\" does not have a palette! (1643)

`#ERR_NEEDPALETTEIMAGE`
: Image data does not have a palette! (1644)

`#ERR_NOPALETTEIMAGE`
: This function cannot be used with palette images! (1645)

`#ERR_PENRANGE`
: Palette pen is out of range! (1646)

`#ERR_DEPTHMISMATCH`
: Incompatible pixel color depth! (1647)

`#ERR_ALLOCCHUNKY`
: Error allocating palette bitmap! (1648)

`#ERR_FINDPALETTE`
: Could not find palette %d! (1649)

`#ERR_DEPTHRANGE`
: Specified palette depth is out of range! (1650)

`#ERR_BGPICPALETTE`
: Current BGPic does not have a palette! (1651)

`#ERR_UNKNOWNPALETTE`
: Unknown standard palette type specified! (1652)

`#ERR_PALETTEFILL`
: Specified fill style cannot be used with palette images! (1653)

`#ERR_DISPLAYDESKTOPPAL`
: Palettes cannot be used together with a desktop display! (1654)

`#ERR_DBPALETTE`
: Hardware double buffers cannot be used in palette mode! (1655)

`#ERR_UNKNOWNICNOUT`
: Unknown icon format specified! (1656)

`#ERR_SAVEICON`
: Error saving icon! (1657)

`#ERR_PALETTEMODE`
: This function can only be used in palette mode! (1658)

`#ERR_NOPALETTEMODE`
: This function cannot be used in palette mode! (1659)

`#ERR_NORTG`
: Cannot find CyberGraphX or Picasso96! To use Hollywood without either CyberGraphX or Picasso96, you need to install the Plananarama plugin! (1660)

`#ERR_GETMENUATTR`
: Error getting menu attributes! (1661)

`#ERR_SETMENUATTR`
: Error setting menu attributes! (1662)

`#ERR_TRAYICON`
: Error setting tray icon! (1663)

#ERR_FONTPATH2
: You must use the inbuilt font engine when specifying font files directly! (1664)

#ERR_MEDIAFOUNDATION
: A Media Foundation error has occurred! (1665)

#ERR_GETIFADDRS
: Error getting interface addresses! (1666)

#ERR_TFVANIM
: Cannot insert a transformed vector anim as a layer! Use draw tags instead of transforming the anim directly! (1667)

#ERR_VECTORANIM
: This functionality is not available for vector anims! (1668)

#ERR_PLAYVIDEO
: Error starting video playback! (1669)

#ERR_TEXTCONVERT2
: Error during text conversion! (1670)

#ERR_TFVTEXTOBJ
: Cannot insert a transformed vector text object as a layer! Use draw tags instead of transforming the text object directly! (1671)

#ERR_GROUPNOTFOUND
: Specified layer group doesn't exist! (1672)

#ERR_MERGEDLAYER
: This functionality isn't supported for merged layers! (1673)

#ERR_REMMERGEDLAYER
: Cannot remove layer that is part of a merged layer! (1674)

#ERR_ALLOCIMAGE
: Error allocating image! (1675)

#ERR_ADVANCEDCONSOLE
: This function is only available in advanced console mode! (1676)

#ERR_COLORTERMINAL
: Terminal doesn't support color mode! (1677)

#ERR_CONSOLE
: A console error has occurred! (1678)

#ERR_FINDCONWIN
: Could not find console window %d! (1679)

#ERR_CONWIN
: Error creating console window! (1680)

#ERR_FREEPARENT
: Attempt to free parent before child! (1681)

#ERR_AMIGAINPUT
: Error initializing AmigaInput! (1682)

28.4 ExitOnError

NAME

ExitOnError – enable/disable Hollywood's error handler

SYNOPSIS

```
ExitOnError(enable)
```

FUNCTION

This function enables or disables Hollywood's error handler. If the error handler is enabled and an error occurs, your script will either be stopped and Hollywood will display the error message, or, in case you have installed a custom error handling callback using `RaiseOnError()`, this custom error handling callback will be executed. See Section 7.7 [Error handling], page 90, for details.

It can be useful to disable the error handler for a very short time if you need to check whether a certain command has succeeded or not. This can be done by enclosing the command in an `ExitOnError()` block. See Section 7.7 [Error handling], page 90, for details. It is not advised to disable the error handler for a longer time because errors can easily accumulate so in general you should only keep the error handler enabled.

In Hollywood 7.1 and up the new question mark syntax to check for errors is preferable to using `ExitOnError()` because it is much shorter and less prone to accidental mistakes. See Section 7.7 [Error handling], page 90, for details.

INPUTS

`enable` `True` to enable the error handler; `False` to disable it

28.5 GetErrorName

NAME

GetErrorName – get string for an error code

SYNOPSIS

```
err$ = GetErrorName(code)
```

FUNCTION

This function returns a string for a specified error code. The string describes the error for the passed code. This command should be called right after `GetLastError()` because the error string might contain some information that will be trashed when the next normal function is executed. Calling condition functions between `GetLastError()` and `GetErrorName()` is no problem. Please see the example for more information.

Be sure to read also the documentation of `GetLastError()` for more detailed information on manual error handling.

See Section 28.3 [Error codes], page 507, for a list of all error codes defined by Hollywood.

INPUTS

`code` an error code as returned by `GetLastError()`

RESULTS

`err$` a string describing the error occurred

EXAMPLE
See Section 28.6 [GetLastError], page 545.

28.6 GetLastError

NAME
GetLastError – get error code for the last command

SYNOPSIS
`code = GetLastError()`

FUNCTION
This function queries Hollywood's internal error flag and returns the result. This flag is zero if the last command executed was successful. If the command failed, an error code which is not zero will be returned. This error code can than be used to query Hollywood for a name string that describes the error occurred. (use `GetErrorName()` then)

Important note: Hollywood's internal error flag will be reset to zero before a command is called. Therefore the error code you will get when you call `GetLastError()` is the error code of the function that was called before `GetLastError()`.

Important note `#2`: This function is only useful if the automatical error handler is disabled. If it is enabled (which is the default), the error handler will break your script immediately when an error occurs. So your script will never reach a `GetLastError()` call if an error occurred and the automatic error handler is enabled. Therefore you will have to call `ExitOnError()` with `False` as the flag to disable Hollywood's error handler.

See Section 28.3 [Error codes], page 507, for a list of all error codes defined by Hollywood.

INPUTS
none

RESULTS

`code`	non-zero if an error occurred, zero for success

EXAMPLE
```
ExitOnError(FALSE)       ; disable automatic error handler
LoadBGPic(1,"blablabla") ; this command will fail!
code=GetLastError()
If code<>0
  err$=GetErrorName(code)
  SystemRequest("An error occurred!",err$,"OK")
  End
EndIf
```

The above code shows how to handle the error that LoadBGPic() will produce. It is important that there is no further command between the LoadBGPic() and the `GetLastError()`. If there would be another command, it would trash the error results of LoadBGPic().

28.7 RaiseOnError

NAME

RaiseOnError – install a custom error handler (V5.2)

SYNOPSIS

```
RaiseOnError(f)
```

FUNCTION

This function can be used to install a custom error handling function. Whenever an error occurs, this function will be called with the following four arguments: An error code, a string describing the error, the name of the last command, and the current line number.

This is useful if you do not want to use Hollywood's inbuilt automatic error handler. Please note that in certain situations the name of the last command and the current line number can be wrong.

Also note that if an error occurs in your custom error handling function, Hollywood will exit with a fatal error. Thus, you should keep the custom error handler as brief and straight-forward as possible.

To uninstall your custom error handler, simply pass `Nil` in the `f` argument.

See Section 28.3 [Error codes], page 507, for a list of all error codes defined by Hollywood.

INPUTS

`f` function that shall be called whenever an error occurs

EXAMPLE

```
Function p_ErrorFunc(code, msg$, cmd$, line)
   DebugPrint(code, msg$, cmd$, line)
EndFunction

RaiseOnError(p_ErrorFunc)

LoadBrush(1, "non_existing_brush.png")
```

The code above installs a custom error function and then tries to load a non-existing brush. This leads to the error function being called and further information will be printed to the debug device.

29 Event library

29.1 BreakEventHandler

NAME
BreakEventHandler – break current event handler cycle (V5.2)

SYNOPSIS
```
BreakEventHandler()
```

FUNCTION
This function can be used to break Hollywood's internal current event handler cycle. This is a lowlevel function and you normally will not need to use this. It is just here for certain emergency situations and debugging purposes.

INPUTS
none

29.2 ChangeInterval

NAME
ChangeInterval – change interval frequency (V2.0)

SYNOPSIS
```
ChangeInterval(id, ms)
```

FUNCTION
This function can be used to change the frequency of a running interval. Just specify the identifier of the interval and the new frequency. See Section 29.26 [SetInterval], page 582, for everything you need to know about intervals.

INPUTS

`id`	identifier of the interval function to modify
`ms`	new interval frequency in milliseconds

29.3 CheckEvent

NAME
CheckEvent – check for event without blocking (V1.9)

SYNOPSIS
```
info = CheckEvent()
```

FUNCTION
This function checks if there is an event in the queue. If there is, `CheckEvent()` will remove it from the queue and run its callback function. If there is no event in the queue, `CheckEvent()` will return immediately.

`CheckEvent()` returns a table that contains information about whether or not it has executed a callback. The following fields will be initialized in that table:

`Action:` Contains the name of the event that caused the callback execution (e.g. `OnMouseDown`). If `CheckEvent()` returns without having ran a callback, this field will be set to an empty string.

`ID:` Contains the identifier of the object that caused the callback execution (e.g. a display identifier). `ID` can also be zero in case an event was caused that has no ID associated.

`Triggered:` Will be set to `True` if `CheckEvent()` has executed a callback.

`NResults:` Contains the number of values that the user callback returned (e.g. 1). This will be 0 if the user callback did not return any values or if no user callback was ran at all.

`Results:` If `NResults` is greater than 0, this table will contain all values that the user callback returned. Otherwise this table will not be present at all. You can easily use this table to pass additional information from your callbacks back to the main scope of the program.

`CheckEvent()` is similar to the popular `WaitEvent()` command with the difference that `WaitEvent()` blocks the script execution until an event arrives whereas `CheckEvent()` immediately exits if there is no event. By using this command you can do something while waiting for an event which would not be possible with `WaitEvent()`.

Note that `CheckEvent()` only handles a single event from the event queue. If you'd like to handle all events that are currently in the event queue, you have to use `CheckEvents()` instead. See Section 29.4 [CheckEvents], page 548, for details.

Please note that generally you should use `CheckEvent()` only if you really need it. Using `WaitEvent()` is normally a much better idea than `CheckEvent()`.

INPUTS

none

RESULTS

`info` table containing information about whether and event occurred or not, and the return value(s) of the user callback in case it has been called

29.4 CheckEvents

NAME

CheckEvents – check for events without blocking (V6.1)

SYNOPSIS

`CheckEvents()`

FUNCTION

This function does the same as `CheckEvent()` but handles all events that are currently queued. `CheckEvent()`, on the other hand, only removes and handles a single event from the queue.

Another difference is that `CheckEvents()` doesn't return any information about the events it has removed and handled. If you need this information, you have to use `CheckEvent()` instead. See Section 29.3 [CheckEvent], page 547, for details.

Please note that you should use `CheckEvents()` only if you really need it. Using `WaitEvent()` is generally a much better idea than `CheckEvents()`.

INPUTS

none

29.5 ClearInterval

NAME

ClearInterval – remove an interval function (V2.0)

SYNOPSIS

```
ClearInterval(id)
```

FUNCTION

This function aborts the calling of the interval function specified by `id`. See Section 29.26 [SetInterval], page 582, for everything you need to know about intervals.

INPUTS

`id`	identifier of the interval function to cancel

EXAMPLE

See Section 29.26 [SetInterval], page 582.

29.6 ClearTimeout

NAME

ClearTimeout – remove a timeout function (V2.0)

SYNOPSIS

```
ClearTimeout(id)
```

FUNCTION

This function stops the timeout specified by `id`. It is not necessary to stop timeout functions that have already been called. Hollywood will clear them automatically after it called them. See Section 29.27 [SetTimeout], page 583, for everything you need to know about timeout functions.

INPUTS

`id`	identifier of the timeout function to cancel

EXAMPLE
See Section 29.27 [SetTimeout], page 583.

29.7 CtrlCQuit

NAME
CtrlCQuit – enable/disable quit by control-c (V2.0)

SYNOPSIS
`CtrlCQuit(enable)`

FUNCTION
By default, all Hollywood scripts can be interrupted at any time just by pressing CTRL-C. If you do not want this, use this function to disable the feature.

Please note: Think twice before disabling CTRL-C quit. For example, if your script runs in a borderless window or full screen, there is no close box to click, so it is quite handy to have CTRL-C quit enabled.

INPUTS
`enable` `True` to enable CTRL-C quit, `False` to disable it

29.8 DeleteButton

NAME
DeleteButton – delete a button (V2.0)

SYNOPSIS
`DeleteButton(id)`

FUNCTION
This function deletes the button specified by `id` from the current background picture.

INPUTS
`id` identifier of the button to delete

29.9 DisableButton

NAME
DisableButton – disable a button (V2.0)

SYNOPSIS
`DisableButton(id)`

FUNCTION
This function temporarily disables the button specified by `id`. You can enable it later by using the `EnableButton()` function. If you want to remove a button completely, use the `DeleteButton()` function.

INPUTS

`id` identifier of the button to disable

29.10 EnableButton

NAME

EnableButton – enable a button (V2.0)

SYNOPSIS

```
EnableButton(id)
```

FUNCTION

This function enables the button specified by `id`. This is only necessary if you have disabled it previously using `DisableButton()`.

INPUTS

`id` identifier of the button to enable

29.11 EscapeQuit

NAME

EscapeQuit – enable/disable quit with escape (V1.5)

SYNOPSIS

```
EscapeQuit(enable)
```

FUNCTION

If you set `enable` to `True`, pressing the escape key will immediately terminate your script.

INPUTS

`enable` `True` to enable escape quit, `False` to disable it

EXAMPLE

```
EscapeQuit(TRUE)
Repeat
  Wait(10)
Forever
```

The above code enters an endless loop which would normally block your program. But using EscapeQuit(`True`) allows the user to terminate it.

29.12 InKeyStr

NAME

InKeyStr – query user input (V1.5)

SYNOPSIS

```
input$ = InKeyStr(type[, maxlen, password, cursor])
```

FUNCTION

This function allows you to easily read input from the user's keyboard. `type` specifies the characters that are allowed to be typed in. `maxlen` can be used to limit the maximum length of the user input (default is 0 which means no limit). If `password` is set to `True`, Hollywood will show an asterisk (*) for every character typed in.

The following types can be specified currently:

`#ALL` Will accept all visible characters

`#ALPHABETICAL`
: Will accept only alphabetical characters; this is not necessarily limited to characters a-z. The user may also type special alphabetical characters that are only available in his language's alphabet

`#ALPHANUMERICAL`
: Will accept alphabetical and numerical characters

`#HEXNUMERICAL`
: Will accept hexadecimal characters (0-9 and a-f)

`#NUMERICAL`
: Will accept 0-9

If you have layers enabled while using this function, you will get a new layer of type `#PRINT` which contains the string the user has typed in (since Hollywood 2.0; in previous versions, layers for each character were added).

Starting with Hollywood 8.0, there is a new optional argument named `cursor`. If this is set to `True`, `InKeyStr()` will show a cursor while the user is typing. In that case it is also possible to use the cursor keys to navigate backwards and forwards and it is also possible to delete characters using the DEL key. The cursor will be drawn in the same color as the text.

Hollywood 8.0 also adds paste support to `InKeyStr()`. Just press CTRL+V (on Windows) or CMD+V (on all other systems) to paste text from the clipboard into the current insert position.

INPUTS

`type` specifies which characters the user is allowed to type in

`maxlen` optional: if you specify this argument, the user will only be able to type in `maxlen` characters; otherwise he can input as many characters as he wants and finish his input by pressing the RETURN key (defaults to 0 which means that the user can input as many characters as he wants)

`password` optional: if set to `True`, Hollywood will display an asterisk (*) instead of the actual character typed in (defaults to `False`)

`cursor` optional: if set to `True`, a cursor indicating the current insert and delete position will be shown (defaults to `False`) (V8.0)

RESULTS

`input$` the string that was typed in

EXAMPLE

```
Print("What is your name? ")
name$ = InKeyStr(#ALPHABETICAL)
Print("Hello", name$, "!")
```

The code above asks the user to enter his name and then it will be output.

29.13 InstallEventHandler

NAME

InstallEventHandler – install/remove an event handler (V2.0)

SYNOPSIS

```
InstallEventHandler(table[, userdata])
```

FUNCTION

You can use this function to install your own event handlers for standard events. You have to pass a table to this function, that tells Hollywood which event handlers you want to install or remove. To install a new handler you need to initialize the corresponding table field with your own function. If you want to remove an event handler, set the corresponding table field to 0.

The following table fields are recognized by this function:

`OnKeyDown:`

The function you specify here will be called each time the user presses a control key, a numerical key, or an English alphabetical key. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnKeyDown`.

`Key:` Will be set to the key which has been pressed.

`ID:` Will be set to the identifier of the display that has received this key stroke.

Note that officially, `OnKeyDown` only supports control keys, numerical keys and English alphabet keys. To listen to non-English keys, use the `VanillaKey` event handler instead. `VanillaKey` supports the complete Unicode range of keys. Note that `OnKeyDown` supports certain non-English keys on some platforms but this is unofficial behaviour and you shouldn't rely on it. If you need to listen to modifier keys like shift, alt, control, etc. use the `OnRawKeyDown` event handler instead (see below).

`OnKeyUp:` The function you specify here will be called each time the user releases a control key, a numerical key, or an English alphabetical key. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnKeyUp`.

`Key:` Will be set to the key which has been released.

`ID:` Will be set to the identifier of the display that has received this key release.

Note that officially, `OnKeyUp` only supports control keys, numerical keys and English alphabet keys. To listen to non-English keys, use the `VanillaKey` event handler instead. `VanillaKey` supports the complete Unicode range of keys. Note that `OnKeyUp` supports certain non-English keys on some platforms but this is unofficial behaviour and you shouldn't rely on it. If you need to listen to modifier keys like shift, alt, control, etc. use the `OnRawKeyUp` event handler instead (see below).

`OnRawKeyDown:`

This event handler can be used to listen to raw key events. The difference between raw key events and normal key events is that raw key events always deliver the raw key without applying any potential modifier keys like shift, alt, control, etc. that might be down as well. For example, when pressing the shift key and the `"1"` key on an English keyboard, `OnKeyDown` will report that the `"!"` key has been pressed whereas `OnRawKeyDown` will report two key events: It will first report that the shift key has been pressed and then it will report that the `"1"` key has been pressed. In contrast to `OnKeyDown`, `OnRawKeyDown` will never combine the shift and the `"1"` key into the `"!"` key. Instead, you will get the raw key events. Also, `OnKeyDown` will never be triggered if a modifier key like shift, alt, control, etc. has been pressed on its own. `OnRawKeyDown`, however, will also be triggered when the user presses a modifier key. Thus, `OnRawKeyDown` is very useful for listening to modifier keys or combinations of character keys and modifier keys, e.g. you can use this event handler to find out if the right alt key and a character key are both down. The function you pass to this event handler will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnRawKeyDown`.

`Key:` Will be set to the key which has been pressed. See Section 29.22 [Raw keys], page 579, for a list of raw keys supported by Hollywood.

`Modifiers:`

Will be set to a combination of modifier keys that are currently down. The following modifier key flags may be set:

`#MODLSHIFT`
: Left shift key

`#MODRSHIFT`
: Right shift key

`#MODLALT` Left alt key

`#MODRALT` Right alt key

`#MODLCOMMAND`
: Left command key

`#MODRCOMMAND`
: Right command key

`#MODLCONTROL`
: Left control key

`#MODRCONTROL`
: Right control key

`ID:` Will be set to the identifier of the display that has received this key stroke.

(V7.1)

`OnRawKeyUp:`
: This event handler will be triggered whenever a raw key has been released. Please see the description of `OnRawKeyDown` above to find out more about the difference between normal key events and raw key events. The function you pass to this event handler will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnRawKeyUp`.

`Key:` Will be set to the key which has been released. See Section 29.22 [Raw keys], page 579, for a list of raw keys supported by Hollywood.

`Modifiers:`
: Will be set to a combination of modifier keys that are currently down. See above in `OnRawKeyDown` for a list of modifier key flags.

`ID:` Will be set to the identifier of the display that has received this key release.

(V7.1)

`VanillaKey:`
: The function you specify here will be called each time the user presses a key or a key combination that results in character that has a graphical representation, including the SPACE character. `VanillaKey` supports the whole Unicode range of characters and it can also handle characters that are generated by multiple key presses, e.g. diacritical characters. This is the event handler to use if your application should be able to handle non-English characters as well. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `VanillaKey`.

`Key:` Will be set to the key which has been pressed.

`ID:` Will be set to the identifier of the display that has received this key stroke.

Note that `VanillaKey` only reports key down and key repeat events. It cannot report key up events because this isn't possible for characters generated by multiple key strokes. Additionally, it will only report printable characters (including the SPACE character). If you need to listen to control keys like ESC, backspace, cursor keys, etc., use the `OnKeyDown` and `OnKeyUp` event handlers. (V7.0)

`OnMouseMove:`

The function you specify here will be called each time the user moves the mouse. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMouseMove`.

`X,Y:` Will be set to the current mouse pointer position. If they are negative, the mouse pointer is outside the display's boundaries.

`ID:` Will be set to the identifier of the display that received this mouse event.

`OnMouseDown:`

The function you specify here will be called each time the user presses the left mouse button. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMouseDown`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V3.1)

`OnMouseUp:`

The function you specify here will be called each time the user releases the left mouse button. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMouseUp`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V3.1)

`OnRightMouseDown`

The function you specify here will be called each time the user presses the right mouse button. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnRightMouseDown`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V3.1)

`OnRightMouseUp:`

The function you specify here will be called each time the user releases the right mouse button. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnRightMouseUp`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V3.1)

`OnWheelDown:`

The function you specify here will be called each time the user moves the mouse wheel in downward direction. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnWheelDown`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V4.0)

`OnWheelUp:`

The function you specify here will be called each time the user moves the mouse wheel in upward direction. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnWheelUp`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V4.0)

`OnMusicEnd:`

The function you specify here will be called each time a music object has finished playing. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMusicEnd`.

`ID:` Identifier of the music object that has stopped.

Please note that this event is only triggered when the music has finished playing. It is not triggered when you call `StopMusic()`.

`OnSampleLoop:`

The function you specify here will be called each time a sample is started. If the sample is only played once, the function will only get called once. If the sample is playing in loop mode, the function you specify will be called each time Hollywood repeats the sample. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnSampleLoop`.

`ID:` Identifier of the sample that was just started or repeated.

`Time:` The time in milliseconds that the sample has been playing now.

`Starts:` The number of times this sample was started. This field starts at 1 and will be increased each time Hollywood loops your sample.

Attention! Use this event handler with care. If you have a short sample that is looped infinitely, your callback function will get called again and again which will kill your script's performance. This event handler allows you to achieve exact timing with sample playback.

`OnSampleEnd:`
The function you specify here will be called each time a sample has finished playing. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnSampleEnd`.

`ID:` Identifier of the sample that has stopped.

`Time:` The time in milliseconds that the sample was playing.

`Starts:` The number of times this sample was started. This field starts at 1 and will be increased each time Hollywood loops your sample.

Please note that this event is only triggered when the sample has finished playing. It is not triggered when you call `StopSample()`.

`OnARexx:` The function you specify here will be called each time a new ARexx message arrives at the Rexx port created with `CreateRexxPort()`. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnARexx`.

`Command:` A string with the command that shall be executed.

`Args:` A preparsed string which contains all the arguments for this command delimited by `NULL` (`"\0"`) characters. You can use the `SplitStr()` function to extract the individual arguments.

`ArgC:` The number of arguments in the `Args` string.

`RawArgs:` The unparsed string containing all the arguments.

Please note that Hollywood might not always separate the arguments in the way you want to have them. In that case, you can use the `RawArgs` field to access the arguments in their original format just as Hollywood has received them. The `Args` and `ArgC` fields are included just for your convenience but some advanced users might sometimes prefer to use `RawArgs` instead.

See Section 16.3 [CreateRexxPort], page 166, for an example of this event handler. (V2.5)

`SizeWindow:`
The function you specify here will be called each time the user resizes the window. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `SizeWindow`.

`Width:` New width of the window

`Height:` New height of the window

`ID:` Will be set to the identifier of the display that has been sized.

`MoveWindow:`
The function you specify here will be called each time the user moves the window. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `MoveWindow`.

`X,Y:` New position of the window on the host screen.

`ID:` Will be set to the identifier of the display that has been moved.

`CloseWindow:`
This function will be called everytime the user presses the close box of the window. You can use this to pop up a requester which asks the user if he really wants to quit. Message fields:

`Action:` Initialized to `CloseWindow`.

`ID:` Will be set to the identifier of the display whose close box has been clicked.

`ActiveWindow:`
The function you specify here will be called each time the Hollywood window becomes active. Message fields:

`Action:` Initialized to `ActiveWindow`.

`ID:` Will be set to the identifier of the display that has been activated.

`InactiveWindow:`
This function will be called everytime the Hollywood window becomes inactive. Message fields:

`Action:` Initialized to `InactiveWindow`.

`ID:` Will be set to the identifier of the display that has lost the focus.

`ShowWindow:`
The function you specify here will be called each time the user brings the hidden Hollywood window back to the screen. Message fields:

`Action:` Initialized to `ShowWindow`.

`ID:` Will be set to the identifier of the display that has returned from minimized mode.

(V3.0)

`HideWindow:`
This function will be called every time the Hollywood window is hidden by the user. Message fields:

`Action:` Initialized to `HideWindow`.

`ID:` Will be set to the identifier of the display that has been minimized.

(V3.0)

`ModeSwitch:`
This function is called every time the user switches the current display mode by pressing the CMD+RETURN (LALT+RETURN on Windows) hotkey. Message fields:

`Action:` Initialized to `ModeSwitch`.

`Mode:` Display mode that Hollywood switched into (can be `#DISPMODE_WINDOWED` or `#DISPMODE_FULLSCREEN`)

`ID:` Display which handled the pressed hotkey.

`Width:` Display width. (V6.0)

`Height:` Display height. (V6.0)

(V4.5)

`OnDropFile:`
This function is called every time the user drops one or multiple icons onto a display. The following fields will be available to your function:

`Action:` Initialized to `OnDropFile`.

`ID:` Identifier of the display over which the files were dropped.

`NumDropFiles:`
The number of files the user dropped over your display. This is usually 1.

`DropFiles:`
A table containing the list of files that were dropped of the display. This table will have exactly `NumDropFiles` entries.

`X,Y:` Contains the position relative to the top left corner of the receiving display over which the files have been dropped.

(V4.5)

`ClipboardChange:`
This function is called every time the contents of the clipboard changes. This is useful to enable/disable paste functionality in your script. Message fields:

`Action:` Initialized to `ClipboardChange`.

`ID:` Display which received this event.

(V4.5)

`OnMidMouseDown:`
The function you specify here will be called each time the user presses the middle mouse button. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMidMouseDown`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V4.5)

`OnMidMouseUp:`
The function you specify here will be called each time the user releases the middle mouse button. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMidMouseUp`.

`ID:` Will be set to the identifier of the display that received this mouse event.

(V4.5)

`OnConnect:`

The function you specify here will be called each time a new client connects to a server created using the `CreateServer()` call. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnConnect`.

`ClientID:`
Identifier of the client that has connected itself to the server.

`ServerID:`
Identifier of the server that the client has connected to.

The `ClientID` is important and you should store it somewhere because you will need it to communicate with the client. You can also use this id to find out the IP address and port number of the client using the commands `GetConnectionIP()` and `GetConnectionPort()`. You can also send data to the client by using `SendData()`. (V5.0)

`OnDisconnect:`

The function you specify here will be called each time a client disconnects from a server created using the `CreateServer()` call. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnDisconnect`.

`ID:` Identifier of the client that has disconnected from the server. You need to call `CloseConnection()` on this ID to remove the client from your server.

When you get this event, do not forget to call `CloseConnection()` on the client ID to fully disconnect the client from your server. This is very important. (V5.0)

`OnReceiveData:`

The function you specify here will be called each time new data is received by an existing connection. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnReceiveData`.

`ID:` Identifier of the connection that has new data.

When you receive this message it means that there is new data available in the connection specified by `ID`, and that you should call `ReceiveData()` now to read this data from the network buffer. (V5.0)

`OnReceiveUDPData:`

The function you specify here will be called each time new data is received by an existing UDP connection. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnReceiveUDPData`.

`ID:` Identifier of the connection that has new data.

When you receive this message it means that there is new data available in the connection specified by `ID`, and that you should call `ReceiveUDPData()` now to read this data from the network buffer. (V5.0)

`OnVideoEnd:`

The function you specify here will be called each time a video has finished playing. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnVideoEnd`.

`ID:` Identifier of the video that has stopped.

Please note that this event is only triggered when the video has finished playing. It is not triggered when you call `StopVideo()`. (V5.0)

`FillMusicBuffer:`

The function you specify here will be called each time the sound server needs more audio data when a dynamic music created using `CreateMusic()` is playing. Your callback will then have to call `FillMusicBuffer()` to feed more audio data to the device. The function you specify here will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `FillMusicBuffer`.

`ID:` Identifier of the music object that has caused this event.

`Samples:` Contains the number of PCM frames Hollywood is requesting from your callback. Your callback must provide this amount of PCM frames to Hollywood so that it can forward this audio data to the system's audio device. See Section 49.7 [FillMusicBuffer], page 978, for details.

`Count:` Contains a global count of how many PCM frames have been sent to the audio device already. Useful for keeping track of how many seconds the music has already been playing.

(V5.0)

`OrientationChange:`

The function you specify here will be called each time the user changes the orientation of his mobile device. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OrientationChange`.

`ID:` Identifier of the display.

`Orientation:`

Constant specifying the new orientation of the device. This will be one of the following constants:

```
#ORIENTATION_PORTRAIT
#ORIENTATION_LANDSCAPE
```

```
#ORIENTATION_PORTRAITREV
#ORIENTATION_LANDSCAPEREV
```

Please note that this event handler is only supported in the mobile version of Hollywood. (V5.0)

`ShowKeyboard:`
The function you specify here will be called each time the software keyboard becomes visible on mobile devices. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `ShowKeyboard`.

`ID:` Identifier of the display.

Please note that this event handler is only supported in the mobile version of Hollywood. (V5.0)

`HideKeyboard:`
The function you specify here will be called each time the software keyboard becomes invisible on mobile devices. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `HideKeyboard`.

`ID:` Identifier of the display.

Please note that this event handler is only supported in the mobile version of Hollywood. (V5.0)

`OnUserMessage:`
The function you specify here will be called each time a new user message arrives at the message port created with `CreatePort()`. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnUserMessage`.

`Command:` A string with the command that shall be executed.

`Args:` A preparsed string which contains all the arguments for this command delimited by `NULL` (`"\0"`) characters. You can use the `SplitStr()` function to extract the individual arguments.

`ArgC:` The number of arguments in the `Args` string.

`RawArgs:` The unparsed string containing all the arguments.

Please note that Hollywood might not always separate the arguments in the way you want to have them. In that case, you can use the `RawArgs` field to access the arguments in their original format just as Hollywood has received them. The `Args` and `ArgC` fields are included just for your convenience but some advanced users might sometimes prefer to use `RawArgs` instead.

See Section 32.1 [CreatePort], page 639, for an example of this event handler. (V5.0)

`Hotkey:` The function you specify here will be called each time the user presses the key combination specified in the `-cxkey` argument. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `Hotkey`.

`ID:` Identifier of the display.

Please note that this event handler is only supported in the AmigaOS compatible versions of Hollywood. (V5.2)

`TrayIcon:`

If you have called the `SetTrayIcon()` function to install an icon into the system tray, the function you specify here will be called each time the user clicks on that icon. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `TrayIcon`.

`ID:` Identifier of the display.

Please note that this event handler is only supported in the Microsoft Windows version of Hollywood. (V5.2)

`OnTouch:` The function you specify here will be called each time the touch screen detects a user interaction such as putting a new finger on the touch screen, lifting a finger, or moving it. This event is only required if you need fine-tuned control over all touch events, e.g. for supporting multi-touch events or gestures. If you only need simple control over the touch screen, it is easier to use the `OnMouseDown`, `OnMouseUp`, and `OnMouseMove` event handlers. Hollywood will always map the primary finger on the touch screen to the left mouse button.

The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnTouch`.

`ID:` Will be set to the identifier of the display that received this mouse event.

`Type:` This is set to a string describing the type of the touch event. This will be set to either `Down`, `Up`, or `Move`.

`Finger:` The finger number this event refers to.

`X:` The current X coordinate of the finger on the touch screen.

`Y:` The current Y coordinate of the finger on the touch screen.

`Pressure:`

Contains the current pressure of the finger. This generally ranges from 0 (no pressure at all) to 1 (normal pressure). Values greater than 1 are also possible, dependingt on the calibration of the touch screen.

`DownTime:`

Contains the time when the finger was put down on the touch screen.

`EventTime:`
Contains the time when the event was generated.

`Size:` Contains the scaled value of the approximate size for the current finger.

`TouchMajor:`
Contains the current length of the major axis of an ellipse describing the touch area for the current finger.

`TouchMinor:`
Contains the current length of the minor axis of an ellipse describing the touch area for the current finger.

`ToolMajor:`
Contains the current length of the major axis of an ellipse describing the tool area for the current finger.

`ToolMinor:`
Contains the current length of the minor axis of an ellipse describing the tool area for the current finger.

`Orientation:`
Contains the current orientation of the touch and tool areas in radians running clockwise from vertical for the current finger. The range is from -PI/2 radians (finger is fully left) to PI/2 radians (finger is fully right).

Please note that this event handler is only supported in the mobile version of Hollywood. (V5.3)

`OnApplicationMessage:`
The function you specify here will be called each time a new message sent through AmigaOS 4's application.library messaging system arrives. If you want to be able to receive application.library messages, you need to have set the `RegisterApplication` tag in `@OPTIONS` to `True` first. The function you specify here will be passed a table as parameter 1 with the following fields initialized:

`Action:` Initialized to `OnApplicationMessage`.

`Sender:` The name of the application that has sent this message.

`Message:` The actual message.

Please note that this event handler is only available on AmigaOS 4. (V6.0)

`OnDockyClick:`
If you have set `RegisterApplication` to `True` in the `@OPTIONS` preprocessor command, the function you specify here will be called every time the user clicks on your application's icon in AmiDock. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnDockyClick`.

`ID:` Identifier of the display.

Please note that this event handler is only supported in the AmigaOS 4 version of Hollywood. (V6.0)

`OnMenuSelect:`
The function you specify here will be called each time the user selects a menu item. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnMenuSelect`.

`ID:` Identifier of the display that the menu strip has been attached to.

`Item:` The identifier of the menu item that has been selected. See Section 39.8 [MENU], page 804, for details.

`Selected:`
If the menu item that has been selected is a toggle menu item (i.e. a menu item that can have two different states), this field will contain the current toggle state. See Section 39.8 [MENU], page 804, for details.

(V6.0)

`RunFinished:`
The function you specify here will be called whenever a program executed asynchronously using the `Run()` command has terminated. The function will receive a message as parameter 1 with the following fields initialized:

`Action:` Initialized to `RunFinished`.

`Program:` This is set to a string that contains the name of the program that was launched using `Run()`.

`Args:` This is set to a string that contains the arguments that were passed to the program launched using `Run()`.

`RunUserData:`
If custom user data is specified in the call to `Run()`, it will be passed on to your callback in this message field. If you don't pass any custom user data, this field won't be initialized at all.

`ReturnCode:`
This tag will only be set if the eponymous tag has been set to `True` when calling `Run()`. In that case, `ReturnCode` will contain the program's return code when it terminates. (V9.0)

(V6.1)

`DirectoryChanged:`
If you are currently monitoring directories using the `MonitorDirectory()` function, the function you specify here will be called whenever a change

inside a monitored directory occurs. The function will receive a message as parameter 1 with the following fields initialized:

`Action:` Initialized to `DirectoryChanged`.

`ID:` Identifier of the directory object in which the change occurred.

`Directory:`
This will be set to a string that contains a fully-qualified path of the directory in which the change occurred.

`MonitorUserData:`
If custom user data was specified in the call to `MonitorDirectory()`, it will be passed to your callback in this message field. If you don't pass any custom user data, this field won't be initialized at all.

`Type:` The type of change. This will only be set if the `ReportChanges` tag has been set to `True` in the call to `MonitorDirectory()`. If that is the case, this will be one of the following types:

`#DIRMONITOR_ADD:`
The file or directory in the `Name` tag has been added to the directory.

`#DIRMONITOR_REMOVE:`
The file or directory in the `Name` tag has been removed from the directory.

`#DIRMONITOR_CHANGE:`
The file or directory in the `Name` tag has been changed in the directory.

(V9.0)

`Name:` This contains the name of the file or directory that has been added, removed, or changed, depending on the value in the `Type` tag (see above). Note that `Name` will only be set if the `ReportChanges` tag has been set to `True` in the call to `MonitorDirectory()`. (V9.0)

(V8.0)

`OnAccelerometer:`
The function you specify here will be called each time the device's sensor reports new accelerometer values. Note that this will typically happen all the time so if you listen to this event, you will get lots of events that can impact the performance of your script.

The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnAccelerometer`.

`ID:` Will be set to the identifier of the display that received this event.

`X:` X accelerometer value from sensor.

`Y:` Y accelerometer value from sensor.

`Z:` Z accelerometer value from sensor.

Please note that this event handler is only supported in the Android version of Hollywood. To learn how to interpret the `X`, `Y`, and `Z` values provided by this event handler, please consult the Android documentation of `SensorEvent`. (V8.0)

`OnGyroscope:`

The function you specify here will be called each time the device's sensor reports new gyroscope values. Note that this will typically happen all the time so if you listen to this event, you will get lots of events that can impact the performance of your script.

The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `OnGyroscope`.

`ID:` Will be set to the identifier of the display that received this event.

`X:` X gyroscope value from sensor.

`Y:` Y gyroscope value from sensor.

`Z:` Z gyroscope value from sensor.

Please note that this event handler is only supported in the Android version of Hollywood. To learn how to interpret the `X`, `Y`, and `Z` values provided by this event handler, please consult the Android documentation of `SensorEvent`. (V8.0)

`RunOutput:`

The function you specify here will be called whenever a program executed asynchronously using the `Run()` command writes data to the console. This data will be redirected to your program and you can process it using this event handler. This makes it possible to capture a program's output. The function will receive a message as parameter 1 with the following fields initialized:

`Action:` Initialized to `RunOutput`.

`Program:` This is set to a string that contains the name of the program that was launched using `Run()`.

`Args:` This is set to a string that contains the arguments that were passed to the program launched using `Run()`.

`Output:` This is set to a string that contains the program's output. Note that this can be of any arbitrary length. Do not assume `Output` to always be a complete line of the program's output, it can also be half a line with the other half delivered the next time the event handler triggers. Depending on the way the program

writes output to the console, it could even be delivered to you character by character so make no assumptions on the actual format of `Output`. The string passed in `Output` will be in UTF-8 format by default. See below how this can be changed.

`RunUserData:`
If custom user data is specified in the call to `Run()`, it will be passed on to your callback in this message field. If you don't pass any custom user data, this field won't be initialized at all.

Note that by default, the `RunOutput` event handler expects programs to output text only. This is why `RunOutput` will make sure to pass only properly UTF-8 encoded text to your callback function. If you don't want `RunOutput` to format the text as UTF-8, you need to set the `RawMode` argument to `True` when calling `Run()`. In that case, `RunOutput` won't do any preformatting and will just forward the program's raw output to you. This means that your event handler callback has to be ready to process binary data as well.

Also note that output will normally not be delivered in real-time because console output is typically buffered. However, if the external program continually outputs text it will arrive pretty instantly at your `RunOutput` event handler callback.

(V9.0)

`ShowSystemBars:`
The function you specify here will be called each time the system bars become visible again when a display is in immersive mode. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `ShowSystemBars`.

`ID:` Will be set to the identifier of the display that received this event.

Please note that this event handler is only supported in the Android version of Hollywood. (V9.0)

`HideSystemBars:`
The function you specify here will be called each time the system bars are hidden when a display is in immersive mode. The function will receive a message as parameter 1 with the following fields:

`Action:` Initialized to `HideSystemBars`.

`ID:` Will be set to the identifier of the display that received this event.

Please note that this event handler is only supported in the Android version of Hollywood. (V9.0)

If you want to remove an event handler, simply set the corresponding field in the table to 0 instead of passing a function.

Starting with Hollywood 3.1 there is an optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful

if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

Starting with Hollywood 7.0 all event messages will contain an additional field named `Timestamp`. This field contains the time stamp when the event was generated. This time stamp is passed in seconds as a fractional number. It is relative to the time Hollywood was started. See Section 55.8 [GetTimestamp], page 1163, for details.

Please note that the event handler functions are only executed while the script is in a `WaitEvent()` loop. You have to use `WaitEvent()` for them!

INPUTS

`table` table which contains information about which handlers to install or remove

`userdata` optional: user specific data that will be passed to the callback function (V3.1)

EXAMPLE

```
Function p_HandlerFunc(msg)
  Switch(msg.action)
  Case "ActiveWindow":
    DebugPrint("Window has become active again!")
  Case "InactiveWindow":
    DebugPrint("Window has become inactive!")
  Case "MoveWindow":
    DebugPrint("User has moved the window to", msg.x, msg.y)
  Case "OnKeyDown":
    If msg.key = "ESC" Then End
  EndSwitch
EndFunction

InstallEventHandler({ActiveWindow = p_HandlerFunc,
   InactiveWindow = p_HandlerFunc,
   MoveWindow = p_HandlerFunc,
   OnKeyDown = p_HandlerFunc})

Repeat
  WaitEvent
Forever
```

The code above installs four event handlers for `ActiveWindow`, `InactiveWindow`, `MoveWindow` and `OnKeyDown`. If the user presses escape, the program will quit. If you want to remove e.g. the event handler `MoveWindow`, just call `InstallEventHandler()` again with the parameter `MoveWindow=0`.

29.14 IsKeyDown

NAME

IsKeyDown – check if a key is pressed (V1.5)

SYNOPSIS

```
state = IsKeyDown(key$[, rawkey])
```

FUNCTION

This function checks if the key specified by `key$` is currently pressed. If it is, this function will return `True` otherwise it will return `False`.

`key$` is a string representing a key on your keyboard. This can be one of the following control keys:

`UP`	Cursor up
`DOWN`	Cursor down
`RIGHT`	Cursor right
`LEFT`	Cursor left
`HELP`	Help key
`DEL`	Delete key
`BACKSPACE`	Backspace key
`TAB`	Tab key
`RETURN`	Return key
`ENTER`	Enter key
`ESC`	Escape
`SPACE`	Space key
`F1 - F16`	Function keys
`INSERT`	Insert key
`HOME`	Home key
`END`	End key
`PAGEUP`	Page up key
`PAGEDOWN`	Page down key
`PRINT`	Print key
`PAUSE`	Pause key

Alternatively, `key$` can also be a character from the English alphabet, e.g. `"A"`, or a string containing a number from 0 to 9. Note that `IsKeyDown()` doesn't support Unicode keys.

Starting with Hollywood 4.0, you can check the status of the modifier keys, too. The following modifier keys can be checked using `IsKeyDown()`:

`LSHIFT`	Left shift key
`RSHIFT`	Right shift key

`LALT`	Left alt key
`RALT`	Right alt key
`LCOMMAND`	Left command key
`RCOMMAND`	Right command key
`LCONTROL`	Left control key
`RCONTROL`	Right control key

Starting with Hollywood 6.1 you can pass the special string `ANY` in `key$` to check for an arbitrary key to be pressed.

Starting with Hollywood 7.1 there is an optional argument `rawkey`. If this argument is set to `True`, `IsKeyDown()` will treat `key$` as a raw key and check if it is down. In that case, `key$` must be one of the raw keys defined by Hollywood. See Section 29.22 [Raw keys], page 579, for details. The difference between normal keys and raw keys is described in the documentation of the `OnRawKeyDown` event handler. See Section 29.13 [InstallEventHandler], page 553, for details.

INPUTS

`key$`	key to check
`rawkey`	optional: `True` if `key$` is a raw key (defaults to `False`) (V7.1)

RESULTS

`state`	`True` if key$ is pressed, `False` otherwise

EXAMPLE

```
Print("Press F1 please.")
Repeat
  VWait
Until IsKeyDown("F1") = True
```

The above code waits until the F1 key is pressed. (you can have that easier by using `WaitKeyDown()`; the one above is only useful if you want to do something while the key is not pressed)

29.15 IsLeftMouse

NAME

IsLeftMouse – check if the left mouse button is pressed

SYNOPSIS

```
pressed = IsLeftMouse()
```

FUNCTION

This function returns `True` if the left mouse button is currently pressed, otherwise `False`.

INPUTS

none

EXAMPLE

```
Repeat
  Wait(2)
Until IsLeftMouse() = True
```

The above code waits until the left mouse button is pressed. (you can have that easier by using `WaitLeftMouse()`; the one above is only useful if you want to do something while the mouse button is not pressed)

29.16 IsMidMouse

NAME

IsMidMouse – check if the middle mouse button is pressed (V4.5)

SYNOPSIS

```
pressed = IsMidMouse()
```

FUNCTION

This function returns `True` if the middle mouse button is currently pressed, otherwise `False`.

INPUTS

none

EXAMPLE

```
Repeat
  Wait(1)
Until IsMidMouse() = True
```

The above code waits until the middle mouse button is pressed. (you can have that easier by using `WaitMidMouse()`; the one above is only useful if you want to do something while the mouse button is not pressed)

29.17 IsRightMouse

NAME

IsRightMouse – check if the right mouse button is pressed (V1.5)

SYNOPSIS

```
pressed = IsRightMouse()
```

FUNCTION

This function returns `True` if the right mouse button is currently pressed, otherwise `False`.

INPUTS

none

EXAMPLE

```
Repeat
  Wait(2)
```

```
Until IsRightMouse() = True
```

The above code waits until the right mouse button is pressed. (you can have that easier by using `WaitRightMouse()`; the one above is only useful if you want to do something while the mouse button is not pressed)

29.18 LeftMouseQuit

NAME

LeftMouseQuit – enable/disable left mouse quit

SYNOPSIS

```
LeftMouseQuit(enable)
```

FUNCTION

If you set `enable` to `True`, your script will be terminated as soon as the user presses the left mouse button.

INPUTS

`enable` `True` to enable left mouse quit, `False` to disable it

EXAMPLE

```
LeftMouseQuit(TRUE)
Repeat
  Wait(10)
Forever
```

The above code enters an endless loop which would normally block your program. But using `LeftMouseQuit(TRUE)` allows the user to terminate it.

29.19 MakeButton

NAME

MakeButton – create a new button (V2.0)

SYNOPSIS

```
[id] = MakeButton(id, #LAYERBUTTON, layerid, t[, userdata])
[id] = MakeButton(id, #SIMPLEBUTTON, x, y, width, height, t[, userdata])
```

DEPRECATED SYNTAX

```
[id] = MakeButton(id, #LAYERBUTTON, layerid, exactcoll, noautohide, t
                      [, userdata]) (V2.5)
```

FUNCTION

This function creates a new button and attaches it to the current BGPic. The button will be given the identifier specified by `id`, or, if you pass `Nil` as `id`, `MakeButton()` will automatically choose an identifier for you. The argument `type` specifies the type of the button. Currently, the following types are supported:

`#SIMPLEBUTTON`:

Creates a standard button. Note that this button is invisible. You need to use graphics in your BGPic to show the user where your button is. The type

`#SIMPLEBUTTON` requires you pass the position and dimensions for the button in argument 3 to 6. The seventh argument is a table argument allowing you to specify the callback functions to be invoked when the button receives an event (see below).

`#LAYERBUTTON:`
This type is new in Hollywood 2.5. It will create a dynamic button which will always share the position and size of the layer specified by `layerid`. Note that layer buttons are closely tied to their layer. Thus, when the layer is deleted, the button will also be deleted. Layer buttons also support some additional options in the table argument accepted by `MakeButton()` (see below for details).

The `t` argument must be a table which specifies the functions that shall be called when a specific event occurs. It can also be used to configure some advanced button options. The table can contain the following fields:

`OnMouseOver:`
The function specified here will be called when the user moves the mouse over the button's area.

`OnMouseOut:`
The function specified here will be called when the mouse pointer leaves the area occupied by this button.

`OnMouseDown:`
The function specified here will be called when the user presses the left mouse button while the pointer is over the button's area.

`OnMouseUp:`
The function specified here will be called when the user releases the left mouse button while the pointer is over the button's area. This event will only be triggered if the user also pressed the left mouse button while the pointer was over the button's area.

`OnRightMouseDown:`
Same as `OnMouseDown` but with the right mouse button.

`OnRightMouseUp:`
Same as `OnMouseUp` but with the right mouse button.

`OnMidMouseDown:`
Same as `OnMouseDown` but with the middle mouse button. (V4.5)

`OnMidMouseUp:`
Same as `OnMouseUp` but with the middle mouse button. (V4.5)

`PixelExact:`
This is only supported for `#LAYERBUTTON`. It specifies whether or not your button should use pixel-exact or rectangular collision detection. If you pass `True` here, events will only be triggered on a pixel-exact collision. This can be useful for irregularly shaped buttons. Defaults to `False`.

`NoAutoHide:`
: This is only supported for `#LAYERBUTTON`. It specifies whether or not the button shall automatically be hidden when the layer is hidden. If you specify `True` here, the button will not automatically disappear when you hide the layer it is attached to. If `NoAutoHide` is `False`, the button will disappear as soon as you hide the layer. Defaults to `False`.

`ZOrder:`
: This is only supported for `#LAYERBUTTON`. It specifies whether or not the layer z-order should be respected when handling events of overlapping layer buttons. Historically, overlapping layer button events were handled in the order of their creation, i.e. if a layer button was created before another layer button and both layers overlapped, the layer button created earlier would receive the events even if it was below the other layer button. By setting the `ZOrder` tag to `True`, you can force Hollywood to handle events of overlapping layer buttons in their stacking order, i.e. layers on top of other layers will receive their events first. For compatibility reasons, this tag defaults to `False`. (V9.0)

If you just want to listen to mouse clicks on a button, it is enough to provide a callback function for the `OnMouseUp` event type. `OnMouseDown` is only required, if you want to highlight the button in some way while the user clicks on it.

Starting with Hollywood 3.1 there is an optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

The callback functions you specify in the event table will be called by Hollywood with one parameter. This parameter is a message table that contains some information about the event. The following fields are provided:

`Action:`
: Contains the name of the event triggered, e.g. `OnMouseUp`, `OnMouseOver` or `OnMouseOut`. This field is a string.

`ID:`
: Contains the identifier of the button that triggered this event. This field is a number.

`X, Y, Width, Height:`
: These fields contain the dimensions of the button that triggered this event.

`MouseDown:`
: This field will be set to `True` if the left mouse button is currently pressed. Useful in connection with the `OnMouseOver` event, so that you can display a differently highlighted version of the button if the user moves the pointer of the button while the left mouse button is pressed.

`RightMouseDown:`
: Same as `MouseDown` but relating to the right mouse button.

`MidMouseDown:`
: Same as `MouseDown` but relating to the middle mouse button. (V4.5)

`Layer:` The identifier of the layer this button is bound to. This is only set for buttons of type `#LAYERBUTTON`. (V4.5)

`LayerName:`
The name of the layer this button is bound to. This is only set for buttons of type `#LAYERBUTTON`. (V4.5)

`UserData:`
Contains the value that you have passed in the `userdata` argument when you created the button.

`Timestamp:`
Contains a time stamp that indicates when the event occurred. See Section 55.8 [GetTimestamp], page 1163, for details. (V7.0)

The advantage of this message is that you can use one and the same function for all your buttons and all events. Just check the fields `Action` and `ID` to find out which button caused the event.

Please note: Buttons are always attached to BGPics. So if you call `MakeButton()` when BGPic 1 is displayed, the button will be attached to BGPic 1. If you display BGPic 2 then, the buttons will go away. Once you switch back to BGPic 1, its buttons will also be re-activated.

Also note that you can only create rectangularly shaped buttons with `#SIMPLEBUTTON`. If you want to have irregularly shaped buttons, create a layer that has a transparency setting (masked or alpha channelled transparency) and then use `#LAYERBUTTON` to attach a button to this layer.

INPUTS

`id` identifier for the new button or `Nil` for auto id selection

`type` type of the new button

`x` x position of the new button

`y` y position of the new button

`width` width of the new button

`height` height of the new button

`t` table that contains callback functions that Hollywood shall call if a specific event occurs and additional arguments (see above)

`userdata` optional: user specific data that will be passed to the callback function (V3.1)

RESULTS

`id` optional: identifier of the button; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
Function p_MyFunc(msg)
  Switch msg.action
  Case "OnMouseUp":
```

```
    DebugPrint("User left-clicked button", msg.id)
  Case "OnMouseOver":
    DebugPrint("User moved mouse over button", msg.id)
  Case "OnRightMouseUp":
    DebugPrint("User right-clicked button", msg.id)
  EndSwitch
EndFunction

; draw the buttons
Box(0, 0, 100, 100, #RED)
Box(200, 200, 100, 100, #BLUE)

; install them
evtmatch = {OnMouseUp = p_MyFunc, OnMouseOver = p_MyFunc,
            OnRightMouseUp = p_MyFunc}
MakeButton(1, #SIMPLEBUTTON, 0, 0, 100, 100, evtmatch)
MakeButton(2, #SIMPLEBUTTON, 200, 200, 100, 100, evtmatch)

Repeat
  WaitEvent
Forever
```

29.20 MouseX

NAME

MouseX – return x-position of mouse (V1.5)

SYNOPSIS

```
pos = MouseX()
```

FUNCTION

This function returns the current x-position of the mouse pointer.

INPUTS

none

RESULTS

`pos` mouse pointer's x-position

29.21 MouseY

NAME

MouseY – return y-position of mouse (V1.5)

SYNOPSIS

```
pos = MouseY()
```

FUNCTION

This function returns the current y-position of the mouse pointer.

INPUTS
none

RESULTS

`pos`	mouse pointer's y-position

29.22 Raw keys

Starting with version 7.1 Hollywood supports raw key events. The following raw keys are currently supported by Hollywood:

`A-Z`	Alphabetical keys. Note that these will always be in upper case because they are raw keys and no modifier keys are applied to raw keys.
`0-9`	Numerical keys
`UP`	Cursor up
`DOWN`	Cursor down
`RIGHT`	Cursor right
`LEFT`	Cursor left
`HELP`	Help key
`DEL`	Delete key
`BACKSPACE`	Backspace key
`TAB`	Tab key
`RETURN`	Return key
`ENTER`	Enter key
`ESC`	Escape
`SPACE`	Space key
`F1-F16`	Function keys
`INSERT`	Insert key
`HOME`	Home key
`END`	End key
`PAGEUP`	Page up key
`PAGEDOWN`	Page down key
`PRINT`	Print key
`PAUSE`	Pause key
`NP0-NP9`	Numpad numeric keys
`NPDEC`	Numpad decimal key

`NPADD`	Numpad addition key
`NPSUB`	Numpad subtraction key
`NPMUL`	Numpad multiplication key
`NPDIV`	Numpad division key
`LSHIFT`	Left shift key
`RSHIFT`	Right shift key
`LALT`	Left alt key
`RALT`	Right alt key
`LCOMMAND`	Left command key
`RCOMMAND`	Right command key
`LCONTROL`	Left control key
`RCONTROL`	Right control key

29.23 ResetKeyStates

NAME

ResetKeyStates – reset internal key and mouse states (V4.6)

SYNOPSIS

```
ResetKeyStates()
```

FUNCTION

This function can be used to reset Hollywood's internal key and mouse states. This is a low-level function and you normally will not need to use this. It is just here for certain emergency situations.

Please note that key states are cached per display. `ResetKeyStates()` will reset the key states of the currently active display.

INPUTS

none

29.24 RunCallback

NAME

RunCallback – push callback into event queue (V9.0)

SYNOPSIS

```
RunCallback(func[, userdata])
```

FUNCTION

This function adds the function specified by `func` to the event queue. The function will be run the next time Hollywood checks the event queue, i.e. when `CheckEvent()` or `WaitEvent()` gets called.

The optional `userdata` argument allows you to specify additional data that will be passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

Your callback function will be called by Hollywood with a single parameter. The parameter is a message table which contains the following fields:

`Action:` Will be always set to `RunCallback`. This field is a string!

`UserData:`
Will be set to what you have specified in the `userdata` argument when calling `RunCallback()`. Note that this field will only be there if you have actually passed a value in `userdata` when calling `RunCallback()`.

INPUTS

`function` function to be added to the event queue

`userdata` optional: user data to be passed to the function when running it

EXAMPLE

```
RunCallback(Function(msg) NPrint(msg.userdata) EndFunction, "Hello 2!")
NPrint("Hello 1!")
Repeat
   WaitEvent
Forever
```

This will first print `"Hello 1!"` and then `"Hello 2!"` because the function that prints `"Hello 2!"` won't be called until Hollywood empties the event queue which happens on `WaitEvent()`.

29.25 SetEventTimeout

NAME

SetEventTimeout – define event cache duration (V1.9)

SYNOPSIS

```
SetEventTimeout(duration)
```

FUNCTION

You can use this function to tell Hollywood for how long it should cache input events. The default value is currently 60000 which means that all events will be cached for 60 seconds. This function can be useful if you have a function that blocks the program flow for some time, e.g. if you do some heavy computing that takes longer than 60 seconds, all events that occurred more than 60 seconds ago will get lost. To prevent that, just increase the event timeout.

Starting with Hollywood 7.0, it is also possible to set `duration` to -1 to disable event timeouts completely. In that case, events will never get lost.

INPUTS

`duration` specifies for how long events will be cached (in milliseconds)

EXAMPLE

```
SetEventTimeout(5000)
```

Sets the event cache duration to 5 seconds (= 5000 milliseconds).

29.26 SetInterval

NAME

SetInterval – install a new interval function (V2.0)

SYNOPSIS

```
[id] = SetInterval(id, func, ms[, userdata])
```

FUNCTION

This function installs a new interval function and assigns the identifier `id` to it. If you pass `Nil` in `id`, `SetInterval()` will automatically choose an identifier and return it. You need to specify a Hollywood function in the `func` argument and the time in milliseconds that defines the interval. The function you specified will then be called again and again and again at the intervals of the specified time. For example if you specify 40 as the interval, your function will be called every 40 milliseconds which corresponds to 25 times a second (25 * 40ms = 1000ms = 1 second). This is enough for most games, intros etc.

The function you specify in `func` will be called for the first time when the time specified in `ms` has elapsed. After that your function will be called repeatedly in intervals of `ms` milliseconds.

You always need to use `WaitEvent()` in connection with this function! If you have an interval function installed, the internal Hollywood timer scheduler will trigger interval events and inform `WaitEvent()` to call your interval function. Intervals do not work without `WaitEvent()`! If you do not use `WaitEvent()`, your interval function will never be called. Interval functions are only called as long as you are in a `WaitEvent()` loop.

This function is very important because it helps you to make sure that your script runs at the same speed on every system. See Section 15.3 [Script timing], page 161, for more information on this issue and also for an example.

You can install as many intervals as you want. Hollywood's internal scheduler will make sure that all interval functions are called correctly.

Please remember that Hollywood does not support multithreading. Therefore your interval functions must not block the script - otherwise the whole script will be blocked. For example, if you have two interval functions installed and one of those functions executes a `Wait(100)`, then the whole script will be blocked for 2 seconds.

You can use the `ClearInterval()` call to stop an interval function.

Starting with Hollywood 3.1 there is an optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

Your interval function will be called by Hollywood with one parameter. The parameter is a message table which contains the following fields:

`Action:` Will be always set to `Interval`. This field is a string.

`ID:` Will be set to the identifier of the interval that Hollywood has just called.

`UserData:`
Will be set to what you have specified in the `userdata` argument when you installed the interval.

This message is useful if you want to handle two or more intervals in the same function. The message tells you then which interval Hollywood executed. If you do not need this message, simply disregard it. For example, the interval function of the code above does not regard the message either.

Last but not least: You should really have a look at the examples that came with Hollywood. Many of them use `SetInterval()` to manage the timing of the script!

INPUTS

`id` identifier for the new interval function or `Nil` for auto id selection; the identifier is needed so that you can stop the interval later using the `ClearInterval()` command

`func` Hollywood function that should be called at the intervals of the specified time

`ms` intervals at which the function shall be called, e.g. 40 will call the function 25 times a second because 25 * 40ms = 1000ms = 1 second

`userdata` optional: user specific data to pass to callback function (V3.1)

RESULTS

`id` optional: identifier of the interval; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

See Section 15.3 [script timing], page 161.

29.27 SetTimeout

NAME

SetTimeout – install a new timeout function (V2.0)

SYNOPSIS

```
[id] = SetTimeout(id, func, timeout[, userdata])
```

FUNCTION

This function installs a new timeout function and assigns the identifier `id` to it. If you pass `Nil` in `id`, `SetTimeout()` will automatically choose an identifier and return it. You need to specify a Hollywood function in the `func` argument and a `timeout` in milliseconds. After this time has elapsed, Hollywood will call your timeout function. This is useful if you need exact timing, for example if you want to synchronize graphics with music. Timeout functions are perfect for that.

You always need to use `WaitEvent()` in connection with this function! If you have a timeout function installed, the internal Hollywood timer scheduler will trigger timeout

events and inform `WaitEvent()` to call your timeout function. Timeouts do not work without `WaitEvent()`! If you do not use `WaitEvent()`, your timeout function will never be called. Timeout functions are only called as long as you are in a `WaitEvent()` loop.

You can install as many timeouts as you want. Hollywood's internal scheduler will make sure that all timeout functions are called correctly.

You can use the `ClearTimeout()` call to stop a timeout function.

Starting with Hollywood 3.1 there is an optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

Your timeout function will be called by Hollywood with one parameter. The parameter is a message table which contains the following fields:

`Action:` Will be always set to `Timeout`. This field is a string!

`ID:` Will be set to the identifier of the timeout that Hollywood has just called.

`UserData:`
Will be set to what you have specified in the `userdata` argument when you installed the timeout.

This message is useful if you want to handle two or more timeouts in the same function. The message tells you then which timeout Hollywood executed. If you do not need this message, simply disregard it.

INPUTS

`id` identifier for the new timeout function or `Nil` for auto id selection; the identifier is needed so that you can stop the timeout later using the `ClearTimeout()` command

`func` Hollywood function to be called after the specified time has elapsed

`timeout` time in milliseconds that specifies the timeout

`userdata` optional: user specific data to pass to callback function (V3.1)

RESULTS

`id` optional: identifier of the timeout; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
Function p_TenSeconds()
  SystemRequest("Hollywood", "Ten seconds are over now!", "OK")
EndFunction

SystemRequest("Hollywood", "I will call the function TenSeconds()\n" ..
   "after 10 seconds have elapsed!\nCheck your watch, then click Go!",
   "Let's go!")

SetTimeout(1, p_TenSeconds, 10000)
```

```
Repeat
  WaitEvent
Forever
```

The code above installs a timeout function that will be called after 10000 milliseconds (= 10 seconds) have elapsed.

29.28 WaitEvent

NAME

WaitEvent – wait for an event to occur

SYNOPSIS

```
info = WaitEvent()
```

FUNCTION

The `WaitEvent()` function puts Hollywood in sleep state. The program will be woken up when an event is triggered. In this case `WaitEvent()` will execute the function that you installed for this event and then it will return. Therefore, you always need to use `WaitEvent()` in a loop. For example:

```
While quit = False
   WaitEvent
Wend
```

Or use an endless loop:

```
Repeat
   WaitEvent
Forever
```

`WaitEvent()` is a core function of the Hollywood language and you should use one of the loops presented above in every script as your main loop. `WaitEvent()` has the advantage that it sleeps until an event is triggered. This is important in a multitasking environment because it saves CPU time. Never use polling loops as they will consume all CPU time. If you need to constantly execute code in your main loop, use `SetInterval()` to install an interval function that gets called 25 times a second by `WaitEvent()`.

The functions that `WaitEvent()` will call when an event is triggered can be installed with the following functions of the Hollywood event library: `MakeButton()`, `SetInterval()`, `SetTimeout()`, and `InstallEventHandler()`

Note that `WaitEvent()` must not be called from any callbacks executed by `WaitEvent()`. In general, you should use `WaitEvent()` only once in your script: in your main loop. If you really have to check for events in a callback executed by `WaitEvent()`, use `CheckEvents()` instead, but this should normally be unnecessary.

`WaitEvent()` returns a table that contains information about the callback function it has just executed. The following fields will be initialized in that table:

`Action:` Contains the name of the event that caused the callback execution (e.g. `OnMouseDown`). If `WaitEvent()` returns without having run a callback, this field will be set to an empty string.

`ID:` Contains the identifier of the object that caused the callback execution (e.g. a display identifier). `ID` can also be zero in case an event was caused that has no identifier associated.

`Triggered:`
Will be set to `True` if `WaitEvent()` has executed a callback and then returned control to the script. If this is set to `False`, then some other internal event has caused `WaitEvent()` to return control to the script.

`NResults:`
Contains the number of values that the user callback returned (e.g. 1). This will be 0 if the user callback did not return any values or if no user callback was ran at all.

`Results:` If `NResults` is greater than 0, this table will contain all values that the user callback returned. Otherwise this table will not be present at all. You can easily use this table to pass additional information from your callbacks back to the main scope of the program.

INPUTS
none

RESULTS

`info` return values from previously executed event function; normally you won't need this and you can ignore it

EXAMPLE
See Section 29.19 [MakeButton], page 574.

See Section 29.26 [SetInterval], page 582.

See Section 29.27 [SetTimeout], page 583.

See Section 29.13 [InstallEventHandler], page 553.

29.29 WaitKeyDown

NAME
WaitKeyDown – wait for the user to press a key (V1.5)

SYNOPSIS
```
WaitKeyDown(key$[, rawkey])
```

FUNCTION
This function halts the program flow until the user presses the key specified by `key$`. See Section 29.14 [IsKeyDown], page 570, for the keys you can specify in key$. Please note that `WaitKeyDown()` cannot be used for Unicode characters. Only characters from the English alphabet are universally supported.

Starting with Hollywood 6.1 you can pass the special string `ANY` in `key$` to wait for an arbitrary key to be pressed.

Starting with Hollywood 7.1 there is an optional argument `rawkey`. If this argument is set to `True`, `WaitKeyDown()` will treat `key$` as a raw key and wait until it is down. In

that case, `key$` must be one of the raw keys defined by Hollywood. See Section 29.22 [Raw keys], page 579, for details. The difference between normal keys and raw keys is described in the documentation of the `OnRawKeyDown` event handler. See Section 29.13 [InstallEventHandler], page 553, for details. Setting `rawkey` to `True` can be useful if you need to wait for a modifier key like shift, alt, control, etc.

INPUTS

`key$`	key to wait for
`rawkey`	optional: `True` if `key$` should be treated as a raw key (defaults to `False`) (V7.1)

EXAMPLE

```
Print("Press Return to continue.")
WaitKeyDown("Return")
```

The code aboves waits until the user presses the return key.

29.30 WaitLeftMouse

NAME

WaitLeftMouse – wait for the user to press the left mouse button

SYNOPSIS

```
WaitLeftMouse()
```

FUNCTION

This function halts the script's execution until the left mouse button is pressed.

INPUTS

none

EXAMPLE

```
Print("Press left mouse to quit.")
WaitLeftMouse
End
```

Wait for the user to press left mouse button.

29.31 WaitMidMouse

NAME

WaitMidMouse – wait for the user to press the middle mouse button (V4.5)

SYNOPSIS

```
WaitMidMouse()
```

FUNCTION

This function halts the script's execution until the middle mouse button is pressed.

INPUTS

none

EXAMPLE

```
Print("Press middle mouse to quit.")
WaitMidMouse
End
```

Wait for the user to press middle mouse button.

29.32 WaitRightMouse

NAME

WaitRightMouse – wait for the user to press the right mouse button (V1.5)

SYNOPSIS

```
WaitRightMouse()
```

FUNCTION

This function halts the script's execution until the right mouse button is pressed.

INPUTS

none

EXAMPLE

```
Print("Press right mouse to quit.")
WaitRightMouse
End
```

Wait for the user to press right mouse button.

30 Graphics library

30.1 ARGB

NAME

ARGB – compose a color with alpha transparency (V2.0)

SYNOPSIS

```
color = ARGB(alpha, rgb)
```

FUNCTION

This function combines an alpha transparency value with a color in RGB notation. A 32-bit ARGB color in the format $AARRGGBB will be returned. This is useful in connection with the commands of the drawing library because they can draw alpha-blended graphics, too.

INPUTS

`alpha`	desired alpha transparency (0-255)
`rgb`	standard RGB color

RESULTS

`color`	ARGB color

EXAMPLE

```
Box(#CENTER, #CENTER, 320, 240, ARGB(128, #WHITE))
```

The code above draws an alpha-blended white rectangle. The mixing ratio is 128 (50%).

30.2 ARGB colors

An ARGB color is a RGB color extended with alpha transparency information. This notation is useful for the functions of the drawing library because they allow you to specify an ARGB color. Hence, you can draw transparent graphics primitives with the Hollywood drawing library.

A standard RGB color is a 24-bit value whereas an ARGB color uses 32 bits. The highest 8 bits are used for the alpha transparency information which can range from 0 to 255. Thus, an ARGB color looks like the following:

```
$AARRGGBB
```

An alpha value of 0 means that there is no transparency at all. A value of 255 means full transparency. This is an inversion of the format used in `SetAlphaIntensity()` where 255 means 0% transparency and 0 means 100% transparency. Please keep that in mind.

You can use the `ARGB()` function to easily combine an alpha transparency value with a 24-bit RGB color.

30.3 BeginDoubleBuffer

NAME

BeginDoubleBuffer – start double buffering for current display (V2.0)

SYNOPSIS

`BeginDoubleBuffer([hardware])`

FUNCTION

This command puts the current display in double buffering mode. The graphics that are drawn after this command was called will not be visible until you call `Flip()`. Double buffering is used to avoid that screen updates are visible to the user. As the name implies, you have two buffers in double buffering mode: A front buffer (visible to the user) and a back buffer (in memory). Your screen updates are always drawn to the back (memory) buffer and when you are finished with that, you call the `Flip()` command to bring the back buffer to the front. After that, you can draw the next update. This technique ensures that no flickering will be visible because `Flip()` will always refresh the whole display in one go.

Double buffers are extremely useful if many graphics have to be drawn in a screen update. If you only need to move a little player image around, you should better use sprites because that is faster. Remember that a double buffered display will always refresh the whole screen. Thus, if you have an application running in 640x480 at 25fps, it will be quite some work for Hollywood because it has to draw a screen of 640x480 25 times a second.

Double buffers are installed on a per display basis. Thus, when you call `BeginDoubleBuffer()`, it will change the currently selected display into a double-buffered one. It will not change all displays to double-buffered! If you want all your displays to be double-buffered, you need to call `BeginDoubleBuffer()` for each of your displays.

Some restrictions apply:

- You cannot use double buffering when layers are enabled.
- You cannot use sprites together with a double buffered display.
- The BGPic that is active when `BeginDoubleBuffer()` is called must not be transparent.

Starting with Hollywood 5.0, there is a new optional argument `hardware` that allows you to enable hardware double buffering. On supported systems, this is much faster than software double buffering because it will completely operate in video memory which can use the GPU for drawing. There are some restrictions, though: If you use a hardware double buffer, you should draw to it using hardware brushes whenever this is possible. All other drawing commands will be much slower! Only by using hardware brushes can you get full hardware accelerated drawing. Using normal drawing functions with a hardware double buffer can even be slower than using them on a software double buffer. This is especially the case with graphics that use an alpha channel, e.g. anti-aliased text or vector shapes, because for alpha channel drawing, Hollywood has to read from the destination device which will be very slow for hardware double buffers because reading from video memory is very slow. Thus, you should try to use hardware brushes

whereever possible when you work with a hardware double buffer. See Section 21.37 [hardware brushes], page 280, for details.

Please note that currently hardware double buffering is only supported on AmigaOS and Android by default. However, plugins that install a display adapter are also able to support hardware double buffers for their display adapter. In that case you can also use hardware double buffers on systems other than AmigaOS and Android. For example, the GL Galore and RebelSDL plugins allow you to use hardware double buffers on Windows, macOS, and Linux. See Section 5.4 [Obtaining plugins], page 66, for details.

Please note that Hollywood might also fall back to single buffering on some systems. Therefore, it is not safe to assume that calling `Flip()` will really switch buffers. It could also just draw the single buffer and then simply let you draw on it again.

INPUTS

`hardware` optional: whether or not to create a hardware double buffer (defaults to `False` which means software double buffering) (V5.0)

EXAMPLE

```
BeginDoubleBuffer()
CreateBrush(1, 64, 64, #RED)
For k = -64 To 640
   Cls
   DisplayBrush(1, k, #CENTER)
   Flip
Next
EndDoubleBuffer()
```

The code above moves a red rectangle from the outer left to the outer right without any visible flickering. This is not a good example because we only move a little image around. It is a lot of overhead to refresh the whole 640x480 pixels just for this little image, so you should better use sprites in this case. Remember that double buffering is only recommended when there are a lot of graphics to draw.

30.4 BeginRefresh

NAME

BeginRefresh – prepare display for optimized refresh (V5.3)

SYNOPSIS

```
BeginRefresh([force])
```

FUNCTION

This command prepares the active display for optimized refreshing. On platforms supporting optimized refreshing, this means that all drawing commands that follow `BeginRefresh()` will be queued until you call `EndRefresh()`. By caching all drawing commands that are called between `BeginRefresh()` and `EndRefresh()`, the latter will be able to refresh your whole display in a single frame which on many platforms is the most efficient way of drawing besides using a double buffer. On platforms that don't support optimized refreshing, calling `BeginRefresh()` and `EndRefresh()` won't

have any effect at all so it is safe to always use it in your scripts and Hollywood will enable it automatically when needed.

Optimized refreshing is always supported on Windows (except when the software renderer is active), macOS (except on PowerPC), iOS and Android. On all other platforms optimized refreshing is only supported when the auto scaling engine is active. See Section 25.18 [Scaling engines], page 401, for details.

To better understand the concept behind optimized refreshing, you need to understand that on some systems all drawing operations will always result in a complete redraw of the whole display, even if just a single pixel is changed. Now imagine the following situation: For every new frame your game has to draw 48 sprites to the screen. On systems that always refresh the whole display even if just a single pixel has changed this means that Hollywood has to redraw the whole screen 48 times and this 50 times a second if your game runs at 50fps. Thus, 2400 complete screen refreshes (48 * 50) would be necessary per second (!). This of course will significantly kill your game's performance. If you encapsulate the drawing of the 48 sprites inside a `BeginRefresh()` and `EndRefresh()` section instead, Hollywood will only have to refresh the screen once per frame (50 times per second), which leads to a much smoother appearance. Thus, using `BeginRefresh()` in this case is absolutely mandatory to ensure a good performance of your game.

Optimized refresh can also greatly enhance the performance of your script when autoscaling is active. If you use normal drawing with autoscaling, Hollywood will have to scale its refresh buffer to the desired dimensions on every single command that draws something. If you use a `BeginRefresh()` section here instead, Hollywood will just have to scale its refresh buffer once `EndRefresh()` is called. This of course can also enhance the performance significantly.

By default, `BeginRefresh()` will only use optimized refresh if its use is either encouraged on the host platform or if autoscaling is enabled. You can change this behaviour by passing `True` in the optional force argument. If `force` is set to `True`, `BeginRefresh()` will always use optimized refresh but this is not recommended because optimized refresh can also be slower than normal refresh in some cases. That is why you should leave the decision whether to use optimized refresh or not to Hollywood.

Please note that you only need to use `BeginRefresh()` sections if your script does its drawing without a double buffer. If your script uses a double-buffer, drawing is already pipelined to one refresh per frame and thus using `BeginRefresh()` is not needed in this case.

INPUTS

`force` optional: force the use of optimized refresh or leave the decision to Hollywood. See above for more information (defaults to `False` which means leave the decision to Hollywood)

EXAMPLE

```
BeginRefresh()
For Local k = 1 To 48
   DisplaySprite(k, sprites[k].x, sprites[k].y)
Next
EndRefresh()
```

The code above updates the positions of 48 sprites using a `BeginRefresh()` section. This means that on systems that always refresh the whole display (e.g. Android) Hollywood just has to update the screen once to reposition all 48 sprites. Without `BeginRefresh()` Hollywood would have to update the screen 48 times which is of course much slower.

30.5 Blue

NAME

Blue – return blue portion of a color (V1.9)

SYNOPSIS

```
b = Blue(color)
```

FUNCTION

This function returns the 8-bit blue component of a color in the RGB format.

INPUTS

`color` input color in RGB format

RESULTS

`b` blue component of the color

EXAMPLE

```
b = Red($C0C024)
```

The above code will return $24 which is the blue component of the color.

30.6 ClearScreen

NAME

ClearScreen – clear screen with a transition effect

SYNOPSIS

```
[handle] = ClearScreen(effect, color[, table])
```

FUNCTION

This function clears the screen with one of Hollywood's transition effects. The color is a RGB value specifying the color of the transition effect.

Note that this function will automatically create a new BGPic filled with the specified color and switch to it.

Starting with Hollywood 5.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Speed:` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter:`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async:` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here DisplayTransitionFX() will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`X:` Specifies the new x-position for the display. If you want the display to keep its current x-position, specify the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

`Y:` Specifies the new y-position for the display. If you want the display to keep its current y-position, specify the special constant `#KEEPPOSITION`. Defaults to `#CENTER`.

Have a look at the documentation of `DisplayTransitionFX()` to see the list of all transition effects that can be used.

INPUTS

`effect` special effect constant (see `DisplayTransitionFX()`)

`color` color that shall be used for the effect

`table` optional: table configuring the transition effect

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
ClearScreen(#VBLINDS32, $000000, {Speed = 10})
```

The above code clears the screen with color black and the effect `#VBLINDS32` and speed of 10.

30.7 Collision

NAME

Collision – check if two objects collide (V2.0)

SYNOPSIS

```
bool = Collision(type, ...)
bool = Collision(#BOX, x1, y1, width1, height1, x2, y2, width2, height2)
bool = Collision(#BRUSH, id1, x1, y1, id2, x2, y2)
bool = Collision(#LAYER, id1, id2)
bool = Collision(#SPRITE, id1, id2)
bool = Collision(#BRUSH_VS_BOX, id, x, y, x2, y2, width2, height2) (V4.5)
bool = Collision(#LAYER_VS_BOX, id, x, y, width, height) (V4.5)
bool = Collision(#SPRITE_VS_BOX, id, x, y, width, height) (V4.5)
bool = Collision(#SPRITE_VS_BRUSH, id1, id2, x, y) (V7.1)
```

FUNCTION

This function checks if two objects collide. There are several possibilities how you can use this function depending on the type you specify.

The following collision types are currently supported:

`#BOX:` You have to specify the position and size of two rectangles and this function will determine if they collide or not. This is a quick calculation but is probably not exact enough for some purposes.

`#BRUSH:` Checks if the two brushes specified by `id1` and `id2` would collide if they were shown at `x1` and `y1` and `x2` and `y2`, respectively. Transparent areas (mask or alpha channel) of the brushes will be fully respected so that you get an exact result if pixels collide or not.

`#LAYER:` Checks if the two layers specified by `id1` and `id2` collide. If the layers have transparent areas, they will be respected. If you use this type, layers must be enabled of course.

`#SPRITE:` Checks if the two sprites specified by `id1` and `id2` collide. If the sprites have transparent areas, they will be respected.

`#BRUSH_VS_BOX:`
Checks if the specified brush collides with the specified rectangular area. If the brush has transparent areas, they will be taken into account. (V4.5)

`#LAYER_VS_BOX:`
Checks if the specified layer collides with the specified rectangular area. If the layer has transparent areas, they will be taken into account. (V4.5)

`#SPRITE_VS_BOX:`
Checks if the specified sprite collides with the specified rectangular area. If the sprite has transparent areas, they will be taken into account. (V4.5)

`#SPRITE_VS_BRUSH:`
Checks if the sprite specified by `id1` collides with the brush specified by `id2` in case the brush was displayed at the position specified by `x` and `y`. (V7.1)

INPUTS

`type` either `#BOX`, `#BRUSH`, `#SPRITE`, `#LAYER`, `#BRUSH_VS_BOX`, `#LAYER_VS_BOX`, `#SPRITE_VS_BOX`, or `#SPRITE_VS_BRUSH` (see above)

`...` optional arguments depend on the type specified (see above)

RESULTS

`bool` `True` for collision, `False` otherwise

EXAMPLE

```
Box(10, 10, 100, 100, #RED)
Box(70, 70, 100, 100, #BLUE)
b = Collision(#BOX, 10, 10, 100, 100, 70, 70, 100, 100)
```

This returns `True` because the rectangles collide.

30.8 CreateClipRegion

NAME

CreateClipRegion – create a clip region (V2.0)

SYNOPSIS

```
[id] = CreateClipRegion(id, type, ...)
[id] = CreateClipRegion(id, #BOX, x, y, width, height)
[id] = CreateClipRegion(id, #SHAPE, id, x, y)
```

FUNCTION

This function can be used to create a clip region which all Hollywood graphics functions will respect. `CreateClipRegion()` creates a new clip region and assigns the identifier `id` to it. If you pass `Nil` in `id`, `CreateClipRegion()` will automatically choose an identifier and return it. Note that `CreateClipRegion()` does not activate the clip region; this has to be done using `SetClipRegion()`.

Clip regions are useful to limit the area of your display, where graphics can be displayed, e.g. if you have a game screen with two parts: The level area and the status bar (lifes, ammo etc.) it can be useful to install a clip region which matches the bounds of the level area so that sprites will never be drawn outside of this area. All Hollywood graphics functions will respect the clip region you install using this function. No graphics will ever be drawn outside the bounds of your clip region.

There are two different clip region types: Rectangular (`#BOX`) and custom shaped (`#SHAPE`) clip regions. Drawing to rectangular clip regions is generally faster. If you want to install a rectangular clip region, you have to specify its `x` and `y` position as well as its `width` and `height`. A custom shaped clip region can be installed by specifying a brush, whose mask is to be used as the clip region. Also, you need to specify `x` and `y` offsets where the clip region shall be positioned on the display, e.g. if you have a mask of size 320x240 but a display of size 640x480, you might want to center the clip region on the display. If you install a custom shaped clip region using a brush's mask, Hollywood will be able to draw to all visible pixels of the mask and all invisible pixels of the mask will be clipped.

See Section 30.31 [SetClipRegion], page 611, for information on how to install a clip region created using CreateClipRegion().

To free a clip region created with `CreateClipRegion()`, use the `FreeClipRegion()` function.

INPUTS

`id`	identifier for the clip region or `Nil` for auto id selection
`type`	type of clip region to install; `#BOX`, `#SHAPE` or `#NONE`
`...`	the following arguments depend on the type; see above

RESULTS

`id`	optional: identifier of the clip region; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

See Section 30.31 [SetClipRegion], page 611.

30.9 DisablePrecalculation

NAME
DisablePrecalculation – disable precalculation (V1.5 only)

SYNOPSIS
`DisablePrecalculation()`

FUNCTION
Attention: This command was removed in Hollywood 1.9. Get a faster CPU.

This function disables precalculation of certain effects.

INPUTS
none

30.10 DisableVWait

NAME
DisableVWait – disable vertical refresh timing (V9.0)

SYNOPSIS
`DisableVWait()`

FUNCTION
This command disables internal vertical refresh timing. `DisableVWait()` is only useful for debugging purposes. You should not use it in your scripts.

INPUTS
none

30.11 EnablePrecalculation

NAME
EnablePrecalculation – enable precalculation (V1.5 only)

SYNOPSIS
`EnablePrecalculation()`

FUNCTION
Attention: This command was removed in Hollywood 1.9. Get a faster CPU.

This function enables precalculation for certain cpu-intensive effects from the library of `DisplayTransitionFX()`. Effects which support precalculation, will be precalculated then. This is only useful on 68k systems. PPC systems should be able to display those cpu-intensive effects in real time.

INPUTS
none

30.12 EnableVWait

NAME
EnableVWait – enable vertical refresh timing (V9.0)

SYNOPSIS
`EnableVWait()`

FUNCTION
This command enables internal vertical refresh timing. `EnableVWait()` is only useful for debugging purposes. You should not use it in your scripts.

INPUTS
none

30.13 EndDoubleBuffer

NAME
EndDoubleBuffer – stop double buffering for current display (V2.0)

SYNOPSIS
`EndDoubleBuffer()`

FUNCTION
This command ends the double buffering mode in the current display. See Section 30.3 [BeginDoubleBuffer], page 590, for more information on double buffering and an example.

INPUTS
none

EXAMPLE
See Section 30.3 [BeginDoubleBuffer], page 590.

30.14 EndRefresh

NAME
EndRefresh – flush drawing queue (V5.3)

SYNOPSIS
`EndRefresh()`

FUNCTION
This command draws all graphics that have been queued since the last call to `BeginRefresh()`. See Section 30.4 [BeginRefresh], page 591, for more information on optimized refreshing and an example.

INPUTS
none

EXAMPLE
See Section 30.4 [BeginRefresh], page 591.

30.15 Flip

NAME

Flip – flip front and back buffers (V2.0)

SYNOPSIS

```
Flip([sync])
```

FUNCTION

This command brings the back buffer to the front and makes the front buffer the back buffer so that you can draw the next screen update.

Starting with Hollywood 5.0, there is an optional argument called `sync`. If this is set to `False`, `Flip()` will not wait for the vertical refresh but will draw the back buffer immediately. This is useful if you would like to refresh the double buffer more often than the refresh rate of the monitor. If not specified, `sync` defaults to `True` which means that `Flip()` will synchronize its double buffer with the monitor refresh.

Please note that Hollywood might also fall back to single buffering on some systems. Therefore, it is not safe to assume that calling `Flip()` will really switch buffers. It could also just draw the single buffer and then simply let you draw on it again.

See Section 30.3 [BeginDoubleBuffer], page 590, for more information on double buffering and an example.

INPUTS

`sync`	optional: whether or not to synchronize double buffer with monitor refresh (defaults to `True`) (V5.0)

EXAMPLE

See Section 30.3 [BeginDoubleBuffer], page 590.

30.16 FreeClipRegion

NAME

FreeClipRegion – free a clip region (V2.0)

SYNOPSIS

```
FreeClipRegion(id)
```

FUNCTION

This function frees a clip region created by CreateClipRegion().

If there are sprites which are attached to this clip region, they will be let out and the clip region is freed immediately.

The behaviour for layers is different: If there are layers which reference this clip region, then Hollywood will keep the clip region until these layers are gone. As soon as there are no more layers which reference this clip region, the Hollywood garbage collector will free the clip region automatically if you have previously called `FreeClipRegion()` on it.

Please see also the documentation of `CreateClipRegion()` for more information on clip regions.

INPUTS

`id` identifier of the clip region to be freed

EXAMPLE

See Section 30.31 [SetClipRegion], page 611.

30.17 GetFPSLimit

NAME

GetFPSLimit – get current frames per second limit (V1.5)

SYNOPSIS

```
fps = GetFPSLimit()
```

FUNCTION

This function returns the current fps limit set using `SetFPSLimit()`. If SetFPSLimit() is not active, 0 will be returned. See Section 30.33 [SetFPSLimit], page 613, for details.

INPUTS

none

RESULTS

`fps` current fps limit or 0

30.18 GetRandomColor

NAME

GetRandomColor – choose a random color (V4.7)

SYNOPSIS

```
color = GetRandomColor()
```

FUNCTION

This function simply chooses a random color and returns it. The color will be returned in RGB notation.

INPUTS

none

RESULTS

`color` a randomly chosen color

30.19 GetRandomFX

NAME

GetRandomFX – choose a random transition effect (V4.0)

SYNOPSIS

```
fx = GetRandomFX(objfx)
```

FUNCTION

This function simply chooses a random transition effect from Hollywood's palette of transition effects. You need to specify in argument 1 whether or not the transition effect shall be applicable for objects or only for background pictures. `GetRandomFX()` then scans the available effects and returns a random effect for you which you can then pass to the special effects functions like `DisplayBrushFX()` and `DisplayTransitionFX()`.

Normally, you do not have to use this function because you can simply pass `#RANDOMEFFECT` with all special effects functions. `GetRandomFX()` is only useful if you want to filter effects, i.e. you want to choose a random effect but not `#WATER1`, for example. In that case, you can call `GetRandomFX()` until it returns an effect that is different from the ones you don't want.

See Section 20.11 [DisplayTransitionFX], page 238, for the different types of Hollywood's transition effects.

INPUTS

`objfx` `True` if you want to have an object effect or `False` for a background picture effect

RESULTS

`fx` a randomly chosen transition effect

EXAMPLE

```
Repeat
   fx = GetRandomFX(FALSE)
Until fx <> #WATER1
DisplayTransitionFX(2, fx)
```

The code above chooses a random background picture effect but never the `#WATER1` effect. After having chosen the effect, background picture 2 is display with this effect.

30.20 GetRealColor

NAME

GetRealColor – return representation of a color on the current screen (V1.5)

SYNOPSIS

```
color = GetRealColor(col)
```

FUNCTION

This function returns the color which represents the specified color on the current screen. On 24-bit and 32-bit screens, the returned color is always the same as the color you specified. On 15-bit and 16-bit screens, the returned color is slightly different from the original color in most cases because those screens do not have 16.7 million colors but only 65536 (16-bit screens) respectively 32768 (15-bit screens).

This is command is mostly used in connection with `ReadPixel()`.

INPUTS

`col` color to find the representation for

RESULTS

`color` color which represents the specified input color

EXAMPLE

```
color = GetRealColor($FFFFFF)
```

Color will get the following values on...

1) 24-bit and 32-bit screens: $FFFFFF 2) 16-bit screens: $F8FCF8 3) 15-bit screens: $F8F8F8

30.21 GrabDesktop

NAME

GrabDesktop – create a brush of the desktop screen (V4.5)

SYNOPSIS

```
[id] = GrabDesktop(id[, table])
```

FUNCTION

This function can be used to copy the contents of desktop screen into a brush. You have to pass an identifier for the new brush that shall be created by this function. If you pass a `Nil` identifier, `GrabDesktop()` will return a handle to the new brush containing the desktop screen to you.

The optional table argument is useful if you only want to grab a portion of the desktop screen. In that case, you can use the optional table argument to define the portion that shall be grabbed.

The optional table argument recognizes the following tags:

`X, Y:` Defines the top left corner of the rectangle on the desktop that shall be grabbed by this function. Defaults to 0/0.

`Width, Height:`
Defines the size of the rectangle that shall be grabbed by this function. Defaults to the size of the desktop.

`PubScreen:`
This tag is only supported in the Amiga versions of Hollywood. It allows you to specify the name of the screen which `GrabDesktop()` should copy to a brush. By default, `GrabDesktop()` will always grab the frontmost screen. (V5.3)

INPUTS

`id` identifier of a brush that shall be created by this function or `Nil` for auto id selection

`table` optional: table specifying the portion of the desktop screen that shall be grabbed (see above)

RESULTS

`id` optional: handle to the newly created brush; this will only be returned if you passed `Nil` in `id`

EXAMPLE

```
desktop_brush = GrabDesktop(Nil)
BrushToGray(desktop_brush)
DisplayBrush(desktop_brush, 0, 0)
```

The code above grabs the whole desktop screen to a brush, converts the brush to gray, and then displays it. The result will be a desktop screen that suddenly lost its color information.

30.22 Green

NAME

Green – return green portion of a color (V1.9)

SYNOPSIS

```
g = Green(color)
```

FUNCTION

This function returns the 8-bit green component of a color in the RGB format.

INPUTS

`color`	input color in RGB format

RESULTS

`g`	green component of the color

EXAMPLE

```
g = Green($24C0C0)
```

The above code will return $C0 which is the green component of the color.

30.23 Intersection

NAME

Intersection – compute intersection of two rectangles (V6.1)

SYNOPSIS

```
ix, iy, iw, ih = Intersection(x1, y1, w1, h1, x2, y2, w2, h2)
```

FUNCTION

This function computes the intersection between the two rectangles whose positions and dimensions are passed to it. If the two rectangles don't intersect, the returned dimensions will be 0.

INPUTS

`x1`	x position of first rectangle
`y1`	y position of first rectangle
`w1`	width of first rectangle
`h1`	height of first rectangle

`x2` x position of second rectangle

`y2` y position of second rectangle

`w2` width of second rectangle

`h2` height of second rectangle

RESULTS

`ix` x position of intersecting rectangle

`iy` y position of intersecting rectangle

`iw` width of intersecting rectangle

`ih` height of intersecting rectangle

EXAMPLE

```
SetFillStyle(#FILLCOLOR)
Box(100, 100, 80, 100, #RED)
Box(160, 120, 100, 40, #YELLOW)

ix, iy, iw, ih = Intersection(100, 100, 80, 100, 160, 120, 100, 40)

Box(ix, iy, iw, ih, #GREEN)
```

The code above computes the intersection of the red and yellow rectangles and visualizes it by drawing a green rectangle.

30.24 IsPicture

NAME

IsPicture – determine if a picture is in a supported format

SYNOPSIS

```
ret, table = IsPicture(file$[, table])
```

FUNCTION

This function will check if the file specified by `file$` is in a supported picture format. If it is, this function will return `True`, otherwise `False`. If this function returns `True`, you can load the picture by calling `LoadBGPic()` or `LoadBrush()`.

New in Hollywood 4.5: This function returns a table now as the second return value. If the specified file is an image, the table will contain some information about the image file. The following fields of the return table will be initialized:

`Width:` Contains the width of the image in pixels.

`Height:` Contains the height of the image in pixels.

`Depth:` Contains the bit depth of the image. (V9.0)

`Alpha:` `True` if image has an alpha channel, `False` otherwise.

`Vector:` `True` if image is in a scalable vector format, `False` otherwise. (V5.0)

`Transparency:`
: `True` if image has a monochrome transparency channel, i.e. a transparent pen in a palette-based image. (V6.0)

`Format:`
: This contains the image format type Hollywood has detected for this file. It can be one of the following special constants:

  `#IMGFMT_BMP:`
  : BMP image.

  `#IMGFMT_PNG:`
  : PNG image.

  `#IMGFMT_JPEG:`
  : JPEG image.

  `#IMGFMT_ILBM:`
  : IFF ILBM image.

  `#IMGFMT_GIF:`
  : GIF image.

  `#IMGFMT_TIFF:`
  : TIFF image.

  `#IMGFMT_NATIVE:`
  : The image is in a format that can be loaded by the host operating system Hollywood is currently running on. For example, if there is a datatype which can load the image on AmigaOS, you can get `#IMGFMT_NATIVE` as the result.

  `#IMGFMT_PLUGIN:`
  : The image is in a format supported by a Hollywood plugin that is installed.

  (V8.0)

This function is much faster than `LoadBrush()` or `LoadBGPic()` because it will not load the picture. It will just check its format header and return whether or not Hollywood can handle it. However, please note that image plugins you have installed might not be as optimized as Hollywood's inbuilt image loader and so they can slow down `IsPicture()` significantly because by default `IsPicture()` will first ask all plugins if they can load the picture and then Hollywood's inbuilt image loader will be asked. Thus, you might want to disable plugins for `IsPicture()` by setting the `Loader` tag (see below) to `Inbuilt`. This means that `IsPicture()` will never ask any plugins if they can load the picture. Only Hollywood's very optimized inbuilt image loader will be asked in that case.

Starting with Hollywood 6.0 this function accepts an optional table argument which allows you to configure further options:

`Loader:`
: This tag allows you to specify one or more format loaders that should be asked to load this picture. This must be set to a string containing the name(s) of one or more loader(s). You might want to set this to `Inbuilt` for the best performance, but then only formats supported by Hollywood's

inbuilt image loader will be recognized (see above for details). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

See Section 21.42 [LoadBrush], page 283, for a list of supported image formats.

INPUTS

`file$` file to check

`table` optional: table configuring further options (V6.0)

RESULTS

`ret` `True` if the picture is in a supported format, `False` otherwise

`table` table initialized with the fields listed above; only valid if first return argument is `True` (V4.5)

30.25 Matrix2D

NAME

Matrix2D – construct 2x2 transformation matrix (V10.0)

SYNOPSIS

```
m = Matrix2D(sx, sy, angle)
```

FUNCTION

This function combines the scaling coefficients specified by `sx` and `sy` and the rotation factor specified by `angle` into a 2x2 transformation matrix and returns it. The 2x2 transformation matrix is returned as a table which has the following fields initialized:

`sx:` Amount of scaling on the x axis.

`rx:` Amount of rotation on the x axis.

`ry:` Amount of rotation on the y axis.

`sy:` Amount of scaling on the y axis.

You could then pass the matrix to functions like `TransformBox()` or `TransformPoint()`.

INPUTS

`sx` scale x coefficient

`sy` scale y coefficient

`angle` rotation angle

RESULTS

`m` 2x2 transformation matrix returned as a table

30.26 MixRGB

NAME

MixRGB – mix two colors (V2.0)

SYNOPSIS

```
color = MixRGB(col1, col2, ratio)
```

FUNCTION

This function mixes the two specified colors at a specified ratio which must be in the range of 0 to 255. The second color will be mixed with the first color at the specified ratio, i.e. if `ratio` is 0, `col1` will returned and if `ratio` is 255, `col2` will be returned.

`ratio` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`col1` color 1 in RGB format

`col2` color 2 in RGB format

`ratio` mixing ratio (0 to 255 or percent specification)

RESULTS

`color` mixed color

EXAMPLE

```
c = MixRGB(#RED, #BLUE, 128)
```

This will mix a pink sort of color.

30.27 Red

NAME

Red – return red portion of a color (V1.9)

SYNOPSIS

```
r = Red(color)
```

FUNCTION

This function returns the 8-bit red component of a color in the RGB format.

INPUTS

`color` input color in RGB format

RESULTS

`r` red component of the color

EXAMPLE

```
r = Red($24C0C0)
```

The above code will return $24 which is the red component of the color.

30.28 RGB

NAME

RGB – compose a color (V1.9)

SYNOPSIS

```
color = RGB(red, green, blue)
```

FUNCTION

This function composes a RGB color by mixing the three basic colors red, green and blue. You must specify an intensity of 0 to 255 for each basic color.

INPUTS

`red`	red intensity (0-255)
`green`	green intensity (0-255)
`blue`	blue intensity (0-255)

RESULTS

`color`	RGB color

EXAMPLE

```
color = RGB(255, 255, 255)  ; mix white
color = RGB(255, 0, 0)      ; mix red
color = RGB(255, 255, 0)    ; mix yellow (red + green)
color = RGB(255, 0, 255)    ; mix magenta (green + blue)
```

30.29 RGB colors

Hollywood accepts colors in the 24-bit RGB format. This means that you have 8-bit per color. RGB24 is the common format for specifying colors today, e.g. it is also used in HTML.

You will usually specify RGB colors in hexadecimal notation starting with a '$' prefix. The general syntax for a RGB color in hexadecimal notation is

```
$RRGGBB
RR: color intensity of red (maximum = 255 = $FF = 100% red)
GG: color intensity of green (maximum = 255 = $FF = 100% green)
BB: color intensity of blue (maximum = 255 = $FF = 100% blue)
```

Colors are generated by mixing the red, green and blue components together. Every color component can have a maximum of 255.

Here are some example colors:

```
#BLACK   = $000000  - black is just 0% of every component
```

```
#WHITE   = $FFFFFF  - white is 100% of every component
#RED     = $FF0000  - pure red is 100% of red
#GREEN   = $00FF00  - 100% of green
#BLUE    = $0000FF  - 100% of blue
#YELLOW  = $FFFF00  - yellow can be mixed with 100% of red and green
#FUCHSIA = $FF00FF  - fuchsia is 100% of red and blue
#GRAY    = $939393  - gray is a bit of everything
```

When Hollywood asks you for a RGB color, you can specify your individual color using hexadecimal notation or also one of the special inbuilt color constants.

#BLACK	$000000	
#MAROON	$800000	
#GREEN	$008000	
#OLIVE	$808000	
#NAVY	$000080	
#PURPLE	$800080	
#TEAL	$008080	
#GRAY	$808080	
#SILVER	$C0C0C0	
#RED	$FF0000	
#LIME	$00FF00	
#YELLOW	$FFFF00	
#BLUE	$0000FF	
#FUCHSIA	$FF00FF	
#AQUA	$00FFFF	
#WHITE	$FFFFFF	

30.30 SaveSnapshot

NAME

SaveSnapshot – take a snapshot (V2.0)

SYNOPSIS

`SaveSnapshot(f$[, mode, fmt, table])`

FUNCTION

This function takes a snapshot and saves it as the file specified in `f$`. The `mode` argument specifies the area to be grabbed. This can be one of the following constants:

`#SNAPWINDOW:`
: Grabs the complete Hollywood display, i.e. with window decorations. This is the default.

`#SNAPDISPLAY:`
: Grabs the display's contents only, i.e. without window decorations.

`#SNAPDESKTOP:`
: Grabs the complete host screen.

Note that if the display is a palette mode display and you want the image file to be palette-based as well, you need to use `#SNAPDISPLAY` as this is the only snap mode that doesn't contain any other graphics besides the display's contents.

The `fmt` argument specifies the desired output image format. This can either be one of the following constants or an image saver provided by a plugin:

`#IMGFMT_BMP:`
: Windows bitmap. Hollywood's BMP saver supports RGB and palette images. `#IMGFMT_BMP` is the default format used by `SaveSnapshot()`.

`#IMGFMT_PNG:`
: PNG format. Hollywood's PNG saver supports RGB and palette images. RGB images also can have an alpha channel, palette images can have a transparent pen. (V2.5)

`#IMGFMT_JPEG:`
: JPEG format. Note that the JPEG format does not support alpha channels or palette-based graphics. The `Quality` field (see below) allows you to specify the quality level for the JPEG image (valid values are 0 to 100 where 100 is the best quality). (V4.0)

`#IMGFMT_GIF:`
: GIF format. Because GIF images are always palette-based, RGB graphics have to be quantized before they can be exported as GIF. You can use the `Colors` and `Dither` tags (see below) to specify the number of palette entries to allocate for the image and whether or not dithering shall be applied. When using `#IMGFMT_GIF` with a palette display, no quantizing will be done. `#IMGFMT_GIF` also supports palette images with a transparent pen. (V4.5)

`#IMGFMT_ILBM:`
: IFF ILBM format. Hollywood's IFF ILBM saver supports RGB and palette images. Palette images can also have a transparent pen, alpha channels are unsupported for this output format. (V4.5)

The optional table argument allows you to configure further parameters:

`Dither:` Set to `True` to enable dithering. This tag is only handled when the destination format is palette-based and the source data is RGB. Defaults to `False` which means no dithering.

`Depth:` Specifies the desired image depth. This is only handled when the format is palette-based and the source data is RGB. Valid values are between 1 (= 2 colors) and 8 (= 256 colors). Defaults to 8. (V9.0)

`Colors:` This is an alternative to the `Depth` tag. Instead of a bit depth, you can pass how many colors the image shall use here. Again, this is only handled for palette-based formats when the source data is RGB. Valid values are between 1 and 256. Defaults to 256.

`Quality:` Here you can specify a value between 0 and 100 indicating the compression quality for lossy compression formats. A value of 100 means best quality, 0 means worst quality. This is only available for image formats that support lossy compression. Defaults to 90 which means pretty good quality.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Here is an overview that shows which formats support which tags:

	BMP	PNG	JPEG	GIF	ILBM
Dither	No	No	No	Yes	No
Colors	No	No	No	Yes	No
Quality	No	No	Yes	No	No

INPUTS

`f$` destination file

`mode` optional: specifies which area shall be grabbed (defaults to `#SNAPWINDOW`)

`fmt` optional: output format; either `#IMGFMT_BMP`, `#IMGFMT_PNG`, (V2.5) `#IMGFMT_JPEG`, `#IMGFMT_GIF` or `#IMGFMT_ILBM` (V4.0) (defaults to `#IMGFMT_BMP`)

`table` optional: table argument for configuring further options (V4.5)

EXAMPLE

```
SaveSnapshot("Snap.bmp")
```

Saves a snapshot of the Hollywood window to "Snap.bmp".

30.31 SetClipRegion

NAME

SetClipRegion – activate a clip region (V2.0)

SYNOPSIS

```
SetClipRegion(id)
```

FUNCTION

This function installs a clip region that has been previously created using the `CreateClipRegion()` command. The clip region will be active until you call `SetClipRegion()` with the special value `#NONE` - this will remove the clip region then. Hollywood will automatically kill the clip region if you display a new background picture.

If a clip region is installed, this will also affect Hollywood's special coordinate constants, e.g. `#RIGHT` means the right-side of the clip region then. Margin settings will also be adapted.

You can also install a clip region while `SelectBrush()` is active. This clip region will then be deactivated when `EndSelect()` is called.

If you have a clip region installed on your main display and call one of the off-screen rendering functions (e.g. `SelectBrush()`), the clip region will be temporarily disabled but restored when you call `EndSelect()`.

If layers are active, every layer can have its private clip region. In case the layer is transformed (scaled or rotated), its clip region will also be transformed. The default clip region of a layer is the clip region that was active when the layer was created. You can change the clip region of a layer by using the style element `ClipRegion` of the `SetLayerStyle()` command.

Exceptions. You cannot use `SetClipRegion()` if...

- current output device is an alpha channel, i.e. `SelectAlphaChannel()` is active
- current output device is a mask, i.e. `SelectMask()` is active

See Section 30.8 [CreateClipRegion], page 596, for details.

INPUTS

`id` identifier of the clip region to install; use `CreateClipRegion()` to create clip regions

EXAMPLE

```
CreateClipRegion(1, #BOX, #CENTER, #CENTER, 320, 240)
SetClipRegion(1)
Circle(0, 0, 100, #RED)
Circle(439, 0, 100, #RED)
Circle(439, 279, 100, #RED)
Circle(0, 279, 100, #RED)
```

Installs a clip region of size 320x240 in the center of a 640x480 display and draws four circles in all corners. However, only parts of the circles will be visible because of the clip region.

30.32 SetDrawTagsDefault

NAME

SetDrawTagsDefault – set default values for standard draw tags (V5.0)

SYNOPSIS

```
SetDrawTagsDefault(table)
```

FUNCTION

This command can be used to modify the default values of the standard draw tags. The standard draw tags are generic options that are supported by most of Hollywood's drawing commands. The standard draw tags are always passed inside an optional table that constitutes the last argument to a function. If a certain draw tag is not specified,

Hollywood will fall back to an internal default setting. This default setting can be modified using `SetDrawTagsDefault()`.

For example, let's assume that you always want to use an anchor point of 0.5/0.5 instead of 0.0/0.0 which is Hollywood's default anchor point. So instead of explicitly passing your desired anchor point to all draw commands that you call, you can simply define this anchor point as the new default anchor point that all drawing commands should use when no other point is given. See below for an example. You could also use `SetDrawTagsDefault()` to change the default insert position for layers from frontmost to backmost position etc.

The table you need to pass to this function can contain all tags that are listed in the documentation of the standard draw tags. For each tag that you specify, you have to provide a default value that Hollywood should use if no other value is given.

See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`table` a table containing one or more of the standard draw tags and a default value for each tag

EXAMPLE

```
SetDrawTagsDefault({AnchorX = 0.5, AnchorY = 0.5})
DisplayBrush(1, 0, 0)
Box(100, 100, 200, 150, #RED)
```

The code above sets 0.5/0.5 as the default anchor point. The calls to `DisplayBrush()` and `Box()` will then use this anchor point because no other anchor point is given. In that case, the drawing commands fall back to the default anchor point which has been modified by the call to `SetDrawTagsDefault()`.

30.33 SetFPSLimit

NAME

SetFPSLimit – limit frames per second (V1.5)

SYNOPSIS

```
SetFPSLimit(fps)
```

FUNCTION

This function restricts the number of frames per second of certain Hollywood commands which call `VWait()`. These commands are: `PlayAnim()`, `MoveBrush()` etc., `DisplayTransitionFX()` etc., and `Flip()`. Under normal circumstances it is not necessary to use this command because Hollywood automatically restricts the drawing speed internally. By default, the commands listed above will never draw more frames per second than the monitor refreshes. You can disable this behaviour with this command but this is not suggested (unless you really know what you are doing).

The best thing is to use the default video synchronizer of Hollywood will should give you the smoothest graphics. Use this command with care or better not at all.

If you pass 0 in `fps`, the default video synchronizer will be restored.

INPUTS

`fps` maximum allowed number of frames per second or 0 to bring back Hollywood's default video synchronizer

30.34 TransformBox

NAME

TransformBox – apply affine transformation to rectangle (V10.0)

SYNOPSIS

```
tx, ty, tw, th = TransformBox(x, y, w, h, m[, origin, anchorx, anchory])
```

FUNCTION

This function applies the 2x2 transformation matrix specified by `m` to the rectangle specified by `x`, `y`, `w` and `h` and returns the size and coordinates of the transformed rectangle. The `origin` parameter is a boolean value which specifies whether or not the rectangle should be moved to the origin before transforming it. The `anchorx` and `anchory` values specify the anchor point to use for the transformation. This should be a floating point value between 0.0 and 1.0 where 0.0 means the left/upper edge and 1.0 indicates the right/bottom edge. `anchorx` and `anchory` both default to 0 which means the upper-left corner of the rectangle is the default anchor point.

The transformation matrix `m` must be passed as a table that has the following fields initialized:

`sx:` Amount of scaling on the x axis.

`rx:` Amount of rotation on the x axis.

`ry:` Amount of rotation on the y axis.

`sy:` Amount of scaling on the y axis.

You can use the `Matrix2D()` function to construct such a matrix. See Section 30.25 [Matrix2D], page 606, for details.

INPUTS

`x` left edge of rectangle

`y` upper edge of rectangle

`w` rectangle width

`h` rectangle height

`m` table containing a 2x2 transformation matrix

`origin` optional: whether or not to move the rectangle to the origin before transforming it (defaults to `True`)

`anchorx` optional: horizontal anchor point (defaults to 0 which means left edge)

`anchory` optional: vertical anchor point (defaults to 0 which means top edge)

RESULTS

`tx` transformed left edge

`ty`	transformed upper edge
`tw`	transformed width
`th`	transformed height

30.35 TransformPoint

NAME

TransformPoint – apply affine transformation to point (V10.0)

SYNOPSIS

```
tx, ty = TransformPoint(px, py, m)
```

FUNCTION

This function applies the 2x2 transformation matrix specified by `m` to the point specified by `px` and `py` and returns the coordinates of the transformed point. The transformation matrix `m` must be passed as a table that has the following fields initialized:

`sx:`	Amount of scaling on the x axis.
`rx:`	Amount of rotation on the x axis.
`ry:`	Amount of rotation on the y axis.
`sy:`	Amount of scaling on the y axis.

You can use the `Matrix2D()` function to construct such a matrix. See Section 30.25 [Matrix2D], page 606, for details.

INPUTS

`px`	horizontal point coordinate
`py`	vertical point coordinate
`m`	table containing a 2x2 transformation matrix

RESULTS

`tx`	transformed horizontal position
`ty`	transformed vertical position

30.36 VWait

NAME

VWait – wait for vertical blank

SYNOPSIS

```
VWait()
```

FUNCTION

This command waits for the next vertical blank to begin.

INPUTS

none

31 Icon library

31.1 AddIconImage

NAME

idx = AddIconImage – add image to icon (V8.0)

SYNOPSIS

```
AddIconImage(id, table)
```

FUNCTION

This function can be used to add a new image to the icon specified by `id`. The image must be specified in the `table` parameter which must be set to a table that recognizes the following fields:

`Type:` This tag allows you to set the source type of the image you wish to add to the icon. This can be `#BRUSH` if you want to add a brush, or `#FILE` if you would like to add an image from an external file source. The default is `#BRUSH`. Note that that default is different to the default used by the `@ICON` preprocessor command, which is `#FILE`.

`Image:` This tag specifies the actual image source and must be set. If `Type` has been set to `#BRUSH`, you have to set this tag to the identifier of a brush you want to add to the icon. Otherwise, `Image` needs to be set to the path of an image file that should be added to the icon. The image file may be in any of the image file formats supported by Hollywood. Note that if the image file specified here has an alpha channel, the alpha channel data is loaded automatically. Also note that in every icon, each image size must only be used once, i.e. it is not possible to add two 48x48 images to a single icon. There can only be one image for each size in every icon. Also note that it's not possible to add a vector graphics image to an icon because vector icons must only contain a single image. See Section 31.3 [CreateIcon], page 619, for details.

`SelImage:`
This tag allows you to include an additional image that should be used as a selected version of the image specified in `Image`. This tag is optional. If you set it, the image specified here must be of exactly the same size as the one specified in `Image`. Besides that, `SelImage` is used in the very same way as `Image`, i.e. it depends on the image type set in `Type` what you have to pass here, either a brush identifier or a path to an external image file.

`Standard:`
This tag allows you to set the image that is about to be added as the standard size for the icon. Setting a standard size is important in some contexts, so that Hollywood knows which image to pick for higher resolutions, e.g. if you designate a 64x64 image inside an icon as the standard size, Hollywood knows to pick the 128x128 image in case the monitor's resolution uses a DPI setting that is twice as high as the normal setting. Obviously, there can be

only one standard image inside every icon, so if there already is a standard image in the icon, an error is generated.

`AddIconImage()` returns the index where the image has been added inside the icon. Note that this isn't necessarily the last index in the icon because the individual images inside icons are sorted by their width in ascending the order. The indices returned by `AddIconImage()` start at 1.

To remove an image from an icon, use the `RemoveIconImage()` function.

INPUTS

`id` identifier of the icon to use

`table` table describing the image to be added to the icon (see above)

RESULTS

`idx` index where the image has been added (starting at 1)

EXAMPLE

```
AddIconImage(1, {Image = "ic16x16.png"})
```

The code above adds the image file "ic16x16.png" to icon 1.

31.2 ChangeApplicationIcon

NAME

ChangeApplicationIcon – change docky icon (V6.0)

SYNOPSIS

```
ChangeApplicationIcon(id1[, id2, type])
```

PLATFORMS

AmigaOS 4 only

FUNCTION

This function can be used to change your application's icon in AmiDock at runtime. You have to pass at least one brush to this function. If you pass a second brush in the optional argument, then this brush will be used as the icon's second state. Both brushes must have the same dimensions. For the best visual appearance, you should only use brushes with alpha channel transparency with this function.

Please note that only standard dockies support two state icons in AmiDock. If your application is represented by an app docky in AmiDock, you may only specify one image here. See Section 16.1 [AmiDock information], page 165, for more information on the difference between standard and app dockies.

Also note that changing the application icon of standard dockies causes a clearly visible relayout of AmiDock and thus is not apt to display animations in AmiDock. If you want to change the icon in a smooth way, you need to use an app docky. See Section 16.1 [AmiDock information], page 165, for details.

Starting with Hollywood 9.0, you can also use Hollywood icons with `ChangeApplicationIcon()`. To do that, you have to pass `#ICON` in the optional `type` argument. In that case, `id1` needs to be the identifier of an icon that should

be used. If `type` has been set to `#ICON`, the `id2` parameter will be ignored. In case of `#ICON`, the `id2` parameter is unnecessary because in contrast to brushes, Hollywood icons can contain images for multiple states so `ChangeApplicationIcon()` can simply use the selected image stored in the icon.

The initial icon for your application in AmiDock can be specified using the `@APPICON` preprocessor command or the `DockyBrush` tag of the `@OPTIONS` preprocessor command.

Please note that this function can only be used if you have set the `RegisterApplication` tag in `@OPTIONS` to `True`. See Section 52.25 [OPTIONS], page 1088, for details.

INPUTS

`id1`	brush or icon that should replace the current docky icon's normal state
`id2`	optional: brush that should replace the current docky icon's selected state
`type`	optional: type of the object passed in `id1` (must be either `#BRUSH` or `#ICON`, defaults to `#BRUSH`) (V9.0)

31.3 CreateIcon

NAME

CreateIcon – create an icon (V8.0)

SYNOPSIS

```
[id] = CreateIcon(id, table)
```

FUNCTION

This function can be used to create an icon from a collection of individual image files or brushes. You have to pass the desired identifier for the new icon in the `id` parameter. `If` you specify `Nil` in the `id` argument, `CreateIcon()` will automatically choose an identifier for this icon and return it to you. In Hollywood, an icon is a collection of the same image in different sizes and color depths.

By using individually designed images for each size instead of just scaling one and the same image to each size, a better quality is achieved, especially when it comes to smaller image sizes, which look much better when they are specifically designed for their resolution. Typical sizes for the individual images within an icon are 16x16, 24x24, 32x32, 48x48, 64x64, 96x96, 128x128, 256x256, and 512x512, but they can also be completely arbitrary. The advantage of having the same image in different sizes in an icon is that Hollywood can choose an appropriate size depending on the screen resolution.

Furthermore, images inside Hollywood icons can also be specifically designed for different color depths. For example, you can provide 24x24 images in various color depths, e.g. in 256 colors (8 bits) and in true color with alpha channel (32 bits). Thus, it is possible to have images of the same size inside an icon as long as they differ in their color depth. This once again gives Hollywood the advantage of choosing the best image from an icon for a certain screen resolution and color depth.

Finally, each image inside an icon set can have two different states: normal and selected. Normally, you only ever need the normal state, but on AmigaOS and compatibles the selected state is sometimes used as well.

On top of the identifier for the new icon that is to be passed in `id`, you also have to pass a table in the `table` parameter to `CreateIcon()`. This table must contain a number of subtables, one for each image size you wish to add to the icon.

The individual subtables can use the following tags:

`Type:` This tag allows you to set the source type of the image you wish to add to the icon. This can be `#BRUSH` if you want to add a brush, or `#FILE` if you would like to add an image from an external file source. The default is `#BRUSH`. Note that that default is different to the default used by the `@ICON` preprocessor command, which is `#FILE`.

`Image:` This tag specifies the actual image source and must be set in every subtable. If `Type` has been set to `#BRUSH`, you have to set this tag to the identifier of a brush you want to add to the icon. The brush can be an RGB or palette brush. Otherwise, `Image` needs to be set to the path of an image file that should be added to the icon. The image file may be in any of the image file formats supported by Hollywood. Note that if the image file specified here has an alpha channel, the alpha channel data is loaded automatically. If it is a palette image and the `LoadPalette` tag is set to `True` (see below), the image's transparent pen will also be loaded automatically. Also note that in every icon, each image size must only be used once for each color depth, i.e. it is not possible to add two 48x48 images that use the same color depth to a single icon. There can only be one image for each size and color depth in every icon. Note that if the image you specify here is in a vector graphics format, i.e. either a vector brush or a file in a vector image format, you mustn't pass any other images because in the case of vector graphics, one image is used for all sizes and Hollywood will automatically render it to all sizes it needs. So if you use vector instead of raster graphics, there must only be one subtable in the table you pass to `CreateIcon()`.

`SelImage:`
This tag allows you to include an additional image that should be used as a selected version of the image specified in `Image`. This tag is optional. If you set it, the image specified here must be of exactly the same size as the one specified in `Image`. Besides that, `SelImage` is used in the very same way as `Image`, i.e. it depends on the image type set in `Type` what you have to pass here, either a brush identifier or a path to an external image file.

`Standard:`
This tag allows you to set the image specified in this subtable as the standard size for the icon. Setting a standard size is important in some contexts, so that Hollywood knows which image to pick for higher resolutions, e.g. if you designate a 64x64 image inside an icon as the standard size, Hollywood knows to pick the 128x128 image in case the monitor's resolution uses a DPI setting that is twice as high as the normal setting. Obviously, there can be only one standard image inside every icon, so setting this tag to `True` twice will result in an error. Also note that it is not necessary to declare a standard icon size, but for many use cases it is recommended to do it.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load the image files specified in `Image` and `SelImage`. If specified, this must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. Obviously, this tag is only used when `Type` is set to `#FILE`.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the files specified in `Image` and `SelImage`. If specified, this must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. Obviously, this tag is only used when `Type` is set to `#FILE`.

`LoadPalette:`
If this tag is set to `True` and `Type` is set to `#FILE`, Hollywood will try to load the image's palette and store the image as a palette image within the icon. If the image has a transparent pen, that transparent pen will be loaded automatically. (V9.0)

This command is also available from the preprocessor: Use `@ICON` to create icons from the preprocessor.

To add and remove individual images from an icon, use the `AddIconImage()` and `RemoveIconImage()` functions.

INPUTS

`id` identifier for the icon or `Nil` for auto id selection

`table` table containing the individual images to be added to the icon (see above)

EXAMPLE

```
CreateIcon(1, {
    {Image = "ic16x16.png", Type = #FILE},
    {Image = "ic24x24.png", Type = #FILE},
    {Image = "ic32x32.png", Type = #FILE},
    {Image = "ic48x48.png", Type = #FILE},
    {Image = "ic64x64.png", Type = #FILE},
    {Image = "ic96x96.png", Type = #FILE},
    {Image = "ic128x128.png", Type = #FILE},
    {Image = "ic256x256.png", Type = #FILE},
    {Image = "ic512x512.png", Type = #FILE},
    {Image = "ic1024x1024.png", Type = #FILE}})
```

The code above creates icon 1 from a set of external images in different sizes ranging from 16x16 pixels to 1024x1024 pixels.

```
CreateIcon(1, {{Image = "icon.svg", Type = #FILE}})
```

The code above creates icon 1 and uses just a single image because the image is in a vector graphics format (SVG) and in that case only a single image must be specified (see above).

31.4 FreeIcon

NAME

FreeIcon – free an icon (V8.0)

SYNOPSIS

```
FreeIcon(id)
```

FUNCTION

This function frees the memory of the icon specified by `id`. To save memory, you should always free icons when you do not need them any longer.

INPUTS

`id` identifier of the icon to free

31.5 GetIconProperties

NAME

GetIconProperties – retrieve properties from an icon (V4.5)

SYNOPSIS

```
t = GetIconProperties(id)
type, tooltypes, deftool$ = GetIconProperties(file$)
```

FUNCTION

This function can be used to get the properties of an icon. This is mostly useful for Amiga icons because they contain metadata besides the image data, e.g. icon type, tooltypes, default tool, and so on.

There are two ways of using this function: You can either pass the identifier of an icon in the first parameter or the filename of an icon file. Passing a filename to `GetIconProperties()` is only supported on AmigaOS and compatibles, passing an icon identifier is supported on all platforms however.

If you choose to pass the identifier of an icon in the first parameter, `GetIconProperties()` will return a table that has the following fields initialized:

`Type:` This tag will be set to the Amiga icon type of the icon. This will be one of the following constants:

`#AMIGAICON_DISK:`
An icon of a drive (e.g. RAM, HD, CD-ROM, etc.)

`#AMIGAICON_DRAWER:`
An icon of a drawer.

`#AMIGAICON_TOOL:`
An icon of a program

`#AMIGAICON_PROJECT:`
An icon of a project. A project is a data file that can be opened by an other program. The program that should be

used to open the project will be passed in `DefaultTool`, e.g. `SYS:Utilities/MultiView`.

`#AMIGAICON_GARBAGE:`
A trashcan icon.

`##AMIGAICON_DEVICE:`
A device icon.

`#AMIGAICON_KICKSTART:`
A Kickstart icon.

(V9.0)

`IconX:` The icon's x position relative to the top-left corner of the drawer it is stored in. (V9.0)

`IconY:` The icon's y position relative to the top-left corner of the drawer it is stored in. (V9.0)

`DrawerX:` In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag will be set to the x position of the new window that will be opened when double-clicking the icon. (V9.0)

`DrawerY:` In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag will be set to the y position of the new window that will be opened when double-clicking the icon. (V9.0)

`DrawerWidth:`
In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag will be set to the width of the new window that will be opened when double-clicking the icon. (V9.0)

`DrawerHeight:`
In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag will be set to the height of the new window that will be opened when double-clicking the icon. (V9.0)

`ViewAll:` In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag will be set to `True` if all files of the drawer should be shown instead of just the ones that have an icon. (V9.0)

`StackSize:`
In case `Type` is set to `#AMIGAICON_TOOL` or `#AMIGAICON_PROJECT`, the desired stack size for the program to be launched. (V9.0)

`DefaultTool:`
In case `Type` is set to `#AMIGAICON_PROJECT`, this will be set to a string containing name (and optionally path) of the program to open the file with. (V9.0)

`ToolTypes:`
If there are tooltypes in the icon, they will be returned in this tag. The `ToolTypes` tag will contain a table that contains a list of subtables, one subtable per tooltype entry. Each subtable will contain the following tags:

`Key:` This tag contains the name of the tooltype.

`Value:` This tag contains the tooltype's value.

`Enabled:` This tag is a boolean that indicates whether or not the tooltype is enabled.

Alternatively, you can also get the tooltypes by using the `RawToolTypes` tag (see below).

(V9.0)

`RawToolTypes:`
If you don't want to use the `ToolTypes` tag for some reason, you can also use `RawToolTypes` to get them. In contrast to `ToolTypes`, the `RawToolTypes` tag doesn't divide tooltypes into their individual constituents (key, value, enabled flag). Instead, the `RawToolTypes` tag will just return the tooltypes as they are stored in the icon, i.e. without any additional processing. This makes it possible to get custom data stored in tooltypes as well. `RawToolTypes` will be set to a table that contains the tooltypes as strings. (V9.0)

`Images:` This tag will be set to a table containing information about the individual images within the icon. The table will contain a subtable for each image in the icon. Each subtable will have the following fields:

`Width:` Image width in pixels.

`Height:` Image height in pixels.

`Standard:`
`True` if the image is the standard image, `False` otherwise.

`Frames:` Number of image frames. This can be either 1 or 2. If it is 1, there is only a normal image, if it is 2, there is a selected state image as well.

(V9.0)

To set the properties of an icon, use the `SetIconProperties()` command.

INPUTS

`id` syntax 1: identifier of an icon to examine

`file$` syntax 2: the icon to examine

RESULTS

`t` syntax 1: table containing icon properties

`type` syntax 2: type of the icon; will be one of the constants from above

`tooltypes`
syntax 2: a table containing a list of all tooltypes; each list entry will have the fields `Key`, `Value`, and `Enabled` initialized

`deftool$` syntax 2: the default tool set for this icon; only set for icons of type `#AMIGAICON_PROJECT`

EXAMPLE

```
t = GetIconProperties(1)
For k = 0 To ListItems(t.ToolTypes) - 1
   DebugPrint("Item:", k, "Key:", t.ToolTypes[k].key,
       "Value:", t.ToolTypes[k].value,
       "Enabled:", t.ToolTypes[k].enabled)
Next
```

The code above gets the properties of icon 1 and then prints all information stored in its tooltypes.

```
type, tt, deftool$ = GetIconProperties("MyIcon.info")
For k = 0 To ListItems(tt) - 1
   DebugPrint("Item:", k, "Key:", tt[k].key, "Value:", tt[k].value,
               "Enabled:", tt[k].enabled)
Next
```

The code above gets the properties of the icon "MyIcon.info" and then prints all information stored in its tooltypes.

31.6 ICON

NAME

ICON – preload an icon for later use (V8.0)

SYNOPSIS

```
@ICON id, filename$[, table]
@ICON id, table
```

FUNCTION

This preprocessor command can be used to preload an icon for later use, either from image file sources or from brush sources. In Hollywood, an icon is a collection of the same image in different sizes and color depths.

By using individually designed images for each size instead of just scaling one and the same image to each size, a better quality is achieved, especially when it comes to smaller image sizes, which look much better when they are specifically designed for their resolution. Typical sizes for the individual images within an icon are 16x16, 24x24, 32x32, 48x48, 64x64, 96x96, 128x128, 256x256, and 512x512, but they can also be completely arbitrary. The advantage of having the same image in different sizes in an icon is that Hollywood can choose an appropriate size depending on the screen resolution.

Furthermore, images inside Hollywood icons can also be specifically designed for different color depths. For example, you can provide 24x24 images in various color depths, e.g. in 256 colors (8 bits) and in true color with alpha channel (32 bits). Thus, it is possible to have images of the same size inside an icon as long as they differ in their color depth. This once again gives Hollywood the advantage of choosing the best image from an icon for a certain screen resolution and color depth.

Finally, each image inside an icon set can have two different states: normal and selected. Normally, you only ever need the normal state, but on AmigaOS and compatibles the selected state is sometimes used as well.

This preprocessor command can be used in two different ways: You can either specify a single file in `filename$` to be loaded as a Hollywood icon or you can specify a table to create a Hollywood icon from individual images which may be specified as either brushes or external files.

In case you use the first syntax, i.e. you pass a single file in `filename$`, `@ICON` expects the file to be in Hollywood's custom PNG icon format and `@ICON` does the same as the `LoadIcon()` command in that case, except that the icon is preloaded and will be linked to the applet or executable on compiling. See Section 31.7 [LoadIcon], page 629, for details.

The first syntax also accepts a table argument that follows the `filename$` parameter. The following fields in the table can be set in case you use the first syntax:

`Link:` Set this field to `False` if you do not want to have this icon linked to your executable/applet when you compile your script. This field defaults to `True` which means that the icon is linked to your executable/applet when Hollywood is in compile mode.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this icon. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Alternatively, you can use the second syntax which allows you to create an icon from a set of source images which may either be external files or Hollywood brushes. In that case, you have to specify a single table argument which must contain a number of subtables, one for each image size you wish to add to the icon. This is similar to the way the `CreateIcon()` command works. See Section 31.3 [CreateIcon], page 619, for details.

The individual subtables can use the following tags:

`Type:` This tag allows you to set the source type of the image you wish to add to the icon. This can be `#BRUSH` if you want to add a brush, or `#FILE` if you would like to add an image from an external file source. The default is `#FILE`. Note that that default is different to the default used by `CreateIcon()` and `AddIconImage()`, which is `#BRUSH`.

`Image:` This tag specifies the actual image source and must be set in every subtable. If `Type` has been set to `#BRUSH`, you have to set this tag to the identifier of a brush you want to add to the icon. The brush can be an RGB or palette brush. Otherwise, `Image` needs to be set to the path of an image file that should be added to the icon. The image file may be in any of the image file formats supported by Hollywood. Note that if the image file specified here has an alpha channel, the alpha channel data is loaded automatically. If it is a palette image and the `LoadPalette` tag is set to `True` (see below), the image's transparent pen will also be loaded automatically. Also note that in every icon, each image size must only be used once for each color depth, i.e. it is not possible to add two 48x48 images that use the same color depth to a single icon. There can only be one image for each size and color depth in every icon. Note that if the image you specify here is in a vector graphics format, i.e. either a vector brush or a file in a vector image format, you mustn't pass any other images because in the case of vector graphics, one image is used for all sizes and Hollywood will automatically render it to all sizes it needs. So if you use vector instead of raster graphics, there must only be one subtable in the table you pass to `@ICON`.

`SelImage:`

This tag allows you to include an additional image that should be used as a selected version of the image specified in `Image`. This tag is optional. If you set it, the image specified here must be of exactly the same size as the one specified in `Image`. Besides that, `SelImage` is used in the very same way as `Image`, i.e. it depends on the image type set in `Type` what you have to pass here, either a brush identifier or a path to an external image file.

`Standard:`

This tag allows you to set the image specified in this subtable as the standard size for the icon. Setting a standard size is important in some contexts, so that Hollywood knows which image to pick for higher resolutions, e.g. if you designate a 64x64 image inside an icon as the standard size, Hollywood knows to pick the 128x128 image in case the monitor's resolution uses a DPI setting that is twice as high as the normal setting. Obviously, there can be only one standard image inside every icon, so setting this tag to `True` twice will result in an error. Also note that it is not necessary to declare a standard icon size, but for many use cases it is recommended to do it.

`Link:` Set this field to `False` if you do not want to have the image files specified in `Image` and `SelImage` linked to your executable/applet when you compile your script. This field defaults to `True` which means that the image files specified in `Image` and `SelImage` will be linked to your executable/applet when Hollywood is in compile mode. Obviously, this tag is only used when `Type` is set to `#FILE`.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load the image files specified in `Image` and `SelImage`. If specified, this must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Load-

ers and adapters], page 92, for details. Obviously, this tag is only used when `Type` is set to `#FILE`.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the files specified in `Image` and `SelImage`. If specified, this must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. Obviously, this tag is only used when `Type` is set to `#FILE`.

`LoadPalette:`
If this tag is set to `True` and `Type` is set to `#FILE`, Hollywood will try to load the image's palette and store the image as a palette image within the icon. If the image has a transparent pen, that transparent pen will be loaded automatically. (V9.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

To load or create icons at runtime, take a look at the `LoadIcon()` and `CreateIcon()` commands.

To add and remove individual images from an icon, use the `AddIconImage()` and `RemoveIconImage()` functions.

INPUTS

`id` a value that is used to identify this icon later in the code

`filename$`
optional: the icon file you want to load (see above)

`table` optional or mandatory, depending on which syntax you use (see above for a discussion)

EXAMPLE

```
@ICON 1, "MyIcon.png"
```

Loads "MyIcon.png" as icon 1.

```
@ICON 1, {
    {Image = "ic16x16.png"},
    {Image = "ic24x24.png"},
    {Image = "ic32x32.png"},
    {Image = "ic48x48.png"},
    {Image = "ic64x64.png"},
    {Image = "ic96x96.png"},
    {Image = "ic128x128.png"},
    {Image = "ic256x256.png"},
    {Image = "ic512x512.png"},
```

```
{Image = "ic1024x1024.png"}}
```

The code above creates icon 1 from a set of external images in different sizes ranging from 16x16 pixels to 1024x1024 pixels.

```
@ICON 1, {{Image = "icon.svg"}}
```

The code above creates icon 1 and uses just a single image because the image is in a vector graphics format (SVG) and in that case only a single image must be specified (see above).

31.7 LoadIcon

NAME

LoadIcon – load an icon (V8.0)

SYNOPSIS

```
[id] = LoadIcon(id, filename$[, table])
```

FUNCTION

This function loads the icon specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadIcon()` will automatically choose an identifier and return it.

In Hollywood, an icon is a collection of the same image in different sizes. By using individually designed images for each size instead of just scaling one and the same image to each size, a better quality is achieved, especially when it comes to smaller image sizes, which look much better when they are specifically designed for their resolution. Additionally, each image inside an icon set can have two different states: normal and selected. Normally, you only ever need the normal state, but on AmigaOS and compatibles the selected state is sometimes used as well. Typical sizes for the individual images within an icon are 16x16, 24x24, 32x32, 48x48, 64x64, 96x96, 128x128, 256x256, and 512x512, but they can also be completely arbitrary. The advantage of having the same image in different sizes in an icon is that Hollywood can choose an appropriate size depending on the screen resolution.

The icon passed in `filename$` must be in Hollywood's custom PNG icon format. You can use `SaveIcon()` to create such icons. Note that although Hollywood icons are normal PNG images, they contain additional metadata which is why you mustn't edit them with your favourite image manipulation tool because that might lead to the loss of said metadata. Hollywood icons should only ever be created by using the `SaveIcon()` function.

If you don't want to use Hollywood's custom icon format, you can also create icons from brushes or normal images using the `CreateIcon()` function or the `@ICON` preprocessor command. Those commands also have the advantage that you can use vector brushes, which can be losslessly scaled to any size, leading to a perfectly crisp look in all kinds of different resolutions.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this icon. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

This command is also available from the preprocessor: Use `@ICON` to preload icons.

INPUTS

`id` identifier for the icon or `Nil` for auto id selection

`filename$`
file to load

`table` optional: further options (see above)

RESULTS

`id` optional: identifier of the icon; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadIcon(1, "test.png")
```

This loads "test.png" as icon 1.

31.8 RemoveIconImage

NAME

RemoveIconImage – remove image from icon (V8.0)

SYNOPSIS

```
RemoveIconImage(id, idx)
```

FUNCTION

This command can be used to remove a single image from the icon specified by `id`. The image to be removed must be specified by its index using the `idx` parameter. Indices start at 1 for the first image and run up to the number of images in the icon. You can query the number of images in an icon by using the `#ATTRNUMENTRIES` attribute with `GetAttribute()`.

Note that the individual images inside icons are sorted by their width in ascending order. This means that the indices passed to `RemoveIconImage()` aren't necessarily the same

as the order of images passed to functions like `CreateIcon()` or the `@ICON` preprocessor command.

Since icons must contain at least one image, it's also not allowed to remove the very last image from an icon.

To add images to icons, use the `AddIconImage()` function.

INPUTS

`id` identifier of the icon to use

`idx` index of the image to remove (starting at 1)

EXAMPLE

```
RemoveIconImage(1, 1)
```

The code above removes the first image from icon 1.

31.9 SaveIcon

NAME

SaveIcon – save icon to a file (V8.0)

SYNOPSIS

```
SaveIcon(id, f$[, fmt, t])
```

FUNCTION

This function saves the icon specified by `id` to the file specified by `f$` in By default, the icon will be saved in Hollywood's custom icon format based on PNG. You can change this by passing a different icon format constant in the `fmt` argument. The only icon format supported internally by Hollywood is `#ICNFMT_HOLLYWOOD`, which is Hollywood's custom icon format based on PNG. Additional icon formats might be made available by Hollywood plugins.

Note that although Hollywood's custom icon format stores icons as seemingly normal PNG images, they contain additional metadata which is why you mustn't edit them with your favourite image manipulation tool because that might lead to the loss of said metadata. Hollywood icons should only ever be created by using the `SaveIcon()` function.

Also note that when using Hollywood's custom icon format the icon specified in `id` mustn't contain any vector graphics. Hollywood icons only support raster graphics because they are based on PNG which is a raster graphics format. If you want to use vector graphics in an icon, you can create such icons using the `CreateIcon()` function and the `@ICON` preprocessor command.

Finally, icons in Hollywood's custom icon format also mustn't contain any palette graphics. That is why `SaveIcon()` will fail if the icon specified by `id` contains palette images.

Starting with Hollywood 9.0, `SaveIcon()` accepts an optional table argument that can contain the following options:

`Compression:`

For icon formats that support compression, you can set this tag to `True` or `False` to enable or disable compression. Defaults to `True`. (V9.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`id` identifier of the icon to save

`f$` destination file

`fmt` optional: desired output icon format (defaults to `#ICNFMT_HOLLYWOOD`) (V9.0)

`t` optional: table containing further arguments (V9.0)

31.10 SetIconProperties

NAME

SetIconProperties – change properties of an icon (V4.5)

SYNOPSIS

```
SetIconProperties(id, table)
SetIconProperties(file$, type[, tooltypes, deftool$]) (Amiga only)
```

FUNCTION

This function can be used to change the properties of an icon. This is mostly useful for Amiga icons because those contain metadata besides the image data, e.g. icon type, tooltypes, default tool, and so on. This metadata can be modified using `SetIconProperties()`.

There are two ways of using this function: You can either pass the identifier of an icon in the first parameter and a table in the second parameter or you can pass the filename of an icon file in the first parameter and more arguments in the following parameters. Passing a filename to `SetIconProperties()` is only supported on AmigaOS and compatibles, passing an icon identifier is supported on all platforms however.

If you choose to pass an icon identifier to `SetIconProperties()`, you have to pass a table in the second argument to specify the actual properties to modify. The following icon properties can currently be modified:

`Type:` This tag allows you to set the desired Amiga icon type for the icon. This must be one of the following constants:

`#AMIGAICON_DISK:`
An icon of a drive (e.g. RAM, HD, CD-ROM, etc.)

`#AMIGAICON_DRAWER:`
An icon of a drawer.

`#AMIGAICON_TOOL:`
An icon of a program

`#AMIGAICON_PROJECT:`
An icon of a project. A project is a data file that can be opened by an other program. The program that should be used to open the project should be passed in the `DefaultTool` tag (see below).

`#AMIGAICON_GARBAGE:`
A trashcan icon.

`##AMIGAICON_DEVICE:`
A device icon.

`#AMIGAICON_KICKSTART:`
A Kickstart icon.

(V9.0)

`IconX:` The icon's x position relative to the top-left corner of the drawer it is stored in. (V9.0)

`IconY:` The icon's y position relative to the top-left corner of the drawer it is stored in. (V9.0)

`DrawerX:` In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag can be used to set the x position of the new window that will be opened when double-clicking the icon. (V9.0)

`DrawerY:` In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag can be used to set the y position of the new window that will be opened when double-clicking the icon. (V9.0)

`DrawerWidth:`
In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag can be used to set the width of the new window that will be opened when double-clicking the icon. (V9.0)

`DrawerHeight:`
In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, this tag can be used to set the height of the new window that will be opened when double-clicking the icon. (V9.0)

`ViewAll:` In case `Type` is set to a container type like `#AMIGAICON_DRAWER`, you can set this tag to `True` to tell Workbench to show all files of the drawer, not just the ones that have an icon. (V9.0)

`StackSize:`
In case `Type` is set to `#AMIGAICON_TOOL` or `#AMIGAICON_PROJECT`, the desired stack size for the program to be launched. (V9.0)

`DefaultTool:`
In case `Type` is set to `#AMIGAICON_PROJECT`, this can be set to a string containing name (and optionally path) of the program to open the file with. (V9.0)

`ToolTypes:`
If you want to add tooltypes to the icon, you have to pass a table in the `ToolTypes` tag. The table must contain a list of subtables, one subtable per tooltype entry. Each subtable can contain the following tags:

`Key:` This tag is mandatory. It specifies the name of the tooltype. Tooltype names should use letters of the English alphabet only. They should always be in upper-case format and must not use any SPACE characters. If you want a SPACE, use an underscore instead (`"_"`). Furthermore, numbers should not be used as the initial characters of a tooltype name.

`Value:` This tag is optional. You can use it to assign a value to the tooltype. If you do not set this value, the tooltype will be a boolean one.

`Enabled:` This tag is optional. It defaults to `True`. If you want to add tooltypes that are initially disabled, you can set this tag to `False`. In that case, the tooltype will be enclosed by parentheses which means that it is disabled.

Alternatively, you can also set tooltypes by using the `RawToolTypes` tag (see below).
(V9.0)

`RawToolTypes:`
If you don't want to use the `ToolTypes` tag for some reason, you can also use `RawToolTypes` to set them. In contrast to `ToolTypes`, the `RawToolTypes` tag doesn't divide tooltypes into their individual constituents (key, value, enabled flag). Instead, the `RawToolTypes` tag will just copy the tooltypes to the icon without any additional processing. This makes it possible to store custom data in tooltypes as well. If you want to do that, just set `RawToolTypes` to a table that contains all tooltypes that should be set as simple strings. (V9.0)

To read the properties of an icon, use the `GetIconProperties()` command.

INPUTS

`id` syntax 1: identifier of icon to modify

`table` syntax 1: table containing icon properties to set

`file$` syntax 2: the icon to modify

`type` syntax 2: new type for the icon; must be one of the constants from above

`tooltypes`
syntax 2, optional: a table containing a list of all tooltypes you want to set; each list entry must have at least the `Key` field set; defaults to {} (empty table)

`deftool$` syntax 2, optional: the default tool to set for this icon; will only be set for icons of type `#AMIGAICON_PROJECT`; defaults to `""`

EXAMPLE

```
SetIconProperties(1, {
    Type = #AMIGAICON_PROJECT,
    DefaultTool = "Hollywood:System/Hollywood",
    ToolTypes = {
       {Key = "BORDERLESS"},
       {Key = "BACKFILL", Value = "GRADIENT"},
       {Key = "STARTCOLOR", Value = "$000000"},
       {Key = "ENDCOLOR", Value = "$0000ff"},
       {Key = "FIXED", Enabled = False}
    }
})
```

The code above sets the type of icon 1 to `#AMIGAICON_PROJECT`, the default tool to "Hollywood:System/Hollywood" and adds some tooltypes.

```
SetIconProperties("MyCoolScript.hws.info", #AMIGAICON_PROJECT, {
{Key = "BORDERLESS"},
{Key = "BACKFILL", Value = "GRADIENT"},
{Key = "STARTCOLOR", Value = "$000000"},
{Key = "ENDCOLOR", Value = "$0000ff"},
{Key = "FIXED", Enabled = False} }, "Hollywood:System/Hollywood")
```

The code above sets Hollywood as the default tool for "MyCoolScript.hws". Furthermore, it adds several tooltypes to the script's icon that tell Hollywood what eye candy it should add to the script (e.g. gradient backfill).

31.11 SetStandardIconImage

NAME

SetStandardIconImage – set icon's standard image (V8.0)

SYNOPSIS

```
SetStandardIconImage(id, idx)
```

FUNCTION

This command can be used to set the standard image of the icon specified by `id`. The image to be made the standard one must be specified by its index using the `idx` parameter. Indices start at 1 for the first image and run up to the number of images in the icon. You can query the number of images in an icon by using the `#ATTRNUMENTRIES` attribute with `GetAttribute()`.

Note that the individual images inside icons are sorted by their width in ascending order. This means that the indices passed to `SetStandardIconImage()` aren't necessarily the same as the order of images passed to functions like `CreateIcon()` or the `@ICON` preprocessor command.

Setting an image inside the icon as the standard one can be important in some contexts so that Hollywood knows which image to pick for higher resolutions, e.g. if you designate a 64x64 image inside an icon as the standard size, Hollywood knows to pick the 128x128 image in case the monitor's resolution uses a DPI setting that is twice as high as the normal setting. Obviously, there can be only one standard image inside every icon, so making one image the standard one will automatically clear the standard flag for any image that was the standard one before. To make no image the standard one, pass 0 in `idx`.

INPUTS

`id`	identifier of the icon to use
`idx`	index of the image to set as standard (starting at 1)

EXAMPLE

```
SetStandardIconImage(1, 1)
```

The code above makes the first image 1 in icon 1 the standard one.

31.12 SetTrayIcon

NAME

SetTrayIcon – install brush as a system tray icon (V5.2)

SYNOPSIS

```
SetTrayIcon(id[, tooltip$, type])
```

PLATFORMS

Microsoft Windows only

FUNCTION

This function can be used to install the image specified in `id` as an icon in the Windows system tray. Whenever the user clicks on this icon, your script will get an event of type `TrayIcon` which you can listen to using `InstallEventHandler()`. The optional argument `tooltip$` can be used to specify a string that should be displayed as a tooltip whenever the mouse hovers over the system tray icon.

The image that you pass to this function should be 16x16 pixels and should use an alpha channel for transparency.

In case you have already installed a system tray icon when you call this function, the icon will be changed to the graphics of the specified brush. If you pass the special value `#NONE` as the brush identifier, the system tray icon will be removed.

Another special value that you can pass to this function is `#DEFAULTICON`. If you pass `#DEFAULTICON` in `id`, `SetTrayIcon()` will use the icon that has been declared using the `@APPICON` preprocessor command, or, in case no `@APPICON` declaration has been made, Hollywood's default icon, the clapperboard.

Starting with Hollywood 8.0 there is an optional `type` argument which allows you to specify the source image type for the tray icon. This defaults to `#BRUSH` which means that you have to pass the identifier of a brush in the `id` argument. However, you can also set the `type` argument to `#ICON`, in which case you have to pass the identifier of a

Hollywood icon in the `id` argument. This has the advantage that Hollywood can choose different images depending on the resolution of the host system's monitor. This is very useful for systems using high DPI monitors. By using an icon that contains an image in several resolutions, you can make sure that the tray icon looks perfectly crisp even on high DPI monitors. See Section 31.3 [CreateIcon], page 619, for details.

Note that if you pass an icon in `id`, you have to make sure to set the 16x16 image as the standard image inside the icon because 16x16 pixels is the default icon size for the Windows system tray. See Section 31.11 [SetStandardIconImage], page 635, for details.

INPUTS

`id` identifier of the image to use as system tray icon or `#NONE` or `#DEFAULTICON`

`tooltip$` optional: text to display as an icon tooltip

`type` optional: type of the image passed in `id`; this can be either `#BRUSH` or `#ICON` (defaults to `#BRUSH`) (V8.0)

EXAMPLE

```
InstallEventHandler({TrayIcon = ...})
SetTrayIcon(1, "My program")
```

The code above enables the tray icon event handler and then installs brush number 1 as a system tray icon.

31.13 SetWBIcon

NAME

SetWBIcon – change Hollywood's iconify icon (V4.5)

SYNOPSIS

```
SetWBIcon(icon$[, ...])
```

PLATFORMS

AmigaOS and compatibles only

FUNCTION

This function can be used to specify your custom icon that Hollywood shall show on the Workbench when it is in iconified state. You must specify an icon file here that is in the `*.info` format. All icons that the currently installed Workbench can read are supported. Thus, if you are on MorphOS or are using an appropriate patch, you could also use PNG icons here.

For best compatibility, however, you should stick to standard icons in the GlowIcon format.

The following special constants can be passed to `icon$`:

`#AMIGAICON_NONE`:
: Pass this if you do not want Hollywood to add an appicon to the Workbench when it is iconified. (V5.2)

`#AMIGAICON_SHOW:`
: Show the app icon. This is useful if you want your app icon to be permanently shown on Workbench screen and not only when your program is iconified. (V6.1)

`#AMIGAICON_HIDE:`
: Hide the app icon. (V6.1)

`#AMIGAICON_SETTITLE:`
: Set text to show below the app icon. This defaults to what you specified in the `@APPTITLE` preprocessor command. The text to show needs to be passed as the second argument. (V6.1)

`#AMIGAICON_SETPOSITION:`
: Change position of the app icon. You have to pass two additional arguments specifying the new x and y position of the app icon. If you omit the two additional arguments, the app icon's position will be reset to the position stored in the `*.info` file. (V6.1)

Note that you might need to call `SetWBIcon()` several times in order to achieve the desired effect. For example, if you'd like to change the app icon and show it permanently, you first have to call `SetWBIcon()` to set the `*.info` file to show and then you have to call `SetWBIcon()` again and pass `#AMIGAICON_SHOW` to permanently show your app icon.

INPUTS

`icon$` icon file to use when iconified or a special constant (see above)

`...` additional arguments depending on the special constant passed (see above)

EXAMPLE

```
SetWBIcon("MyCoolProg.info")
```

This code uses the program's icon as its default WB icon.

32 IPC library

32.1 CreatePort

NAME

CreatePort – create a message port for your script (V5.0)

SYNOPSIS

`CreatePort(name$)`

FUNCTION

This function will create a message port for your script and assign the specified name to it. In order to receive messages sent by `SendMessage()`, your script needs to have a message port. Other Hollywood applications can then communicate with your script by sending messages to this port. All messages that arrive at your message port will be forwarded to the callback function which you need to install using the `InstallEventHandler()` function (use the `OnUserMessage` event handler). If you do not install this event handler, you will not get any notifications on incoming messages.

Please remember that message port names are always given in case sensitive notation. Thus, "MYPORT" and "myport" denote two different message ports. For style reasons it is suggested that you use only upper case characters for your port name. Furthermore, each message port must be unique in the system. If you specify a port name which is already in use, this function will fail. Thus, make sure that you use a unique name.

Please note that every Hollywood script can only have one message port. If you have already created a message port and call this function again, the old message port will be deleted.

See Section 29.13 [InstallEventHandler], page 553, for more information on how the user callback function will be called.

INPUTS

`name$` desired name for your message port

EXAMPLE

```
Function p_EventFunc(msg)
  Switch msg.action
  Case "OnUserMessage"
    Switch msg.command
    Case "EXIT"
      DebugPrint("Exit received! Quitting now.")
      End
    Default
      Local t = SplitStr(msg.args, "\0")
      DebugPrint(msg.command, "called with", msg.argc, "argument(s)")
      For Local k = 1 To msg.argc
         DebugPrint("Argument", k .. ":", t[k - 1])
      Next
    EndSwitch
```

```
  EndSwitch
EndFunction
CreatePort("MY_COOL_PORT_123")
InstallEventHandler({OnUserMessage = p_EventFunc})
Repeat
  WaitEvent
Forever
```

Save the code above as a Hollywood script and run it with Hollywood. Then save the following code as a new Hollywood script and run it:

```
SendMessage("MY_COOL_PORT_123", "INIT", "Value1", "Value2", "Value3")
SendMessage("MY_COOL_PORT_123", "DO_SOMETHING", "Argument1")
SendMessage("MY_COOL_PORT_123", "EXIT")
```

The code above will then communicate with the first script. You can see that the messages are arriving from the debug output of script number one.

32.2 SendMessage

NAME

SendMessage – send message to a message port (V5.0)

SYNOPSIS

`SendMessage(port$, cmd$[, ...])`

FUNCTION

This function sends the command specified in `cmd$` to the message port specified in `port$`. The command specified in `cmd$` must not contain any space characters. Additionally, you can send an unlimited number of arguments to the message port. Just pass them as optional arguments after the command name. The optional arguments must be passed as strings.

The port specified in `port$` must have been created previously by a call to `CreatePort()`. Please remember that port names are case sensitive, i.e. `"MYPORT"` and `"myport"` denote two different message ports. For style guide reasons, port names are usually in upper case only.

The message will be sent to the specified message port in form of a `OnUserMessage` event that will be forwarded to the callback you specified when installing this event handler using `InstallEventHandler()`.

INPUTS

`port$`	name of the port you want to address
`cmd$`	the command(s) you want to send to that port
`...`	optional: additional string arguments to send to the port

EXAMPLE

See Section 32.1 [CreatePort], page 639.

33 Joystick library

33.1 ConfigureJoystick

NAME
ConfigureJoystick – set joystick options (V10.0)

SYNOPSIS
`ConfigureJoystick(t)`

FUNCTION
This function can be used to configure several joystick options. You have to pass a table as the sole function parameter `t`. This table can contain the following tags:

`UseAmigaInput:`
This tag is only supported on AmigaOS 4. If you set it to `True`, OS4's AmigaInput system will be used to query joystick states. The advantage of using AmigaInput instead of lowlevel.library, which is the default on Amiga, is that you can query more than 7 buttons and users don't have to use the AmigaInput prefs to map their controllers to lowlevel.library ports.

INPUTS

`t`	table containing one or more options (see above)

33.2 CountJoysticks

NAME
CountJoysticks – return number of joysticks currently plugged in (V4.6)

SYNOPSIS
`r = CountJoysticks()`

FUNCTION
This function counts the number of joysticks currently plugged in. You can then query the single joysticks using commands like `JoyDir()` and `JoyButton()`. This function is useful to check if there is a joystick available at all. If it returns 0, then there is currently no joystick that is recognized by Hollywood.

INPUTS
none

RESULTS

`r`	number of joysticks available or 0 if none

33.3 JoyAxisX

NAME

JoyAxisX – return state of joystick's x axis (V10.0)

SYNOPSIS

```
state = JoyAxisX(port[, idx])
```

FUNCTION

This function returns the current x axis state of the joystick at the port specified by `port`. The x axis state is returned in the range of -1000 to 1000. The optional argument `idx` can be used to specify the index of the joystick to use in case there are multiple joysticks on a controller. Joystick indices start at 0.

`port` can range from 0 to the number of joysticks currently plugged in minus 1. You can find out the number of joysticks currently available using the `CountJoysticks()` function. Please note that under AmigaOS, port 0 addresses the standard joystick port although this is port 1 on classic Amiga hardware. Hollywood switches these ports for cross-platform consistency where port 0 shall always refer to the default joystick.

INPUTS

`port`	port number (usually 0 for the standard Joystick port)
`idx`	optional: joystick index to query (defaults to 0)

RESULTS

`state`	state of the joystick x axis, ranging from -1000 to 1000

33.4 JoyAxisY

NAME

JoyAxisY – return state of joystick's y axis (V10.0)

SYNOPSIS

```
state = JoyAxisY(port[, idx])
```

FUNCTION

This function returns the current y axis state of the joystick at the port specified by `port`. The y axis state is returned in the range of -1000 to 1000. The optional argument `idx` can be used to specify the index of the joystick to use in case there are multiple joysticks on a controller. Joystick indices start at 0.

`port` can range from 0 to the number of joysticks currently plugged in minus 1. You can find out the number of joysticks currently available using the `CountJoysticks()` function. Please note that under AmigaOS, port 0 addresses the standard joystick port although this is port 1 on classic Amiga hardware. Hollywood switches these ports for cross-platform consistency where port 0 shall always refer to the default joystick.

INPUTS

`port`	port number (usually 0 for the standard Joystick port)
`idx`	optional: joystick index to query (defaults to 0)

RESULTS

`state` state of the joystick y axis, ranging from -1000 to 1000

33.5 JoyAxisZ

NAME

JoyAxisZ – return state of joystick's z axis (V10.0)

SYNOPSIS

```
state = JoyAxisZ(port[, idx])
```

FUNCTION

This function returns the current z axis state of the joystick at the port specified by `port`. The z axis state is returned in the range of -1000 to 1000. The optional argument `idx` can be used to specify the index of the joystick to use in case there are multiple joysticks on a controller. Joystick indices start at 0.

`port` can range from 0 to the number of joysticks currently plugged in minus 1. You can find out the number of joysticks currently available using the `CountJoysticks()` function. Please note that under AmigaOS, port 0 addresses the standard joystick port although this is port 1 on classic Amiga hardware. Hollywood switches these ports for cross-platform consistency where port 0 shall always refer to the default joystick.

INPUTS

`port` port number (usually 0 for the standard Joystick port)

`idx` optional: joystick index to query (defaults to 0)

RESULTS

`state` state of the joystick z axis, ranging from -1000 to 1000

33.6 JoyButton

NAME

JoyButton – check if joystick button is pressed (V1.5)

FORMERLY KNOWN AS

JoyFire (V1.5 - V10.0)

SYNOPSIS

```
pressed = JoyButton(port[, button])
```

FUNCTION

This function returns `True` if a button of the Joystick plugged into the port specified by `port` has been pressed. Otherwise `False` is returned. The optional argument `button` specifies which button to look for. If you are looking for a specific button, specify the number of this button (must be between 1 and 32). If you are looking for multiple buttons, specify 0 and this function will return a 32-bit mask in which each of the 32 bits indicates the state of the button (pressed or not pressed).

`port` can range from 0 to the number of joysticks currently plugged in minus 1. You can find out the number of joysticks currently available using the `CountJoysticks()` function. Please note that under AmigaOS, port 0 addresses the standard joystick port although this is port 1 on classic Amiga hardware. Hollywood switches these ports for cross-platform consistency where port 0 shall always refer to the default joystick.

INPUTS

`port` port number (usually 0 for the standard Joystick port)

`button` optional: button to look for or 0 for all buttons (defaults to 1 which means look for first, i.e. fire, button) (V4.6)

RESULTS

`pressed` `True` if button is pressed, otherwise FALSE; if you passed 0 for `button`, then this will be a 32-bit mask indicating the states of all 32 buttons

EXAMPLE

```
While fire = FALSE
  fire = JoyButton(0)
  VWait
Wend
```

The above code waits until the user presses fire.

33.7 JoyDir

NAME

JoyDir – return direction of joystick (V1.5)

SYNOPSIS

```
dir = JoyDir(port[, idx])
```

FUNCTION

This function returns the direction of the Joystick plugged into the port specified by `port`. The optional argument `idx` can be used to specify the index of the joystick to use in case there are multiple joysticks on a controller. Joystick indices start at 0.

One of the following states will be returned:

`#JOYUP` Joystick direction is up

`#JOYUPRIGHT`
Joystick direction is up-right

`#JOYRIGHT`
Joystick direction is right

`#JOYDOWNRIGHT`
Joystick direction is down-right

`#JOYDOWN` Joystick direction is down

`#JOYDOWNLEFT`
Joystick direction is down-left

`#JOYLEFT` Joystick direction is left

`#JOYUPLEFT`
Joystick direction is up-left

`#JOYNODIR`
no direction selected

`port` can range from 0 to the number of joysticks currently plugged in minus 1. You can find out the number of joysticks currently available using the `CountJoysticks()` function. Please note that under AmigaOS, port 0 addresses the standard joystick port although this is port 1 on classic Amiga hardware. Hollywood switches these ports for cross-platform consistency where port 0 shall always refer to the default joystick.

INPUTS

`port` port number (usually 0 for the standard Joystick port)

`idx` optional: joystick index to query (defaults to 0) (V10.0)

RESULTS

`dir` current Joystick state (one of the constants from above)

EXAMPLE

```
While state <> #JOYRIGHT
  state = JoyDir(0)
  VWait
Wend
```

The above code waits until the user moves the Joystick in port 0 to right.

33.8 JoyHat

NAME

JoyHat – return state of joystick hat (V10.0)

SYNOPSIS

```
state = JoyHat(port[, idx])
```

FUNCTION

This function returns the state of the hat of the joystick at the port specified by `port`. Joystick hats are also known as point-of-view d-pads. The returned state will be -1 if the hat is in the center, otherwise a value between 0 and 27000 will be returned. The optional argument `idx` can be used to specify the index of the joystick hat to use in case there are multiple hats on a controller. Indices start at 0.

`port` can range from 0 to the number of joysticks currently plugged in minus 1. You can find out the number of joysticks currently available using the `CountJoysticks()` function. Please note that under AmigaOS, port 0 addresses the standard joystick port although this is port 1 on classic Amiga hardware. Hollywood switches these ports for cross-platform consistency where port 0 shall always refer to the default joystick.

INPUTS

`port` port number (usually 0 for the standard Joystick port)

`idx`	optional: joystick index to query (defaults to 0)

RESULTS

`state`	state of the joystick hat, ranging from -1 (center) to 27000

34 Layers library

34.1 Overview

Hollywood offers you a powerful yet easy to use layer system which should be able to realize everything you need for your application. Layers are children of a background picture. Every background picture has its own attached layers. Hollywood's layer system is not enabled at startup. You have to enable it manually by calling the `EnableLayers()` command. Alternatively, you can use the `Layers` tag in `@DISPLAY` or `CreateDisplay()`. Once layers are enabled, every object displayed on the screen will be on its own layer.

Note that layers are enabled/disabled on a per display basis. Thus, it is absolutely possible to mix layered and non-layered displays. For instance, if you have two displays, display 1 could use layers and display 2 could be non-layered. This is perfectly possible.

What you should try to avoid is disabling layers in a display in which they have been enabled before. This is possible to do but it should be avoided in any case because layered and non-layered modes are distinctly different.

Let's have a look at a brief example now:

```
EnableLayers()
DisplayBGPic(2)
DisplayBrush(1, #CENTER, #CENTER)
Plot(100, 100, #RED)
Print("Hello World!")
Box(50, 50, 100, 100, #BLUE)
```

The above code displays 4 different object types and attaches at the same time 4 layers to the background picture number 2 because layers were enabled. Every displayed object gets its own layer now, therefore we have the following layers now for background picture 2:

```
Layer id 1: Brush 1 at coordinates #CENTER : #CENTER
Layer id 2: A red pixel at 100 : 100
Layer id 3: Text "Hello World!"
Layer id 4: A blue box at 50 : 50 with dimensions 100 : 100
```

Now you can do everything you like with those layers, e.g. you can hide them, move them, swap foreground priorities or remove them. Hollywood offers many functions that can handle layers.

Please note that layer ids are dynamic. For example if the above code would now call the command

```
RemoveLayer(2)
```

then the layer ids would be changed. After this command returns we would have the following layers for background picture 2:

```
Layer id 1: Brush 1 at coordinates #CENTER : #CENTER
Layer id 2: Text "Hello World!"
Layer id 3: A blue box at 50 : 50 with dimensions 100 : 100
```

You see that the text "Hello World!" has now layer id 2 and the box is now at layer 3.

Starting with Hollywood 2.0, there is a new command available: `SetLayerName()`. You can use it to give your layers a unique name so you can simply address the layer through its

name a instead of its id. This is very useful if you have many layers and you do not want to remember their ids. All functions that work with layers accept a name string in addition to a numeric id now. Here is our example again:

```
Layer id 1: Brush 1 at coordinates #CENTER : #CENTER
Layer id 2: A red pixel at 100 : 100
Layer id 3: Text "Hello World!"
Layer id 4: A blue box at 50 : 50 with dimensions 100 : 100
```

Now we do the following:

```
SetLayerName(1, "brush: 1")
SetLayerName(2, "red pixel")
SetLayerName(3, "text: hello world")
SetLayerName(4, "blue box")
```

Now we could remove layer 2 by calling

```
RemoveLayer("red pixel")
```

We do not have to care about the fact that the layer ids have changed now because all layers have names and so we can easily address them.

Please keep in mind that layers are always background picture private. For example if you now call a

```
DisplayBGPic(3)
```

you will not have any layers you could access. If you call now

```
DisplayBGPic(2)
```

again, Hollywood will display your background picture 2 together with all layers attached to it. So you can safely switch between background pictures and you do not have to display all your data again. If you have layers enabled, Hollywood will display all layers attached to a background picture automatically with `DisplayBGPic()`.

To save memory it is advised however to call `FreeLayers()` when you do not need them any longer.

Please also make sure that you call `EnableLayers()` before displaying the objects you want to access as layers. For example, the following code will not work:

```
DisplayBrush(1, #CENTER, #CENTER)
EnableLayers()
Undo(#BRUSH, 1)
```

Every command that outputs graphics will check if layers are enabled and if they are, it will add a layer. Therefore the above example cannot work because layers are enabled after `DisplayBrush()` is called. So you have to use the following code:

```
EnableLayers()
DisplayBrush(1, #CENTER, #CENTER)
Undo(#BRUSH, 1)
```

This will work fine then.

If you plan to use layers in your whole application, it is recommended to call `EnableLayers()` right at the start of your code. This ensures that layers are always enabled.

Once a layer is on the display you can change its appearance very easily. Hollywood offers a wide range of layer manipulation function. The most powerful of them is `SetLayerStyle()` which can be used to change nearly all of the layer's attributes with just a single call. It can even change the attributes of multiple layers at once! Furthermore, you can rotate a layer using `RotateLayer()` and scale it using `ScaleLayer()`. It is also possible to show and hide layers with an effect from Hollywood's wide range of transition effects using the `ShowLayerFX()` and `HideLayerFX()` functions.

34.2 AddMove

NAME

AddMove – add object to move list (V1.5)

SYNOPSIS

```
[id] = AddMove(id, type, sourceid[, par1, par2, par3])
```

FUNCTION

This function adds an object to the move list specified by `id`. If the move list is not existing yet, it is created by this function. You can also pass `Nil` in `id` which will cause `AddMove()` to create a new move list in any case and return its id. Move lists are used for optimized drawing using DoMove(). The optional parameters `par1`, `par2` and `par3` specify different things depending on which object type you passed over.

The following types are currently supported by `AddMove()`:

`#BRUSH` Adds the brush with id `sourceid` to the move list; `par1` specifies x-position and `par2` the y-position for the brush; `par3` is not used

`#HIDEBRUSH`
Hides the brush with id specified by `sourceid`; optional parameters are not used

`#HIDELAYER`
Hides the layer specified by `sourceid`; optional parameters are not used

`#INSERTBRUSH`
Inserts the brush specified by `sourceid` into the layer position specified by `par3`; `par1` specifies the x-position for the brush and `par2` the y-position; See Section 34.19 [InsertLayer], page 663, for more information on inserting layers.

`#LAYER` Adds the layer specified by `sourceid` to the move list; `par1` specifies the x-position for the layer, `par2` the y-position; new in Hollywood 4.0: `par3` can be used to specify a visibility mode: 0 means `"show layer"`, 1 means `"hide layer"`, and 2 means `"keep current visibility setting"` (i.e. layer stays hidden if it is currently hidden); `par3` defaults to 0 which means always show the layer even if it is currently hidden

`#NEXTFRAME`
Displays a new frame of an anim layer; `par1` specifies the new x-position for the layer, `par2` the y-position; `par3` specifies the frame to be displayed; specify 0 to display the next frame, -1 to display the last frame of the animation (V2.0)

`#NEXTFRAME2`
: Same as `#NEXTFRAME` but takes a layer id as `sourceid`; this makes it possible to address anim layers directly; `par1` specifies the new x-position for the layer, `par2` the y-position; `par3` specifies the frame to be displayed; specify 0 to display the next frame, -1 to display the last frame of the animation (V2.5)

`#REMOVELAYER`
: Removes the layer specified by `sourceid` from the background picture's layer cache; optional parameters are not used

`#TEXTOBJECT`
: Adds the text object specified by `sourceid` to the move list; `par1` specifies x-position and `par2` the y-position; `par3` is not used

`#UNDO` Adds an `Undo()` operation to the move list; `sourceid` specifies the type of the object to be undone, `par1` specifies the identifier of the object to be undone, `par2` specifies the undo level; See Section 34.60 [Undo], page 708, for details.

After you have filled the move list with objects you can call `DoMove()` to draw the new display.

Please note: It is not possible to have multiple objects of the same type and id in your move list. For example, you cannot do the following:

```
DisplayBrush(1, #LEFT, #TOP)
DisplayBrush(1, #RIGHT, #BOTTOM)

/* This will not work */
AddMove(1, #BRUSH, 1, #CENTER, #CENTER)
AddMove(1, #BRUSH, 1, #LEFTOUT, #TOPOUT)
/* This will not work */

DoMove(1)
```

The above code will not work because you are using brush 1 two times in the same move list. Hollywood does not know which brush to use then which leads to unpredictable results.

See Section 34.7 [DoMove], page 654, for details.

INPUTS

`id` identifier of the move list or `Nil` to create a new move list

`type` type of the object to add (see list above)

`sourceid` depends on the specified type (see list above)

`par1` depends on the specified type (see list above)

`par2` depends on the specified type (see list above)

`par3` depends on the specified type (see list above)

RESULTS

`id` optional: identifier of the move list; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

See Section 34.7 [DoMove], page 654.

34.3 ClearMove

NAME

ClearMove – clear move list (V1.5)

SYNOPSIS

```
ClearMove(id)
```

FUNCTION

This function clears all objects which are in the move list specified by `id`. After you have called this command, your move list is empty again and can be filled with new objects.

See Section 34.7 [DoMove], page 654, for more information of Hollywood's move lists.

INPUTS

`id` identifier of the move list to clear

EXAMPLE

See Section 34.7 [DoMove], page 654.

34.4 CopyLayer

NAME

CopyLayer – clone a layer (V9.1)

SYNOPSIS

```
CopyLayer(id, pos[, t])
```

FUNCTION

This command clones the layer specified by `id` and inserts the copy into the layer position specified by `pos`. The special value 0 can be passed in `pos` to insert the cloned layer as the last one. `CopyLayer()` will clone all layer attributes except the layer's name because that must be unique. You can use the optional table argument `t` to specify a name for the cloned layer.

The optional table argument supports these tags:

`Name:` If you want to assign a name to the new layer, set this tag to a string containing the desired name. By default, the new layer won't be given a name.

`Hidden:` This tag can be set to `True` to automatically hide the new layer after creation. Defaults to `False`.

You need to enable layers before you can use this command. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier or name of the layer to clone

`pos` desired position for the new layer or 0 for last layer

`t` optional: table containing further parameters

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
Box(0, 0, 320, 480, #RED)
CopyLayer(1, 2, {Hidden = True})
SetLayerStyle(2, {X = 320, Color = #BLUE, Hidden = False})
```

The code above creates a filled red rectangle layer, clones it, changes the color of the cloned layer to blue and positions it next to the red layer.

34.5 CreateLayer

NAME

CreateLayer – create a new layer (V4.7)

SYNOPSIS

`CreateLayer(x, y, width, height[, table])`

FUNCTION

This command can be used to insert a new layer to the current BGPic. The layer will be of the dimensions specified in `width` / `height` and it will appear at the specified position. This function will create either a layer of type `#BRUSH` or of type `#ANIM`. If you want to create an anim layer, you will have to pass the desired number of frames in the `Frames` tag in the optional table argument.

The optional table argument recognizes the following tags:

`Frames` Specifies the number of frames for this layer. If this is set to 1, `CreateLayer()` will create a brush layer. Otherwise an anim layer containing the specified number of frames will be created. Defaults to 1 (which means that by default, `CreateLayer()` will create a brush layer).

`Color` Specifies the initial RGB color of the layer. This defaults to $000000 (i.e. black).

`Mask` Set this tag to `True` if `CreateLayer()` should attach a mask to the new layer. If this is `True`, `AlphaChannel` must be `False`. Defaults to `False`.

`AlphaChannel`
Set this tag to `True` if `CreateLayer()` should attach an alpha channel to the new layer. If this is set to `True`, `Mask` must be set to `False`. Defaults to `False`.

`Clear` This tag is only handled if either `AlphaChannel` or `Mask` was set to `True`. If that is the case, `Clear` specifies whether or not the mask or alpha channel should be cleared (i.e. fully transparent) or not (i.e. opaque). This defaults to `False` which means that by default, the new mask or alpha channel will be opaque.

Additionally, you can pass one or more of the standard tags in the optional table argument. Using these tags you can for instance control the insert position of the layer, assign a name to it, and modify the anchor point settings of this layer. See Section 27.17 [Standard draw tags], page 501, for details.

`CreateLayer()` is the preferred way of creating an empty layer that you later want to modify using the `SelectLayer()` command. Of course, you could also create an empty brush using `CreateBrush()` and then insert it as a layer using `DisplayBrush()` or `InsertLayer()` but this is not as effective as using the new `CreateLayer()` function because when you then call `SelectLayer()` on a layer that was created from an existing brush source, Hollywood first has to create a copy of the layer because `SelectLayer()` shall only modify the layer contents and not the contents of the brush that was used to create the layer. This is not very critical with brush layers, but imagine an anim layer with some dozens of frames! Using `SelectLayer()` on such an anim layer would be very expensive and would take quite some time. In these cases, `CreateLayer()` is really much more effective.

INPUTS

`x` desired x position for the new layer

`y` desired y position for the new layer

`width` desired layer width

`height` desired layer height

`table` optional: table configuring further options; can be one or more of the tags listed above or from the standard tags

EXAMPLE

```
CreateLayer(#CENTER, #CENTER, 100, 100, {Color = #RED})
SelectLayer(1)
Circle(0, 0, 50, #WHITE)
EndSelect
```

The code above creates a new 100x100 red layer and then draws a white circle onto it.

34.6 DisableLayers

NAME

DisableLayers – disable layers for current display (V1.5)

SYNOPSIS

```
DisableLayers()
```

FUNCTION

This function disables layers in the currently selected display.

Please note that this function does not free any layers that are attached to a background picture. They will be kept until you free them or until Hollywood is closed. So you can also disable layers temporarily and enable them again later and you will not lose any layers.

Please note though that it is generally not advised to switch between layered and non-layered modes because both modes are not really compatible with each other. Thus, the best idea is certainly to define whether or not a display should use layers at display creation time (e.g. when calling `@DISPLAY` or `CreateDisplay()`) and then stick to this decision. Mixing layered and non-layered sections in the same display is really only recommended if you know exactly what you are doing.

See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

none

34.7 DoMove

NAME

DoMove – draw a move list (V1.5)

SYNOPSIS

```
DoMove(id)
```

FUNCTION

This function draws all objects which have been added to the move list specified by `id` (using `AddMove()`) at once. This is very useful if you want to display animated graphics with different objects. If you painted every object with `DisplayBrush()` your display would certainly flicker a lot. This can be prevented by updating the display with one drawing operation. `DoMove()` lets you realize that: You add all objects that shall be drawn to a move list (using `AddMove()`) and then you call `DoMove()` which will draw the whole move list using just a single draw operation.

Implementation of `DoMove()` is that it scans the move list you specify and looks what objects shall be drawn. For every object that is in the list Hollywood will check if the object is already on the screen. If it is, Hollywood will move the object to the new position. If it is not on the screen, it will be drawn on the screen. Therefore if all objects that shall be drawn are already on the screen and shall just be moved with `DoMove()`, all layer positions will be kept. If there are objects in the move list that are not currently on the screen, they will be drawn and will get the top most layer position assigned.

After `DoMove()` is finished, you should call `ClearMove()`. This will clear the move list you specify and you can use it again with new object positions.

This function requires enabled layers.

INPUTS

`id` identifier of the move list to draw

EXAMPLE

```
EnableLayers()
```

```
For x = 0 To 400
AddMove(1, #BRUSH, 1, x, 0)
AddMove(1, #BRUSH, 2, x, 100)
AddMove(1, #BRUSH, 3, x, 200)
AddMove(1, #BRUSH, 4, x, 300)
DoMove(1)
ClearMove(1)
Next
```

The code above scrolls brushes 1 to 4 from 0 to 400. You will see no flickering because we use the move list technique.

34.8 DumpLayers

NAME

DumpLayers – print internal information about layers (V2.0)

SYNOPSIS

`DumpLayers([what])`

FUNCTION

This function prints internal information about the layers in the current BGPic to the debug device. This is mostly useful for debugging purposes. The information that is printed by `DumpLayers()` includes position and size information, the layer's visibility flag as well as the internal storage size of a basic Hollywood layer.

The `what` argument can be used to control the information that should be printed. Internally, Hollywood layers can have up to three different representations. A normal representation without any transformations, a transformed representation and a layerscale-transformed representation. The transformed representation is created by functions such as `RotateLayer()` and `ScaleLayer()` whereas the layerscale-transformed manifestation of a layer represents either the normal or the transformed layer with additional transformations added by the layer scaling engine.

The following values are currently accepted by the `what` argument:

`0:` Print information about the normal representation of the layer. This is the default.

`1:` Print information about the transformed representation of the layer.

`2:` Print information about the layerscale-transformed representation of the layer.

Note that even if no layerscale-transformation is currently active, level 2 always represents the physical appearance of a layer. So if you need to know details about the physical appearance of a layer, always pass 2 in the `what` parameter, even if no transformation are currently active.

Also note that the position and size information printed by this function is separated into the real physical position and size and the position and size as maintained by the script. The real physical position and size information is printed in brackets.

All this information, however, is probably of not much use for normal programmers. This function is mainly here for debugging purposes. If you need to query layer attributes for your script, use the `GetLayerStyle()` or the `GetAttribute()` function using the `#LAYER` source type. See Section 34.15 [GetLayerStyle], page 659, for details. See Section 43.4 [GetAttribute], page 858, for details.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`what` flag to control which information should be printed (see above); defaults to 0

34.9 EnableLayers

NAME

EnableLayers – enable layers for current display (V1.5)

SYNOPSIS

```
EnableLayers()
```

FUNCTION

This function enables layers in the currently selected display. In order to use any of the layer functions in this display, you need to call this function first. Alternatively, you could enable layers already at display creation time by using the `Layers` tag in either the `@DISPLAY` or `CreateDisplay()` commands.

Please note also that it is generally not advised to switch between layered and non-layered modes because both modes are not really compatible with each other. Thus, the best idea is certainly to define whether or not a display should use layers at display creation time (e.g. when calling `@DISPLAY` or `CreateDisplay()`) and then stick to this decision. Mixing layered and non-layered sections in the same display is really only recommended if you know exactly what you are doing.

See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

none

34.10 FreeLayers

NAME

FreeLayers – free background picture's layers (V1.5)

SYNOPSIS

```
FreeLayers([keep])
```

FUNCTION

This function frees all layers associated with the current background picture. You should call this command when you are finished with layers on a background picture because it releases quite some memory.

By default, `FreeLayers()` will free all layers but it will also draw them into the background picture. This means that there will be no visible change after you have called `FreeLayers()`. The layers will be gone but it will look as if they were still there because they will be drawn into the background picture. If you don't want that, set the `keep` parameter to `False`. In that case, the layers will be removed and they won't be drawn into the background. This is the same as calling `RemoveLayers()`.

Please note: Layers will not be freed when you display a new background picture. You have to free them on your own.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`keep` optional: whether to draw the layers into the background picture or not (defaults to `True`) (V4.0)

34.11 GetLayerAtPos

NAME

GetLayerAtPos – return topmost layer at specified position (V4.7)

SYNOPSIS

```
id, name$ = GetLayerAtPos(x, y)
```

FUNCTION

This function returns the topmost layer at the specified position. This is useful when creating some interactive user interface where layers can be moved with the mouse or hovering over a layer changes the style of that very layer. The position passed to this function is relative to the top-left corner of the display, i.e. a position of (0,0) means the top-left corner.

`GetLayerAtPos()` returns the identifier of the topmost layer at the specified position as well as the name of that layer. If the layer does not have a name, an empty string is returned as the second return value. If there is no layer at the specified position at all, 0 is returned as the identifier and an empty string as the name.

INPUTS

`x` x position to query

`y` y position to query

RESULTS

`id` identifier of the topmost layer at this position or 0 if there is no layer at this position

`name$` name of the topmost layer or empty string (`""`) if the layer does not have a name or no layer was found

34.12 GetLayerGroupMembers

NAME

GetLayerGroupMembers – return all members of a layer group (V10.0)

SYNOPSIS

```
t = GetLayerGroupMembers(group$)
```

FUNCTION

This function finds all members of the layer group specified by `group$` and returns them in a table. The order in which the group members are returned in the table is arbitrary.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`group$`	name of layer group to use

RESULTS

`t`	table containing all group members

EXAMPLE

```
t = GetLayerGroupMembers("mygroup")
For Local k = 0 To ListItems(t) - 1 Do DebugPrint(t[k])
```

The code above gets all members of the layer group "mygroup" and prints them.

34.13 GetLayerGroups

NAME

GetLayerGroups – return layer groups in current BGPic (V10.0)

SYNOPSIS

```
t = GetLayerGroups()
```

FUNCTION

This function collects all layer groups in the current BGPic and returns them in a table. The order in which the groups are returned in the table is arbitrary.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

none

RESULTS

`t`	table containing all layer groups

EXAMPLE

```
t = GetLayerGroups()
For Local k = 0 To ListItems(t) - 1 Do DebugPrint(t[k])
```

The code above gets all layer groups and prints them.

34.14 GetLayerPen

NAME

GetLayerPen – get pen color from layer's palette (V9.0)

SYNOPSIS

```
color = GetLayerPen(id, pen)
```

FUNCTION

This function gets the color of the pen specified by `pen` from the palette of the layer specified by `id`. The color will be returned as an RGB color.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of layer to use
`pen`	pen you want to get (starting from 0)

RESULTS

`color`	color of the pen, specified as an RGB color

EXAMPLE

```
color = GetLayerPen(1, 0)
```

The code gets the color of the first pen of layer 1.

34.15 GetLayerStyle

NAME

GetLayerStyle – get the style of a layer (V4.0)

SYNOPSIS

```
t = GetLayerStyle(id)
```

FUNCTION

This function returns all style attributes of the specified layer. The different attributes are returned in a table which you can then examine. The contents of the style table returned by this function depend on the type of the layer that you specified. For a complete overview of all style elements that will be returned by this function, please have a look at the documentation of the `SetLayerStyle()` command which contains a list of the layer style elements and to which layer types they apply. See Section 34.48 [SetLayerStyle], page 689, for details.

Please note that this command always queries all attributes so it can sometimes get quite slow. If you need only some basic information about a layer, it could be faster to use `GetAttribute()` instead.

INPUTS

`id`	identifier of the layer to examine

RESULTS

`t`	a table containing all style attributes for this layer

EXAMPLE

```
t = GetLayerStyle(1)
Print("This layer is at position", t.x, ":", t.y, "!")
```

The code above queries the style of layer 1 and displays its position then.

34.16 GroupLayer

NAME

GroupLayer – add layer(s) to group (V10.0)

SYNOPSIS

```
GroupLayer(group$, layer1[, layer2, ...])
```

FUNCTION

This function can be used to add one or more layers to the layer group specified by `group$`. If the layer group specified by `group$` doesn't exist yet, it will be automatically created by `GroupLayer()`. Layer groups are simply referenced by a name string that can contain any characters as long as the group's name isn't already used by a layer. The layer(s) that should be added to the group must be specified by their id in the parameters after `group$`. You can pass an unlimited number of layers to this function.

Once you have finished grouping your layers, you can then pass the group's name to most functions that deal with layers, e.g. you could show a group of layers by simply passing the name of your layer group to `ShowLayer()`. You could also move all layers of a layer group at once by passing the layer group to `MoveLayer()` etc.

Note that when passing groups instead of single layers to functions of the layer library, those functions won't treat the layer group as an own entity but will simply apply the respective operation on all layers that are part of the group. For example, if you call `MoveLayer()` on a layer group and pass 100:100 as the new position, Hollywood won't move the group as a whole to position 100:100 but all group members individually will be moved to 100:100 so that after the call all layers that are part of the group will appear at 100:100, i.e. they all will be at the same position which might not be what you expected. If you want to move layers that are part of a group and preserve their individual position within the group, you need to call `TranslateLayer()` instead because that allows moving layers relative to their current position. See Section 34.59 [TranslateLayer], page 708, for details.

Layers can also be added to a group right when they are created by passing the group's name in the `Group` tag of the standard drawing tags accepted by all Hollywood functions that add a layer. See Section 27.17 [Standard drawing tags], page 501, for details.

To remove a layer from a group, use the `UngroupLayer()` function. See Section 34.62 [UngroupLayer], page 711, for details. As soon as a group doesn't have any more layers attached, it will be automatically deleted.

Another way of grouping layers is to merge them. In comparison to grouping layers, merging layers means to turn them into a single layer. One advantage of merged layers is that they are treated as a whole, for example when showing or hiding them using transition effects. Grouped layers, on the other hand, will show transition effects individually for each group member. See Section 34.24 [MergeLayers], page 666, for details.

You need to enable layers before you can use `GroupLayer()`. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`group$`	name of the group to add the layer(s) to
`layer1`	first layer to add to the group
`...`	further layers to add to the group

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
Box(0, 0, 100, 100, #RED, {Hidden = True})
Box(100, 0, 100, 100, #GREEN, {Hidden = True})
Box(200, 0, 100, 100, #BLUE, {Hidden = True})
GroupLayer("mygroup", 1, 2, 3)
TranslateLayer("mygroup", 170, 190)
ShowLayerFX("mygroup", #SCROLLSOUTH)
```

The code above creates three hidden 100x100 rectangles, groups them and then moves the group to the center of the 640x480 display and scrolls them in from the south.

34.17 HideLayer

NAME

HideLayer – hide a layer (V1.5)

SYNOPSIS

```
HideLayer(id)
```

FUNCTION

This function hides the layer or layer group specified by `id`. The layer will not be removed. It will just be hidden. You can make it visible again, by calling `ShowLayer()`. If you want to remove it completely, use `RemoveLayer()` or `Undo()`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of the layer or layer group to hide

EXAMPLE

```
EnableLayers()
NPrint("Hello World!")
WaitLeftMouse
HideLayer(1)
WaitLeftMouse
ShowLayer(1)
```

The code above prints "Hello World!" to the display, then hides this text and displays it again.

34.18 HideLayerFX

NAME

HideLayerFX – hide a layer with transition effects (V1.9)

SYNOPSIS

```
[handle] = HideLayerFX(id[, table])
```

FUNCTION

This function is an extended version of the `HideLayer()` command. It hides the layer or layer group specified by `id` and uses one of the many Hollywood transition effects. You can also specify the speed for the transition and an optional argument.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of all supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `HideLayerFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`NoBorderFade`
If the layer to be hidden has a border, do not gradually fade out the border but remove it in one go at the end of the transition effect. (V5.0)

`BorderFX:`
If the layer to be hidden has a border, Hollywood will only apply the transition effect to the border if the layer is a transparent layer with text or pixel graphics. For non-transparent and vector graphics layers a generic fade effect will be used instead because otherwise there would be visual glitches between the penultimate and final effect frame because of differences in the border algorithms. If you don't care about this glitch and want to force Hollywood to always apply the transition effect to the border, set this tag to `True`. To force Hollywood to always use the generic fade mode, set this tag to `False`. (V9.0)

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of the layer or layer group to hide

`table` optional: table configuring the transition

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
HideLayerFX(5, {Type = #CROSSFADE}) ; new syntax

OR

HideLayerFX(5, #CROSSFADE)          ; old syntax
```

The above code hides layer 5 with a nice crossfade transition.

34.19 InsertLayer

NAME

InsertLayer – insert a new layer (V1.5)

SYNOPSIS

`InsertLayer(pos, type, id, x, y[, hidden])`

FUNCTION

This function inserts a new layer of the object type specified by `type` and the object id specified by `id` into layer position `pos`. All the following layers will be moved downwards and therefore they will get a new id. The new layer will also be displayed at the position specified by `x` and `y`. If you specify 0 as `pos`, the layer will be inserted as the last layer.

The following object types are currently supported:

`#BRUSH` Inserts the brush specified by `id` at x, y

`#TEXTOBJECT`
Inserts the text object specified by `id` at x, y

`#ANIM` Inserts the anim specified by `id` at x, y (V2.0)

`#VIDEO` Inserts the video specified by `id` at x, y (V6.0)

Starting with Hollywood 1.9 you can specify the optional argument `hidden`, which will insert a hidden layer which you can bring to front using `ShowLayer()` or `ShowLayerFX()`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`pos` desired position for the layer or 0 for last layer

`type` type of the object to be inserted (see list above)

`id` identifier of the object to be inserted

`x`	x-position for the new layer
`y`	y-position for the new layer
`hidden`	optional: `True` if the layer shall be hidden (defaults to `False`) (V1.9)

EXAMPLE

```
EnableLayers()
SetFillStyle(#FILLCOLOR)
Box(0, 0, 100, 100, #RED)
Circle(#CENTER, #CENTER, 50, #BLUE)
TextOut(#RIGHT, #BOTTOM, "Hello World")
InsertLayer(1, #BRUSH, 1, #CENTER, #CENTER)
```

The code above inserts brush 1 as the first layer. This means that all the other layers will be re-positioned. The red rectangle will get layer position 2 now (was layer 1), the blue circle will be layer 3 (was layer 2) and the "Hello World" text will be layer 4 (was layer 3).

34.20 LayerExists

NAME

LayerExists – check if specified layer exists (V4.6)

SYNOPSIS

```
ret = LayerExists(layer$)
```

FUNCTION

This command simply checks whether or not the specified layer exists. Obviously, you must pass a layer name here, not a layer id as layer ids are per se existent.

INPUTS

`layer$`	layer name to check

RESULTS

`ret`	`True` if the layer exists, `False` otherwise

34.21 LayerGroupExists

NAME

LayerGroupExists – check if group exists (V10.0)

SYNOPSIS

```
ok = LayerGroupExists(group$)
```

FUNCTION

This function checks if the layer group specified by `group$` exists. If it does, `True` is returned, `False` otherwise.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`group$` name of the layer group

RESULTS

`ok` `True` if the group exists, `False` if it doesn't

34.22 LayerToBack

NAME

LayerToBack – move layer to backmost z-position (V5.0)

SYNOPSIS

```
LayerToBack(layer[, swap])
```

FUNCTION

This command moves the specified layer all the way to the background. `LayerToBack()` is a convenience function. The same could be achieved by using `SwapLayers()` or `SetLayerZPos()`.

If the optional argument `swap` is set to `False`, the layer is brought to the back by simply moving it to the back. This is different from the default behaviour which simply swaps the positions of the back layer and the layer specified by `layer`. If `swap` is set to `False`, `layer` can also be the name of a layer group.

INPUTS

`layer` layer to move to the background

`swap` set this to `False` if the layers shouldn't swap positions but the specified layer should just be moved to the back (defaults to `True`) (V7.1)

34.23 LayerToFront

NAME

LayerToFront – move layer to frontmost z-position (V5.0)

SYNOPSIS

```
LayerToFront(layer[, swap])
```

FUNCTION

This command moves the specified layer all the way to the front. `LayerToFront()` is a convenience function. The same could be achieved by using `SwapLayers()` or `SetLayerZPos()`.

If the optional argument `swap` is set to `False`, the layer is brought to the front by simply moving it to the front. This is different from the default behaviour which simply swaps the positions of the front layer and the layer specified by `layer`. If `swap` is set to `False`, `layer` can also be the name of a layer group.

INPUTS

`layer` layer to move to the front

`swap` set this to `False` if the layers shouldn't swap positions but the specified layer should just be moved to the front (defaults to `True`) (V7.1)

34.24 MergeLayers

NAME

MergeLayers – merge layers into new layer (V10.0)

SYNOPSIS

```
MergeLayers(layer1[, layer2, ..., t])
```

FUNCTION

This function merges the layers specified by `layer1`, `layer2`, etc. into a new layer while preserving the source layers. By default, `MergeLayers()` will automatically hide the source layers, but this behaviour can be changed by setting the `AutoHide` tag to `False` in the optional table argument. Instead of single layers, you can also pass layer groups that should be merged to this function.

The new layer that is created by `MergeLayers()` will use the special `#MERGED` layer type. Layers of type `#MERGED` can't be transformed (except by the layer scaling engine) and they'll typically contain all settings of their child layers, e.g. shadow, border, filters, transparency settings etc. rendered into the layer, although this can be changed using certain tags in the optional table argument. In comparison to layer groups created using `GroupLayer()`, one advantage of merged layers is that when showing them using transition effects they will be treated as a whole whereas showing layers that are part of a group created by `GroupLayer()` using transition effects would apply the transitions to each group member individually which might not always look as good as when the transitions are applied to the layers as a whole. This limitation of `GroupLayer()` can thus be overcome by using `MergeLayers()`.

The optional table argument `t` can contain the following tags:

`AutoHide:`
: Specifies whether or not the source layers should be automatically hidden by `MergeLayers()`. By default, `MergeLayers()` will automatically hide the layers that are merged into a new one. If you don't want that, set this tag to `False`. Defaults to `True`.

`MergeShadow:`
: Specifies whether or not any potential layer shadow should be merged into the new layer as well. This defaults to `True`. If you set this to `False`, no shadow effect from any of the source layers will be merged into the new layer so the new layer will appear without any shadow. Of course, it's possible to add a shadow to the new layer using `SetLayerShadow()` or the standard drawing tags.

`MergeBorder:`
: Specifies whether or not any potential layer border effect should be merged into the new layer as well. This defaults to `True`. If you set this to `False`, no border effect from any of the source layers will be merged into the new

layer so the new layer will appear without any border effect. Of course, it's possible to add a border effect to the new layer using `SetLayerBorder()` or the standard drawing tags.

`MergeFilter:`
Specifies whether or not any potential layer filter should be merged into the new layer as well. This defaults to `True`. If you set this to `False`, no filter from any of the source layers will be merged into the new layer so the new layer will appear without any filters. Of course, it's possible to add filters to the new layer using `SetLayerFilter()` or the standard drawing tags.

`MergeTransparency:`
Specifies whether or not any potential layer transparency should be merged into the new layer as well. This defaults to `True`. If you set this to `False`, no transparency from any of the source layers will be merged into the new layer so the new layer will appear without any transparency setting. Of course, it's possible to set the transparency of the new layer using `SetLayerTransparency()` or the standard drawing tags.

`MergeFX:` Specifies whether or not any potential layer transition effect should be merged into the new layer as well. This defaults to `True`. If you set this to `False`, no transition effect from any of the source layers will be merged into the new layer so the new layer will appear without any transition effects.

Furthermore, the optional table argument also supports Hollywood's standard drawing tags. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

Note that merged layers aren't updated automatically when their source layers change their graphics. You need to use the `RefreshLayer()` function to force an update of a merged layer. See Section 34.30 [RefreshLayer], page 671, for details. Also note that only visible layers will be merged. Hidden layers will be ignored by `MergeLayers()`.

You need to enable layers before you can use GroupLayer(). See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`layer1`	first layer or layer group to merge
`...`	further layers or layer groups to merge
`t`	optional: table containing further options (see above)

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
SelectBGPic(1)
Box(0, 0, 100, 100, #RED)
Box(100, 0, 100, 100, #GREEN)
Box(200, 0, 100, 100, #BLUE)
MergeLayers(1, 2, 3, {Name = "newlayer", Hidden = True})
MoveLayer("newlayer", #CENTER, #CENTER)
EndSelect
```

```
ShowLayerFX("newlayer", #ZOOMCENTER)
```

The code above creates three hidden 100x100 rectangles, merges them to a new layer, moves this new layer to the center and then shows the merged layer with a transition effect. Note that we use `SelectBGPic()` to make sure nothing is drawn before our call to `ShowLayerFX()`.

34.25 ModifyLayerFrames

NAME

ModifyLayerFrames – change number of anim layer frames (V4.7)

SYNOPSIS

```
ModifyLayerFrames(id, frames[, pos])
```

FUNCTION

This function can be used to extend or shrink the frames of an anim layer. If you specify a positive value in `frames`, then the anim layer is extended by this number of frames. If you specify a negative value, the number of frames specified are removed from the anim layer.

The optional argument `pos` can be used to specify where the new frames shall be inserted or from where the frames shall be removed, respectively. If you do not specify the optional argument or set it to 0, frames are added at the end of the anim layer or removed from the end of the anim layer, respectively.

This command works only with anim layers that have their frames buffered entirely in memory. You cannot use it for anim layers that load their frames dynamically from disk.

INPUTS

`id`	identifier of the anim layer to modify
`frames`	number of frames to insert (if value is positive) or number of frames to remove (if value is negative)
`pos`	optional: where to insert or remove frames (defaults to 0 which means insert at/remove from the end)

EXAMPLE

```
ModifyLayerFrames(1, -5, 1)
```

The code above removes the first five frames from anim layer number 1.

34.26 MoveLayer

NAME

MoveLayer – move layer to a new position (V1.9)

SYNOPSIS

```
MoveLayer(id, xa, ya, xb, yb[, table])
MoveLayer(id, x, y) (V9.1)
```

FUNCTION

This function can be used to either scroll the layer specified by `id` to a new position or simply move it to a new position without scrolling.

If you pass the `xa`, `ya`, `xb` and `yb` arguments, `MoveLayer()` will scroll the layer specified by `id` softly from the position specified by `xa` and `ya` to the position specified by `xb` and `yb`. Further configuration options are possible using the optional argument `table`. You can specify the move speed, special effect, and whether or not the move shall be asynchronous. See Section 21.46 [MoveBrush], page 287, for more information on the optional table argument.

If you just pass the `x` and `y` arguments, `MoveLayer()` will simply move the layer to the position specified by `x` and `y`. In that case, `id` can also be the name of a layer group.

For all coordinates you can specify the special constant `#USELAYERPOSITION`. Hollywood will use the current position of the layer then.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	id or name of the layer to use
`xa`	source x position
`ya`	source y position
`xb`	destination x position
`yb`	destination y position
`table`	optional: further configuration for this move

EXAMPLE

```
MoveLayer(5, #LEFTOUT, #CENTER, #RIGHTOUT, #CENTER)
```

Scrolls layer 5 from the outside left to the outside right of the display and centers it vertically.

```
MoveLayer(4, #USELAYERPOSITION, #USELAYERPOSITION, #LEFTOUT, #CENTER)
```

Scrolls layer 4 from its current position out of the screen.

```
MoveLayer(5, #CENTER, #CENTER)
```

Moves layer 5 to the screen center.

34.27 NextFrame

NAME

NextFrame – display a new frame of an anim layer (V2.0)

SYNOPSIS

```
NextFrame(id[, x, y, frame])
```

FUNCTION

This function displays a new frame of an anim layer. If you omit the optional `frame` argument or set it to 0, `NextFrame()` will show the next frame of the anim layer. If you pass -1 in the `frame` argument, the last frame will be displayed. The `x` and `y` arguments can be used to move the layer to a new position while changing the frame. If you do not need them, pass `#USELAYERPOSITION` which will keep the layer where it is.

Starting with Hollywood 9.0, this function can also be used with text layers that are in list mode to show the next list items. See Section 54.39 [TextOut], page 1149, for details.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier or name of the anim layer
`x`	optional: new x-position for the layer (defaults to `#USELAYERPOSITION`)
`y`	optional: new y-position for the layer (defaults to `#USELAYERPOSITON`)
`frame`	optional: which frame to show (defaults to 0 which means that the next frame shall be shown)

EXAMPLE

```
EnableLayers
InsertLayer(1, #ANIM, 1, 0, #CENTER)
For k = 0 To 400 Step 3
   NextFrame(1, k, #USELAYERPOSITION)
   Wait(5)
Next
```

Plays the anim number 1 while moving it from x-position 0 to 400.

34.28 PauseLayer

NAME

PauseLayer – pause a playing video layer (V6.0)

SYNOPSIS

`PauseLayer(id)`

FUNCTION

This function pauses the video layer specified by `id`. This video layer must be playing when you call this command. You can resume playback later by using the `ResumeLayer()` command.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier or name of the video layer to pause

34.29 PlayLayer

NAME

PlayLayer – play a currently stopped video layer (V6.0)

SYNOPSIS

```
PlayLayer(id)
```

FUNCTION

This function starts playback of the video layer specified by `id`. You can stop playback by calling `StopLayer()`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier or name of the video layer to be played

34.30 RefreshLayer

NAME

RefreshLayer – refresh a layer (V10.0)

SYNOPSIS

```
RefreshLayer(id)
```

FUNCTION

This function refreshes the layer or layer group specified by `id`. This is normally not needed because Hollywood will refresh layers automatically whenever it is needed. There is one exception, though: Due to performance reasons, merged layers created using `MergeLayers()` won't be refreshed automatically when the graphics of one of their source layers change. Thus, you must manually tell merged layers to refresh by calling `RefreshLayer()` on them whenever you want them to refresh their graphics and that's the reason why this function exists.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of the layer or layer group to be refreshed

34.31 RemoveLayer

NAME

RemoveLayer – remove a layer (V1.5)

SYNOPSIS

```
RemoveLayer(id)
```

FUNCTION

This function removes the layer or layer group specified by `id`. This is basically the same as the `Undo()` command with the exception that this function accepts layer ids directly. With `Undo()` you would have to specify a type, an id and maybe also an undo-level, now you can just specify the layer id which should be much more convenient.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of layer or layer group to be removed

34.32 RemoveLayerFX

NAME

RemoveLayerFX – remove a layer with transition effects (V3.0)

SYNOPSIS

```
[handle] = RemoveLayerFX(id[, table])
```

FUNCTION

This function is an extended version of the `RemoveLayer()` command. It removes the layer or layer group specified by `id` and uses one of the many Hollywood transition effects. You can also specify the speed for the transition and an optional argument.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `RemoveLayerFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`NoBorderFade`
If the layer to be removed has a border, do not gradually fade out the border but remove it in one go at the end of the transition effect. (V5.0)

`BorderFX:`

If the layer to be removed has a border, Hollywood will only apply the transition effect to the border if the layer is a transparent layer with text or pixel graphics. For non-transparent and vector graphics layers a generic fade effect will be used instead because otherwise there would be visual glitches between the penultimate and final effect frame because of differences in the border algorithms. If you don't care about this glitch and want to force Hollywood to always apply the transition effect to the border, set this tag to `True`. To force Hollywood to always use the generic fade mode, set this tag to `False`. (V9.0)

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of the layer or layer group to remove

`table` optional: table configuring the transition

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
RemoveLayerFX(5, #CROSSFADE)              ; old syntax

OR

RemoveLayerFX(5, {Type = #CROSSFADE})     ; new syntax
```

The above code removes layer 5 with a nice crossfade transition.

34.33 RemoveLayers

NAME

RemoveLayers – remove all layers (V8.0)

SYNOPSIS

`RemoveLayers()`

FUNCTION

This function removes all layers in the current background picture. When this function returns, they will no longer be visible. If you want to remove all layers but still keep their graphics on screen, use `FreeLayers()` instead.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

none

34.34 RenderLayer

NAME
RenderLayer – render layer to brush layer (V10.0)

SYNOPSIS
`RenderLayer(id)`

FUNCTION
This function converts the layer specified by `id` to a brush layer. This usually means sacrificing quality because brushes are rasterized and thus cannot be scaled or transformed without losses in quality which is why this function is probably of not much use.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of the layer to be rendered

34.35 ResumeLayer

NAME
ResumeLayer – resume a paused video layer (V6.0)

SYNOPSIS
`ResumeLayer(id)`

FUNCTION
This function resumes the playback of the paused video layer specified by `id`. You can pause the playback of a video layer using the `PauseLayer()` command.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier or name of the video layer to be resumed

34.36 RotateLayer

NAME
RotateLayer – rotate a layer (V4.0)

SYNOPSIS
`RotateLayer(id, angle[, smooth])`

FUNCTION
This function rotates the layer or layer group specified by `id` by the specified angle (in degrees). A positive angle rotates anti-clockwise, a negative angle rotates clockwise. Additionally, you can choose to have the rotated graphics interpolated by passing `True` in the `smooth` argument. The graphics will then be rotated using anti-alias.

If the specified layer is a vector layer (e.g. circle, polygon, TrueType text or a rectangle) Hollywood will rotate the layer without any loss in quality because vector graphics can be freely transformed. Thus, the `smooth` argument does not have any function if the specified layer is a vector layer. If the layer uses raster graphics, however, normal raster-based rotation will be used.

In contrast to rotating brushes using `RotateBrush()` layers always keep their original data so there will not be any loss in quality if you rotate a layer forth by some degrees and then back by the same degrees. This is perfectly possible and does not generate any quality losses with `RotateLayer()`.

INPUTS

`id`	layer or layer group to rotate
`angle`	rotation angle in degrees
`smooth`	optional: whether or not anti-aliased rotation shall be used (only applicable if the layer is not a vector layer)

34.37 ScaleLayer

NAME

ScaleLayer – scale a layer (V4.0)

SYNOPSIS

```
ScaleLayer(id, width, height[, smooth])
```

FUNCTION

This command scales the layer or layer group specified by `id` to the specified width and height. Optionally, you can choose to have the scaled graphics interpolated by passing `True` in the `smooth` argument. The graphics will then be scaled using anti-alias.

If the specified layer is a vector layer (e.g. circle, polygon, TrueType text or a rectangle), Hollywood will scale the layer without any loss in quality because vector graphics can be freely transformed. Thus, the `smooth` argument does not have any function if the specified layer is a vector layer. If the layer uses raster graphics, however, normal raster-based rotation will be used.

In contrast to scaling brushes using `ScaleBrush()` layers always keep their original data so there will not be any loss in quality if you scale a layer to (20,15) and then back to (640,480). This is perfectly possible.

The `width` and `height` arguments can also be a string containing a percent specification, e.g. `"50%"`.

If you prefer to work with relative scaling factors instead of absolute pixel values, then you should use the `ScaleX` and `ScaleY` tags of the `SetLayerStyle()` function instead.

INPUTS

`id`	identifier of the layer to scale
`width`	desired new width for the layer
`height`	desired new height for the layer

`smooth` optional: whether or not anti-aliased scaling shall be used (only applicable if the layer is not a vector layer)

EXAMPLE

```
ScaleLayer(1,640,480)
```

Scales layer 1 to a resolution of 640x480.

34.38 SeekLayer

NAME

SeekLayer – seek to a certain position in a video layer (V6.0)

SYNOPSIS

```
SeekLayer(id, pos)
```

FUNCTION

You can use this function to seek to the specified position in the video layer specified by `id`. The video layer does not have to be playing. If the video layer is playing and you call `SeekLayer()`, it will immediately skip to the specified position. The position is specified in milliseconds. Thus, if you want to skip to the position 3:24, you would have to pass the value 204000 because 3 * 60 * 1000 + 24 * 1000 = 204000.

Please note that video seeking is a complex operation. There are video formats which do not have any position lookup tables so that Hollywood first has to approximate the seeking position and then do some fine- tuning and keyframe seeking so that the final position can always be a bit off from the position you specified in `SeekLayer()`. It can also happen that Hollywood will not seek directly to a keyframe so there might be artefacts from previous frames left on the screen.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier or name of the video layer to seek

`pos` new position for the video (in milliseconds)

34.39 SelectLayer

NAME

SelectLayer – select layer as output device (V4.7)

SYNOPSIS

```
SelectLayer(id, [, mode, frame, combomode])
```

FUNCTION

This function selects the specified layer as the current output device. This means that all graphics data that is rendered by Hollywood will be written to this layer. When `EndSelect()` is called, the layer will be refreshed automatically to reflect the changes you made to it. You have to specify a layer identifier in the first argument. If that layer

is an anim layer, you will also have to specify the frame you would like to select in the third argument.

The optional `mode` argument defaults to `#SELMODE_NORMAL` which means that only the color channels of the layer will be altered when you draw to it. The transparency channel of the layer (can be either a mask or an alpha channel) will never be altered. You can change this behaviour by using `#SELMODE_COMBO` in the optional `mode` argument. If you use this mode, every Hollywood graphics command that is called after `SelectLayer()` will draw into the color and transparency channel of the layer. If the layer does not have a transparency channel, `#SELMODE_COMBO` behaves the same as `#SELMODE_NORMAL`.

Starting with Hollywood 5.0 you can use the optional `combomode` argument to specify how `#SELMODE_COMBO` should behave. If `combomode` is set to 0, the color and transparency information of all pixels in the source image are copied to the destination image in any case - even if the pixels are invisible. This is the default behaviour. If `combomode` is set to 1, only the visible pixels are copied to the destination image. This means that if the alpha value of a pixel in the source image is 0, i.e. invisible, it will not be copied to the destination image. Hollywood 6.0 introduces the new combomode 2. If you pass 2 in `combomode`, Hollywood will blend color channels and alpha channel of the source image into the destination image's color and alpha channels. When you draw the destination image later, it will look as if the two images had been drawn on top of each other consecutively. Please note that the `combomode` argument is only supported together with `#SELMODE_COMBO`. It doesn't have any effect when used with the other modes. Please note that the `combomode` argument is only supported together with `#SELMODE_COMBO`. It doesn't have any effect when used with the other modes.

An alternative way to draw into the transparency channels of a layer is to do this separately using `SelectMask()` or `SelectAlphaChannel()`. These two commands, however, will write data to the transparency channel only. They will not touch the color channel. So if you want both channels, color and transparency, to be affected, you need to use `SelectLayer()` with `mode` set to `#SELMODE_COMBO`.

When you are finished with rendering to your layer and want to use your display as output device again, just call `EndSelect()`. If your layer is visible, Hollywood will refresh it automatically now to reflect the changes you made to it. It is important to take into account that your changes won't be visible before you call `EndSelect()`.

Note that you must not call any commands which modify your layer while it is selected as the output device. For example, you must not call `SetLayerStyle()` or `RemoveLayer()` while it is the output device.

Only commands that output graphics directly can be used after `SelectLayer()`. You may not call animated functions like `MoveAnim()` or `DisplayBrushFX()` while `SelectLayer()` is active.

Please note that if you use this command on a vector layer (for example a polygon or text layer), the layer will get rasterized automatically. This means that, effectively, the former vector layer will now be a brush layer. The difference between the two is only visible when it comes to transforming the layer: A vector layer can be freely transformed without any losses in quality. A rasterized brush layer, on the other hand, will always have losses in quality when it is transformed.

INPUTS

`id` layer which shall be used as output device

`mode` optional: rendering mode to use (see above); this can be either `#SELMODE_NORMAL` or `#SELMODE_COMBO`; defaults to `#SELMODE_NORMAL`

`frame` optional: in case the specified layer is an anim layer, this argument specifies which frame to select (first frame=1)

`combomode` optional: mode to use when `#SELMODE_COMBO` is active (see above); defaults to 0 (V5.0)

EXAMPLE

```
SelectLayer(1)
SetFillStyle(#FILLCOLOR)
Box(0, 0, 320, 256, #RED)
EndSelect()
```

The code above draws a 320x256 rectangle to layer 1.

34.40 SetLayerAnchor

NAME

SetLayerAnchor – change anchor point of layer (V4.5)

SYNOPSIS

```
SetLayerAnchor(id, ax, ay)
```

FUNCTION

This function can be used to change the anchor point of a layer. The anchor point is a point inside the layer that is used as the origin for all layer transformations (scale, rotate) and also the position of a layer is always relative to the anchor point. Sometimes the anchor point is also referred to as the 'hot spot' of a layer.

The anchor point can be any point inside the layer ranging from 0.0/0.0 (top left corner of the layer) to 1.0/1.0 (bottom right corner of the layer). The center of the layer would be defined by an anchor point of 0.5/0.5.

For example, if you want to have a layer that shall be rotated around its center point, then you need to set this layer's anchor point to 0.5/ 0.5. If it shall be rotated around its top left corner, you have to use 0.0/0.0 as the anchor point. To rotate around the layer's bottom right corner, use 1.0/1.0 as the anchor point. The usual setting is to rotate around the center, so you should normally set the anchor point to 0.5/0.5.

When using an anchor point different than 0.0/0.0, keep in mind that all position specifications will be relative to the anchor point now. This means that a position of 0:0 does not necessarily mean that the layer will appear at the top-left display corner. For example, if you have a layer with an anchor point of 1.0/1.0, moving this layer to position 0:0 (top left corner of display) would make the layer pretty much invisible because its anchor point is set to the bottom-right corner of the layer. Thus, if you move a layer with a bottom-right anchor point to position 0:0, it means that the bottom-right corner

of the layer will actually appear at 0:0. This obviously means that only a single pixel of the layer will be visible. The rest will be off-screen.

By default, all layers use an anchor point of 0.0/0.0.

Starting with Hollywood 10.0, this function can also operate on layer groups so you can also pass the name of a layer group to this function.

INPUTS

`id`	identifier of a layer
`ax`	x coordinate of anchor point; must be between 0.0 and 1.0
`ay`	y coordinate of anchor point; must be between 0.0 and 1.0

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
Box(300, 200, 300, 200, #RED)
WaitLeftMouse
SetLayerAnchor(1, 0.5, 0.5)
WaitLeftMouse
SetLayerAnchor(1, 1.0, 1.0)
WaitLeftMouse
```

The code above demonstrates three different anchor points: First, at 0.0/0.0, then at 0.5/0.5, finally at 1.0/1.0. You can see that the layer will move with every call to `SetLayerAnchor()`. That is because the position of a layer is always relative to its anchor point. Thus, the layer will move although its position will always be 300:200.

34.41 SetLayerBorder

NAME

SetLayerBorder – enable/disable border for layer (V5.0)

SYNOPSIS

```
SetLayerBorder(layer, enable[, color, size])
```

FUNCTION

This command can be used to enable or disable a border effect for the specified layer or layer group depending on whether the `enable` argument is set to `True` or `False`. In the third argument you can specify the color of the border as an ARGB color value. The optional `size` argument can be used to control the size of the border. The size value specifies the desired border size on each side of the layer.

You can also use the `SetLayerStyle()` function to enable/disable the border frame of a layer, or to modify the border's parameters.

INPUTS

`layer`	layer or layer group to use
`enable`	whether to enable or disable the layer border frame (`True` means enable, `False` means disable)

`color` optional: color that shall be used by the border in ARGB notation (defaults to `#BLACK`)

`size` optional: size of border on each side (defaults to 2)

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
Box(#CENTER, #CENTER, 320, 240, #RED)
SetLayerBorder(1, True, #WHITE, 10)
```

The code above draws a red box to the center of the display and then adds a 10 pixel white border frame to it.

34.42 SetLayerDepth

NAME

SetLayerDepth – set layer palette depth (V9.0)

SYNOPSIS

```
SetLayerDepth(id, depth[, t])
```

FUNCTION

This function sets the depth of the palette of the layer specified by `id` to the depth specified in `depth`. `depth` must be a bit depth ranging from 1 (= 2 colors) to 8 (= 256 colors). See Section 44.1 [Palette overview], page 889, for details. Note that if the specified depth is less than that of the pixel data attached to the palette, the pixel data will be remapped to match the new depth.

The following tags are supported by the optional table argument `t`:

`Frame:` If the layer is an anim layer, you can set this tag to specify the frame whose depth should be set. Frames are counted from 1. Defaults to the anim layer's current frame.

`Remap:` If this tag is set to `False`, out-of-range pens will not be remapped to existing pens but instead they will simply be set to the pen specified in the `ClipPen` tag (see below), i.e. no remapping will take place. Note that `Remap` is only effective when reducing colors. If the new depth has more pens than the old depth, `Remap` won't do anything. (V10.0)

`ClipPen:` This is only used in case the `Remap` tag is set to `False` (see above). In that case, out-of-range pens will not be remapped to existing pens but will simply be set to the pen specified in the `ClipPen` tag, i.e. no remapping will take place. Note that `ClipPen` is only effective when reducing colors. If the new depth has more pens than the old depth, `ClipPen` won't do anything. (V10.0)

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of layer to modify

`depth` desired new palette depth (ranging from 1 to 8)

`t` optional: table argument containing further options (see above)

EXAMPLE

```
SetLayerDepth(1, 8)
```

The code above changes the depth of layer 1's palette to 8 (= 256 colors).

34.43 SetLayerFilter

NAME

SetLayerFilter – enable/disable filters for layer (V5.0)

SYNOPSIS

```
SetLayerFilter(layer, table)
```

FUNCTION

This command can be used to control which filters are applied to a layer, and in what order they will be applied to the layer. You have to pass a table to this function that contains a number of subtables, each of which contains information for a single layer filter. The following tags are supported for each subtable:

`Name` Contains the name of the filter that this subtable element shall configure. This tag is mandatory and must always be specified in every subtable so that `SetLayerFilter()` knows the filter the subtable is addressing. Please see below for a list of supported filter types.

`Args` Contains an array of arguments for the filter specified in the `Name` tag. The values passed here depend on the filter specified. Some filters like `XFlip` or `Gray` do not require any arguments at all. In that case, you do not have to pass the `Args` table. Please see below to learn about the arguments required by the single filters. Also note that `SetLayerFilter()` has fallback arguments for every filter it supports. Thus, you can also leave out arguments. In that case, `SetLayerFilter()` will use its default settings for the respective filter.

`Disable` This tag can be used to enable or disable a filter. Pass `False` here to enable the filter, or `True` to disable it. This tag is optional. If it is not specified, the filter will get enabled by default.

`Priority` This tag allows you to specify a priority level for the current filter. A priority level is simply a numeric value which is then used by `SetLayerFilter()` to find out the order in which the filters should be applied. The priority level must be between 0 (= lowest priority) and 255 (= highest priority). As an example, if you assign a priority of 10 to the `Blur` filter, and a priority of 9 to the `Tint` filter, the blur filter will be applied before the tint filter. This tag will default to 0 if not specified.

A list of supported layer filters follows below. Please note that the arguments must not be passed to the filter as a named table tag but sequentially in the `Args` array. I.e. for the `Modulate` filter, you would put the brightness setting in array element 0, the

saturation setting in array element 1, and the hue setting in element 2. The order in which the single arguments are listed below corresponds to the order in which they are expected in the `Args` table. Here is the list now:

`Blur` This filter will apply a Gaussian blur to the layer. The following arguments need to be passed:

- `Radius:` Specifies the blur radius. The greater the value you specify here, the longer the blurring will take.

`Charcoal` This filter will apply a charcoal filter to the layer. The following arguments need to be passed:

- `Radius:` Specifies the effect radius. The greater the value you specify here, the longer the calculation will take.

`Contrast` This filter will enhance or reduce color contrast in the layer. The following arguments need to be passed:

- `Inc:` Pass `True` here to enhance the color contrast, or `False` to decrease the color contrast.
- `Repeat:` Specifies how many times the effect should be repeated. This is useful for a more pronounced effect. By default this is set to 1 which means that the effect is only applied once. If you would like to have two passes, specify 2 here, etc. Remember that the greater the number you specify here is, the longer the computation of the result will take.

`Edge` This filter will apply an edge detection filter to the layer. The following arguments need to be passed:

- `Radius:` Specifies the effect radius. The greater the value you specify here, the longer the calculation will take.

`Emboss` This filter will apply an emboss filter to the layer. The following arguments need to be passed:

- `Radius:` Specifies the effect radius. The greater the value you specify here, the longer the calculation will take.

`Gamma` This filter can be used to apply gamma correction to the layer. The following arguments need to be passed:

- `Red:` Gamma correction for red color channel.
- `Green:` Gamma correction for green color channel.
- `Blue:` Gamma correction for blue color channel.

Each value must be floating point value. A value of 1.0 means no change, a value smaller than 1.0 darkens the channel, a value greater than 1.0 lightens it. See Section 21.34 [GammaBrush], page 278, for details.

`Grayscale`
This filter will map the layer to gray. There are no arguments for this filter.

`Invert` This filter will invert the colors of the layer. There are no arguments for this filter.

`Modulate` This filter can be used to modulate brightness, saturation, and hue values of a layer. The following arguments need to be passed:

`Brightness:`
: Desired brightness modulation.

`Saturation:`
: Desired saturation modulation.

`Hue:` Desired hue modulation.

Each value must be floating point value. A value of 1.0 means no change, a value smaller than 1.0 reduces the brightness/saturation/hue, while a value greater than 1.0 enhances it. See Section 21.45 [ModulateBrush], page 286, for details.

`Monochrome`
: This filter will apply a black and white filter to this layer. The following arguments need to be passed:

`Dither:` Specifies whether or not dithering should be used. Pass `True` or `False` here. Dithering looks better, but is of course slower.

`OilPaint` This filter will apply an oil paint filter to the layer. The following arguments need to be passed:

`Radius:` Specifies the effect radius. The greater the value you specify here, the longer the calculation will take.

`Pixelate` This filter will zoom the pixel cells of the layer to the specified size. The following arguments need to be passed:

`CellSize:`
: Specifies the desired zoom size. Every pixel of the layer will be zoomed to this size, starting from the top-left corner of the layer.

See Section 21.50 [PixelateBrush], page 291, for details.

`Quantize` This filter will reduce the number of colors in the layer. The following arguments need to be passed:

`Colors:` Desired number of colors. This must be between 1 and 256.

`Dither:` `True` to enable dithering, `False` to disable it.

See Section 21.52 [QuantizeBrush], page 292, for details. (V6.0)

`SepiaTone`
: Applies a sepia-tone filter to the layer. The following arguments need to be passed:

`Level:` Desired sepia-toning level. This must be between 0 and 255, or alternatively it can be a string containing a percentage specification. The usual setting is `"80%"` (i.e. a level of about 204).

See Section 21.66 [SepiaToneBrush], page 306, for details.

`Sharpen` Applies a sharpening filter to the layer. The following arguments need to be passed:

`Radius:` Specifies the sharpen radius. The greater the value you specify here, the longer the calculation will take.

`Solarize` Applies a solarization effect to the layer. The following arguments need to be passed:

`Level:` Desired solarization level (must be between 0 and 255).

See Section 21.75 [SolarizeBrush], page 312, for details.

`Swirl` Swirls the layer by the specified number of degrees. The following arguments need to be passed:

`Degrees:` Specifies the desired swirling amount. This can be between 0 (no swirling) and 360 (full swirl).

`Tint` This filter will tint the layer with the specified color at the specified ratio. The following arguments need to be passed:

`Color:` Specifies the tinting color in RGB format.

`Ratio:` Specifies the tinting ratio. This can be a value between 0 (= no tinting) and 255 (= full tinting), or a string containing a percentage specification (e.g. `"50%"` corresponds to a ratio of 128).

`WaterRipple`
This filter will apply water ripples to the layer. The following arguments need to be passed:

`Wavelength:`
Desired wavelength for the effect.

`Ampltiude:`
Desired ripple amplitude.

`Phase:` Desired ripple phase.

`CX:` X center point of water ripple.

`CY:` Y center point of water ripple.

See Section 21.81 [WaterRippleBrush], page 316, for details.

`XFlip` This will mirror the layer on the x-axis. There are no arguments for this filter.

`YFlip` This will mirror the layer on the y-axis. There are no arguments for this filter.

To disable all layer filters, you can pass the special value 0 instead of a table in the second argument. `SetLayerFilter()` will then cancel all filters that are currently active on the specified layer.

Please note that this command will not reset any existing filter settings when it is called. Instead, all existing filters settings will be kept and the new settings will merely be merged with the old ones. So if you have a layer that has several filters attached and you only want to change the configuration of one of these filters, it is sufficient to just pass a subtable for this single filter to `SetLayerFilter()`. It is not necessary to pass all the other filters to `SetLayerFilter()` again.

Also note that layer filters can get quite heavy on the CPU; especially when using transition effects on a layer that has filters attached. In that case, the filters have to be recalculated for each new frame of the transition effect. Depending on the complexity of the filter, this can take some time.

You can also use the `SetLayerStyle()` command to change the configuration of one or more layer filters.

Starting with Hollywood 10.0, this function can also operate on layer groups so you can also pass the name of a layer group to this function.

INPUTS

`layer` layer or layer group to use

`table` a table containing one or more subtables that contain a description of filters to apply to or remove from the layer; see above for more information; to remove all filters from a layer, pass 0 here instead of a table

EXAMPLE

```
table = {
{Name = "YFlip"},
{Name = "Modulate", Args = {1.0, 2.0, 1.0}, Priority = 10},
{Name = "Swirl", Args = {128}, Priority = 9} }
SetLayerFilter(1, table)
```

The code above increases the saturation of layer 1 by 200%, swirls the layer by 180 degrees and then mirrors it on the y-axis.

```
SetLayerFilter(1, {{Name = "YFlip", Disable = True}})
```

The code above removes the "YFlip" filter from layer 1 but keeps the other two filters (modulate and swirl).

34.44 SetLayerName

NAME

SetLayerName – assign a layer name (V2.0)

SYNOPSIS

```
SetLayerName(id, name$)
```

FUNCTION

You can use this function to assign a name to the layer specified by `id`. This is very useful if you have multiple layers whose identifiers change constantly (e.g. because you frequently remove and add layers). If you give your layers names, you do not have to

worry about on which position the layer currently resides. You can easily access it by just using its name. All functions which accept layer id's, will also accept names.

Please note that the name for the layer must be unique within the current background picture's layer cache. Layer names are case insensitive, i.e. "layer1" is the same layer as "LAYER1".

To find out which id a named layer currently occupies, you can use the `#ATTRLAYERID` attribute with the `GetAttribute()` command.

If you want to assign a name to the newest layer, simply pass 0 and Hollywood will automatically use the top layer. To remove a layer's name, pass an empty string in `name$`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of the layer to be named or 0 for the last layer added
`name$`	desired name for the layer

EXAMPLE

```
EnableLayers()
SetFillStyle(#FILLCOLOR)

Box(0, 0, 100, 100, #RED)      ; create layer 1
Box(50, 50, 100, 100, #GREEN)  ; create layer 2

SetLayerName(1, "redbox")      ; give them names
SetLayerName(2, "greenbox")    ; give them names

SwapLayers("redbox", "greenbox")  ; swap 'em! Now greenbox is layer 1
                                  ; and redbox is layer 2!

ShowLayer("redbox", #RIGHT, #BOTTOM)  ; move layer 2 to bottom-right
ShowLayer("greenbox", #LEFT, #TOP)    ; move layer 1 to top-left
```

You see that it is much easier to work with string names for layers instead of layer id's which are relative to the layer's position.

34.45 SetLayerPalette

NAME

SetLayerPalette – change layer palette (V9.0)

SYNOPSIS

```
SetLayerPalette(id, palid[, t])
```

FUNCTION

This function replaces the palette of the layer specified by `id` with the palette specified by `palid`. The optional table argument `t` allows you to specify some further options. The following tags are currently supported by the optional table argument `t`:

`Remap:` If this is set to `True`, the pixels of the layer will be remapped to match the colors of the new palette as closely as possible. By default, there will be no remapping and the actual pixel data of the layer will remain untouched. If you want remapping, set this tag to `True` but be warned that remapping all pixels will of course take much more time than just setting a new palette without remapping. Defaults to `False`.

`Dither:` If the `Remap` tag (see above) has been set to `True`, you can use the `Dither` tag to specify whether or not dithering should be used. Defaults to `True` which means dithering should be used.

`CopyCycleTable:`
Palettes can have a table containing color cycling information. If you set this tag to `True`, this cycle table will be copied to the layer as well. Defaults to `False`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of layer to use
`palid`	identifier of palette to copy to layer
`t`	optional: table for specifying further options (see above)

34.46 SetLayerPen

NAME

SetLayerPen – change layer palette pen (V9.0)

SYNOPSIS

```
SetLayerPen(id, pen, color)
```

FUNCTION

This function sets the color of the pen specified by `pen` to the color specified by `color` in the palette of the layer specified by `id`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of layer
`pen`	pen you want to modify (starting from 0)
`color`	new color for the pen, must be specified as an RGB color

EXAMPLE

```
SetLayerPen(1, 0, #RED)
```

The code above sets pen 0 to red in the palette of layer 1.

34.47 SetLayerShadow

NAME

SetLayerShadow – enable/disable drop shadow for layer (V5.0)

SYNOPSIS

```
SetLayerShadow(layer, enable[, color, radius, size, dir])
```

FUNCTION

This command can be used to enable or disable a shadow effect for the specified layer or layer group depending on whether the `enable` argument is set to `True` or `False`. In the third argument you can specify the color of the shadow. This will usually be `#BLACK` but combined with a transparency value because simple opaque black does not look too good as a shadow. You can use the `ARGB()` function to combine a transparency value and a color into an ARGB color. The optional arguments `radius` and `size` can be used to control the shadow's smoothness and size. Usually, both values are set to about the same value. Finally, the `dir` argument can be used to control the shadow's direction. This argument must be set to one of Hollywood's directional constants. See Section 27.5 [Directional constants], page 491, for details.

Please note that drop shadows can become quite heavy on the CPU because Hollywood has to recalculate them whenever the layer's contents change. Normally, this does not happen too often. There is one prominent exception, though: When you run a transition effect on a layer that has a drop shadow. In that case, Hollywood has to remake the drop shadow for every new frame of the transition effect. On slower systems this can quite possibly kill the show so that you might want to turn off drop shadows before running a transition effect on a layer.

You can also use the `SetLayerStyle()` function to enable/disable the drop shadow of a layer, or to modify the drop shadow's parameters.

INPUTS

`layer`	layer or layer group to use
`enable`	whether to enable or disable the layer drop shadow (`True` means enable, `False` means disable)
`color`	optional: color that shall be used by the drop shadow inARGB notation (defaults to $80000000 which means black with 50% transparency)
`radius`	optional: radius for shadow smoothing (defaults to 5)
`size`	optional: size of shadow shift from main layer (defaults to 4)
`dir`	optional: light direction of drop shadow (defaults to `#SHDWSOUTHEAST`)

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
Box(#CENTER, #CENTER, 320, 240, #RED)
SetLayerShadow(1, True)
```

The code above draws a red box to the center of the display and then adds a shadow to it.

34.48 SetLayerStyle

NAME

SetLayerStyle – change the style of one or more layers (V4.0)

SYNOPSIS

```
SetLayerStyle(id1, style1, ...)
```

FUNCTION

This command can be used to modify nearly all attributes of one or more layers or layer groups with a single call. It is a very powerful command which can be used to realize complex animation mechanisms in a very easy and straightforward way. For each layer or layer group whose style you want to modify, you always have to pass the layer or group id followed by a table containing the attributes you want to change. You can repeat this pattern as many times as you need it.

The configuration of the style table depends on the type of layer specified. However, some style elements can be used with all layer types. These generic layer styles will be covered first. The specific layer styles dependent on layer type will be dealt with below.

The following style elements are generic and can thus be used with every layer:

`X,Y` Specifies the position for the layer. If not specified the layer will keep its current position.

`Width,Height`
Can be used to scale the layer to new dimensions. This can either be a numerical value specifying a new pixel size or a string containing a percent specification (e.g. `"50%"`). See Section 34.37 [ScaleLayer], page 675, for details.

`Rotate` This style element can be used to control layer rotation. You have to pass a value in degrees here. A positive value means anti-clockwise rotation, negative values rotate in clockwise direction. See Section 34.36 [RotateLayer], page 674, for details..

`SmoothScale`
Specifies whether or not smooth scaling and rotation shall be used on this layer. This is only applicable for non-vector layers and of course it only makes sense when the layer is scaled or rotated.

`Transparency`
Use this style element to modify the transparency setting of a layer. This can be either a value ranging from 0 (= no transparency) to 255 (= full transparency) or a string containing a percent specification (e.g. `"50%"` which means half transparency). See Section 34.50 [SetLayerTransparency], page 702, for details.

`Tint` Use this style element to modify the tint setting of a layer. This can be either a value ranging from 0 (= no tinting) to 255 (= opaque tinting) or a string containing a percent specification (e.g. `"50%"` which means half tinting). If this is set to non zero, the layer will be tinted with the color specified in `TintColor` at the specified level. See Section 34.49 [SetLayerTint], page 701, for details.

`TintColor`
: Specifies the color to use for tinting. Can only be used in union with the `Tint` style element.

`Hidden`
: You can use this style element to show or hide the specified layer. Set it to `True` to hide the layer or to `False` to show it. See Section 34.17 [HideLayer], page 661, for details.

`Type`
: This style element allows you to change the layer type. Please note that if you change the type of a layer, you will most likely need to provide further information to `SetLayerStyle()`. For instance, if you choose to convert a `#ELLIPSE` layer into a `#BRUSH` layer, it is mandatory that you also specify the `ID` style element to tell Hollywood which brush shall replace the ellipse layer. Hollywood will try to inherit general style elements like color, position, size from the previous layer type, but for certain type conversions you need to specify additional elements. If `Type` is specified, only style elements that are supported by the newly set type will be handled. (V4.5)

`ClipRegion`
: Use this style to change the clipping region of this layer. Every layer can have its private clip region. See Section 30.31 [SetClipRegion], page 611, for details.

`ScaleX, ScaleY`
: This is an alternative way of scaling the layer. You have to pass a floating point value here that indicates a scaling factor. For example, 0.5 means half the size, 2.0 means twice the size. This is especially convenient if you would like to keep the proportions of the layer that you want to scale. If you use the same factor for `ScaleX` and `ScaleY`, the proportions of the layer will remain intact. Please note that `ScaleX` / `ScaleY` and `Width` / `Height` are mutually exclusive. You must not mix both groups. Either use `ScaleX` / `ScaleY` or stick to `Width` / `Height`. (V4.5)

`Transform`
: This tag allows you to apply a 2x2 transformation matrix to this layer. Transformation matrices are useful if you want to apply scaling and rotation at the same time, or if you want to mirror a layer. You have to pass a table to `Transform`. The table must contain the four constituents of a 2x2 transformation matrix in the following order: `sx`, `rx`, `ry`, `sy`. See Section 21.78 [TransformBrush], page 314, for more about transformation matrices. Please note that the `Transform` tag is mutually exclusive with the following tags: `Width` / `Height` / `ScaleX` / `ScaleY` / `Rotate`. You must not combine it with any of these tags. (V4.5)

`AnchorX, AnchorY`
: You can use these two tags to change the anchor point of this layer. The anchor point can be any point between 0.0/ 0.0 (top left corner of the layer) and 1.0/1.0 (bottom right corner of the layer). The center of the layer would be defined by an anchor point of 0.5/0.5. All transformations (scaling, rotation etc.) will be applied relative to the anchor point. Also, the position of a layer

is always relative to its anchor point. See Section 34.40 [SetLayerAnchor], page 678, for details. (V4.5)

`NoClipTransform`
This tag can be used to disable automatic clip region transformation. By default, when you transform a layer, its clip region will be transformed in the same vein. To forbid this behaviour, set `NoClipTransform` to `True`.

`TextureX, TextureY`
These tags only work with graphics primitives that are filled using `#FILLTEXTURE` style. If that is the case, you can use these tags to control the start offset inside the texture brush. See Section 27.14 [SetFillStyle], page 498, for details. (V4.6)

`Z` This tag can be used to change the z-position of this layer. The z-position of a layer is its position in the hierarchy of layers. The first (i.e. backmost layer) has a z-position of 1, the last (i.e. frontmost) layer's z-position is equal to the number of layers currently present. You need to pass the new desired z-position for the specified layer in this tag. The layer will then assume exactly this z-position, existing layers that are on or after this z-position will be shifted down. To move a layer all the way to the front (i.e. highest z-position), you can pass the special value 0; to move a layer all the way to the back, specify 1. See Section 34.53 [SetLayerZPos], page 704, for details. (V4.7)

`TranslateX, TranslateY`
These two can be used to translate a layer by a specified delta x and delta y offset. A layer translation means moving the layer relative by the specified delta offsets relative to its current position. Thus, a translation of (1,1) would move the layer one pixel to the right, and one pixel to the bottom. See Section 34.59 [TranslateLayer], page 708, for details. Please note: These two tags and the `X` / `Y` tags are mutually exclusive. You cannot use them together. (V4.7)

`Shadow` This tag can be used to turn the drop shadow of a layer on and off. If you set this tag to `True`, the drop shadow will be turned on, else it will be turned off. If you are turning it on, you can configure the look of the shadow using the `ShadowDir`, `ShadowSize`, `ShadowColor`, and `ShadowRadius` tags. See below for more information. See Section 34.47 [SetLayerShadow], page 688, for details. (V5.0)

`ShadowDir`
Specifies the direction of the shadow for this layer. This must be one of Hollywood's directional constants. This tag is only handled when the shadow is currently turned on. (V5.0)

`ShadowColor`
Specifies the color of the shadow for this layer. This must be an ARGB value that can contain a transparency setting. This tag is only handled when the shadow is currently turned on. (V5.0)

`ShadowPen`
: Specifies the pen to use for drawing the layer's shadow when palette is `#PALETTEMODE_PEN`. See Section 44.35 [SetShadowPen], page 918, for details. (V9.0)

`ShadowSize`
: Specifies the size of the shadow for this layer. This tag is only handled when the shadow is currently turned on. (V5.0)

`ShadowRadius`
: Specifies the shadow radius for this layer. This tag is only handled when the shadow is currently turned on. (V5.0)

`Border`
: This tag can be used to turn the border frame of a layer on and off. If you set this tag to `True`, the border frame will be turned on, else it will be turned off. If you are turning it on, you can configure the look of the border using the `BorderSize` and `BorderColor` tags. See below for more information. See Section 34.41 [SetLayerBorder], page 679, for details. (V5.0)

`BorderColor`
: Specifies the color of the border for this layer. This must be an ARGB value that can contain a transparency setting. This tag is only handled when the border is currently turned on. (V5.0)

`BorderPen`
: Specifies the pen to use for drawing the layer's border when palette mode is `#PALETTEMODE_PEN`. See Section 44.22 [SetBorderPen], page 907, for details. (V9.0)

`BorderSize`
: Specifies the size of the border for this layer. This tag is only handled when the border is currently turned on. (V5.0)

`Filters`
: This tag can be used to apply filters to a layer, remove filters from a layer, or modify the parameters of already applied filters. You have to pass a table to this tag that describes the desired configuration of the single filters. See Section 34.43 [SetLayerFilter], page 681, for details. (V5.0)

`ClearFilters`
: This tag can be used to remove all filters from a layer. Simply specify `True` here and `SetLayerStyle()` will remove all filters that are currently active from this layer.

`PaletteMode`
: Specifies the palette mode to use for the layer when Hollywood is in palette mode. This must be one of the palette modes supported by `SetPaletteMode()`. See Section 44.31 [SetPaletteMode], page 914, for details. (V9.0)

`DitherMode`
: Specifies the dither mode to use for the layer when Hollywood is in palette mode. This must be one of the dither modes supported by

`SetDitherMode()`. See Section 44.26 [SetDitherMode], page 910, for details. (V9.0)

`IgnoreAnchor`
: If this tag is set to `True`, Hollywood will ignore the layer's anchor point when changing its position or style. This means that the anchor point will be treated as 0/0. This can be useful e.g. if you'd like to change the text of a layer whose anchor point is not 0/0. In such cases changing the text would also re-position the layer. This can be prevented by setting `IgnoreAnchor` to `True`. `IgnoreAnchor` can also be useful in case you want to position layers, whose anchor point is not 0/0, relative to their top-left corner. (V9.1)

`Refresh` Force a layer refresh. This is only useful for merged layers created by `MergeLayers()` because they don't refresh automatically when the graphics of one or more of its source layers are changed. See Section 34.24 [MergeLayers], page 666, for details. (V10.0)

The following style elements are dependent on a specific layer type:

`#ANIM` layers can use the following elements:

`ID` This table element can be used to assign a new animation to the specified layer. The old animation will then be replaced with the animation specified in `ID`.

`Frame` You can use this style element to display a specific frame of the animation that sits on this layer. Frames are counted from 1. You can pass the special value 0 to display the next frame in the animation. See Section 34.27 [NextFrame], page 669, for details.

`#BRUSH`, `#BRUSHPART` and `#BGPICPART` layers recognize the following elements:

`ID` This table element can be used to assign a new brush/bgpic to the specified layer. The old brush/bgpic will then be replaced with the new one specified by `ID`. This is useful for example for a slideshow in which a layer shall display a new picture every `n` seconds.

`PartX, PartY, PartWidth, PartHeight`
: These four elements allow you to configure the visible portion of the brush or bgpic. `PartX` and `PartY` specify the x and y coordinates inside the brush/bgpic and `PartWidth` and `PartHeight` specifies the size of the tile that shall be visible. This is useful for displaying only a part of a brush or bgpic. For more information please read the notes on `DisplayBrushPart()` and `DisplayBGPicPart()`. Please note that these elements are not restricted to layer types `#BRUSHPART` and `#BGPICPART` only but they can also be used with layers of type `#BRUSH`. If you use one of the `PartXXX` elements on `#BRUSH` layer, the layer will automatically be changed into a `#BRUSHPART` layer.

The following elements are generic for layer types `#ARC`, `#BOX`, `#CIRCLE`, `#ELLIPSE`, `#POLYGON`, and `#VECTORPATH`:

`Color` Specifies the color of the layer in ARGB notation.

`DrawPen:` When palette mode is `#PALETTEMODE_PEN`, `DrawPen` specifies the pen that should be used to draw this layer. See Section 44.27 [SetDrawPen], page 911, for details. (V9.0)

`FormStyle`
This allows you to change the form style of the layer. You can pass one or more styles here. If you pass multiple form styles, you need to use the bitwise Or operator (`|`). See Section 27.15 [SetFormStyle], page 499, for possible combinations. If you want to remove a form style from a layer, use the `FormStyleClear` element. Note: As of Hollywood 5.0 the only reasonable style to set using this tag is `#ANTIALIAS` because the shadow and border settings are now better controlled using their separate tags (see above).

`FormStyleClear`
All form styles which you set in this element will be removed from the layer. Multiple form styles have to be separated by the bitwise Or operator (`|`). This is the counterpart to the `FormStyle` element. Note: As of Hollywood 5.0 the only reasonable style to unset using this tag is `#ANTIALIAS` because the shadow and border settings are now better controlled using their separate tags (see above).

`FillStyle`
You can use this style element to change the filling style for this layer. See Section 27.14 [SetFillStyle], page 498, for details.

`GradientStyle`
Specifies the style of the gradient if filling style is set to `#FILLGRADIENT`. This can be `#LINEAR`, `#RADIAL`, or `#CONICAL`.

`GradientAngle`
Specifies the orientation of the gradient if filling style is set to `#FILLGRADIENT`. The angle is expressed in degrees. Only possible for `#LINEAR` and `#CONICAL` gradients.

`GradientStartColor, GradientEndColor`
Use these two to configure the colors of the gradient if filling style is set to `#FILLGRADIENT`.

`GradientCenterX, GradientCenterX`
Sets the center point for gradients of type `#RADIAL` or `#CONICAL`. Must be a floating point value between 0.0 and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientBalance`
This tag controls the balance point for gradients of type `#CONICAL`. Must be a floating point value between 0.0 and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientBorder`
: This tag controls the border size for gradients of type `#RADIAL`. Must be a floating point value between 0.0 and and 1.0. See Section 20.6 [CreateGradientBGPic], page 232, for details. (V5.0)

`GradientColors`
: This tag can be used to create a gradient between more than two colors. This has to be set to a table that contains sequences of alternating color and stop values. See Section 20.6 [CreateGradientBGPic], page 232, for details. If this tag is used, the `GradientStartColor` and `GradientEndColor` tags are ignored. (V5.0)

`OutlineThickness`
: If filling style is set to `#FILLNONE` this value can be used to configure the thickness of the outline. See Section 27.14 [SetFillStyle], page 498, for details.

`TextureBrush`
: If filling style is set to `#FILLTEXTURE` you can change the currently used texture with this style element. Simply pass the identifier of a brush in this style element to switch to a new texture.

In addition to the elements above, layers of type `#ARC` accept the following style elements:

`RadiusA, RadiusB`
: These two values specify the x and y radii of the partial ellipse.

`StartAngle, EndAngle`
: These two values specify the start and end angles of the partial ellipse. See Section 27.1 [Arc], page 487, for details.

`Clockwise`
: Specifies whether or not elliptic arc shall be drawn in clockwise direction. See Section 27.1 [Arc], page 487, for details. (V4.5)

Layers of type `#BOX` accept the following additional style elements:

`SizeX, SizeY`
: You can use these two values to change the dimensions of the rectangle. `SizeX` specifies the rectangle width and `SizeY` specifies its height.

`RoundLevel`
: Specifies the rounding levels for the four corners of the rectangle. A value of 0 means no rounding. A value of 100 means completely round corners. See Section 27.2 [Box], page 488, for details.

`CornerA, CornerB, CornerC, CornerD`
: These four tags allow you to fine-tune the corner rounding of the rectangle. You can specify a rounding level (0 to 100) for every corner of the rectangle thus allowing you to create a rectangle where not all corners are rounded, or where the different corners use different rounding levels. These tags will override any setting specified in the `RoundLevel` tag. (V5.0)

Layers of type `#CIRCLE` accept the following additional style elements:

`Radius` Specifies the radius of the circle. See Section 27.3 [Circle], page 489, for details.

Layers of type `#ELLIPSE` accept the following additional style elements:

`RadiusA, RadiusB`
These two values specify the x and y radii of the ellipse. See Section 27.6 [Ellipse], page 491, for details.

Layers of type `#LINE` accept the following additional style elements:

`Thickness`
Specifies the thickness of the line. See Section 27.10 [Line], page 494, for details.

`X1,Y1,X2,Y2`
Use these tags to change the line orientation. Please note that these tags are mutually exclusive with the generic `X` / `Y` tags. If you use those tags, you must not use these tags and vice versa. (V4.6)

`Arrowhead`
This tag allows you to turn the line into an arrow. It can be set to one of the following tags:

`#ARROWHEAD_NONE`
No arrowhead. This is the default mode.

`#ARROWHEAD_SINGLE`
Add arrowhead to end of line.

`#ARROWHEAD_DOUBLE`
Add arrowhead to start and end of line.

(V9.1)

Layers of type `#POLYGON` accept the following additional style elements:

`Vertices` This style element can be used to change the look of the polygon by passing a new set of vertices to it. You have to set this style element to a table of vertices containing a sequence of x and y coordinates where both coordinates define one vertex. It uses the same format as with the `Polygon()` command except that you do not have to specify the number of vertices in the table. `SetLayerStyle()` will determine this automatically.

`#PRINT` and `#TEXTOUT` layers recognize the following style elements:

`Color` Specifies the color of the text in ARGB notation.

`DrawPen:` When palette mode is `#PALETTEMODE_PEN`, `DrawPen` specifies the pen that should be used to draw the text. See Section 44.27 [SetDrawPen], page 911, for details. (V9.0)

`FontStyle`
This allows you to change the font style of the layer. You can pass one or more styles here. If you pass multiple font styles, you need to use the bitwise Or operator (`|`). See Section 54.33 [SetFontStyle], page 1143, for possible combinations. If you want to remove a font style from a layer, use the `FontStyleClear` element. Note: As of Hollywood 5.0 this tag should no longer be used to set shadow and border styles. For these styles, you should better use their new separate tags (see above).

`FontStyleClear`
All font styles which you set in this element will be removed from the layer. Multiple font styles have to be separated by the bitwise Or operator (`|`). This is the counterpart to the `FontStyle` element. Note: As of Hollywood 5.0 this tag should no longer be used to unset shadow and border styles. For these styles, you should better use their new separate tags (see above).

`Font` You can use this style element to change the font of the text layer. See Section 54.31 [SetFont], page 1139, for details. Alternatively, you can also specify the new font by setting the `ID` tag (see below).

`FontSize` You can use this style element to change the font size of the specified text layer. See Section 54.31 [SetFont], page 1139, for details.

`ID` You can use this style element to change the font of the text layer. Just set this tag to the ID of the new font and the font will be changed. See Section 54.41 [UseFont], page 1156, for details. Alternatively, you can also specify the new font by name by setting the `Font` tag (see above). (V10.0)

`Text` This style element allows you to change the contents of the text layer. You can replace the whole old contents of the layer with some new text.

`Align` Allows you to change the text alignment after a newline characters. Possible values are `#LEFT`, `#RIGHT`, `#CENTER`, and `#JUSTIFIED`. The default alignment is `#CENTER`.

`LeftMargin, RightMargin`
Allows you to change the margin settings of the current text layer. `LeftMargin` is only used for layers of type `#PRINT` but `RightMargin` can also be used for text objects and `#TEXTOUT` layers. See Section 54.34 [SetMargins], page 1144, for details.

`CursorX, CursorY`
Allows you to change the cursor position of this layer. This is only possible with layers of type `#PRINT`. Also note that if you specify `CursorX` / `CursorY` you must not specify `X` / `Y`. `CursorX` / `CursorY` and `X` / `Y` are mutually exclusive and hence must not be used together. (V4.5)

`Tabs` Allows you to modify the tabulator positions for this layer. This is only possible with layers of type `#PRINT`. `Tabs` takes a table of tabulator positions. See Section 54.3 [AddTab], page 1118, for details. (V4.5)

`Encoding` Allows you to change the character encoding of this text layer. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V4.7)

`Linespacing:`
Allows you to adjust the space between lines. You can set this to a positive or negative value. A positive value will increase the space between lines, a negative value will decrease it. (V9.0)

`Charspacing:`
Allows you to adjust the space between characters. You can set this to a positive or negative value. A positive value will increase the space between characters, a negative value will decrease it. (V10.0)

`Tabs:` Allows you to modify the tab stops for this layer. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`ListMode:`
Allows you to enable or disable list mode for this text layer. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`DefListBullet:`
Use this tag to set the default bullet to use when the text layer is in list mode. See Section 54.39 [TextOut], page 1149, for a list of available bullets. (V9.0)

`ListBullet:`
Use this tag to specify a custom set of bullets to use when the text layer is in list mode. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`DefListIndent:`
This tag can be used to specify the number of spaces to use for indenting list items when the text layer is in list mode. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`ListIndent:`
Use this tag to specify a custom set of indentation levels to use when the text layer is in list mode. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`DefListOffset:`
When the text layer is in list mode and uses a numbered bullet type you can use this tag to specify a starting offset for the numbering. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`ListOffset:`
When the text layer is in list mode and uses a numbered bullet type you can use this tag to specify a custom set of starting offsets for the individual list numbering. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`DefListSpacing:`
: This tag can be used to specify the line spacing between the list items when the text layer is in list mode. See Section 54.39 [TextOut], page 1149, for details. (V9.1)

`ListSpacing:`
: Use this tag to specify a custom set of line spacings to use when the text layer is in list mode. See Section 54.39 [TextOut], page 1149, for details. (V9.1)

`Frame:`
: When the text layer is in list mode, you can use this tag to display a specific frame of the list. Frames are counted from 1. To show the next frame, you can pass the special value 0 to `Frame`. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`FrameMode:`
: When the text layer is in list mode, you can use this tag configure the frame mode that the list should use. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`BulletColor:`
: When the text layer is in list mode, you can use this tag to change the bullet color. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`BulletPen:`
: When the text layer is in list mode, you can use this tag to change the bullet pen. See Section 54.39 [TextOut], page 1149, for details. (V9.0)

`Wordwrap:`
: This tag can be used to set a wordwrapping width for the text layer. Whenever a word exceeds the specified width, it will be wrapped to the next line. Set `Wordwrap` to 0 to disable wordwrapping. (V9.0)

Layers of type `#TEXTOBJECT` accept the following additional style elements:

`ID`
: This style element can be used to associate a new text object with this layer. Just pass the identifier of the desired text object here and it will replace the current text object of this layer. (V4.5)

Layers of type `#VECTORPATH` accept the following additional style elements:

`ID`
: This style element can be used to associate a new vector path object with this layer. Just pass the identifier of the desired vector path object here and it will replace the current path of this layer. (V5.0)

`LineJoin`
: Allows you to change the line join style of this layer. See Section 56.33 [SetLineJoin], page 1195, for details. (V5.0)

`LineCap`
: Allows you to change the line cap style of this layer. See Section 56.32 [SetLineCap], page 1195, for details. (V5.0)

`FillRule`
: Allows you to change the fill rule style of this layer. See Section 56.31 [SetFillRule], page 1194, for details. (V5.0)

`Dashes` This style element can be used to change the dash pattern for outline vector drawing. You have to pass a table here that contains a dash pattern in the same format as described in the documentation of the `SetDash()` command. When passing the `Dashes` tag, you should also pass the `DashOffset` tag (see below) to define the starting offset of the dash pattern. (V5.0)

`DashOffset`
This tag can be used to modify the starting offset of the dash pattern of this layer. This tag is usually specified together with the `Dashes` tag (see above). See Section 56.30 [SetDash], page 1193, for details. (V5.0)

`VectorEngine:`
This tag can be used to set the vectorgraphics renderer that should be used to draw this layer. See Section 56.35 [SetVectorEngine], page 1196, for details. (V6.0)

`#VIDEO` layers recognize the following elements:

`ID` This tag can only be queried by using `GetLayerStyle()`. It contains the identifier of the video that has been assigned to this layer. You cannot currently assign a new video object to a layer using `SetLayerStyle()`. (V6.0)

`PartX, PartY, PartWidth, PartHeight`
These four elements allow you to configure the visible portion of the video layer. `PartX` and `PartY` specify the x and y coordinates inside the video and `PartWidth` and `PartHeight` specifies the size of the tile that shall be visible. This is useful for displaying only a part of a video. (V6.0)

INPUTS

`id1` identifier of the layer or layer group whose style you want to change

`style1` table containing one or more style elements from the lists above

`...` optional: you can repeat the id/style sequence as often as you need so you can modify the styles of many layers with just a single call

EXAMPLE

```
SetLayerStyle(1, {x = #LEFT, y = #TOP}, 4, {x = #CENTER, y = #CENTER},
5, {x = #RIGHT, y = #BOTTOM}, "mylayer", {x = 100, y = 100})
```

The call above changes the position of several layers. Layer 1 is moved to the top left corner, layer 4 to the center, layer 5 to the bottom right corner, and layer "mylayer" is moved to 100:100.

```
Box(0, 0, 100, 100, #BLUE)
WaitLeftMouse
SetLayerStyle(1, {Color = #RED})
```

The code above draws a blue box on the screen, waits for the left mouse button and then changes the color of the box to red.

```
SetLayerStyle(1, {Frame = 0})
```

The code above displays the next frame of layer 1 (which must be of type `#ANIM`).

```
Polygon(#CENTER, #CENTER, {0, 0, 319, 0, 319, 159, 0, 159}, 4, #RED)
WaitLeftMouse
SetLayerStyle(1, {Vertices = {0, 159, 160, 0, 319, 159}, Color = #YELLOW})
```

The code above draws a red rectangular polygon, waits for left mouse and then changes the rectangular polygon into a yellow triangular polygon.

```
Box(0, 0, 100, 100, #RED)
WaitLeftMouse
SetLayerStyle(1, {Type = #BRUSH, ID = 1})
```

The code above draws a red box on the screen, waits for the left mouse button and then replaces the red rectangle by brush number 1. The layer type is changed from `#BOX` to `#BRUSH`.

34.49 SetLayerTint

NAME

SetLayerTint – set layer tinting (V2.0)

FORMERLY KNOWN AS

SetLayerLight (V1.5)

SYNOPSIS

```
SetLayerTint(id, tintcolor, tintlevel)
```

FUNCTION

This function can be used to tint a layer or layer group with a specified color at a given level. This is useful if you want to lighten the layer or layer group (use #WHITE as tintcolor) or darken it (use #BLACK as tintcolor). Of course you can also use other colors. Level ranges from 0 to 255 where 0 means no tinting (layer default setting) and 255 means full tinting which will make the layer appear fully in the specified color.

Starting with Hollywood 2.0, tintlevel can also be a string containing a percent specification, e.g. `"50%"`.

Starting with Hollywood 5.0, this function will simply install a filter of type `Tint` in the specified layer. See Section 34.43 [SetLayerFilter], page 681, for details.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of the layer or layer group to use

`tintcolor`
RGB color to use for tinting

`tintlevel`
tint level to apply (0 to 255 or percent specification)

EXAMPLE

```
EnableLayers()
DisplayBrush(1, #CENTER, #CENTER)
SetLayerTint(1, #BLACK, 128)
```

The code above darkens layer 1 (= brush 1) with ratio 50% (= 128).

34.50 SetLayerTransparency

NAME

SetLayerTransparency – set transparency of a layer (V1.5)

SYNOPSIS

```
SetLayerTransparency(id, level)
```

FUNCTION

This function can be used to set the transpareny level of a layer or layer group. The transparency level must be between 0 and 255, where 0 means no transparency (layer default setting) and 255 is full transparency which means that you will not see the layer any more (in that case it is of course more efficient to just hide the layer using `HideLayer()`. Please note that this is just the other way round from `SetAlphaIntensity()` where 255 means no transparency and 0 means full transparency.

Starting with Hollywood 2.0, level can also be a string containing a percent specification, e.g. `"50%"`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of the layer or layer group to use

`level` transparency level to apply (0 to 255 or percent specification)

EXAMPLE

```
EnableLayers()

; do not display the text, just add the layer
SelectBGPic(1)
TextOut(#RIGHT, #BOTTOM, "Hello World")
EndSelect

; now it will be displayed!
SetLayerTransparency(1, 128)
```

The code above creates layer 1 (text "Hello World") and makes it appear with a transparency of 50% (= 128).

34.51 SetLayerTransparentPen

NAME

SetLayerTransparentPen – set transparent pen of layer palette (V9.0)

SYNOPSIS

```
SetLayerTransparentPen(id, pen)
```

FUNCTION

This function sets the transparent pen of the palette of the layer specified by `id` to the pen specified in `pen`. Pens are counted from 0.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier of layer to use
`pen`	desired transparent pen (starting from 0)

EXAMPLE

```
SetLayerTransparentPen(1, 4)
```

The code makes pen 4 in the palette of layer 1 transparent.

34.52 SetLayerVolume

NAME

SetLayerVolume – modify volume of a video layer (V6.0)

SYNOPSIS

```
SetLayerVolume(id, volume)
```

FUNCTION

This function modifies the volume of the video layer specified by `id`. If the video layer is currently playing, the volume will be modified on-the-fly which can be used for sound fades etc. The `volume` argument can also be a string containing a percent specification, e.g. `"50%"`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id`	identifier or name of the video layer
`volume`	new volume for the video (range: 0=mute until 64=full volume or percent specification)

34.53 SetLayerZPos

NAME

SetLayerZPos – change the z-position of a layer (V4.6)

SYNOPSIS

```
SetLayerZPos(layer, zpos)
```

FUNCTION

This command can be used to change a layer's z-position. The z-position of the layer is its position in the hierarchy of layers. The first (i.e. backmost) layer has a z-position of 1, the last (i.e. frontmost) layer's z-position is equal to the number of layers currently present. You need to pass the new desired z-position for the specified layer to this function. The layer will then assume exactly this z-position, existing layers that are on or after this z-position will be shifted down. To move a layer all the way to the front (i.e. highest z-position), you can pass the special value 0 for the `zpos` argument. To move a layer all the way to the back, specify 1 in the `zpos` argument.

You can also pass a layer name in the `zpos` argument. In that case, the layer specified in the first argument will assume the z-position of the layer in the second argument.

INPUTS

`layer`	layer whose z position shall be changed
`zpos`	new z position for the layer or 0 to move the layer to the highest z position

34.54 ShowLayer

NAME

ShowLayer – show or move a layer (V1.5)

SYNOPSIS

```
ShowLayer(id[, x, y])
```

FUNCTION

This function shows the hidden layer or layer group specified by `id`. You can hide layers and layer groups by calling `HideLayer()`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

Starting with Hollywood 2.0 there are two optional arguments `x` and `y`. You can use these two arguments to re-position the specified layer. So you can move a layer to a new position using this function. Both arguments default to `#USELAYERPOSITION` which means that the layer will not be moved if you do not specify these arguments.

INPUTS

`id`	identifier of the layer or layer group to show
`x`	optional: new x-position for the layer (defaults to `#USELAYERPOSITION`)
`y`	optional: new y-position for the layer (defaults to `#USELAYERPOSITION`)

EXAMPLE

See Section 34.17 [HideLayer], page 661.

34.55 ShowLayerFX

NAME

ShowLayerFX – display a hidden layer with transition effects (V1.9)

SYNOPSIS

```
[handle] = ShowLayerFX(id[, table])
```

FUNCTION

This function is an extended version of the `ShowLayer()` command. It shows the hidden layer or layer group specified by `id` using one of the many transition effects supported by Hollywood. You can also specify the speed for the transition and an optional argument.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of all supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `ShowLayerFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`NoBorderFade`
If the layer to be shown has a border, do not gradually fade in the border but display it in one go at the end of the transition effect. (V5.0)

`BorderFX:`
If the layer to be shown has a border, Hollywood will only apply the transition effect to the border if the layer is a transparent layer with text or pixel graphics. For non-transparent and vector graphics layers a generic fade effect will be used instead because otherwise there would be visual glitches between the penultimate and final effect frame because of differences in the border algorithms. If you don't care about this glitch and want to force Hollywood to always apply the transition effect to the border, set this tag to `True`. To force Hollywood to always use the generic fade mode, set this tag to `False`. (V9.0)

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier of the layer or layer group to show

`table` optional: table configuring the transition effects

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
ShowLayerFX(5, #CROSSFADE)              ; old syntax

OR

ShowLayerFX(5, {Type = #CROSSFADE})     ; new syntax
```

The above code shows layer 5 with a nice crossfade transition.

34.56 StopLayer

NAME

StopLayer – stop a currently playing video layer (V6.0)

SYNOPSIS

`StopLayer(id)`

FUNCTION

This function stops the video layer specified by `id`. You can restart playback by calling `PlayLayer()`.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`id` identifier or name of the video layer to be stopped

34.57 SwapLayers

NAME

SwapLayers – swap two layers (V1.5)

SYNOPSIS

`SwapLayers(a, b)`

FUNCTION

This function swaps the positions of the layer id's a and b. This can be very useful if you need to re-position your layers.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`a`	identifier or name of layer 1
`b`	identifier or name of layer 2

EXAMPLE

```
EnableLayers()
SetFillStyle(#FILLCOLOR)
Box(0, 0, 150, 150, #RED)
Box(0, 0, 100, 100, #GREEN)
WaitLeftMouse
SwapLayers(1, 2)
```

The code above draws a red and a green filled box and then swaps layers which means that the smaller green box is suddenly not visible any more. Before the swapping, the red box was layer 1 and the green box layer 2. After the swapping, the green box is layer 1 and the red box is layer 2.

34.58 TransformLayer

NAME

TransformLayer – apply affine transformation to a layer (V4.5)

SYNOPSIS

```
TransformLayer(id, sx, rx, ry, sy[, smooth])
```

FUNCTION

This function can be used to apply affine transformation to a layer or layer group. You have to pass a 2x2 transformation matrix to this function that will define how each point in the layer will be transformed. This function is useful if you want to apply rotation and scaling at the same time. The optional argument `smooth` can be set to `True` if Hollywood should use interpolation during the transformation. This yields results that look better but interpolation is quite slow.

If the specified layer is a vector layer (e.g. circle, polygon, TrueType text or a rectangle) Hollywood will be able to transform the layer without any loss in quality because vector graphics can be freely transformed. Thus, the `smooth` argument does not have any function if the specified layer is a vector layer. If the layer uses raster graphics, however, normal raster-based rotation will be used.

In contrast to transforming brushes using `TransformBrush()` layers always keep their original data so there will not be any loss in quality if you transform a layer to (20,15) and then back to (640,480). This is perfectly possible.

See Section 21.78 [TransformBrush], page 314, for more information about how to set up a transformation matrix.

INPUTS

`id`	identifier of the layer or layer group to transform
`sx`	scale x factor; must never be 0
`rx`	rotate x factor

`ry`	rotate y factor
`sy`	scale y factor; must never be 0
`smooth`	optional: whether or not affine transformation should use interpolation

EXAMPLE

```
angle = Rad(45)    ; convert degrees to radians
TransformLayer(1, Cos(angle), Sin(angle), -Sin(angle), Cos(angle))
```

The code above rotates layer number 1 by 45 degrees using a 2x2 transformation matrix.

34.59 TranslateLayer

NAME

TranslateLayer – translate a layer (V4.7)

SYNOPSIS

```
TranslateLayer(id, dx, dy)
```

FUNCTION

This command will translate a layer or layer group by the delta offsets specified by `dx` and `dy`. A translation means that the layer is moved relative to its current position by the specified offset. Thus, a translation of (100,-100) would move the layer 100 pixels towards the right and 100 pixels towards the top. A translation of (0,0) would not move the layer at all. Translations are very useful for moving layers independent of its current position.

Alternatively, you can also use the `TranslateX` and `TranslateY` tags of the powerful `SetLayerStyle()` command.

INPUTS

`id`	identifier of the layer or layer group to translate
`dx`	delta x offset (0 means no x translation)
`dy`	delta y offset (0 means no y translation)

EXAMPLE

```
TranslateLayer(1, -50, -50)
```

The code above moves the first layer 50 pixels in top-left direction.

34.60 Undo

NAME

Undo – undo a graphics operation

SYNOPSIS

```
Undo(type[, id, level, quiet])
```

FUNCTION

This function undoes the graphics operation specified by `type` and optionally `id`. You need to enable layers in order to use this function. Hollywood keeps an internal buffer of

all graphics operations it performs, e.g. displaying brush 2. If you want to remove brush 2 now from the display, just call `Undo(#BRUSH,2)`. The following types are possible:

`#ANIM` Remove anim specified by `id` from the display

`#ARC` Remove arc drawn with `Arc()`

`#BGPICPART`
Remove graphics displayed with `DisplayBGPicPart()`

`#BOX` Remove rectangle drawn with `Box()`

`#BRUSH` Remove brush specified by `id` from the display

`#BRUSHPART`
Remove graphics displayed with `DisplayBrushPart()`

`#CIRCLE` Remove circle drawn with `Circle()`

`#ELLIPSE` Remove ellipse drawn with `Ellipse()`

`#LINE` Remove line drawn with `Line()`

`#MERGED:` Remove merged layer created by `MergeLayers()`.

`#PLOT` Remove a pixel displayed with `Plot()`

`#POLYGON` Remove polygon drawn with `Polygon()`

`#PRINT` Remove text printed with `Print()` or `NPrint()`

`#TEXTOBJECT`
Remove text object specified by `id` from the display

`#TEXTOUT` Undo the last `TextOut()` command; id is not required

`#VECTORPATH`
Undo the last `DrawPath()` command; id is not required

`#VIDEO` Remove video specified by `id` from the display

The optional argument `id` is only required for types which use an identifier (`#ANIM`, `#BGPICPART`, `#BRUSH`, `#BRUSHPART`, `#TEXTOBJECT`, `#VIDEO`). The other types do not require the `id` argument. Please set `id` to 0 for the commands.

The `level` argument specifies the undo level to use. The argument is optional and defaults to 1. Undo level defines on which level the object to undo is. For example, if you display brush 3 four times on the display and now you want to remove the first one of all brushes 3, you will have to specify a level of 4. To remove the last one you have to set undo level to 1, which is also the default. Therefore if `level` is not explicitly specified or set to 1, Hollywood will undo the object last displayed of the specified type.

The `quiet` argument is also optional. If you set it to `True`, Hollywood will only remove the specified object from it is internal object lists but will leave it on the display. If set to `False`, Hollywood will also remove it from the screen.

INPUTS

`type` one of the type constants (see list above)

`id` optional: only required for types which require an associated id (defaults to 0)

`level` optional: undo level (defaults to 1)

`quiet` optional: `True` if object shall only be removed internally but not from the display (defaults to `False`)

EXAMPLE

```
EnableLayers()
DisplayBrush(1, #CENTER, #CENTER)
WaitLeftMouse
Undo(#BRUSH, 1)
```

The above code displays brush 1 in the center of the display, waits for a mouse click and then removes it.

```
EnableLayers()
Print("Hello ")
Print("This ")
Print("Is ")
Print("An ")
Print("Undo ")
Print("Test!")
WaitLeftMouse
Undo(#PRINT, 0, 6)
Undo(#PRINT, 0, 5)
Undo(#PRINT, 0, 4)
```

The above code prints "Hello This Is An Undo Test!" on the display, waits for a mouse click and then removes the texts "Hello", "This" and "Is" by using the optional level argument of the `Undo()` command.

34.61 UndoFX

NAME

UndoFX – undo a graphics operation with transition fx

SYNOPSIS

```
[handle] = UndoFX(type, id[, table])
```

FUNCTION

This function is like the `Undo()` command but it uses a transition effect to undo the operation. See Section 34.60 [Undo], page 708, for details.

Remember to have layers turned on when using this command!

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of all supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `UndoFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using AsyncDrawFrame(). See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

`UndoLevel`
Specifies the undo level for this operation. See Section 34.60 [Undo], page 708, for information on the undo level.

`NoBorderFade`
If the layer to be removed has a border, do not gradually fade out the border but remove it in one go at the end of the transition effect. (V5.0)

`BorderFX:`
If the layer to be removed has a border, Hollywood will only apply the transition effect to the border if the layer is a transparent layer with text or pixel graphics. For non-transparent and vector graphics layers a generic fade effect will be used instead because otherwise there would be visual glitches between the penultimate and final effect frame because of differences in the border algorithms. If you don't care about this glitch and want to force Hollywood to always apply the transition effect to the border, set this tag to `True`. To force Hollywood to always use the generic fade mode, set this tag to `False`. (V9.0)

INPUTS

`type` one of the type constants (See Section 34.60 [Undo], page 708, for details.)

`id` identifier of the object

`table` optional: transition effect configuration

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

34.62 UngroupLayer

NAME

UngroupLayer – remove layer(s) from group (V10.0)

SYNOPSIS

```
UngroupLayer(layer1[, layer2, ...])
```

FUNCTION

This function removes the specified layer(s) from their layer group. The layers you pass to this function don't have to part of the same group. They are ungrouped from whatever group they belong to. You can pass as many layers as you want to this function. Note that as soon as a group doesn't have any more layers attached, it will be automatically deleted.

To add layers to a group, use the `GroupLayer()` function. See Section 34.16 [GroupLayer], page 660, for details.

You need to enable layers before you can use this function. See Section 34.1 [Layers introduction], page 647, for details.

INPUTS

`layer1`	first layer to ungroup
`...`	further layers to ungroup

35 Legacy library

35.1 ACTIVEWINDOW

NAME
ACTIVEWINDOW – window got active / OBSOLETE

SYNOPSIS
`Label(ACTIVEWINDOW)`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `InstallEventHandler()` instead of it.

Hollywood will `Gosub()` to this label whenever the window becomes active again. You normally do not need this event but if you want to do something when your window becomes active again, use this event.

INPUTS
none

35.2 BreakWhileMouseOn

NAME
BreakWhileMouseOn – break next WhileMouseOn() command (V1.9) / OBSOLETE

SYNOPSIS
`BreakWhileMouseOn()`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function breaks the next `WhileMouseOn()` command that will be executed. You usually use this function if your ONBUTTONCLICK label shall return immediately to your event loop instead of the `WhileMouseOn()` command in your ONBUTTONOVER label.

Normally, if you have an ONBUTTONCLICK and an ONBUTTONOVER label, Hollywood will return to your ONBUTTONOVER label after the mouse button was released because the mouse is still over your button after the user clicked the mouse. If you want Hollywood to return to your main event loop after a mouse click, use this command.

Note: This command is only required in rare cases.

INPUTS
none

EXAMPLE
```
While(quit = FALSE)
   WaitEvent
```

```
Wend

Label(ONBUTTONOVER1)
Print("Mouse over button 1")
WhileMouseOn
Print("Mouse no longer over button 1")
Return

Label(ONBUTTONCLICK1)
Print("Mouse click on button 1")
WhileMouseDown
Print("Mouse button released")
BreakWhileMouseOn
Return
```

Have a look at the above code. The `BreakWhileMouseOn()` causes Hollywood to return immediately to the `WaitEvent()` loop. If there was no `BreakWhileMouseOn()` in the code above, Hollywood would return to the `WhileMouseOn()` command and wait until the user moves the mouse out of the button area.

35.3 ClearEvents

NAME

ClearEvents – clear user events (V1.5) / OBSOLETE

SYNOPSIS

```
ClearEvents()
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function clears all events.

INPUTS

none

EXAMPLE

```
CreateButton(1,10,10,110,110)
CreateButton(2,130,130,230,230)
CreateKeyDown(1,"ESC")
ClearEvents
End

Label(SIZEWINDOW)
Return

Label(ONJOYFIRE1)
Return
```

```
Label(CLOSEWINDOW)
Return
```

The above code defines some events and calls `ClearEvents()`. This function will now clear the following events: Button 1, Button 2 and Keydown 1. The events SIZEWINDOW, ONJOYFIRE1 and CLOSEWINDOW will not be cleared because they are only declared as labels. If you want to get rid of them, you can disable them.

35.4 CLOSEWINDOW

NAME

CLOSEWINDOW – user clicked the window's close box / OBSOLETE

SYNOPSIS

```
Label(CLOSEWINDOW)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `InstallEventHandler()` instead of it.

Hollywood will `Gosub()` to this label when the user clicks the close box of the window. This is useful if you want to pop up a requester and ask if he really wants to quit, for example.

INPUTS

none

35.5 CreateButton

NAME

CreateButton – create a new button / OBSOLETE

SYNOPSIS

```
CreateButton(id,x1,y1,x2,y2)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `MakeButton()` instead.

Use this command to declare a button on your display. The button is defined by the coordinates x1:y1 to x2:y2. If the user moves the mouse over the button, Hollywood will `Gosub()` to the label with the name ONBUTTONOVER and the number you specified for the button. If the user clicks the button, Hollywood will jump to the label with the name ONBUTTONCLICK and the number of your button.

INPUTS

`id`	desired identifier for the button
`x1`	source left edge

y1	source top edge
x2	destination left edge
y2	destination top edge

EXAMPLE

```
CreateButton(1,0,0,200,200)
CreateButton(2,201,0,400,200)
CreateKeyDown(1,"ESC")

While(quit=FALSE)
  WaitEvent
Wend
End

Label(ONBUTTONOVER1)
Print("Mouse over button 1")
WhileMouseOn
Print("Mouse out of button 1")
Return

Label(ONBUTTONCLICK1)
Print("User clicked button 1")
WhileMouseDown
Print("User released left mouse button")
Return

Label(ONBUTTONRIGHTCLICK1)     ; requires Hollywood 1.5
Print("User right-clicked button 1")
WhileRightMouseDown
Print("User released right mouse button")
Return

Label(ONBUTTONOVER2)
Print("Mouse over button 2")
WhileMouseOn
Print("Mouse out of button 2")
Return

Label(ONBUTTONCLICK2)
Print("User clicked button 2")
WhileMouseDown
Print("User released left mouse button")
Return

Label(ONBUTTONRIGHTCLICK2)
Print("User right-clicked button 2")    ; requires Hollywood 1.5
WhileRightMouseDown
```

```
Print("User released right mouse button")
Return

Label(ONKEYDOWN1)
quit=TRUE
Return
```

The above code creates two buttons on the screen and monitors the user activity. If he presses the escape key, this demo will quit. This example shows a good way of handling the user input: It is advised that you use a loop like

```
While(quit=False)
   WaitEvent
Wend
```

to handle the user's input. However you have to make sure that you always return to the `WaitEvent()` in the loop.

35.6 CreateKeyDown

NAME

CreateKeyDown – create a new keydown object / OBSOLETE

SYNOPSIS

`CreateKeyDown(id,key$)`

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

Use this command to declare a key to monitor. If the user presses this key, Hollywood will `Gosub()` to the label called ONKEYDOWN with the number as specified in this command.

`Key$` is a string representing a key on your keyboard. This can be one of the following control keys:

`UP`	Cursor up
`DOWN`	Cursor down
`RIGHT`	Cursor right
`LEFT`	Cursor left
`HELP`	Help key
`DEL`	Delete key
`BACKSPACE`	Backspace key
`TAB`	Tab key
`RETURN`	Return/enter key

`ESC`	Escape
`SPACE`	Space key
`F1 - F10`	Function keys

The other keys can be accessed by just specifying the character of the key in the string, e.g. `"A"`, `"!"` or `"-"`.

The following keys cannot be monitored: Alt keys, command keys and the control key.

INPUTS

`id`	desired identifier for the button
`key$`	key to monitor

EXAMPLE

```
CreateKeydown(1,"ESC")
While(quit=FALSE)
   WaitEvent
Wend
End

Label(ONKEYDOWN1)
quit=TRUE
Return
```

The above code waits for the user to press the escape key. Then it quits. A code structure like above is recommended for your applications.

35.7 DisableEvent

NAME

DisableEvent – disable an event / OBSOLETE

SYNOPSIS

```
DisableEvent(type,id)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `DisableButton()` instead.

This function disables the event specified by type and id. Once an event is disabled, it will not be monitored any longer. However, you can enable it again later by using the `EnableEvent()`.

Do not forget to specify the '#' prefix for all events because you are passing constants!

INPUTS

`type`	event type (e.g. `#ONBUTTONOVER`, `#ONBUTTONCLICK`, `#CLOSEWINDOW`)
`id`	event number to disable

EXAMPLE

```
DisableEvent(#ONBUTTONOVER,1)
```

```
DisableEvent(#ONBUTTONCLICK,1)
DisableEvent(#ONBUTTONRIGHTCLICK,1)
```

The above code completely disables the monitoring of button 1 (every button has three events `#ONBUTTONOVER`, `#ONBUTTONCLICK` and `#ONBUTTONRIGHTCLICK`). Therefore if you want to disable a button completely you will have to call `DisableEvent()` thrice.

35.8 DisableEventHandler

NAME

DisableEventHandler – disable Hollywood's event handler / OBSOLETE

SYNOPSIS

```
DisableEventHandler()
```

FUNCTION

Attention: This function is no longer supported as of Hollywood 1.9. Please use `CheckEvent()` instead.

This function disables the internal event handler of Hollywood. See Section 35.9 [EnableEventHandler], page 719, for more information about the internal event handler.

INPUTS

none

EXAMPLE

See Section 35.9 [EnableEventHandler], page 719.

35.9 EnableEventHandler

NAME

EnableEventHandler – enable Hollywood's event handler / OBSOLETE

SYNOPSIS

```
EnableEventHandler()
```

FUNCTION

Attention: This function is no longer supported as of Hollywood 1.9. Please use `CheckEvent()` instead.

This function enables the internal event handler of Hollywood. This means, that you do not have to call `WaitEvent()` any longer because Hollywood will always check if there are any events that occurred. Using the internal event handler is useful if you want to call some functions when there is no input but also monitor user input, e.g. if you are doing a slide show with some effects you cannot call `WaitEvent()` every second but you still want that the user can quit the show by pressing some button on your screen. Then it would be wise to call `EnableEventHandler()`. Once it is enabled, you can do what you want but all events are still monitored.

Please note: Use this function only when you really need it. It has major disadvantages compared to an input loop together with `WaitEvent()` because you will never

know when an event was raised and from where. If Hollywood's event handler is enabled, events can be raised always. It could even happen, that an event breaks commands that are still busy, e.g. `DisplayTransitionFX()`. It is not a good idea to use `EnableEventHandler()` in your projects because you will lose the control of your application. `EnableEventHandler()` is also very likely to be removed from Hollywood in future versions. So you should stay on the safe side, which means: Use an input loop with `WaitEvent()`.

INPUTS

none

EXAMPLE

```
EnableEventHandler
DisplayBGPic(1)
Wait(200)
DisplayBGPic(2)
...
Label(ONBUTTONCLICK1)
End
```

The above code enables the event handler and then starts a slide show but the user will always be able to press a button although you do not call `WaitEvent()`.

35.10 EnableEvent

NAME

EnableEvent – enable an event / OBSOLETE

SYNOPSIS

```
EnableEvent(type,id)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `EnableButton()` instead.

This function enables the event specified by type and id. Once an event is enabled, it will be monitored by Hollywood. You will only have to call this function if you disabled the event before because by default, all events are enabled.

Do not forget to specify the '#' prefix for all events because you are passing constants!

INPUTS

`type` event type (e.g. `#ONBUTTONOVER`, `#ONBUTTONCLICK`, `#CLOSEWINDOW`)

`id` event number to enable

EXAMPLE

```
EnableEvent(#ONBUTTONOVER,1)
EnableEvent(#ONBUTTONCLICK,1)
EnableEvent(#ONBUTTONRIGHTCLICK,1)
```

The above code completely enables the monitoring of button 1 (every button has three events `#ONBUTTONOVER`, `#ONBUTTONCLICK` and `#ONBUTTONRIGHTCLICK`). Therefore if you want to enable a button completely you will have to call `EnableEvent()` thrice.

35.11 GetEventCode

NAME

GetEventCode – get event specific code (V1.5) / OBSOLETE

SYNOPSIS

```
code = GetEventCode()
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function is used right after Hollywood jumped to an event label. If an event has to tell you additional information, you can get this information through this function.

For example, ONBUTTONCLICKALL returns the identifier of the button that caused Hollywood to jump to the label.

INPUTS

none

RESULTS

`code` event specific code

EXAMPLE

See Section 35.20 [ONBUTTONCLICKALL], page 725.

35.12 Gosub

NAME

Gosub – call a subroutine / OBSOLETE

SYNOPSIS

```
Gosub(label)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x library. You should not use it any longer. Please use functions instead of labels.

This function jumps to the subroutine specified by label. The subroutine can call `Return()` to return to the point from where it was called.

INPUTS

`label` identifier of a label (defined with `Label()`)

EXAMPLE

```
a$="Hello World"
```

```
Gosub(PRINTTEXT)
WaitLeftMouse
End

Label(PRINTTEXT)
Print(a$)
Return
```

The above code prints the text "Hello World" on the screen and waits for the left mouse. Then quits.

35.13 Goto

NAME

Goto – jump to a new location / OBSOLETE

SYNOPSIS

`Goto(label)`

FUNCTION

Attention: This function is part of the Hollywood 1.x library. You should not use it any longer. Please use functions instead of labels.

This function jumps to the location specified by label. Execution will continue there and it is not possible to get back to the point from where the label was called. If you need this, use the `Gosub()` command.

INPUTS

`label` identifier of a label (defined with `Label()`)

EXAMPLE

```
a$="Hello World"
Goto(PRINTTEXT)
WaitLeftMouse         ; this code will never be reached
End                   ; !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Label(PRINTTEXT)
Print(a$)
WaitLeftMouse
End
```

The above code prints the text "Hello World" on the screen and waits for the left mouse. Then quits.

35.14 INACTIVEWINDOW

NAME

INACTIVEWINDOW – window got inactive / OBSOLETE

SYNOPSIS

```
Label(INACTIVEWINDOW)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `InstallEventHandler()` instead of it.

Hollywood will `Gosub()` to this label whenever the window becomes inactive. You normally do not need this event but if you want to do something when your window becomes inactive, use this event.

INPUTS

none

35.15 Label

NAME

Label – declare a new label / OBSOLETE

SYNOPSIS

```
Label(name)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x library. You should not use it any longer. Please use functions instead of labels.

This function declares a new label with the specified name. You can jump to this label with the `Gosub()` and `Goto()` commands then. Please note that name is not a string! You need to specify a variable name that will be used as the reference for this label.

INPUTS

`name` identifier to use

EXAMPLE

See Section 35.12 [Gosub], page 721.

See Section 35.13 [Goto], page 722.

35.16 ModifyButton

NAME

ModifyButton – modify button data / OBSOLETE

SYNOPSIS

```
ModifyButton(id,x1,y1,x2,y2)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function changes the configuration of the button specified by `id`.

INPUTS

`id`	identifier of the button
`x1`	desired new left edge position
`y1`	desired new top edge position
`x2`	desired new destination left edge position
`y2`	desired new destination top edge position

35.17 ModifyKeyDown

NAME

ModifyKeyDown – modify keydown object data / OBSOLETE

SYNOPSIS

```
ModifyKeyDown(id,key$)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function changes the configuration of the keydown specified by `id`.

See Section 35.6 [CreateKeyDown], page 717, for the keys you can specify in key$.

INPUTS

`id`	identifier of the button
`key$`	desired new key to monitor

35.18 MOVEWINDOW

NAME

MOVEWINDOW – user moved the window / OBSOLETE

SYNOPSIS

```
Label(MOVEWINDOW)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `InstallEventHandler()` instead of it.

Hollywood will `Gosub()` to this label whenever the window is moved. This is useful to know when your window is transparent. Those windows will be closed and reopened when they are moved, therefore you may need to redraw somethings after the window was moved.

INPUTS

none

35.19 ONBUTTONCLICK

NAME

ONBUTTONCLICK – user clicked a button / OBSOLETE

SYNOPSIS

```
Label(ONBUTTONCLICKx)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This event is raised when the user clicks on a declared button. Hollywood will `Gosub()` then to the label with the name ONBUTTONCLICKx where x is the identifier of the button. For example, Hollywood will `Gosub()` to the label called ONBUTTONCLICK15 when the user clicks on the button with the number 15.

INPUTS

`x`	button number

EXAMPLE

See Section 35.5 [CreateButton], page 715.

35.20 ONBUTTONCLICKALL

NAME

ONBUTTONCLICKALL – user clicked any button (V1.5) / OBSOLETE

SYNOPSIS

```
Label(ONBUTTONCLICKALL)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

If you specify this event and it is enabled, Hollywood will always jump to this label when it gets an ONBUTTONCLICK event. If you want to know, which button caused the label jump, just call `GetEventCode()`. It will return which button caused the event.

This event is very useful if you have e.g. 100 or more buttons and every button just does a `Gosub()` to a sub-routine which receives an id which button was pressed. Instead of defining 100 labels now, you can just use ONBUTTONCLICKALL together with `GetEventCode()`.

For example, if you have 3 buttons with id's from 1 to 3, Hollywood would normally jump to ONBUTTONCLICK1, ONBUTTONCLICK2 or ONBUTTONCLICK3 if one of those buttons was clicked. But if you define an ONBUTTONCLICKALL event, Hollywood will always jump to this label.

INPUTS

none

EXAMPLE

```
CreateButton(1,10,10,100,100)
CreateButton(2,130,130,230,230)
CreateButton(3,260,260,360,360)
CreateKeyDown(1,"F1")
CreateKeyDown(2,"F2")
CreateKeyDown(3,"F3")
While(quit = FALSE)
   WaitEvent
Wend

Label(ONBUTTONCLICKALL)
WhileMouseDown
id = GetEventCode()
Print("User left-clicked button # ")
Print(id)
Return

Label(ONBUTTONRIGHTCLICKALL)
WhileRightMouseDown
id = GetEventCode()
Print("User right-clicked button # ")
Print(id)
Return

Label(ONBUTTONOVERALL)
WhileMouseOn
id = GetEventCode()
Print("User moved mouse over button # ")
Print(id)
Return

Label(ONKEYDOWNALL)
WhileKeyDown
id = GetEventCode()
Print("User invoked keydown # ")
Print(id)
Return
```

The above code uses the ALL special events together with GetEventCode() to find out button events that occurred.

35.21 ONBUTTONOVER

NAME

ONBUTTONOVER – user moved the mouse over a button / OBSOLETE

SYNOPSIS

```
Label(ONBUTTONOVERx)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This event is raised when the user moves the mouse over a declared button. Hollywood will `Gosub()` then to the label with the name ONBUTTONOVERx where x is the identifier of the button. For example, Hollywood will `Gosub()` to the label called ONBUTTONOVER15 when the user moves the mouse over the button with the number 15.

INPUTS

`x`	button number

EXAMPLE

See Section 35.5 [CreateButton], page 715.

35.22 ONBUTTONOVERALL

NAME

ONBUTTONOVERALL – user moved mouse over any button (V1.5) / OBSOLETE

SYNOPSIS

```
Label(ONBUTTONOVERALL)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

If you specify this event and it is enabled, Hollywood will always jump to this label when it gets an ONBUTTONOVER event. If you want to know, which button caused the label jump, just call `GetEventCode()`. It will return which button caused the event.

This event is very useful if you have e.g. 100 or more buttons and every button just does a `Gosub()` to a sub-routine which receives an id which button was pressed. Instead of defining 100 labels now, you can just use ONBUTTONOVERALL together with `GetEventCode()`.

INPUTS

none

EXAMPLE

See Section 35.20 [ONBUTTONCLICKALL], page 725.

35.23 ONBUTTONRIGHTCLICK

NAME

ONBUTTONRIGHTCLICK – user right-clicked a button (V1.5) / OBSOLETE

SYNOPSIS

```
Label(ONBUTTONRIGHTCLICKx)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This event is raised when the user right-clicks on a declared button. Hollywood will `Gosub()` then to the label with the name ONBUTTONRIGHTCLICKx where x is the identifier of the button. For example, Hollywood will `Gosub()` to the label called ONBUTTONRIGHTCLICK15 when the user clicks on the button with the number 15.

INPUTS

`x`	button number

EXAMPLE

See Section 35.5 [CreateButton], page 715.

35.24 ONBUTTONRIGHTCLICKALL

NAME

ONBUTTONRIGHTCLICKALL – user right-clicked any button (V1.5) / OBSOLETE

SYNOPSIS

```
Label(ONBUTTONRIGHTCLICKALL)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

If you specify this event and it is enabled, Hollywood will always jump to this label when it gets an ONBUTTONRIGHTCLICK event. If you want to know, which button caused the label jump, just call `GetEventCode()`. It will return which button caused the event.

This event is very useful if you have e.g. 100 or more buttons and every button just does a `Gosub()` to a sub-routine which receives an id which button was pressed. Instead of defining 100 labels now, you can just use ONBUTTONRIGHTCLICKALL together with `GetEventCode()`.

INPUTS

none

EXAMPLE

See Section 35.20 [ONBUTTONCLICKALL], page 725.

35.25 ONJOYDOWN

NAME
ONJOYDOWN – user moved Joystick down (V1.5)

SYNOPSIS
`Label(ONJOYDOWNx)`

FUNCTION
Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x down.

INPUTS

`x`	port number (usually 1 for the standard Joystick port)

EXAMPLE
See Section 35.31 [ONJOYUP], page 731.

35.26 ONJOYDOWNLEFT

NAME
ONJOYDOWNLEFT – user moved Joystick down left (V1.5)

SYNOPSIS
`Label(ONJOYDOWNLEFTx)`

FUNCTION
Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x down left.

INPUTS

`x`	port number (usually 1 for the standard Joystick port)

EXAMPLE
See Section 35.31 [ONJOYUP], page 731.

35.27 ONJOYDOWNRIGHT

NAME
ONJOYDOWNRIGHT – user moved Joystick down right (V1.5)

SYNOPSIS
`Label(ONJOYDOWNRIGHTx)`

FUNCTION
Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x down right.

INPUTS

`x`	port number (usually 1 for the standard Joystick port)

EXAMPLE
See Section 35.31 [ONJOYUP], page 731.

35.28 ONJOYFIRE

NAME
ONJOYFIRE – user pressed Joystick button (V1.5)

SYNOPSIS
`Label(ONJOYFIREx)`

FUNCTION
Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user presses the fire button of the Joystick that is plugged in port x.

INPUTS

`x`	port number (usually 1 for the standard Joystick port)

EXAMPLE
See Section 35.31 [ONJOYUP], page 731.

35.29 ONJOYLEFT

NAME
ONJOYLEFT – user moved Joystick up right (V1.5)

SYNOPSIS
`Label(ONJOYLEFTx)`

FUNCTION
Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x up left.

INPUTS

`x`	port number (usually 1 for the standard Joystick port)

EXAMPLE
See Section 35.31 [ONJOYUP], page 731.

35.30 ONJOYRIGHT

NAME
ONJOYRIGHT – user moved Joystick right (V1.5)

SYNOPSIS
`Label(ONJOYRIGHTx)`

FUNCTION

Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x right.

INPUTS

x	port number (usually 1 for the standard Joystick port)

EXAMPLE

See Section 35.31 [ONJOYUP], page 731.

35.31 ONJOYUP

NAME

ONJOYUP – user moved Joystick up (V1.5)

SYNOPSIS

```
Label(ONJOYUPx)
```

FUNCTION

Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x up.

INPUTS

x	port number (usually 1 for the standard Joystick port)

EXAMPLE

```
While(quit = FALSE)
WaitEvent
Wend

Label(ONJOYUP1)
NPrint("Joy up")
Return

Label(ONJOYUPRIGHT1)
NPrint("Joy up right")
Return

Label(ONJOYRIGHT1)
NPrint("Joy right")
Return

Label(ONJOYDOWNRIGHT1)
NPrint("Joy down right")
Return

Label(ONJOYDOWN1)
NPrint("Joy down")
```

```
Return

Label(ONJOYDOWNLEFT1)
NPrint("Joy down left")
Return

Label(ONJOYLEFT1)
NPrint("Joy left")
Return

Label(ONJOYUPLEFT1)
NPrint("Joy up left")
Return

Label(ONJOYFIRE1)
NPrint("Joy fire")
Return
```

The above code shows how to query input from the Joystick.

35.32 ONJOYUPLEFT

NAME

ONJOYUPLEFT – user moved Joystick up right (V1.5)

SYNOPSIS

```
Label(ONJOYUPLEFTx)
```

FUNCTION

Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x up left.

INPUTS

x	port number (usually 1 for the standard Joystick port)

EXAMPLE

See Section 35.31 [ONJOYUP], page 731.

35.33 ONJOYUPRIGHT

NAME

ONJOYUPRIGHT – user moved Joystick up right (V1.5)

SYNOPSIS

```
Label(ONJOYUPRIGHTx)
```

FUNCTION

Attention: This label is no longer supported in Hollywood 2.0.

This event is raised when the user moves the Joystick plugged in port x up right.

INPUTS

`x` port number (usually 1 for the standard Joystick port)

EXAMPLE

See Section 35.31 [ONJOYUP], page 731.

35.34 ONKEYDOWN

NAME

ONKEYDOWN – user pressed a key / OBSOLETE

SYNOPSIS

```
Label(ONKEYDOWNx)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This event is raised when the user presses a monitored key. Hollywood will `Gosub()` then to the label with the name ONKEYDOWNx where x is the identifier of the keydown. For example, Hollywood will Gosub() to the label called ONKEYDOWN15 when the user presses the key with the number 15.

INPUTS

`x` key number

EXAMPLE

See Section 35.6 [CreateKeyDown], page 717.

35.35 ONKEYDOWNALL

NAME

ONKEYDOWNALL – user invoked any keydown (V1.5) / OBSOLETE

SYNOPSIS

```
Label(ONKEYDOWNALL)
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

If you specify this event and it is enabled, Hollywood will always jump to this label when it gets an ONKEYDOWN event. If you want to know, which keydown caused the label jump, just call `GetEventCode()`. It will return which keydown caused the event.

This event is very useful if you have e.g. 100 or more keydown's and every keydown just does a `Gosub()` to a sub-routine which receives an id which key was pressed. Instead of defining 100 labels now, you can just use ONKEYDOWNALL together with `GetEventCode()`.

INPUTS
none

EXAMPLE
See Section 35.20 [ONBUTTONCLICKALL], page 725.

35.36 RemoveButton

NAME
RemoveButton – remove a button / OBSOLETE

SYNOPSIS
`RemoveButton(id)`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `DeleteButton()` instead.

This function will remove the button specified by `id` from Hollywood's button database. Events of this button will no longer be received.

INPUTS

`id`	button to delete

EXAMPLE
```
RemoveButton(1)
```
Deletes button 1.

35.37 RemoveKeyDown

NAME
RemoveKeyDown – remove a keydown object / OBSOLETE

SYNOPSIS
`RemoveKeyDown(id)`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function will remove the keydown object specified by `id` from Hollywood's keydown database. Events of this keydown will no longer be received.

INPUTS

`id`	keydown to delete

EXAMPLE
```
RemoveKeydown(1)
```
Deletes keydown object 1.

35.38 Return

NAME
Return – return to last Gosub() / OBSOLETE

SYNOPSIS
`Return()`

FUNCTION
Attention: This function is part of the Hollywood 1.x library. You should not use it any longer. Please use functions instead of labels.

This function will return to the point where the last routine was gosub'ed from.

INPUTS
none

EXAMPLE
See Section 35.12 [Gosub], page 721.

35.39 SIZEWINDOW

NAME
SIZEWINDOW – user changed the window size / OBSOLETE

SYNOPSIS
`Label(SIZEWINDOW)`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use `InstallEventHandler()` instead of it.

Hollywood will `Gosub()` to this label when the user changes the window size. If your application shall support resizable windows, you must place some code after this label which redraws/repositions your objects.

INPUTS
none

35.40 WhileKeyDown

NAME
WhileKeyDown – halt until key is up (V1.5) / OBSOLETE

SYNOPSIS
`WhileKeyDown()`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function can be used after an ONKEYDOWN event occurred. If you call this function, Hollywood will wait until the user releases the key which caused the event.

If you do not call this function after your Label(ONKEYDOWNx) you may receive several events if the user holds down the key a bit longer.

INPUTS

none

35.41 WhileMouseDown

NAME

WhileMouseDown – halt until the mouse button is up / OBSOLETE

SYNOPSIS

```
WhileMouseDown()
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function must be called after a ONBUTTONCLICK event occurred. If you call this function, Hollywood will wait until the user has finished his mouse button click. It is absolutely necessary to call this function right after the ONBUTTONCLICK event occurred because even if the user clicks the mouse only once, several events may be generated. To prevent this, call this function right after the ONBUTTONCLICK occurred.

INPUTS

none

EXAMPLE

See Section 35.5 [CreateButton], page 715.

35.42 WhileMouseOn

NAME

WhileMouseOn – halt until the user moves mouse out of a button / OBSOLETE

SYNOPSIS

```
WhileMouseOn()
```

FUNCTION

Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function can be called after a ONBUTTONOVER event occurred. If you call this function, Hollywood will wait until the user moves the mouse out of the button it is currently over. This is useful when you want to display a brush when the mouse is over a button (hover effect).

This function is optional after an ONBUTTONOVER event (other than the `WhileMouseDown()` function which is required after a ONBUTTONCLICK event!)

INPUTS
none

EXAMPLE
See Section 35.5 [CreateButton], page 715.

35.43 WhileRightMouseDown

NAME
WhileRightMouseDown – halt until the right mouse button is up (V1.5) / OBSOLETE

SYNOPSIS
`WhileRightMouseDown()`

FUNCTION
Attention: This function is part of the Hollywood 1.x event library. You should not use it any longer. Please use the functions of the new library from Hollywood 2.0 which is much better.

This function must be called after a ONBUTTONRIGHTCLICK event occurred. If you call this function, Hollywood will wait until the user has finished his mouse button click. It is absolutely necessary to call this function right after the ONBUTTONRIGHTCLICK event occurred because even if the user clicks the mouse only once, several events may be generated. To prevent this, call this function right after the ONBUTTONRIGHTCLICK occurred.

INPUTS
none

EXAMPLE
See Section 35.5 [CreateButton], page 715.

36 Locale library

36.1 Overview

Hollywood allows you to localize your programs by importing text strings from so-called catalog files which you open using the `OpenCatalog()` function or the `@CATALOG` preprocessor command. For compatibility reasons, these catalogs can be in the `IFF CTLG` format established by Commodore in the early 1990s but this isn't recommended because `IFF CTLG` has no proper Unicode support and will only work correctly if the system's locale matches the catalog's locale.

Thus, it is recommended to use Hollywood's own catalog format instead. This alternative catalog format is simply a plain text file that contains a list of catalog strings, one per line, in the UTF-8 character encoding. If newline characters occur in a catalog string, you must write them as `"\n"`. Backslashes must be written as `"\\"`. If you need to split a string across multiple lines, append a single backslash at the end of the respective line. Such a single backslash at the end of the line signals Hollywood that the string continues in the next line. If there is no backslash at the end of the line, Hollywood knows that this is the end of the string. Empty lines are ignored completely and can be included for readability reasons.

The text file must be in the UTF-8 character encoding, with or without BOM. Comments can be included by using a semicolon as the very first character in the line. This makes Hollywood ignore the rest of the line. Semicolons at other positions in the line don't have any significance for the parser. If the first character of your string needs to be a semicolon, you need to prefix it with a backslash to tell the parser that this is not a comment.

Here is a simple example of a catalog definition in Hollywood's own catalog format:

```
; this is a comment
This is the first string!
;
This is the second string!
;
This is the third string\nand it has two lines!
;
This is the fourth string \
and it has only one line \
but it is split across four lines \
in the catalog file!
;
This is the fifth and last string!
```

To get a string from the catalog file, just call `GetCatalogString()` and pass the index of the string you want to retrieve. For example, to get the fifth string from the catalog file shown above, you'd have to do the following:

```
Print(GetCatalogString(4, "default string"))
```

The string passed in the second argument to `GetCatalogString()` is the default string and it will be returned only if the requested string index could not be found in the catalog file.

36.2 CATALOG

NAME

CATALOG – preload a catalog for later use (V9.0)

SYNOPSIS

```
@CATALOG name$[, table]
```

FUNCTION

This preprocessor command can be used to preload the specified catalog in the user's language. If a catalog in the user's language does not exist, this preprocessor command will not show an error. This is normal behaviour because you always have to provide default English strings for every entry you try to get from a catalog using the `GetCatalogString()` function. Thus, it's not a problem if `@CATALOG` specifies a catalog that doesn't exist. In that case, `GetCatalogString()` will simply fall back to the default English strings without throwing an error.

Note that there currently can only be a single catalog per application. Thus, `@CATALOG` should only be used once per script. Also note that `name$` must not be a filename but the name of a catalog that is stored within a Hollywood catalog directory structure. See Section 36.11 [OpenCatalog], page 755, for details on how such a Hollywood catalog structure is organized.

The advantage of using `@CATALOG` instead of `OpenCatalog()` is that if you use `@CATALOG`, the catalogs for all available languages will be linked into your executable or applet when you compile your script. This makes it easier to distribute your project because you don't have to include the catalog directory structure with your application. Also, when compiling Hollywood applets for mobile systems like Android or iOS, it's much better to have all external files linked into the applet.

In addition to the `name$` parameter, `@CATALOG` also accepts an optional table argument. The following fields in the table argument are currently available:

`Link:` Set this field to `False` if you do not want to have this catalog linked to your executable/applet when you compile your script. This field defaults to `True` which means that the catalog will be linked to your executable/applet when Hollywood is in compile mode.

To load a catalog at runtime, use the `OpenCatalog()` function. See Section 36.11 [OpenCatalog], page 755, for details.

INPUTS

`name$` name of the catalog to open

`table` optional: table containing further options

EXAMPLE

```
@CATALOG "Hollywood.catalog"

; this is our default English catalog
def$ = {}
def$[0] = "Welcome to Hollywood!"
def$[1] = "Written by Andreas Falkenhahn"
```

```
def$[2] = "What do you wanna do?"

; if Hollywood.catalog is not available in the
; user's language; the English strings will be
; used
For k = 0 To 2
   c$[k] = GetCatalogString(k, def$[k])
Next
```

The code above opens "Hollywood.catalog" and prints the first three entries from that catalog.

36.3 CloseCatalog

NAME
CloseCatalog – close an open catalog

SYNOPSIS
`CloseCatalog()`

FUNCTION
This function will close the currently opened catalog. You do not really have to call this function, because Hollywood will close the catalog by itself when it terminates.

INPUTS
none

EXAMPLE
See Section 36.11 [OpenCatalog], page 755.

36.4 FormatDate

NAME
FormatDate – format date template (V10.0)

SYNOPSIS
`d$ = FormatDate(fmt$, date$[, isdst])`

FUNCTION
This function formats the date passed in `date$` according to the format template passed in `fmt$` and returns the result. This allows you to convert date and time format templates returned by `GetLocaleInfo()` to human-readable dates. The date string passed in `date$` must be in the default date and time notation used by Hollywood:

```
dd-mmm-yyyy hh:mm:ss
```

The `dd` part is a two digit date specification (with leading zeros) and the `mmm` constituent is a string with three characters identifying the month. This can be `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`. The `yyyy` is a four digit year specification whereas `hh` specifies the hours, `mm` the minutes and `ss` the seconds.

The date template passed in `date$` can contain string literals as well the following tokens:

`%a`	Abbreviated weekday name
`%A`	Weekday name
`%b`	Abbreviated month name
`%B`	Month name
`%d`	Day number with leading zeros
`%-d`	Day number without leading zeros
`%H`	Hour using 24-hour style with leading zeros
`%-H`	Hour using 24-hour style without leading zeros
`%I`	Hour using 12-hour style with leading zeros
`%-I`	Hour using 12-hour style without leading zeros
`%m`	Month number with leading zeros
`%-m`	Month number without leading zeros
`%M`	Number of minutes with leading zeros
`%-M`	Number of minutes without leading zeros
`%S`	Number of seconds with leading zeros
`%-S`	Number of seconds without leading zeros
`%y`	Year using two digits with leading zeros
`%-y`	Year using two digits without leading zeros
`%Y`	Year using four digits with leading zeros

Note that depending on the platform Hollywood is running on some more tokens might be supported but only the ones listed above are guaranteed to work on all platforms.

Only the tokens listed above will be replaced in `fmt$`. All other characters won't be replaced and will remain as string literals in the returned string. If you want to have a percent character as a string literal in the date template string, you need to escape it by using two percent characters (%%).

The optional argument `isdst` specifies whether or not daylight saving time is active at the specified date. Normally, you don't have to specify this argument because Hollywood will automatically query this information from the timezone database. It is only necessary to pass this information in case the specified time is ambiguous, i.e. when switching from daylight saving time back to standard time, a certain period of time (typically an hour) is repeated in the night. In Germany, for example, clocks are set back from 3am to 2am when switching from daylight saving time to standard time. This means that the hour between 2am and 3am happens twice: Once in daylight saving time, once in standard time. The `isdst` argument allows you to specify which hour you are referring to.

INPUTS

`fmt$`	date template string

`date$` date string in the Hollywood date notation

`isdst` optional: whether or not daylight saving time is active at the specified date (defaults to -1 which means that this information should be retrieved from the local timezone database)

RESULTS

`d$` formatted date

EXAMPLE

```
d$ = FormatDate(GetLocaleInfo().DateTimeFormat, GetDate(#DATELOCAL))
DebugPrint(d$)
```

The code above prints the date and time formatted according to the rules of the current locale.

36.5 GetCatalogString

NAME

GetCatalogString – get a string from a catalog

SYNOPSIS

```
s$ = GetCatalogString(id, default$)
```

FUNCTION

This function extracts the string with the number `id` from the currently opened catalog file. If there is no open catalog, the string specified in `default$` is returned.

INPUTS

`id` specifies which string to return (starting from 0)

`default$` string to return if there is no open catalog or if there is no string with the specified id

EXAMPLE

See Section 36.11 [OpenCatalog], page 755.

36.6 GetCountryInfo

NAME

GetCountryInfo – get information about country (V7.1)

SYNOPSIS

```
t = GetCountryInfo(ctry)
```

FUNCTION

This function can be used to retrieve additional information about a country. You have to pass one of Hollywood's country constants in the `ctry` argument. See Section 36.9 [GetSystemCountry], page 746, for a list of countries.

`GetCountryInfo()` will then return a table with the following fields initialized:

`Alpha2:` Alpha-2 country code as defined by ISO 3166-1.

`Alpha3:` Alpha-3 country code as defined by ISO 3166-1.

INPUTS

`ctry` one of the country constants defined by Hollywood (see above)

RESULTS

`t` table containing country information

EXAMPLE

```
t = GetCountryInfo(GetSystemCountry())
Print(t.Alpha2, t.Alpha3)
```

On a German system, this will print "DE" and "DEU".

36.7 GetLanguageInfo

NAME

GetLanguageInfo – get information about language (V7.1)

SYNOPSIS

```
t = GetLanguageInfo(lang)
```

FUNCTION

This function can be used to retrieve additional information about a language. You have to pass one of Hollywood's language constants in the `lang` argument. See Section 36.10 [GetSystemLanguage], page 751, for a list of languages.

`GetLanguageInfo()` will then return a table with the following fields initialized:

`Code:` Two letter language code as defined by ISO 639.

`Name:` The language's ISO name.

INPUTS

`lang` one of the language constants defined by Hollywood (see above)

RESULTS

`t` table containing language information

EXAMPLE

```
t = GetLanguageInfo(GetSystemLanguage())
Print(t.Code, t.Name)
```

On a German system, this will print "DE" and "german".

36.8 GetLocaleInfo

NAME

GetLocaleInfo – get information about locale (V10.0)

SYNOPSIS

```
t = GetLocaleInfo()
```

FUNCTION

This function can be used to retrieve information about the currently active locale. `GetLocaleInfo()` will then return a table with the following fields initialized:

`DecimalPoint:`
: Contains the locale's decimal point, e.g. `"."` on an English and `","` on a German system.

`ThousandSeparator:`
: Contains the locale's thousand separator, e.g. `","` on an English and `"."` on a German system.

`Currency:`
: Contains the locale's default currency symbol, e.g. `"€"` on a German system.

`DateFormat:`
: Contains the long date format template for the current locale. See Section 36.4 [FormatDate], page 741, for a description of the individual template constituents.

`ShortDateFormat:`
: Contains the short date format template for the current locale. See Section 36.4 [FormatDate], page 741, for a description of the individual template constituents.

`TimeFormat:`
: Contains the time format template for the current locale. See Section 36.4 [FormatDate], page 741, for a description of the individual template constituents.

`DateTimeFormat:`
: Contains the combined date and time format template for the current locale. See Section 36.4 [FormatDate], page 741, for a description of the individual template constituents.

`Days:` Contains an array of the weekday names in the current locale, e.g. `"Monday"`, `"Tuesday"`, etc.

`AbDays:` Contains an array of the abbreviated weekday names in the current locale, e.g. `"Mon"`, `"Tue"`, etc.

`Months:` Contains an array of the month names in the current locale, e.g. `"January"`, `"February"`, etc.

`AbMonths:`
: Contains an array of the abbreviated month names in the current locale, e.g. `"Jan"`, `"Feb"`, etc.

`Language:`
: The language name in the locale's language, e.g. `"deutsch"` on a German system. Note that the exact string you get here depends on the host OS so this is not really portable.

INPUTS

none

RESULTS

`t` table containing locale information

EXAMPLE

```
d$ = FormatDate(GetLocaleInfo().DateTimeFormat, GetDate(#DATELOCAL))
DebugPrint(d$)
```

The code above prints the date and time formatted according to the rules of the current locale.

36.9 GetSystemCountry

NAME

GetSystemCountry – retrieve current user's country (V5.0)

SYNOPSIS

```
ctry = GetSystemCountry()
```

FUNCTION

This function can be used to retrieve the country setting of the current system. The following countries are currently supported:

```
#COUNTRY_UK (0)
#COUNTRY_USA (1)
#COUNTRY_AUSTRALIA (2)
#COUNTRY_BELGIUM (3)
#COUNTRY_BULGARIA (4)
#COUNTRY_BRAZIL (5)
#COUNTRY_CANADA (6)
#COUNTRY_CZECHREPUBLIC (7)
#COUNTRY_DENMARK (8)
#COUNTRY_GERMANY (9)
#COUNTRY_SPAIN (10)
#COUNTRY_FRANCE (11)
#COUNTRY_GREECE (12)
#COUNTRY_ITALY (13)
#COUNTRY_LIECHTENSTEIN (14)
#COUNTRY_LITHUANIA (15)
#COUNTRY_LUXEMBOURG (16)
#COUNTRY_HUNGARY (17)
#COUNTRY_MALTA (18)
#COUNTRY_MONACO (19)
#COUNTRY_NETHERLANDS (20)
#COUNTRY_NORWAY (21)
#COUNTRY_POLAND (22)
#COUNTRY_PORTUGAL (23)
#COUNTRY_ROMANIA (24)
#COUNTRY_RUSSIA (25)
#COUNTRY_SANMARINO (26)
```

```
#COUNTRY_SLOVAKIA (27)
#COUNTRY_SLOVENIA (28)
#COUNTRY_SWITZERLAND (29)
#COUNTRY_FINLAND (30)
#COUNTRY_SWEDEN (31)
#COUNTRY_TURKEY (32)
#COUNTRY_IRELAND (33)
#COUNTRY_AUSTRIA (34)
#COUNTRY_ICELAND (35)
#COUNTRY_ANDORRA (36)
#COUNTRY_UKRAINE (37)
#COUNTRY_UNKNOWN (38)
#COUNTRY_AFGHANISTAN (39)
#COUNTRY_ALANDISLANDS (40)
#COUNTRY_ALBANIA (41)
#COUNTRY_ALGERIA (42)
#COUNTRY_AMERICANSAMOA (43)
#COUNTRY_ANGOLA (44)
#COUNTRY_ANGUILLA (45)
#COUNTRY_ANTARCTICA (46)
#COUNTRY_ANTIGUAANDBARBUDA (47)
#COUNTRY_ARGENTINA (48)
#COUNTRY_ARMENIA (49)
#COUNTRY_ARUBA (50)
#COUNTRY_AZERBAIJAN (51)
#COUNTRY_BAHAMAS (52)
#COUNTRY_BAHRAIN (53)
#COUNTRY_BANGLADESH (54)
#COUNTRY_BARBADOS (55)
#COUNTRY_BELARUS (56)
#COUNTRY_BELIZE (57)
#COUNTRY_BENIN (58)
#COUNTRY_BERMUDA (59)
#COUNTRY_BHUTAN (60)
#COUNTRY_BOLIVIA (61)
#COUNTRY_BESISLANDS (62)
#COUNTRY_BOSNIAANDHERZEGOVINA (63)
#COUNTRY_BOTSWANA (64)
#COUNTRY_BOUVETISLAND (65)
#COUNTRY_BRUNEI (66)
#COUNTRY_BURKINAFASO (67)
#COUNTRY_BURUNDI (68)
#COUNTRY_CAMBODIA (69)
#COUNTRY_CAMEROON (70)
#COUNTRY_CAPEVERDE (71)
#COUNTRY_CAYMANISLANDS (72)
#COUNTRY_CENTRALAFRICANREPUBLIC (73)
```

```
#COUNTRY_CHAD (74)
#COUNTRY_CHILE (75)
#COUNTRY_CHINA (76)
#COUNTRY_CHRISTMASISLAND (77)
#COUNTRY_COCOSISLANDS (78)
#COUNTRY_COLOMBIA (79)
#COUNTRY_COMOROS (80)
#COUNTRY_CONGO (81)
#COUNTRY_COOKISLANDS (82)
#COUNTRY_COSTARICA (83)
#COUNTRY_IVORYCOAST (84)
#COUNTRY_CROATIA (85)
#COUNTRY_CUBA (86)
#COUNTRY_CURACAO (87)
#COUNTRY_CYPRUS (88)
#COUNTRY_DJIBOUTI (89)
#COUNTRY_DOMINICA (90)
#COUNTRY_DOMINICANREPUBLIC (91)
#COUNTRY_DRCONGO (92)
#COUNTRY_ECUADOR (93)
#COUNTRY_EGYPT (94)
#COUNTRY_ELSALVADOR (95)
#COUNTRY_EQUATORIALGUINEA (96)
#COUNTRY_ERITREA (97)
#COUNTRY_ESTONIA (98)
#COUNTRY_ETHIOPIA (99)
#COUNTRY_FALKLANDISLANDS (100)
#COUNTRY_FAROEISLANDS (101)
#COUNTRY_FIJI (102)
#COUNTRY_FRENCHGUIANA (103)
#COUNTRY_FRENCHPOLYNESIA (104)
#COUNTRY_GABON (105)
#COUNTRY_GAMBIA (106)
#COUNTRY_GEORGIA (107)
#COUNTRY_GHANA (108)
#COUNTRY_GIBRALTAR (109)
#COUNTRY_GREENLAND (110)
#COUNTRY_GRENADA (111)
#COUNTRY_GUADELOUPE (112)
#COUNTRY_GUAM (113)
#COUNTRY_GUATEMALA (114)
#COUNTRY_GUERNSEY (115)
#COUNTRY_GUINEA (116)
#COUNTRY_GUINEABISSAU (117)
#COUNTRY_GUYANA (118)
#COUNTRY_HAITI (119)
#COUNTRY_HOLYSEE (120)
```

```
#COUNTRY_HONDURAS (121)
#COUNTRY_HONGKONG (122)
#COUNTRY_INDIA (123)
#COUNTRY_INDONESIA (124)
#COUNTRY_IRAN (125)
#COUNTRY_IRAQ (126)
#COUNTRY_ISLEOFMAN (127)
#COUNTRY_ISRAEL (128)
#COUNTRY_JAMAICA (129)
#COUNTRY_JAPAN (130)
#COUNTRY_JERSEY (131)
#COUNTRY_JORDAN (132)
#COUNTRY_KAZAKHSTAN (133)
#COUNTRY_KENYA (134)
#COUNTRY_KIRIBATI (135)
#COUNTRY_NORTHKOREA (136)
#COUNTRY_SOUTHKOREA (137)
#COUNTRY_KUWAIT (138)
#COUNTRY_KYRGYZSTAN (139)
#COUNTRY_LAOS (140)
#COUNTRY_LATVIA (141)
#COUNTRY_LEBANON (142)
#COUNTRY_LESOTHO (143)
#COUNTRY_LIBERIA (144)
#COUNTRY_LIBYA (145)
#COUNTRY_MACAO (146)
#COUNTRY_MACEDONIA (147)
#COUNTRY_MADAGASCAR (148)
#COUNTRY_MALAWI (149)
#COUNTRY_MALAYSIA (150)
#COUNTRY_MALDIVES (151)
#COUNTRY_MALI (152)
#COUNTRY_MARSHALLISLANDS (153)
#COUNTRY_MARTINIQUE (154)
#COUNTRY_MAURITANIA (155)
#COUNTRY_MAURITIUS (156)
#COUNTRY_MAYOTTE (157)
#COUNTRY_MEXICO (158)
#COUNTRY_MICRONESIA (159)
#COUNTRY_MOLDOVA (160)
#COUNTRY_MONGOLIA (161)
#COUNTRY_MONTENEGRO (162)
#COUNTRY_MONTSERRAT (163)
#COUNTRY_MOROCCO (164)
#COUNTRY_MOZAMBIQUE (165)
#COUNTRY_MYANMAR (166)
#COUNTRY_NAMIBIA (167)
```

```
#COUNTRY_NAURU (168)
#COUNTRY_NEPAL (169)
#COUNTRY_NEWCALEDONIA (170)
#COUNTRY_NEWZEALAND (171)
#COUNTRY_NICARAGUA (172)
#COUNTRY_NIGER (173)
#COUNTRY_NIGERIA (174)
#COUNTRY_NIUE (175)
#COUNTRY_NORFOLKISLAND (176)
#COUNTRY_OMAN (177)
#COUNTRY_PAKISTAN (178)
#COUNTRY_PALAU (179)
#COUNTRY_PALESTINE (180)
#COUNTRY_PANAMA (181)
#COUNTRY_PAPUANEWGUINEA (182)
#COUNTRY_PARAGUAY (183)
#COUNTRY_PERU (184)
#COUNTRY_PHILIPPINES (185)
#COUNTRY_PITCAIRN (186)
#COUNTRY_PUERTORICO (187)
#COUNTRY_QATAR (188)
#COUNTRY_REUNION (189)
#COUNTRY_RWANDA (190)
#COUNTRY_SAINTBARTHELEMY (191)
#COUNTRY_SAINTHELENA (192)
#COUNTRY_SAINTKITTSANDNEVIS (193)
#COUNTRY_SAINTLUCIA (194)
#COUNTRY_SAINTVINCENT (195)
#COUNTRY_SAMOA (196)
#COUNTRY_SAOTOMEANDPRINCIPE (197)
#COUNTRY_SAUDIARABIA (198)
#COUNTRY_SENEGAL (199)
#COUNTRY_SERBIA (200)
#COUNTRY_SEYCHELLES (201)
#COUNTRY_SIERRALEONE (202)
#COUNTRY_SINGAPORE (203)
#COUNTRY_SOLOMONISLANDS (204)
#COUNTRY_SOMALIA (205)
#COUNTRY_SOUTHAFRICA (206)
#COUNTRY_SOUTHSUDAN (207)
#COUNTRY_SRILANKA (208)
#COUNTRY_SUDAN (209)
#COUNTRY_SURINAME (210)
#COUNTRY_SWAZILAND (211)
#COUNTRY_SYRIA (212)
#COUNTRY_TAIWAN (213)
#COUNTRY_TAJIKISTAN (214)
```

```
#COUNTRY_TANZANIA (215)
#COUNTRY_THAILAND (216)
#COUNTRY_TIMOR (217)
#COUNTRY_TOGO (218)
#COUNTRY_TONGA (219)
#COUNTRY_TRINIDADANDTOBAGO (220)
#COUNTRY_TUNISIA (221)
#COUNTRY_TURKMENISTAN (222)
#COUNTRY_TUVALU (223)
#COUNTRY_UGANDA (224)
#COUNTRY_UAE (225)
#COUNTRY_URUGUAY (226)
#COUNTRY_UZBEKISTAN (227)
#COUNTRY_VANUATU (228)
#COUNTRY_VENEZUELA (229)
#COUNTRY_VIETNAM (230)
#COUNTRY_YEMEN (231)
#COUNTRY_ZAMBIA (232)
```

INPUTS

none

RESULTS

`ctry` country setting of current user

36.10 GetSystemLanguage

NAME

GetSystemLanguage – retrieve current user's language (V5.0)

SYNOPSIS

```
lang = GetSystemLanguage()
```

FUNCTION

This function can be used to retrieve the default language of the current user. The following languages are currently supported:

```
#LANGUAGE_ENGLISH (0)
#LANGUAGE_GERMAN (1)
#LANGUAGE_DUTCH (2)
#LANGUAGE_ITALIAN (3)
#LANGUAGE_FRENCH (4)
#LANGUAGE_SPANISH (5)
#LANGUAGE_PORTUGUESE (6)
#LANGUAGE_SWEDISH (7)
#LANGUAGE_DANISH (8)
#LANGUAGE_FINNISH (9)
#LANGUAGE_NORWEGIAN (10)
#LANGUAGE_POLISH (11)
```

```
#LANGUAGE_HUNGARIAN (12)
#LANGUAGE_GREEK (13)
#LANGUAGE_CZECH (14)
#LANGUAGE_TURKISH (15)
#LANGUAGE_CROATIAN (16)
#LANGUAGE_RUSSIAN (17)
#LANGUAGE_UNKNOWN (18)
#LANGUAGE_ABKHAZIAN (19)
#LANGUAGE_AFAR (20)
#LANGUAGE_AFRIKAANS (21)
#LANGUAGE_AKAN (22)
#LANGUAGE_ALBANIAN (23)
#LANGUAGE_AMHARIC (24)
#LANGUAGE_ARABIC (25)
#LANGUAGE_ARAGONESE (26)
#LANGUAGE_ARMENIAN (27)
#LANGUAGE_ASSAMESE (28)
#LANGUAGE_AVARIC (29)
#LANGUAGE_AVESTAN (30)
#LANGUAGE_AYMARA (31)
#LANGUAGE_AZERBAIJANI (32)
#LANGUAGE_BAMBARA (33)
#LANGUAGE_BASHKIR (34)
#LANGUAGE_BASQUE (35)
#LANGUAGE_BELARUSIAN (36)
#LANGUAGE_BENGALI (37)
#LANGUAGE_BIHARI (38)
#LANGUAGE_BISLAMA (39)
#LANGUAGE_BOSNIAN (40)
#LANGUAGE_BRETON (41)
#LANGUAGE_BULGARIAN (42)
#LANGUAGE_BURMESE (43)
#LANGUAGE_CATALAN (44)
#LANGUAGE_CHAMORRO (45)
#LANGUAGE_CHECHEN (46)
#LANGUAGE_CHICHEWA (47)
#LANGUAGE_CHINESE (48)
#LANGUAGE_CHUVASH (49)
#LANGUAGE_CORNISH (50)
#LANGUAGE_CORSICAN (51)
#LANGUAGE_CREE (52)
#LANGUAGE_DIVEHI (53)
#LANGUAGE_DZONGKHA (54)
#LANGUAGE_ESPERANTO (55)
#LANGUAGE_ESTONIAN (56)
#LANGUAGE_EWE (57)
#LANGUAGE_FAROESE (58)
```

```
#LANGUAGE_FIJIAN (59)
#LANGUAGE_FULAH (60)
#LANGUAGE_GALICIAN (61)
#LANGUAGE_GEORGIAN (62)
#LANGUAGE_GREENLANDIC (63)
#LANGUAGE_GUARANI (64)
#LANGUAGE_GUJARATI (65)
#LANGUAGE_HAITIAN (66)
#LANGUAGE_HAUSA (67)
#LANGUAGE_HEBREW (68)
#LANGUAGE_HERERO (69)
#LANGUAGE_HINDI (70)
#LANGUAGE_HIRIMOTU (71)
#LANGUAGE_INTERLINGUA (72)
#LANGUAGE_INDONESIAN (73)
#LANGUAGE_INTERLINGUE (74)
#LANGUAGE_IRISH (75)
#LANGUAGE_IGBO (76)
#LANGUAGE_INUPIAQ (77)
#LANGUAGE_IDO (78)
#LANGUAGE_ICELANDIC (79)
#LANGUAGE_INUKTITUT (80)
#LANGUAGE_JAPANESE (81)
#LANGUAGE_JAVANESE (82)
#LANGUAGE_KANNADA (83)
#LANGUAGE_KANURI (84)
#LANGUAGE_KASHMIRI (85)
#LANGUAGE_KAZAKH (86)
#LANGUAGE_CENTRALKHMER (87)
#LANGUAGE_KIKUYU (88)
#LANGUAGE_KINYARWANDA (89)
#LANGUAGE_KIRGHIZ (90)
#LANGUAGE_KOMI (91)
#LANGUAGE_KONGO (92)
#LANGUAGE_KOREAN (93)
#LANGUAGE_KURDISH (94)
#LANGUAGE_KUANYAMA (95)
#LANGUAGE_LATIN (96)
#LANGUAGE_LUXEMBOURGISH (97)
#LANGUAGE_GANDA (98)
#LANGUAGE_LIMBURGAN (99)
#LANGUAGE_LINGALA (100)
#LANGUAGE_LAO (101)
#LANGUAGE_LITHUANIAN (102)
#LANGUAGE_LUBAKATANGA (103)
#LANGUAGE_LATVIAN (104)
#LANGUAGE_MANX (105)
```

```
#LANGUAGE_MACEDONIAN (106)
#LANGUAGE_MALAGASY (107)
#LANGUAGE_MALAY (108)
#LANGUAGE_MALAYALAM (109)
#LANGUAGE_MALTESE (110)
#LANGUAGE_MAORI (111)
#LANGUAGE_MARATHI (112)
#LANGUAGE_MARSHALLESE (113)
#LANGUAGE_MONGOLIAN (114)
#LANGUAGE_NAURU (115)
#LANGUAGE_NAVAJO (116)
#LANGUAGE_NORTHNDEBELE (117)
#LANGUAGE_NEPALI (118)
#LANGUAGE_NDONGA (119)
#LANGUAGE_NORWEGIANBOKMAL (120)
#LANGUAGE_NORWEGIANNYNORSK (121)
#LANGUAGE_SICHUANYI (122)
#LANGUAGE_SOUTHNDEBELE (123)
#LANGUAGE_OCCITAN (124)
#LANGUAGE_OJIBWA (125)
#LANGUAGE_CHURCHSLAVIC (126)
#LANGUAGE_OROMO (127)
#LANGUAGE_ORIYA (128)
#LANGUAGE_OSSETIAN (129)
#LANGUAGE_PANJABI (130)
#LANGUAGE_PALI (131)
#LANGUAGE_PERSIAN (132)
#LANGUAGE_PASHTO (133)
#LANGUAGE_QUECHUA (134)
#LANGUAGE_ROMANSH (135)
#LANGUAGE_RUNDI (136)
#LANGUAGE_ROMANIAN (137)
#LANGUAGE_SANSKRIT (138)
#LANGUAGE_SARDINIAN (139)
#LANGUAGE_SINDHI (140)
#LANGUAGE_NORTHERNSAMI (141)
#LANGUAGE_SAMOAN (142)
#LANGUAGE_SANGO (143)
#LANGUAGE_SERBIAN (144)
#LANGUAGE_GAELIC (145)
#LANGUAGE_SHONA (146)
#LANGUAGE_SINHALA (147)
#LANGUAGE_SLOVAK (148)
#LANGUAGE_SLOVENIAN (149)
#LANGUAGE_SOMALI (150)
#LANGUAGE_SOUTHERNSOTHO (151)
#LANGUAGE_SUNDANESE (152)
```

```
#LANGUAGE_SWAHILI (153)
#LANGUAGE_SWATI (154)
#LANGUAGE_TAMIL (155)
#LANGUAGE_TELUGU (156)
#LANGUAGE_TAJIK (157)
#LANGUAGE_THAI (158)
#LANGUAGE_TIGRINYA (159)
#LANGUAGE_TIBETAN (160)
#LANGUAGE_TURKMEN (161)
#LANGUAGE_TAGALOG (162)
#LANGUAGE_TSWANA (163)
#LANGUAGE_TONGA (164)
#LANGUAGE_TSONGA (165)
#LANGUAGE_TATAR (166)
#LANGUAGE_TWI (167)
#LANGUAGE_TAHITIAN (168)
#LANGUAGE_UIGHUR (169)
#LANGUAGE_UKRAINIAN (170)
#LANGUAGE_URDU (171)
#LANGUAGE_UZBEK (172)
#LANGUAGE_VENDA (173)
#LANGUAGE_VIETNAMESE (174)
#LANGUAGE_WALLOON (175)
#LANGUAGE_WELSH (176)
#LANGUAGE_WOLOF (177)
#LANGUAGE_WESTERNFRISIAN (178)
#LANGUAGE_XHOSA (179)
#LANGUAGE_YIDDISH (180)
#LANGUAGE_YORUBA (181)
#LANGUAGE_ZHUANG (182)
#LANGUAGE_ZULU (183)
```

INPUTS

none

RESULTS

`lang` default language of current user

36.11 OpenCatalog

NAME

OpenCatalog – open a locale catalog

SYNOPSIS

```
OpenCatalog(name$[, version])
```

FUNCTION

This function tries to open the catalog specified by `name$` in the user's language. If a catalog in the user's language does not exist, this function will not fail. You have to specify alternative English strings in the `GetCatalogString()` function that will be used if there is no catalog in the user's language.

By default, Hollywood will search for catalogs inside a sub-directory named "Catalogs" inside the current directory. For example, if the current user's language is `#LANGUAGE_GERMAN` and you try to open "MyProgram.catalog" using `OpenCatalog()`, Hollywood will look in the following places for the catalog:

```
<current-directory>/Catalogs/deutsch/MyProgram.catalog
<current-directory>/Catalogs/german/MyProgram.catalog
```

If the current user's language is `#LANGUAGE_FRENCH`, Hollywood will look in these places for the catalog:

```
<current-directory>/Catalogs/français/MyProgram.catalog
<current-directory>/Catalogs/french/MyProgram.catalog
```

Note that on AmigaOS and compatibles, Hollywood will also scan Locale:Catalogs for the catalog.

Note that it is recommended to use international names for the language sub-directory, i.e. "german" instead of "deutsch" and "french" instead of "français". The native names are only supported for compatibility reasons with AmigaOS-based systems.

The catalog specified in `name$` can be either in the `IFF CTLG` format developed by Commodore, or it can be in a platform-neutral format defined by Hollywood. It is recommended to use Hollywood's platform-neutral format because `IFF CTLG` doesn't support Unicode and has some other restrictions and potential compatibility issues. See Section 36.1 [Catalog format], page 739, for details.

This command is also available from the preprocessor. Use `@CATALOG` to load catalogs from the preprocessor.

INPUTS

`name$` catalog to open

`version` optional: version that the catalog must have at least; if omitted this function will accept any version

EXAMPLE

```
OpenCatalog("Hollywood.catalog")

; this is our default English catalog
def$ = {}
def$[0] = "Welcome to Hollywood!"
def$[1] = "Written by Andreas Falkenhahn"
def$[2] = "What do you want to do?"

; if Hollywood.catalog is not available in the
; user's language; the English strings will be
; used
For k = 0 To 2
```

```
   c$[k] = GetCatalogString(k, def$[k])
Next

CloseCatalog()
```

The code above opens "Hollywood.catalog" and prints the first three entries from that catalog.

37 Math library

37.1 Abs

NAME
Abs – return absolute value (V1.5)

SYNOPSIS
```
result = Abs(val)
```

FUNCTION
This function returns the absolute value of `val`.

INPUTS

`val`	source value

RESULTS

`result`	absolute value of `val`

37.2 ACos

NAME
ACos – calculate arccosine (V2.0)

SYNOPSIS
```
result = ACos(x)
```

FUNCTION
Calculates the arccosine of `x`.

INPUTS

`x`	value whose arccosine is to be calculated

RESULTS

`result`	arccossine of `x`

37.3 Add

NAME
Add – add two values

SYNOPSIS
```
result = Add(value1, value2)
```

FUNCTION
Adds `value2` to `value1` and returns the result.

INPUTS

`value1`	base value

`value2` number to add

RESULTS

`result` result of the addition

EXAMPLE

```
a=99
a=Add(a,1)
Print(a)
```

This will print "100" to the screen.

37.4 ASin

NAME

ASin – calculate arcsine (V2.0)

SYNOPSIS

```
result = ASin(x)
```

FUNCTION

Calculates the arcsine of `x`.

INPUTS

`x` value whose arcsine is to be calculated

RESULTS

`result` arcsine of `x`

37.5 ATan

NAME

ATan – calculate arctangent (V2.0)

SYNOPSIS

```
result = ATan(x)
```

FUNCTION

Calculates the arctangent of `x`.

INPUTS

`x` value whose arctangent is to be calculated

RESULTS

`result` arctangent of `x`

37.6 ATan2

NAME
ATan2 – calculate arctangent of y/x (V2.0)

SYNOPSIS
```
result = ATan2(y, x)
```

FUNCTION
Calculates the arctangent of `y/x`. If `x` is 0, `ATan2()` returns 0.

INPUTS

`y`	denominator
`x`	numerator

RESULTS

`result`	arctangent of `y/x`

37.7 BitClear

NAME
BitClear – clear a bit (V2.0)

SYNOPSIS
```
n = BitClear(x, b)
```

FUNCTION
Clears bit number `b` in `x`.

INPUTS

`x`	source value
`b`	bit to clear (0-31)

RESULTS

`n`	cleared value

EXAMPLE
```
Print(BinStr(BitClear(Val("%11111111"), 2)))
```
The code above clears bit number 2 in the value %11111111 which results in the value %11111011.

37.8 BitComplement

NAME
BitComplement – complement a value (V2.0)

SYNOPSIS
```
n = BitComplement(x)
```

FUNCTION
This function inverts all bits in `x`. `x` is treated as a 32-bit integer.

INPUTS

`x` source value

RESULTS

`n` inverted value

EXAMPLE

```
Print(BinStr(BitComplement(Val("%11110000"))))
```

The code above inverts the value %11110000 and returns it to you as a 32-bit integer value (%11111111111111111111111100001111)

37.9 BitSet

NAME
BitSet – set a bit (V2.0)

SYNOPSIS

```
n = BitSet(x, b)
```

FUNCTION
Sets bit number `b` in `x` and returns the result.

INPUTS

`x` source value

`b` bit to set

RESULTS

`n` result of operation

EXAMPLE

```
Print(BinStr(BitSet(Val("%10111111"), 6)))
```

The code above sets bit 6 in the value %10111111 and returns the result which is %11111111.

37.10 BitTest

NAME
BitTest – test if a bit is set (V2.0)

SYNOPSIS

```
bool = BitTest(x, b)
```

FUNCTION
Tests if bit number `b` is set in `x` and returns `True` if this is the case, `False` otherwise.

INPUTS

`x` source value

`b` bit to test

RESULTS

`bool` `True` if the bit is set, else `False`

EXAMPLE

```
Print(BitTest(Val("%10101111"), 4))
```

Returns `False` because bit number 4 is not set in %10101111.

37.11 BitXor

NAME

BitXor – bitwise xor two values (V2.0)

SYNOPSIS

```
r = BitXor(a, b)
```

FUNCTION

Performs a bitwise xor operation on `a` and `b` and returns the result. The exclusive-or operation will set each bit in the resulting value only if the corresponding bit is set in one of the source values. If the bit is set in both source values, it will not be set in the resulting value.

INPUTS

`a` source value 1

`b` source value 2

RESULTS

`r` result of the bitwise xor

EXAMPLE

```
Print(BinStr(BitXor(Val("%11010001"), Val("%10110010"))))
```

Performs exclusive-or on the values %11010001 and %10110010 which results in the value %01100011.

37.12 Cast

NAME

Cast – cast a number to a new signed/unsigned type (V3.0)

SYNOPSIS

```
result = Cast(val, sign, type)
```

FUNCTION

This function can be used to cast a value to a different type. This is normally not needed in Hollywood because Hollywood only knows one variable type for numbers (all numbers in Hollywood are stored as signed 64-bit floating point values; Hollywood does not differentiate between byte, short, integer, and floating point types internally). Thus,

what you receive in `result` will not really be a variable of the type you have cast it to, but it will just be clipped to the boundaries of the variable type you specify.

However, this function can come in quite handy when it comes to do signed and unsigned conversions. You might want to know which number you get when you want to convert 41234 to a signed short value. You can use this function to do that.

`Cast()` accepts three arguments: The first argument is the value that shall be cast, the second argument specifies whether or not you want a signed value and must be either `True` (= cast to signed) or `False` (= cast to unsigned). The last argument finally specifies the type the value shall be cast to. This can be `#INTEGER` (32-bit), `#SHORT` (16-bit) or `#BYTE` (8-bit).

INPUTS

`val`	source value to be cast
`sign`	`True` (signed cast) or `False` (unsigned cast)
`type`	destination type for the value (`#INTEGER`, `#SHORT` or `#BYTE`)

RESULTS

`result`	result of the cast operation

EXAMPLE

```
Print(Cast(41234, TRUE, #SHORT))
```

The code above casts the number 41234 to a signed short (16-bit) value and prints the result which is -(2^16-41234) = -24302.

37.13 Ceil

NAME

Ceil – calculate ceiling of a value (V2.0)

SYNOPSIS

```
result = Ceil(x)
```

FUNCTION

Calculates the ceiling of `x`. The ceiling of a value is the smallest integer that is greater than or equal to it, e.g. the ceiling of 1.5 is 2, the ceiling of -1.5 is -1.

INPUTS

`x`	value whose ceiling is to be calculated

RESULTS

`result`	ceiling of `x`

37.14 Cos

NAME

Cos – calculate cosine (V1.5)

SYNOPSIS

```
result = Cos(x)
```

FUNCTION

Calculates the cosine of the angle `x`.

INPUTS

`x`	angle in radians

RESULTS

`result`	cosine of `x`

37.15 Deg

NAME

Deg – convert radians to degrees (V2.0)

SYNOPSIS

```
result = Deg(x)
```

FUNCTION

Converts the radians specified in `x` to degrees.

INPUTS

`x`	radian value which should be converted to degrees

RESULTS

`result`	degrees of `x`

37.16 Div

NAME

Div – divide value by a factor

SYNOPSIS

```
result = Div(value1, value2)
```

FUNCTION

Divides `value1` by `value2` and returns the result.

Note that although the division will use floating point precision, `value2` must not be 0. If you need to divide by zero in floating point, use `RawDiv()` instead. See Section 37.40 [RawDiv], page 776, for details.

INPUTS

`value1`	numerator
`value2`	denominator

RESULTS

`result`	result of the division

EXAMPLE

```
a=16
Div(a,4)
Print(a)
```

This will print "4" to the screen.

37.17 EndianSwap

NAME

EndianSwap – swap byte order of a value (V6.0)

SYNOPSIS

```
result = EndianSwap(val, bits)
```

FUNCTION

This function swaps the byte order in `val`. The additional parameter `bits` specifies how many bits should be taken into account and can be either 16 or 32.

INPUTS

`val`	input value
`bits`	operation length in bits

RESULTS

`result`	swapped bytes

EXAMPLE

```
DebugPrint(HexStr(EndianSwap($ABCD, 16)))
```

This prints $CDAB.

37.18 Exp

NAME

Exp – calculate the exponential of a value (V2.0)

SYNOPSIS

```
result = Exp(x)
```

FUNCTION

Calculates the exponential value of `x` (e ^ x).

INPUTS

`x`	value whose exponential is to be calculated

RESULTS

`result`	natural exponential of `x`

37.19 Floor

NAME
Floor – calculate the floor of a value (V2.0)

SYNOPSIS
```
result = Floor(x)
```

FUNCTION
Calculates the floor of `x`. The floor of a value is the largest integer that is less than or equal to it, e.g. the floor for 1.5 is 1, the floor for -1.5 is -2.

INPUTS

`x`	value whose floor is to be calculated

RESULTS

`result`	floor of `x`

37.20 Frac

NAME
Frac – return fractional part of a float (V1.5)

SYNOPSIS
```
result = Frac(val)
```

FUNCTION
This function returns the fractional part of a float.

INPUTS

`val`	source value

RESULTS

`result`	fractional part of `val`

EXAMPLE
```
a = Frac(3.14156)
```
The variable `a` is set to 0.14156.

37.21 FrExp

NAME
FrExp – extract mantissa and exponent from real number (V2.0)

SYNOPSIS
```
m, exp = FrExp(x)
```

FUNCTION
This function can be used to extract the mantissa and exponent of `x`. The mantissa is returned in the first return value, the exponent in the second.

INPUTS

`x`	floating point value to use

RESULTS

`m`	mantissa of floating point value
`exp`	exponent of floating point value

37.22 Hypot

NAME

Hypot – calculate the hypotenuse (V5.0)

SYNOPSIS

```
h = Hypot(x, y)
```

FUNCTION

This function can be used to calculate the hypotenuse of a right triangle. You have to pass the length of the two sides in `x` and `y`. A call to `Hypot()` is the same as the square root of x*x + y*y.

INPUTS

`x`	length of triangle side
`y`	length of triangle side

RESULTS

`h`	hypotenuse of triangle

37.23 Int

NAME

Int – return integer part of a float (V1.5)

SYNOPSIS

```
result = Int(val)
```

FUNCTION

This function returns the integer part of a floating point number.

INPUTS

`val`	source value

RESULTS

`result`	integer part of `val`

EXAMPLE

```
a = Int(4.5)
```

This call returns 4.

37.24 IsFinite

NAME

IsFinite – check for finiteness (V9.0)

SYNOPSIS

```
result = IsFinite(x)
```

FUNCTION

Checks if `x` is a finite value. A finite value is defined as any floating point value that is neither NaN nor infinity.

See Section 37.26 [IsNan], page 770, for details.

See Section 37.25 [IsInf], page 769, for details.

INPUTS

`x`	value to check

RESULTS

`result`	`True` if `x` is a finite value, `False` otherwise

EXAMPLE

```
a=RawDiv(1,0)  ; infinity, non-finite
b=RawDiv(0,0)  ; NaN, non-finite
c=RawDiv(5,2)  ; 2.5, finite
Print(IsFinite(a), IsFinite(b), IsFinite(c))
```

This will print "0 0 1" to the screen because the first two values are non-finite whereas the last value is finite. Note that we need to use `RawDiv()` here because the division operator as well as `Div()` will not allow a division by zero.

37.25 IsInf

NAME

IsInf – check for infinity (V9.0)

SYNOPSIS

```
result = IsInf(x)
```

FUNCTION

Checks if `x` is an infinity value (positive or negative). Positive and negative infinity values are generated when dividing 1/-1 by zero in floating point.

Hollywood also has a predefined constant named `#INF` that contains the infinity value but the preferred way of checking against infinity is using `IsInf()`.

INPUTS

`x`	value to check

RESULTS

`result`	`True` if `x` is an infinity value, `False` otherwise

EXAMPLE

```
a=RawDiv(1,0)
Print(IsInf(a))
```

This will print "1" to the screen because the division 1/0 will yield an infinity value. Note that we need to use `RawDiv()` here because the division operator as well as `Div()` will not allow a division by zero.

37.26 IsNan

NAME

IsNan – check if value is NaN (V9.0)

SYNOPSIS

```
result = IsNan(x)
```

FUNCTION

Checks if `x` is a special NaN value (not-a-number). NaN is a special return value for undefined floating point numbers such as the result of 0/0 or the square root of negative numbers.

Note that you should not test for NaN by comparing a number with itself, expecting to get `False`. This won't work on all platforms. Using `IsNan()` is the only portable way to check if a value is NaN.

The value of NaN is also in a predefined constant named `#NAN` but due to the design of Hollywood's parser you may only access this value using the `GetConstant()` function. Using it literally in a script, i.e. outside a string, will fail.

INPUTS

`x`	value to check

RESULTS

`result`	`True` if `x` is a NaN value, `False` otherwise

EXAMPLE

```
a=RawDiv(0,0)
Print(IsNan(a))
```

This will print "1" to the screen because the result of 0/0 is NaN. Note that we need to use `RawDiv()` here because the division operator as well as `Div()` will not allow a division by zero.

37.27 Ld

NAME

Ld – calculate base-2 logarithm (V1.5)

SYNOPSIS

```
result = Ld(val)
```

FUNCTION

This function calculates and returns the base-2 logarithm of `val`.

INPUTS

`val` source value

RESULTS

`result` base-2 logarithm of val

EXAMPLE

```
a = Ld(8)
```

This returns 3.

37.28 LdExp

NAME

LdExp – compute real number from mantissa and exponent (V2.0)

SYNOPSIS

```
r = LdExp(m, exp)
```

FUNCTION

This function can be used to compute a real number from the specified mantissa and exponent. It returns the value resulting from multiplying `m` by 2 raised to the power of `exp`.

INPUTS

`m` mantissa

`exp` exponent

RESULTS

`r` real number result

37.29 Limit

NAME

Limit – limit the range of a number (V2.0)

SYNOPSIS

```
n = Limit(x, low, high)
```

FUNCTION

Limits the range of `x`. If `x` is greater than or equal to `low` and less than or equal to `high`, the value of `x` is returned. If `x` is less than `low`, then `low` is returned. If `x` is greater than `high`, then `high` is returned.

This function ensures that the value of `x` stays in the boundaries defined by `low` and `high`.

INPUTS

`x` value to examine

RESULTS

`n` result of limit operation

37.30 Ln

NAME

Ln – calculate natural logarithm (base e) (V1.5)

SYNOPSIS

```
result = Ln(val)
```

FUNCTION

This function calculates and returns the natural logarithm of `val` (using base e).

INPUTS

`val` source value

RESULTS

`result` natural logarithm of `val`

37.31 Log

NAME

Log – calculate logarithm for any base (V1.5)

SYNOPSIS

```
result = Log(val, base)
```

FUNCTION

This function calculates and returns the logarithm for any base of `val`.

INPUTS

`val` source value

`base` logarithm base

RESULTS

`result` logarithm of `val` from `base`

EXAMPLE

```
a = Log(100, 10)
```

This returns 2.

37.32 Max

NAME

Max – return maximum value (V1.5)

SYNOPSIS

```
result = Max(a, b, ...)
```

FUNCTION

This function compares `a` and `b` and returns the value which is greater.

New in V2.0: You can pass any number of arguments to this function now. It will always return the maximum value of all input values.

INPUTS

`a`	value a
`b`	value b
`...`	any number of additional values

RESULTS

`result`	maximum value

EXAMPLE

```
a = Max(9, 10)
```

This returns 10.

37.33 Min

NAME

Min – return minimum value (V1.5)

SYNOPSIS

```
result = Min(a, b, ...)
```

FUNCTION

This function compares `a` and `b` and returns the value which is smaller.

New in V2.0: You can pass any number of arguments to this function now. It will always return the minimum value of all input values.

INPUTS

`a`	value a
`b`	value b
`...`	any number of additional values

RESULTS

`result`	minimum value

EXAMPLE

```
a = Min(9, 10)
```

This returns 9.

37.34 Mod

NAME
Mod – calculate remainder (V1.5)

SYNOPSIS
```
result = Mod(a, b)
```

FUNCTION
This function calculates the remainder of the division `a` / `b`.

INPUTS

`a` numerator

`b` denominator

RESULTS

`result` remainder of division

EXAMPLE
```
a = Mod(30, 4)
```
This returns 2 because 30 / 4 is 7 with a remainder of 2.

37.35 Mul

NAME
Mul – multiply two values

SYNOPSIS
```
result = Mul(value1, value2)
```

FUNCTION
Multiplies `value1` by `value2` and returns the result.

INPUTS

`value1` source value

`value2` multiplier

RESULTS

`result` result of multiplication

EXAMPLE
```
a=5
a=Mul(a,5)
Print(a)
```
This will print "25" to the screen.

37.36 NearlyEqual

NAME

NearlyEqual – check for near equality (V10.0)

SYNOPSIS

```
result = NearlyEqual(x, y)
```

FUNCTION

This function compares `x` and `y` and returns `True` if they are nearly equal. This function is only useful for floating point values. The advantage of this function over Hollywood's equality operator is that comparing floating point numbers using the equality operator can lead to problems in case there are extremely minimal differences on the bit level, e.g. caused by (de)serialization. That's why it's recommended to compare floating point numbers against near equality instead of absolute equality for more reliability.

INPUTS

`x`	first operand for comparison
`y`	second operand for comparison

RESULTS

`result`	`True` if `x` and `y` are nearly equal, `False` otherwise

37.37 Pi

NAME

Pi – returns the value of pi

SYNOPSIS

```
result = Pi()
```

FUNCTION

This function returns the value of pi.

Since pi is a constant, calling this function to get it is unnecessary overhead. Instead, you can just use Hollywood's inbuilt constant `#PI` to get the value of pi.

INPUTS

none

RESULTS

`result`	value of pi, will be the same as the `#PI` constant

EXAMPLE

```
Print(Pi() = #PI)
```

This will print `"1"` because the return value of `Pi()` will be the same as `#PI`.

37.38 Pow

NAME
Pow – calculate x raised to the power of y (V1.5)

SYNOPSIS
```
result = Pow(x, y)
```

FUNCTION
This function calculates `x` raised to the power of `y`.

INPUTS

`x`	base
`y`	exponent

RESULTS

`result`	calculation result

EXAMPLE
```
a = Pow(2, 8)
```
This returns 256.

37.39 Rad

NAME
Rad – convert degrees to radians (V2.0)

SYNOPSIS
```
result = Rad(x)
```

FUNCTION
Converts the degrees specified in `x` to radians.

INPUTS

`x`	degree value which should be converted to radians

RESULTS

`result`	radians of `x`

37.40 RawDiv

NAME
RawDiv – divide value by a factor (V9.0)

SYNOPSIS
```
result = RawDiv(value1, value2)
```

FUNCTION
Divides `value1` by `value2` using floating point precision and returns the result.

This function does exactly the same as `Div()` except that it also allows a division by zero, making it possible to generate special values like NaN or infinity.

See Section 37.26 [IsNan], page 770, for details.

See Section 37.25 [IsInf], page 769, for details.

INPUTS

`value1` numerator

`value2` denominator

RESULTS

`result` result of the division

EXAMPLE

```
a=RawDiv(16,4)
Print(a)
```

This will print "4" to the screen.

37.41 Rnd

NAME

Rnd – generate a random number

SYNOPSIS

```
result = Rnd(range)
```

FUNCTION

Generates a random integer number in the range of 0 to `range` (exclusive).

INPUTS

`range` upper integer boundary of the random generator

RESULTS

`result` a random number

EXAMPLE

```
num=Rnd(49)
```

Well, I cannot predict what value `num` will receive. I can only say that it will not be greater than 48 and not less than zero.

37.42 RndF

NAME

RndF – generate a random float (V1.5)

SYNOPSIS

```
result = RndF()
```

FUNCTION

This function returns a random floating point number in the range of 0.0 to 1.0 (exclusive).

Note that before Hollywood 8.0 this function's upper boundary was documented as 1.0 (inclusive). This was wrong. The value returned by `RndF()` is guaranteed to be less than 1.0.

INPUTS

none

RESULTS

`result` a random float in the range of 0.0 to 1.0 (exclusive)

EXAMPLE

```
num = RndF()
```

`num` is set to random floating point number between 0.0 and 1.0.

37.43 RndStrong

NAME

RndStrong – generate a strongly random number (V7.1)

SYNOPSIS

```
result = RndStrong(type, param)
```

FUNCTION

This function can be used to generate cryptographically secure pseudo random numbers. Numbers returned by `RndStrong()` are much more random than those generated by `Rnd()` or `RndF()` both of which aren't appropriate for anything cryptography related.

`RndStrong()` can operate in two different modes: If you pass `#INTEGER` in `type`, it will return a single integer value that won't be smaller than 0 and won't be bigger than the integer passed in `param` (but it could be equal to `param`). If you pass `#STRING` in `type`, `RndStrong()` will generate a string of `param` random bytes, i.e. when passing `#STRING`, `param` specifies the desired length of the string.

Be warned that `RndStrong()` is very slow in comparison to `Rnd()` and `RndF()`. That is why you shouldn't call it too often but rather cache its results if you need lots of very random numbers. For example, you could call `RndStrong()` with `type` set to `#STRING` and `param` to 65536 to make it generate a string containing 64kb worth of random numbers. Once you've consumed those, you could call it again for more random numbers.

Also note that on AmigaOS 3.x and AROS `RndStrong()` currently falls back to `Rnd()` because those operating systems don't offer cryptography-proof randomizers.

INPUTS

`type` type of data to generate; can be either `#INTEGER` or `#STRING` (see above)

`param` greatest acceptable integer number if `type` is `#INTEGER` or the length of the string to generate if `type` is `#STRING`

RESULTS

`result` random number(s) either as a string or integer value

EXAMPLE

```
num=RndStrong(#INTEGER, 49)
```

Well, I cannot predict what value `num` will receive. I can only say that it will not be greater than 49 and not smaller than zero.

37.44 Rol

NAME

Rol – left bit rotation (V3.0)

SYNOPSIS

```
result = Rol(a, x[, length])
```

FUNCTION

This function rotates the bits of value `a` to the left by `x` bits. Bit rotation means that the bits just circle inside `a`, i.e. bits moved out of the left side are appended to the right side.

The optional argument `length` allows you to specify the length of the rotate operation. By default, this is `#INTEGER` which means that `a` will be regarded as a 32-bit integer value. If you want to do a 16-bit or an 8-bit rotation, you need to use `#SHORT` and `#BYTE`, respectively.

INPUTS

`a` source value

`x` number of bits to rotate

`length` optional: bit length for this operation (defaults to `#INTEGER` which means 32-bit rotation); use `#SHORT` for 16-bit rotation and `#BYTE` for 8-bit rotation

RESULTS

`result` rotated value

EXAMPLE

```
r = Rol(Val("%10011110"), 4, #BYTE)
Print(BinStr(r, #BYTE))
```

The code above rotates the binary number %10011110 4 bits to the left and prints the result which is %11101001.

37.45 Ror

NAME

Ror – right bit rotation (V3.0)

SYNOPSIS

```
result = Ror(a, x[, length])
```

FUNCTION

This function rotates the bits of value `a` to the right by `x` bits. Bit rotation means that the bits just circle inside `a`, i.e. bits moved out of the right side are appended to the left side.

The optional argument allows you to specify the length of the rotate operation. By default, this is `#INTEGER` which means that `a` will be regarded as a 32-bit integer value. If you want to do a 16-bit or an 8-bit rotation, you need to use `#SHORT` and `#BYTE`, respectively.

INPUTS

`a`	source value
`x`	number of bits to rotate
`length`	optional: bit length for this operation (defaults to `#INTEGER` which means 32-bit rotation); use `#SHORT` for 16-bit rotation and `#BYTE` for 8-bit rotation

RESULTS

`result`	rotated value

EXAMPLE

```
r = Ror(Val("%10011110"), 2, #BYTE)
Print(BinStr(r, #BYTE))
```

The code above rotates the binary number %10011110 2 bits to the right and prints the result which is %10100111.

37.46 Round

NAME

Round – round a float (V1.5)

SYNOPSIS

```
result = Round(x)
```

FUNCTION

This function rounds `x` to the next intenger.

INPUTS

`x`	float to round

RESULTS

`result`	integer result

EXAMPLE

```
a = Round(3.7)
```

This returns 4.

37.47 Rt

NAME

Rt – calculate root (V1.5)

SYNOPSIS

```
result = Rt(x, y)
```

FUNCTION

This function calculates and returns the `y` root of value `x`.

INPUTS

`x`	source value
`y`	root to calculate

RESULTS

`result`	`y` root of `x`

EXAMPLE

```
a = Rt(27, 3)
```

This returns 3.

37.48 Sar

NAME

Sar – shift bits to the right (V3.0)

SYNOPSIS

```
result = Sar(a, x[, bignum])
```

FUNCTION

This function shifts `a` by `x` bits to the right padding the holes with the most significant bit of `a`. This is called an arithmetic shift. `a` is converted to a signed 32-bit integer variable before the shift (unless `bignum` is set to `True`).

Starting with Hollywood 9.0, there is an optional `bignum` argument. If this is set to `True`, `Sar()` will be able to operate on integers larger than 2^31 but keep in mind that `Sar()` still won't be possible to use the full 64-bit integer range because Hollywood's numeric type is a 64-bit floating point number and is thus limited to integers in the range of [-9007199254740992,9007199254740992].

INPUTS

`a`	source value
`x`	number of bits to shift
`bignum`	optional: whether or not to use 64-bit integers (defaults to `False`) (V9.0)

RESULTS

`result`	integer result

EXAMPLE

```
a = Sar(-256, 3)
```

This returns -32.

37.49 Sgn

NAME

Sgn – return the sign of a value (V2.0)

SYNOPSIS

```
sign = Sgn(x)
```

FUNCTION

Returns the sign of `x`. If `x` is less than 0, -1 is returned. If `x` is greater than 0, 1 is returned. If `x` is equal to 0, 0 is returned.

INPUTS

`x`	value to examine

RESULTS

`sign`	sign of `x`

37.50 Shl

NAME

Shl – shift bits to the left (V1.5)

SYNOPSIS

```
result = Shl(a, x[, bignum])
```

FUNCTION

This function shifts `a` by `x` bits to the left, padding the holes with zero bits. This is called a logical shift. `a` is converted to an unsigned 32-bit integer variable before the shift (unless `bignum` is set to `True`).

Starting with Hollywood 9.0, there is an optional `bignum` argument. If this is set to `True`, `Shl()` will be able to operate on integers larger than 2^31 but keep in mind that `Shl()` still won't be possible to use the full 64-bit integer range because Hollywood's numeric type is a 64-bit floating point number and is thus limited to integers in the range of [-9007199254740992,9007199254740992].

INPUTS

`a`	source value
`x`	number of bits to shift
`bignum`	optional: whether or not to use 64-bit integers (defaults to `False`) (V9.0)

RESULTS

`result`	integer result

EXAMPLE

```
a = Shl(256, 3)
```

This returns 2048.

37.51 Shr

NAME

Shr – shift bits to the right (V1.5)

SYNOPSIS

```
result = Shr(a, x[, bignum])
```

FUNCTION

This function shifts `a` by `x` bits to the right, padding the holes with zero bits. This is called a logical shift. `a` is converted to an unsigned 32-bit integer variable before the shift (unless `bignum` is set to `True`).

Starting with Hollywood 9.0, there is an optional `bignum` argument. If this is set to `True`, `Shr()` will be able to operate on integers larger than 2^31 but keep in mind that `Shr()` still won't be possible to use the full 64-bit integer range because Hollywood's numeric type is a 64-bit floating point number and is thus limited to integers in the range of [-9007199254740992,9007199254740992].

INPUTS

`a`	source value
`x`	number of bits to shift
`bignum`	optional: whether or not to use 64-bit integers (defaults to `False`) (V9.0)

RESULTS

`result`	integer result

EXAMPLE

```
a = Shr(256, 3)
```

This returns 32.

37.52 Sin

NAME

Sin – calculate sine (V1.5)

SYNOPSIS

```
result = Sin(x)
```

FUNCTION

Calculates the sine of the angle `x`.

INPUTS

`x`	angle in radians

RESULTS

`result` sine of `x`

37.53 Sqrt

NAME

Sqrt – calculate square root (V1.5)

SYNOPSIS

```
result = Sqrt(x)
```

FUNCTION

This function calculates and returns the square root of `x`.

INPUTS

`x` source value

RESULTS

`result` square root of `x`

EXAMPLE

```
a = Sqrt(64)
```

This returns 8.

37.54 Sub

NAME

Sub – subtract value from a value

SYNOPSIS

```
result = Sub(value1, value2)
```

FUNCTION

Subtracts `value2` from `value1` and returns the result.

INPUTS

`value1` minuend

`value2` subtrahend

RESULTS

`result` result of the subtraction

EXAMPLE

```
a=1
a=Sub(a,1)
Print(a)
```

The above code will print "0" to the screen.

37.55 Tan

NAME
Tan – calculate tangent (V1.5)

SYNOPSIS
```
result = Tan(x)
```

FUNCTION
Calculates the tangent of the angle `x`.

INPUTS

`x`	angle in radians

RESULTS

`result`	tangent of `x`

37.56 Wrap

NAME
Wrap – wrap values (V1.5)

SYNOPSIS
```
result = Wrap(x, low, high)
```

FUNCTION
Wrap will wrap the result of `x` if `x` is greater than or equal to `high`, or less than `low`. If `x` is less than `low`, then `x-low+high` is returned. If `x` is greater than or equal to `high`, then `x-high+low` is returned.

INPUTS

`x`	source value
`low`	low value
`high`	high value

RESULTS

`result`	result of operation

38 Memory block library

38.1 AllocMem

NAME

AllocMem – allocate a new memory block (V2.0)

SYNOPSIS

```
[id] = AllocMem(id, size)
```

FUNCTION

This command allocates a new memory block of the specified size and makes it available under the handle id, or, if you specify `Nil` in `id` `AllocMem()` automatically chooses an identifier and returns it. The memory will not be initialized and is therefore filled with random data. If you want to initialize it to zero, use the `FillMem()` command.

INPUTS

`id`	identifier for the memory block or `Nil` for auto id selection
`size`	size for the memory block in bytes

RESULTS

`id`	optional: identifier of the memory block; will only be returned when you pass `Nil` as argument 1 (see above)

38.2 AllocMemFromPointer

NAME

AllocMemFromPointer – initialize memory block from pointer (V6.0)

SYNOPSIS

```
[id] = AllocMemFromPointer(id, ptr, size)
```

FUNCTION

This command can be used to convert a pointer of type `#LIGHTUSERDATA` into a memory block that you can read from and write to using the memory block functions. The new memory block object will be made available under the handle `id`, or, if you specify `Nil` as `id`, `AllocMemFromPointer()` automatically chooses an identifier and returns it.

Note that `AllocMemFromPointer()` will not make a local copy of the memory pointed to by parameter 2. It will just allocate a container object so that you can access the memory data using the memory block functions. The `size` argument is only used to prevent read or write operations that are outside the memory block's boundaries. If you don't know the size of the memory block, you can also pass 0 in the `size` argument. In that case, Hollywood will never forbid any read and write operation on this memory block object.

Be warned that this is a dangerous function and should only be used by people who know what they are doing. Reading from or writing to non-allocated memory can easily crash your program.

INPUTS

`id`	identifier for the memory block or `Nil` for auto id selection
`ptr`	`#LIGHTUSERDATA` variable pointing to a memory block
`size`	size of the memory block in bytes or 0 if you don't know the size

RESULTS

`id`	optional: identifier of the memory block; will only be returned when you pass `Nil` as argument 1 (see above)

38.3 AllocMemFromVirtualFile

NAME

AllocMemFromVirtualFile – initialize memory block from virtual file (V6.1)

SYNOPSIS

```
[id] = AllocMemFromVirtualFile(id, vf$)
```

FUNCTION

This command can be used to access the raw memory contents of a virtual string file created using `DefineVirtualFileFromString()`. The new memory block object will be made available under the handle `id`, or, if you specify `Nil` as `id`, `AllocMemFromVirtualFile()` automatically chooses an identifier and returns it.

Note that `AllocMemFromVirtualFile()` will not make a local copy of the memory owned by the virtual string file. It will just allocate a container object so that you can access the memory data using the memory block functions.

Also note that when dealing with writeable virtual string files their memory representation can change with every single write operation performed on the virtual string file. Thus, it is not safe to access the virtual string file's memory through a container obtained from `AllocMemFromVirtualFile()` after a write operation to this virtual string file. Instead, you have to obtain a new container after every write operation and free the old one using `FreeMem()` first. Everything else will crash sooner or later.

Also note that it is forbidden to write to the memory block allocated by this function unless the virtual string file was created as writable.

Be warned that this is a dangerous function and should only be used by people who know what they are doing. Reading from or writing to non-allocated memory can easily crash your program and cause all sorts of issues.

INPUTS

`id`	identifier for the memory block or `Nil` for auto id selection
`vf$`	virtual string file allocated by `DefineVirtualFileFromString()`

RESULTS

`id`	optional: identifier of the memory block; will only be returned when you pass `Nil` as argument 1 (see above)

38.4 CopyMem

NAME

CopyMem – copy data between memory blocks (V2.0)

SYNOPSIS

```
CopyMem(src, dst, size[, src_offset, dst_offset])
```

FUNCTION

This command copies size bytes from the memory block with the id `src` to the block with the id `dst` starting from `src_offset` in the source block and from `dst_offset` in the destination block.

Please note that `src` and `dst` must not be the same blocks.

INPUTS

`src` source memory block to read data from

`dst` destination memory block to copy to

`size` size in bytes to copy

`src_offset`
optional: offset in the source block from where to start reading (defaults to 0 = beginning of the block)

`dst_offset`
optional: offset in the destination block from where to start writing (defaults to 0 = beginning of the block)

38.5 DecreasePointer

NAME

DecreasePointer – decrease pointer (V6.0)

SYNOPSIS

```
ptr = DecreasePointer(ptr, amount)
```

FUNCTION

This function decreases the specified pointer of type `#LIGHTUSERDATA` by the amount of bytes specified in parameter 2. Since you shouldn't use pointers in Hollywood, this function is really only useful when debugging code or doing some experimental stuff with Hollywood.

To increase a pointer, you can use the `IncreasePointer()` function. See Section 38.11 [IncreasePointer], page 792, for details.

INPUTS

`ptr` pointer passed as a `#LIGHTUSERDATA` variable

`amount` number of bytes to decrease

RESULTS

`ptr` new pointer of type `#LIGHTUSERDATA`

38.6 DumpMem

NAME

DumpMem – dump contents of a memory block (V2.0)

SYNOPSIS

```
DumpMem(id[, len, offset])
```

FUNCTION

This function dumps the contents of the memory block specified by `id` to the debug device (usually a console window). The optional argument allows you to specify the length in bytes that shall be dumped. If it is omitted, the whole block will be dumped. The optional argument offset can be used to start dumping from an offset inside the block.

Because of the raw data that is usually in memory blocks, Hollywood will do a hex dump including ASCII notation (if possible). The format is the following:

```
xxxxxxxx: bb bb bb bb bb bb bb bb bb bb bb bb bb bb cccccccccccccccc
```

x: offset in hexadecimal notation

b: 16 bytes per line

c: the 16 bytes in ASCII notation or '.' if the character is not graphical

INPUTS

`id` identifier of the memory block to be dumped

`len` optional: length in bytes (defaults to 0 which means that the complete block will be dumped)

`offset` optional: offset in the block from where to start dumping (defaults to 0 = beginning of the block)

38.7 FillMem

NAME

FillMem – fill a memory block (V2.0)

SYNOPSIS

```
FillMem(id, val, size[, offset, type])
```

FUNCTION

This function can be used to fill the whole memory block or a part of it with a specified value. `id` specifies the memory block to use, `val` is the value to use for filling and `size` specifies the size in bytes for the filling operation. You can use the optional argument offset to fine-tune the filling operation by specifying an offset in the memory block here (in bytes). `type` specifies the type of the value and can be `#BYTE` (1 byte), `#SHORT` (2 bytes) or `#INTEGER` (4 bytes).

If you use `#SHORT` or `#INTEGER` as the filling type, the `size` argument must be a multiple of 2 or 4 respectively. Also, `offset`, if specified, must be a multiple of 2 or 4 respectively.

INPUTS

`id`	memory block to use
`val`	value to fill block with
`size`	size of the filling operation in bytes; must be a multiple of 2 or 4 if type is `#SHORT` or `#INTEGER`
`offset`	optional: offset in the block where the filling should start (defaults to 0 which means the beginning of the block); must be a multiple of 2 or 4 if type is `#SHORT` or `#INTEGER`
`type`	optional: type of val; currently supported are `#BYTE`, `#SHORT` and `#INTEGER` (defaults to `#BYTE`)

EXAMPLE

```
AllocMem(1, 65536)
FillMem(1, 0, 65536)
```

Allocate a block of 64kb and initialize it to 0.

38.8 FreeMem

NAME

FreeMem – free a memory block (V2.0)

SYNOPSIS

```
FreeMem(id)
```

FUNCTION

This command releases the memory occupied by the block specified by `id`.

INPUTS

`id`	memory block to free

38.9 GetMemPointer

NAME

GetMemPointer – get raw address of memory block (V6.0)

SYNOPSIS

```
ptr = GetMemPointer(id[, offset])
```

FUNCTION

This function returns the raw address of the memory block passed in `id`. Optionally, you can specify an offset in bytes that should be added to the address before returning it. The pointer will be returned as a variable of type `#LIGHTUSERDATA`. It will stay valid until you call `FreeMem()` on the memory block object.

This function is only useful in connection with functions which expect parameters of type `#LIGHTUSERDATA`. There are currently no Hollywood functions which can handle `#LIGHTUSERDATA` parameters but plugins might want to use `#LIGHTUSERDATA` parameters for certain tasks in case using tables is too slow.

INPUTS

`id`	memory block whose address should be returned
`offset`	optional: offset in bytes to add to the address before returning it (defaults to 0)

RESULTS

`ptr`	pointer to the raw data of the specified memory block

38.10 GetMemString

NAME

GetMemString – get string from memory block (V7.1)

SYNOPSIS

```
s$ = GetMemString(id[, offset, length])
```

FUNCTION

This function returns `length` bytes starting at `offset` from the memory block specified by `id`. Both the `offset` and `length` parameters must be specified in bytes. If omitted, `offset` defaults to 0 (i.e. the beginning of the memory block) and `length` also defaults to 0 which means all remaining bytes starting from the specified offset are returned.

Note that since Hollywood strings can also contain raw binary data the string that is returned by `GetMemString()` isn't necessarily a valid UTF-8 string but contains the raw binary data copied from the specified memory block.

INPUTS

`id`	memory block to use
`offset`	optional: offset in bytes defining where to start fetching bytes (defaults to 0)
`length`	optional: number of bytes to fetch or 0 to fetch all remaining bytes in the memory block (defaults to 0)

RESULTS

`s$`	contents of the specified memory block range

38.11 IncreasePointer

NAME

IncreasePointer – increase pointer (V6.0)

SYNOPSIS

```
ptr = IncreasePointer(ptr, amount)
```

FUNCTION

This function increases the specified pointer of type `#LIGHTUSERDATA` by the amount of bytes specified in parameter 2. Since you shouldn't use pointers in Hollywood, this

function is really only useful when debugging code or doing some experimental stuff with Hollywood.

To decrease a pointer, you can use the `DecreasePointer()` function. See Section 38.5 [DecreasePointer], page 789, for details.

INPUTS

`ptr` pointer passed as a `#LIGHTUSERDATA` variable

`amount` number of bytes to increase

RESULTS

`ptr` new pointer of type `#LIGHTUSERDATA`

38.12 MemToTable

NAME

MemToTable – return memory block contents as a table (V6.0)

SYNOPSIS

```
t = MemToTable(id, type[, table])
```

FUNCTION

This function can be used to return the contents of a memory block (or part of a memory block) as a table. The `type` argument specifies the data type of the elements that should be read from the memory block and stored inside a table. This can be either `#BYTE` (1 byte), `#SHORT` (2 bytes), `#INTEGER` (4 bytes), `#FLOAT` (4 bytes), or `#DOUBLE` (8 bytes).

The optional `table` argument can be used to set additional parameters for the operation. The following table fields are currently recognized:

`Items:` The number of items to be read from the memory block. Note that this is not a size in bytes, but an item count. So if you set the `type` argument to `#INTEGER` and set `Items` to 4, 16 bytes will be read from the memory block. Defaults to all items that are in the memory block.

`Offset:` This tag can be used to specify an offset in bytes inside the memory block that defines where `MemToTable()` should start to read elements. Defaults to 0 which means read from the beginning of the memory block.

`Signed:` If this tag is set to `True`, `MemToTable()` will treat all elements of type `#BYTE`, `#SHORT`, and `#INTEGER` as signed values. Defaults to `False`.

`EndianSwitch:`
If this tag is set to `True`, `MemToTable()` will switch byte order for all multi-byte data types. This can be useful if you need to convert between big and little endian values. Defaults to `False`.

To convert a table back into a memory block, use the `TableToMem()` function. See Section 38.16 [TableToMem], page 797, for details.

INPUTS

`id` memory block to use

`type` data type of the elements to read (see above)

`table` optional: table configuring further parameters (see above)

RESULTS

`t` a table containing as many elements as specified in the `Items` tag

EXAMPLE

```
AllocMem(1, 26)
For Local k = 0 To 25 Do Poke(1, k, 'A' + k, #BYTE)
Local t = MemToTable(1, #BYTE)
For Local k = 0 To 25 Do Print(Chr(t[k]))
```

This prints the alphabet from a memory block source.

38.13 Peek

NAME

Peek – look inside a memory block (V2.0)

SYNOPSIS

```
val = Peek(id, offset[, type, len, endian])
```

FUNCTION

This function allows you to look inside a memory block at the specified offset. `type` specifies the data type for which you want to look. This can be `#BYTE`, `#SHORT`, `#INTEGER`, `#FLOAT`, `#DOUBLE` or `#STRING`. `#BYTE` will read one byte from the block, `#SHORT` reads two bytes, `#INTEGER` and `#FLOAT` four bytes, `#DOUBLE` eight bytes, and `#STRING` reads from the memory block until it encounters a non-graphical character or a `NULL` character.

Starting with Hollywood 2.5, you can specify the optional parameter `len`. This parameter can only be used with type `#STRING`. If specified, `Peek()` will read exactly `len` bytes from the specified memory block location and return it as a string. You can use this to read raw data from memory blocks because `Peek()` will not terminate at non-graphical or `NULL` characters any more if `len` is specified. If `len` is 0, which is the default setting, `Peek()` will read bytes until it encounters a non-graphical or `NULL` character.

Starting with Hollywood 6.0 there is a new `endian` parameter which allows you to specify the byte order that should be used when reading the data from the memory block. This can be set to the following types:

`#BIGENDIAN:`
: Big endian byte order, MSB first. This is the default.

`#NATIVEENDIAN:`
: Native endian byte order. If you use this type, the byte order will depend on the default byte order on the host system, i.e. big endian on big endian systems, little endian on little endian systems. Be careful using this type because it limits portability.

`#LITTLEENDIAN:`
: Little endian byte order, LSB first. (V8.0)

INPUTS

`id` identifier of the memory block to be used

`offset` where to peek (in bytes)

`type` optional: data type to peek for (defaults to `#INTEGER`)

`len` optional: number of bytes to read (works only in connection with type `#STRING`) (defaults to 0 which means read until a non-graphical or `NULL` character) (V2.5)

`endian` optional: byte order to use (defaults to `#BIGENDIAN`) (V6.0)

RESULTS

`val` contents of the memory block at the specified offset; can be a value or a string (if type was set to `#STRING`)

38.14 Poke

NAME

Poke – write to a memory block (V2.0)

SYNOPSIS

```
Poke(id, offset, val[, type, endian])
```

FUNCTION

This function writes the value or string specified in `val` to the memory block with the identifier `id` at the specified offset. `Type` defaults to `#INTEGER` and specifies the type of `val`. You can also use the following types: `#BYTE` (1 byte), `#SHORT` (2 bytes), `#FLOAT` (4 byte single-precision floating point number), `#DOUBLE` (8 byte double-precision floating point number) or `#STRING`. Poking a string into a memory block occupies the number of characters in the string plus 1 byte.

Starting with Hollywood 6.0 there is a new `endian` parameter which allows you to specify the byte order that should be used when writing the data to the memory block. This can be set to the following types:

`#BIGENDIAN:`
Big endian byte order, MSB first. This is the default.

`#NATIVEENDIAN:`
Native endian byte order. If you use this type, the byte order will depend on the default byte order on the host system, i.e. big endian on big endian systems, little endian on little endian systems. Be careful using this type because it limits portability.

`#LITTLEENDIAN:`
Little endian byte order, LSB first. (V8.0)

INPUTS

`id` identifier of the memory block to be used

`offset` where to poke (in bytes)

`val`	data to poke; can be string or number
`type`	optional: data type to poke (defaults to `#INTEGER`)
`endian`	optional: byte order to use (defaults to `#BIGENDIAN`) (V6.0)

EXAMPLE

```
AllocMem(1, 1024)
Poke(1, 0, "Hello World!", #STRING)
Print(Peek(1, 0, #STRING))
```

This will print "Hello World!" to the screen.

38.15 ReadMem

NAME

ReadMem – read raw data from a file (V2.0)

SYNOPSIS

```
ReadMem(file_id, blk_id, len[, offset])
```

FUNCTION

This function allows you to read `len` bytes raw data from an open file (specified by `file_id`, use `OpenFile()` to open files) to a memory block (specified by `blk_id`). Additionally you can specify the optional `offset` argument to define where in the memory block the raw data shall be stored. The data from the source file is read from the current cursor position in the file which you can modify using the `Seek()` command.

INPUTS

`file_id`	identifier of an open file
`blk_id`	identifier of a memory block
`len`	bytes to read from the file
`offset`	optional: where to store the data in the block (defaults to 0 = beginning of the block)

EXAMPLE

```
len = FileSize("C:SetPatch")
OpenFile(1, "C:SetPatch", #MODE_READ)
AllocMem(1, len)
ReadMem(1, 1, len)
CloseFile(1)
OpenFile(1, "Ram:Copy_of_SetPatch", #MODE_WRITE)
WriteMem(1, 1, len)
CloseFile(1)
FreeMem(1)
```

Makes a copy of the SetPatch program in RAM: by using the two raw data i/o functions `ReadMem()` and `WriteMem()`.

38.16 TableToMem

NAME

TableToMem – write table contents to memory block (V6.0)

SYNOPSIS

```
TableToMem(t, id, type[, table])
```

FUNCTION

This function writes the contents of the table passed as the first parameter to the memory block object specified by `id`. The third parameter specifies the data type of the elements that should be written to the memory block. This can be either `#BYTE` (1 byte), `#SHORT` (2 bytes), `#INTEGER` (4 bytes), `#FLOAT` (4 bytes), or `#DOUBLE` (8 bytes).

The optional `table` argument can be used to set additional parameters for the operation. The following table fields are currently recognized:

`Offset:` This tag can be used to specify an offset in bytes inside the memory block that defines where `TableToMem()` should start to write elements. Defaults to 0 which means start writing at the beginning of the memory block.

`EndianSwitch:`
If this tag is set to `True`, `TableToMem()` will switch byte order for all multi-byte data types when writing them to the memory block. This can be useful if you need to convert between big and little endian values. Defaults to `False`.

If there are more elements in the table than the memory block can store, this function will issue an error.

To convert a memory block back into a table, use the `MemToTable()` function. See Section 38.12 [MemToTable], page 793, for details.

INPUTS

`t`	table whose contents should be written to the memory block
`id`	memory block to use
`type`	data type of the elements to write (see above)
`table`	optional: table configuring further parameters (see above)

38.17 WriteMem

NAME

WriteMem – write raw data to a file (V2.0)

SYNOPSIS

```
WriteMem(file_id, blk_id, len[, offset])
```

FUNCTION

This function writes `len` bytes from the memory block specified by `blk_id` to the file specified by `file_id`. The data is read from the memory block at the `offset`, which you can specify in the homonymous optional argument. The data is written to the file at the current cursor position which you can modify using the `Seek()` command.

INPUTS

`file_id`	identifier of an open file (use `OpenFile()`)
`blk_id`	identifier of a memory block
`len`	bytes to write to the file
`offset`	optional: where to begin reading the data from the block (defaults to 0 = beginning of the block)

EXAMPLE

See Section 38.15 [ReadMem], page 796.

39 Menu library

39.1 CreateMenu

NAME

CreateMenu – create a menu strip (V6.0)

SYNOPSIS

`[id] = CreateMenu(id, table)`

FUNCTION

This function can be used to create a menu strip that can later be attached to one or more displays by calling `SetDisplayAttributes()` on an existing display or by specifying the `Menu` tag in the `@DISPLAY` preprocessor command or in the `CreateDisplay()` call.

You have to pass an identifier for the new menu strip or `Nil`. If you pass `Nil`, `CreateMenu()` will return a handle to the new menu strip which you can then use to refer to this menu strip.

You also need to pass a table containing the actual menu tree definition to this function. Menus are defined as a tree structure that is composed of a master table that contains various subtables. See Section 39.8 [MENU], page 804, for a detailed description of menu tree tables.

This command is also available from the preprocessor: Use `@MENU` to create menu strips at startup time!

INPUTS

`id`	identifier for the menu strip or `Nil` for auto id select
`table`	menu tree definition

RESULTS

`id`	optional: identifier of the new menu; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateMenu(1, {
    {"File", {
        {"New", ID = "new"},
        {"Open...", ID = "open"},
        {""},
        {"Close", ID = "close", Flags = #MENUITEM_DISABLED},
        {""},
        {"Save", Flags = #MENUITEM_DISABLED, Hotkey = "S"},
        {"Compress", ID = "cmp", Flags = #MENUITEM_TOGGLE},
        {""},
        {"Export image...", {
            {"JPEG...", ID = "jpeg"},
            {"PNG...", ID = "png"},
            {"BMP...", ID = "bmp"}}},
```

```
        {""},
        {"Dump state", ID = "dump"},
        {""},
        {"Quit", ID = "quit", Hotkey = "Q"}}},

    {"Edit", {
        {"Cut", ID = "cut"},
        {"Copy", ID = "copy"},
        {"Paste", ID = "paste"}}},

    {"?", {
        {"About...", ID = "about"}}}
    })

SetDisplayAttributes({Menu = 1})
```

The code above creates a menu strip and attaches it to the current display.

39.2 DeselectMenuItem

NAME

DeselectMenuItem – deselect a toggle menu item (V6.0)

SYNOPSIS

`DeselectMenuItem(id, item$[, detached])`

FUNCTION

This function can be used to deselect a toggle menu item. The optional argument `detached` specifies whether a menu strip that is attached to a display or a detached menu strip should be used. If `detached` is `False` (which is also the default), the menu strip of the display specified by `id` will be used. If `detached` is `True`, the menu strip specified by `id` will be used. In other words: If you set `detached` to `True`, you need to pass the identifier of a menu strip in `id`; otherwise you need to pass the identifier of a display in `id`.

Note that when setting `detached` to `True` your operation will never have any effect on menu strips attached to a display. Setting `detached` to `True` is typically only used with menu strips that are shown as popup menus using the `PopupMenu()` function. It's impossible to address display menu strips when setting `detached` to `True` because a single menu strip can be attached to multiple displays.

On AmigaOS 4 you can also pass the special value of 0 for `id`. In that case, the context menu of the docky will be used.

INPUTS

`id`	identifier of a display or a menu strip (see above)
`item$`	identifier of the item inside the menu strip
`detached`	optional: `False` if `id` specifies a display, `True` if it specifies a menu strip object (defaults to `False`) (V10.0)

39.3 DisableMenuItem

NAME

DisableMenuItem – disable a menu item (V6.0)

SYNOPSIS

```
DisableMenuItem(id, item$[, detached])
```

FUNCTION

This function can be used to disable a menu item. The optional argument `detached` specifies whether a menu strip that is attached to a display or a detached menu strip should be used. If `detached` is `False` (which is also the default), the menu strip of the display specified by `id` will be used. If `detached` is `True`, the menu strip specified by `id` will be used. In other words: If you set `detached` to `True`, you need to pass the identifier of a menu strip in `id`; otherwise you need to pass the identifier of a display in `id`.

Note that when setting `detached` to `True` your operation will never have any effect on menu strips attached to a display. Setting `detached` to `True` is typically only used with menu strips that are shown as popup menus using the `PopupMenu()` function. It's impossible to address display menu strips when setting `detached` to `True` because a single menu strip can be attached to multiple displays.

On AmigaOS 4 you can also pass the special value of 0 for `id`. In that case, the context menu of the docky will be used.

INPUTS

`id`	identifier of a display or a menu strip object (see above)
`item$`	identifier of the item inside the menu strip
`detached`	optional: `False` if `id` specifies a display, `True` if it specifies a menu strip object (defaults to `False`) (V10.0)

39.4 EnableMenuItem

NAME

EnableMenuItem – enable a menu item (V6.0)

SYNOPSIS

```
EnableMenuItem(id, item$[, detached])
```

FUNCTION

This function can be used to enable a menu item. The optional argument `detached` specifies whether a menu strip that is attached to a display or a detached menu strip should be used. If `detached` is `False` (which is also the default), the menu strip of the display specified by `id` will be used. If `detached` is `True`, the menu strip specified by `id` will be used. In other words: If you set `detached` to `True`, you need to pass the identifier of a menu strip in `id`; otherwise you need to pass the identifier of a display in `id`.

Note that when setting `detached` to `True` your operation will never have any effect on menu strips attached to a display. Setting `detached` to `True` is typically only used

with menu strips that are shown as popup menus using the `PopupMenu()` function. It's impossible to address display menu strips when setting `detached` to `True` because a single menu strip can be attached to multiple displays.

On AmigaOS 4 you can also pass the special value of 0 for `id`. In that case, the context menu of the docky will be used.

INPUTS

`id`	identifier of a display or a menu strip (see above)
`item$`	identifier of the item inside the menu strip
`detached`	optional: `False` if `id` specifies a display, `True` if it specifies a menu strip object (defaults to `False`) (V10.0)

39.5 FreeMenu

NAME

FreeMenu – free a menu strip (V6.0)

SYNOPSIS

```
FreeMenu(id)
```

FUNCTION

This command can be used to the free the specified menu strip. Please note that only menu strips that are no longer attached to a display can be freed. To detach a menu strip from a display, call `SetDisplayAttributes()` on the display passing the special value -1 in the `Menu` tag.

INPUTS

`id`	identifier of the menu strip to free

39.6 IsMenuItemDisabled

NAME

IsMenuItemDisabled – check if a menu item is disabled (V6.0)

SYNOPSIS

```
result = IsMenuItemDisabled(id, item$[, detached])
```

FUNCTION

This function can be used to check whether a menu item is disabled or not. The optional argument `detached` specifies whether a menu strip that is attached to a display or a detached menu strip should be used. If `detached` is `False` (which is also the default), the menu strip of the display specified by `id` will be used. If `detached` is `True`, the menu strip specified by `id` will be used. In other words: If you set `detached` to `True`, you need to pass the identifier of a menu strip in `id`; otherwise you need to pass the identifier of a display in `id`.

Note that when setting `detached` to `True` your operation will never have any effect on menu strips attached to a display. Setting `detached` to `True` is typically only used

with menu strips that are shown as popup menus using the `PopupMenu()` function. It's impossible to address display menu strips when setting `detached` to `True` because a single menu strip can be attached to multiple displays.

On AmigaOS 4 you can also pass the special value of 0 for `id`. In that case, the context menu of the docky will be used.

INPUTS

`id`	identifier of a display or a menu strip (see above)
`item$`	identifier of the item inside the menu strip
`detached`	optional: `False` if `id` specifies a display, `True` if it specifies a menu strip object (defaults to `False`) (V10.0)

RESULTS

`result`	`True` if the menu item is disabled, `False` otherwise

39.7 IsMenuItemSelected

NAME

IsMenuItemSelected – check if a menu item is selected (V6.0)

SYNOPSIS

```
result = IsMenuItemSelected(id, item$[, detached])
```

FUNCTION

This function can be used to check whether a toggle or radio menu item is selected or not. The optional argument `detached` specifies whether a menu strip that is attached to a display or a detached menu strip should be used. If `detached` is `False` (which is also the default), the menu strip of the display specified by `id` will be used. If `detached` is `True`, the menu strip specified by `id` will be used. In other words: If you set `detached` to `True`, you need to pass the identifier of a menu strip in `id`; otherwise you need to pass the identifier of a display in `id`.

Note that when setting `detached` to `True` your operation will never have any effect on menu strips attached to a display. Setting `detached` to `True` is typically only used with menu strips that are shown as popup menus using the `PopupMenu()` function. It's impossible to address display menu strips when setting `detached` to `True` because a single menu strip can be attached to multiple displays.

On AmigaOS 4 you can also pass the special value of 0 for `id`. In that case, the context menu of the docky will be used.

INPUTS

`id`	identifier of a display or a menu strip (see above)
`item$`	identifier of the item inside the menu strip
`detached`	optional: `False` if `id` specifies a display, `True` if it specifies a menu strip object (defaults to `False`) (V10.0)

RESULTS

`result`	`True` if the menu item is selected, `False` otherwise

39.8 MENU

NAME

MENU – create a menu strip (V6.0)

SYNOPSIS

```
@MENU id, table
```

FUNCTION

This preprocessor command can be used to create a menu strip that can later be attached to one or more displays by calling `SetDisplayAttributes()` on an existing display or by specifying the `Menu` tag in the `@DISPLAY` preprocessor command or in the `CreateDisplay()` call.

You need to pass an identifier for the menu strip to this preprocessor command as well as the actual menu definition. Menus are defined as a tree structure that is composed of a master table that contains various subtables. There are two different types of subtables:

1. Menu tables: These tables contain a heading for the menu in the table element at index 0 and a list of single menu items in the table element at index 1. The list of single menu items, of course, is another subtable that is composed of another number of tables that describe a menu item each (see below). The table you pass to `@MENU` must start with a number of menu tables because every menu item needs a parent menu that it belongs it. These parent menus are described in the menu tables. You can also nest menus, i.e. it is possible to insert a submenu among a number of menu items.
2. Menu item tables: A menu item subtable is a table that describes a single menu item. The name of the menu item that is to be shown in the menu has to be passed at table index 0. If you pass an empty string (`""`) at table index 0, Hollywood will insert a horizontal divider bar instead of a selectable menu item. These divider bars can be used to group related menu items together. There must not be any element at table index 1 for menu item tables. Instead, you can use a number of other tags to configure things like menu item type, hotkey, and identifier. See below for a list of possible tags.

Menu item tables recognize the following tags:

`ID:` Here you can specify a string that identifies this menu item. This string will be passed to your event handler callback so that you know which menu item has been selected by the user. The identifier specified here is also necessary if you want to use functions like `DisableMenuItem()` or `SelectMenuItem()` to manually modify the state of menu items.

`Flags:` This tag allows you to set some flags for this menu item. This can be set to a bitmask containing one or more of the following flags:

`#MENUITEM_TOGGLE:`
If this flag is set, this menu item will be created as a toggle menu item. Toggle menu items have two different states (selected and deselected) and the window manager usually renders them with

a checkmark indicating their current state. You can manually modify the toggle state of a menu item by calling the functions `SelectMenuItem()` and `DeselectMenuItem()` or by setting the `#MENUITEM_SELECTED` flag (see below). The toggle state can be checked by calling the `IsMenuItemSelected()` function.

`#MENUITEM_RADIO:`
: Set this flag to make the menu item part of a radio group. All neighbouring menu items which have `#MENUITEM_RADIO` set, will be included in the same radio group. All menu items inside a radio group will be mutually exclusive, i.e. only one menu item of a radio group can be active at a time. You can manually modify the state of a radio menu item by calling the functions `SelectMenuItem()` or by setting the `#MENUITEM_SELECTED` flag (see below). The radio state can be checked by calling the `IsMenuItemSelected()` function. Since radio groups always need an active item, it is not possible to call `DeselectMenuItem()` on a radio menu item. If you want to deselect a radio menu item, you need to select a different radio menu item using `SelectMenuItem()` and then the previously selected radio menu item will automatically be deselected. (V7.1)

`#MENUITEM_SELECTED:`
: If you have set the `#MENUITEM_TOGGLE` or `#MENUITEM_RADIO` flag to create a toggle or a radio menu item, you can set this flag to put the menu item into selected state. See above for more information on toggle and radio menu items.

`#MENUITEM_DISABLED:`
: If you set this flag, the menu item will be grayed out so that the user won't be able to select it. You can also manually disable a menu item by calling the function `DisableMenuItem()`. To enable a disabled menu item, use `EnableMenuItem()`. The disabled state of a menu item can be checked by calling the `IsMenuItemDisabled()` function.

`Hotkey:`
: This tag can be set to a character that acts as a key shortcut for this menu item. For the best cross-platform compatibility, this tag should be set to a string that contains one character only, e.g. `"q"` for a quit shortcut. Some platforms also support custom hotkeys like `"CTRL+F1"` but in that case you often have to implement the key handling on your own because the window manager does not support listening to these special shortcuts. If you pass a one character string, however, automatic hotkey listening will work on all platforms.

Menu strips can also be created at runtime by using the `CreateMenu()` command. See Section 39.1 [CreateMenu], page 799, for details.

You can attach menu strips to displays by using the `SetDisplayAttributes()` function or by specifying the `Menu` tag in the `@DISPLAY` preprocessor command

or in the `CreateDisplay()` call. To detach a menu strip from a display, call `SetDisplayAttributes()` passing the special value -1 in the `Menu` tag.

To get notified when the user selects items from the menu, you have to listen to the `OnMenuSelect` event handler. This can be done by installing a listener callback for this event using `InstallEventHandler()`. See Section 29.13 [InstallEventHandler], page 553, for details.

Please note that menu strips are not supported for displays in fullscreen mode. They will only work if your display is in windowed mode.

Note that on Android, it's normally not possible to place menu items in the root level of the action bar's options menu because on desktop systems menu items always have to be members of certain root groups (e.g. "File", "Edit", "View", etc.) There can't be any menu items outside such root groups. When using menu strips on Android, Hollywood will of course replicate the desktop menu behaviour by creating individual submenus for those root groups. This means, however, that the user has to tap at least twice to select a menu item because there won't be any menu items in the root level, they will always be in submenus instead. So if you don't want Hollywood to create those submenus but just place all items in the root level, set the `SingleMenu` tag in the `@DISPLAY` preprocessor command to `True`. This is especially useful if there are only a few menu items and it doesn't make sense to place them in submenus.

Also note that menu strips are currently unsupported on Linux and iOS.

INPUTS

`id`	a value that is used to identify this menu strip
`table`	menu tree definition (see above)

EXAMPLE

```
@MENU 1, {
    {"File", {
        {"New", ID = "new"},
        {"Open...", ID = "open"},
        {""},
        {"Close", ID = "close", Flags = #MENUITEM_DISABLED},
        {""},
        {"Save", Flags = #MENUITEM_DISABLED, Hotkey = "S"},
        {"Compress", ID = "cmp", Flags = #MENUITEM_TOGGLE},
        {""},
        {"Export image...", {
            {"JPEG...", ID = "jpeg"},
            {"PNG...", ID = "png"},
            {"BMP...", ID = "bmp"}}},
        {""},
        {"Dump state", ID = "dump"},
        {""},
        {"Quit", ID = "quit", Hotkey = "Q"}}},

    {"Edit", {
        {"Cut", ID = "cut"},
```

```
        {"Copy", ID = "copy"},
        {"Paste", ID = "paste"}}},

    {"?", {
        {"About...", ID = "about"}}}
    }

@DISPLAY {Menu = 1}
```

The code above creates a menu strip that is attached then attached to the default display.

39.9 PopupMenu

NAME

PopupMenu – show popup menu (V10.0)

SYNOPSIS

`PopupMenu(id[, x, y])`

FUNCTION

This function shows the menu strip specified by `id` as a popup menu. Popup menus are also known as context menus and are typically shown when the user presses the right mouse button at a certain area of the display. The menu strip passed to `PopupMenu()` must have been created using either `CreateMenu()` or `@MENU` and it must only consist of a single strip. The title of the menu strip is ignored.

The optional arguments `x` and `y` allow you to specify the desired position of the popup menu. Note that this must be passed in coordinates relative to the screen's top-left corner, i.e. if you pass 0 for `x` and `y`, the popup menu will appear in the top-left corner of the screen. If you omit the `x` and `y` arguments, the popup menu will be shown at the location of the mouse pointer.

`PopupMenu()` will block the script execution until the user has selected a menu item or closed the popup menu. Just as with normal menu events, popup menu events will be sent to your script through the `OnMenuSelect` event handler. See Section 29.13 [InstallEventHandler], page 553, for details.

Note that on AmigaOS and compatibles the right mouse button is typically reserved for accessing the screen menu. If you want to be able to listen to the right mouse button to show a popup menu, you need to set the `TrapRMB` tag to `True`. See Section 25.8 [DISPLAY], page 380, for details.

INPUTS

`id`	identifier of a menu strip
`x`	optional: desired x position for the popup menu
`y`	optional: desired y position for the popup menu

EXAMPLE

```
CreateMenu(1, {{"Unused", {
   {"Cut", ID = "cut"},
```

```
    {"Copy", ID = "copy"},
    {"Paste", ID = "paste"},
    {""},
    {"Undo", ID = "undo"},
    {"Redo", ID = "redo"}
}}})
PopupMenu(1)
```

The code above defines and shows a popup menu.

39.10 SelectMenuItem

NAME

SelectMenuItem – select a toggle or radio menu item (V6.0)

SYNOPSIS

```
SelectMenuItem(id, item$[, detached])
```

FUNCTION

This function can be used to select a toggle or radio menu item. The optional argument `detached` specifies whether a menu strip that is attached to a display or a detached menu strip should be used. If `detached` is `False` (which is also the default), the menu strip of the display specified by `id` will be used. If `detached` is `True`, the menu strip specified by `id` will be used. In other words: If you set `detached` to `True`, you need to pass the identifier of a menu strip in `id`; otherwise you need to pass the identifier of a display in `id`.

Note that when setting `detached` to `True` your operation will never have any effect on menu strips attached to a display. Setting `detached` to `True` is typically only used with menu strips that are shown as popup menus using the `PopupMenu()` function. It's impossible to address display menu strips when setting `detached` to `True` because a single menu strip can be attached to multiple displays.

On AmigaOS 4 you can also pass the special value of 0 for `id`. In that case, the context menu of the docky will be used.

Please note that radio groups can only have one active item at a time: This means that if you select a new radio menu item using `SelectMenuItem()`, the previously selected radio menu item will automatically be deselected.

INPUTS

`id`	identifier of a display or a menu strip (see above)
`item$`	identifier of the item inside the menu strip
`detached`	optional: `False` if `id` specifies a display, `True` if it specifies a menu strip object (defaults to `False`) (V10.0)

40 Mobile support library

40.1 CallJavaMethod

NAME

CallJavaMethod – call method of Java activity (V8.0)

SYNOPSIS

```
[ret] = CallJavaMethod(name$[, t, type1, value1, type2, value2, ...])
```

PLATFORMS

Android only

FUNCTION

This is a powerful function that allows you to call directly into the Java code of Hollywood's Android activity. The Java code may then access the whole Android API to enhance your app with custom features unavailable in Hollywood.

You have to pass the name of the method to call in the `name$` argument. Note that Java is a case-sensitive language so the method name you pass in `name$` must exactly match its definition in the Java code.

Optionally, you can pass a table in the second argument. This table currently supports the following tags:

`Static:` Set this to `True` if the method you'd like to call is static. By default, `CallJavaMethod()` expects the method to be non-static.

`ReturnType:`
Set this tag to configure the return data type for the method passed in `name$`. This must be one of the following predefined constants:

`#BYTE:` Java's `byte` data type, a signed 8-bit quantity.

`#SHORT:` Java's `short` data type, a signed 16-bit quantity.

`#INTEGER:`
Java's `int` data type, a signed 32-bit quantity.

`#FLOAT:` Java's `float` data type, a 32-bit floating point number.

`#DOUBLE:` Java's `double` data type, a 64-bit floating point number.

`#BOOLEAN:`
Java's `boolean` data type, a boolean value (`True` or `False`).

`#STRING:` Java's `String` data type, a text string.

`#VOID:` No return value.

This tag defaults to `#VOID`, i.e. the method doesn't return any value.

`ReturnArray:`
If this tag is set to `True`, the method passed in `name$` is expected to return an array of the data type specified in `ReturnType`. Note that if `ReturnArray` is set to `True`, `ReturnType` must not be set to `#VOID`. Defaults to `False`.

After the optional table argument, `CallJavaMethod()` accepts an unlimited number of `type` and `value` pairs. These pairs can be used to pass parameters to the method specified in `name$`. For each `type` there must be a corresponding `value` argument directly after it.

The following predefined constants are currently supported for the `type` argument:

`#BYTE:` Java's `byte` data type, a signed 8-bit quantity.

`#SHORT:` Java's `short` data type, a signed 16-bit quantity.

`#INTEGER:`
Java's `int` data type, a signed 32-bit quantity.

`#FLOAT:` Java's `float` data type, a 32-bit floating point number.

`#DOUBLE:` Java's `double` data type, a 64-bit floating point number.

`#BOOLEAN:`
Java's `boolean` data type, a boolean value (`True` or `False`).

`#STRING:` Java's `String` data type, a text string.

The value that follows each `type` argument must correspond to the type specified for that parameter, e.g. if you pass `#STRING` in a `type` argument, a string must follow after the `#STRING` argument.

Here is an example Java method:

```
public int littleTest(String s, int v) {
    Log.v("Test", "Got data: " + s + " " + v);
    return 50;
}
```

Note that it is important to declare the method using the `public` keyword because it is accessed from outside its class. To call the Java method `littleTest()` from your Hollywood script using `CallJavaMethod()`, you'd have to use the following code:

```
r = CallJavaMethod("littleTest", {ReturnType = #INTEGER},
        #STRING, "Hello Java!", #INTEGER, 10)
```

Since we have declared the Java method as returning an integer value and its implementation on the Java side returns 50, the Hollywood variable `r` will be set to 50 as soon as `CallJavaMethod()` returns.

Note that this function can only be used in connection with the Hollywood APK Compiler because the Hollywood Player doesn't allow you to inject any custom code into its activity. This is only supported by the Hollywood APK Compiler.

The Java methods you declare will all be part of a subclass of Hollywood's Android `Activity`. Thus, you can call any of the `Activity` methods directly from the methods called by `CallJavaMethod()`. Keep in mind, though, that Java methods run by `CallJavaMethod()` will not be executed on the main (UI) thread but on Hollywood's VM thread. So if you need to access Android APIs that can only be called from the UI thread (like most of the `View-related` APIs), you must first delegate from the Hollywood thread to the main thread or the code won't work.

INPUTS

`name$` name of method to call

`t`	optional: table configuring further options (see above)
`type1`	optional: type of first parameter to pass to method (see above for possible values)
`value1`	optional: actual value of first parameter
`...`	optional: unlimited number of method parameters

RESULTS

`ret`	optional: in case `ReturnType` is not `#VOID`, the value returned by the Java method

40.2 GetAsset

NAME

GetAsset – get handle to Android asset (V6.1)

SYNOPSIS

```
handle$ = GetAsset(f$)
```

FUNCTION

This function can be used to get a handle to an Android asset linked to an APK file compiled by the Hollywood APK Compiler, which is an external add-on for Hollywood. The string returned by this function can then be passed to all Hollywood functions that deal with files, e.g. `LoadBrush()` or `OpenMusic()`. Keep in mind, though, that assets are read-only. Trying to write to an asset handle will result in an error.

Note that since Android is based on Linux, asset names are case sensitive. Thus, the name you pass to this function must exactly match the name specified with the Hollywood APK Compiler or Hollywood will report a "File not found" error.

For convenience reasons, `GetAsset()` is also supported by all other Hollywood versions but it simply returns the string passed to the function when used outside APKs generated by the Hollywood APK Compiler.

INPUTS

`f$`	name of the asset to obtain

RESULTS

`handle$`	handle to the asset which can then be passed to all Hollywood functions that deal with files

EXAMPLE

```
LoadBrush(1, GetAsset("test.png"))
```

The code above loads the asset "test.png" into brush number 1. For this code to work you need to link a file named "test.png" (case must match exactly!) to your APK with the Hollywood APK Compiler.

40.3 HideKeyboard

NAME
HideKeyboard – hide the software keyboard (V5.0)

SYNOPSIS
`HideKeyboard()`

PLATFORMS
Mobile platforms only

FUNCTION
This function can be used to hide the host system's software keyboard on mobile devices. As soon as the software keyboard is hidden, the user will no longer be able to trigger any `OnKeyDown` or `OnKeyUp` events.

To show the software keyboard again, use the `ShowKeyboard()` function.

INPUTS
none

40.4 PerformSelector

NAME
PerformSelector – perform selector (V7.0)

SYNOPSIS
`[ret] = PerformSelector(s$[, ...])`

PLATFORMS
iOS only

FUNCTION
This function can be used to perform a selector of your application's delegate, i.e. it allows you to make calls from your Hollywood script into native code. The name of the selector has to be passed as a string in `s$`.

The selector specified by `s$` will then be run with an `NSMutableArray` as its sole argument. Inside that array, index 1 will contain the `lua_State` and index 2 will contain a pointer to a `hwPluginAPI` structure, allowing you to access all public APIs and especially the Lua VM. Indices 3 and 4 contain the `UIViewController` and `UIView`, respectively. Upon return, you must set index 0 to an `NSValue` containing an `int` which specifies the return code of your function. This is all very similar to the way functions in Hollywood plugins are executed. So please see the Hollywood SDK documentation for more details (especially the chapters concerning writing library plugins).

You need to implement the desired selector in your application's delegate in native code. This is what a custom selector might look like in Objective C:

```
- (void)MyTestSelector:(NSMutableArray *) args
{
   // get essential pointers from Hollywood
   lua_State *L = (lua_State *)
```

```
        [((NSValue *) [args objectAtIndex:1]) pointerValue];
    hwPluginAPI *hwcl = (hwPluginAPI *)
        [((NSValue *) [args objectAtIndex:2]) pointerValue];

    // we return 1 because we push one string
    int retval = 1;

    // print string at stack index 2
    printf("%s\n", hwcl->LuaBase->luaL_checklstring(L, 2, NULL));

    // push return value
    hwcl->LuaBase->lua_pushstring(L, "Test return value");

    // set return value
    [args replaceObjectAtIndex:0 withObject:[NSValue value:&retval
        withObjCType:@encode(int*)]];
}
```

This selector will print the string argument that is passed to the `PerformSelector()` call in argument 2. It will then return the string "Test return value" to the Hollywood script. You could run this selector from your Hollywood script like this:

```
DebugPrint(PerformSelector("MyTestSelector", "Test"))
```

This code will pass the string "Test" to the `MyTestSelector` method. `DebugPrint()` will print "Test return value" because that is the return value of `MyTestSelector`.

Keep in mind that your selector function will not be run on the main (UI) thread but on Hollywood's VM thread. So when accessing UIKit functionality (or other frameworks that need to run on the main thread) you need to delegate the respective code to the main thread first.

INPUTS

`s$` name of selector to run

RESULTS

`ret` optional: return values of your selector function

40.5 ShowKeyboard

NAME

ShowKeyboard – show the software keyboard (V5.0)

SYNOPSIS

`ShowKeyboard()`

PLATFORMS

Mobile platforms only

FUNCTION

This function can be used to show the host system's software keyboard on mobile devices. As soon as the software keyboard is visible, the user will be able to enter text that

will then be sent to your script in the form of `OnKeyDown` and `OnKeyUp` events. Thus, you should install the appropriate event handlers using `InstallEventHandler()` before calling `ShowKeyboard()`.

To hide the software keyboard, use the `HideKeyboard()` function.

INPUTS

none

40.6 ShowToast

NAME

ShowToast – show a short message (V5.3)

SYNOPSIS

```
ShowToast(s$[, x, y, long])
```

PLATFORMS

Mobile platforms only

FUNCTION

This function can be used to show a short message (a so-called "toast") that disappears automatically after a certain period of time. You have to pass the message to display in the string argument `s$`. The optional arguments allow you to specify the desired position of the message on the screen and whether the presentation time should be long or short. You can also use Hollywood's special coordinate constants in the `x` and `y` arguments.

INPUTS

`s$`	message to show
`x`	optional: x position for the message
`y`	optional: y position for the message
`long`	optional: whether or not the presentation duration should be long or short (defaults to `False` which means a short presentation duration)

EXAMPLE

```
ShowToast("Hello World!", #CENTER, #CENTER)
```

The code above shows the message "Hello World!" in the center of the screen and hides it automatically after a short period of time.

40.7 Vibrate

NAME

Vibrate – vibrate device (V8.0)

SYNOPSIS

```
Vibrate(ms)
```

PLATFORMS

Android only

FUNCTION

This function can be used to vibrate the device for the duration specified by `ms`. The duration must be specified in milliseconds.

INPUTS

`ms`	duration in milliseconds to vibrate

EXAMPLE

```
Vibrate(1000)
```

The code above will vibrate the device for one second.

41 Mouse pointer library

41.1 CreatePointer

NAME

CreatePointer – create a new mouse pointer (V4.0)

SYNOPSIS

```
[id] = CreatePointer(id, type, ...)
[id] = CreatePointer(id, #SPRITE, srcid[, frame, spotx, spoty])
[id] = CreatePointer(id, #BRUSH, srcid[, spotx, spoty])
[id] = CreatePointer(id, #POINTER, ptrtype)
```

FUNCTION

This function creates a new mouse pointer and assigns the identifier `id` to it. If you pass `Nil` in `id`, `CreatePointer()` will automatically choose an identifier and return it. The mouse pointer object created by this function can be displayed later by calling the `SetPointer()` function. Mouse pointers can be created either from a sprite or brush source, or you can choose a predefined mouse pointer. The calling convention of `CreatePointer()` depends on the type you specify as the second argument.

For the types `#SPRITE` and `#BRUSH` you have to specify the identifier of the object that shall be used as the source for the pointer graphics. The mouse pointer created by this function is independent of the source object, so you can free the source object after calling `CreatePointer()`.

If you specify `#POINTER` as the type, you have to provide an additional argument that defines which predefined mouse pointer image you want to obtain. Currently, this can be `#STDPTR_SYSTEM` for the system's standard pointer and `#STDPTR_BUSY` for the system's standard wait pointer.

The spotx and spoty arguments specify the hot spot inside the mouse pointer. The hot spot is the mouse pointer's pixel that is used to click. If the mouse pointer image is an arrow, then the hot spot is usually exactly at the tip of the arrow. If you do not specify the spotx & spoty arguments, `CreatePointer()` will use the center of the image as the hot spot.

Please note that not all systems can handle true colour mouse pointers. If the system does not support true colour mouse pointers, Hollywood will reduce the colors. Also, your image data might get scaled because some systems impose limits on the maximum mouse pointer size.

On AmigaOS 3, `CreatePointer()` also supports palette brushes and sprites. Pointers on classic Amiga hardware are always palette-based because they are implemented using hardware sprites so if you pass palette brushes or sprites to `CreatePointer()` on AmigaOS 3 you have full control over the exact pens used by the pointer which is more convenient than using 32-bit graphics because those will first have to be mapped down to palette graphics on AmigaOS 3 and you won't have any control over the palette pens in the remapped brush or sprite.

INPUTS

`id` id for the mouse pointer or `Nil` for auto id selection

`type` type from which to take the source data

`...` further arguments depend on the type specified (see above)

RESULTS

`id` optional: identifier of the mouse pointer; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreatePointer(1, #BRUSH, 2, 0, 0)
SetPointer(1)
```

The code above creates a new mouse pointer 1 from the brush with the id 2. The hot spot will be at position 0:0 (i.e. the top-left corner of the image). After creating the mouse pointer it will be displayed using SetPointer().

41.2 FreePointer

NAME

FreePointer – free a mouse pointer (V4.0)

SYNOPSIS

```
FreePointer(id)
```

FUNCTION

This function frees the mouse pointer specified by `id`. The mouse pointer must have been created previously using `CreatePointer()`. Please note that the mouse pointer must not be currently active. You may only free mouse pointers that are not displayed currently.

INPUTS

`id` identifier of the mouse pointer to be freed

41.3 HidePointer

NAME

HidePointer – hide the mouse pointer in the current display

SYNOPSIS

```
HidePointer()
```

FUNCTION

This function hides the mouse pointer. Use this command with care because it might confuse the user. You can bring the pointer back to the display using `ShowPointer()`.

Please note that every display has its private pointer setting. Thus, this function will only hide the mouse pointer in the current display. If the user activates another display, the mouse pointer will be visible again.

INPUTS

none

41.4 MovePointer

NAME
MovePointer – move the mouse pointer

SYNOPSIS
`MovePointer(x, y)`

FUNCTION
This function moves the mouse pointer to the location specified by `x` and `y`. Use this function with care because it might confuse the user.

INPUTS

`x`	desired new x position of the pointer
`y`	desired new y position of the pointer

EXAMPLE
```
MovePointer(#CENTER, #CENTER)
```
The above code moves the pointer to the center of your display.

41.5 SetPointer

NAME
SetPointer – change mouse pointer of current display (V4.0)

SYNOPSIS
`SetPointer(id)`

FUNCTION
This function displays the mouse pointer specified by `id`. The mouse pointer must have been previously created by `CreatePointer()`.

Please note that every display has its private pointer setting. Thus, this function will only set the mouse pointer in the current display. If you want to change the mouse pointer of all your displays, you need to call `SetPointer()` for each of them (after making each active using `SelectDisplay()`).

Please note: This function has already been available since version 1.5 but its functionality changed completely in version 4.0. The old command is no longer supported.

INPUTS

`id`	identifier of the mouse pointer to be displayed

EXAMPLE
See Section 41.1 [CreatePointer], page 817.

41.6 ShowPointer

NAME

ShowPointer – show the mouse pointer in the current display

SYNOPSIS

`ShowPointer()`

FUNCTION

This function brings the mouse pointer back after it has been hidden using `HidePointer()`.

INPUTS

none

42 Network library

42.1 CloseConnection

NAME

CloseConnection – disconnect from server (V5.0)

SYNOPSIS

```
CloseConnection(id)
```

FUNCTION

This function disconnects from the server specified in `id` and closes the connection. The connection which you need to specify here must have been established by the `OpenConnection()` command.

INPUTS

`id`	identifier of the connection that shall be terminated

42.2 CloseServer

NAME

CloseServer – shutdown an existing server (V5.0)

SYNOPSIS

```
CloseServer(id)
```

FUNCTION

This function shuts down the server specified by the identifier argument. The server you pass here must have been created by the `CreateServer()` command.

It is important that you disconnect all clients from your server using the `CloseConnection()` command before you call `CloseServer()`.

INPUTS

`id`	identifier of the server to shut down

42.3 CloseUDPObject

NAME

CloseUDPObject – close existing UDP object (V5.0)

SYNOPSIS

```
CloseUDPObject(id)
```

FUNCTION

This function closes the UDP object specified in the identifier argument. The UDP object you pass here must have been created earlier by the command `CreateUDPObject()`.

INPUTS

`id`	identifier of the UDP object to close

42.4 CreateServer

NAME

CreateServer – open a new server (V5.0)

SYNOPSIS

```
[id] = CreateServer(id[, port, ip$, backlog, protocol])
```

FUNCTION

This command can be used to establish a new server that is ready to take incoming connections on the local host. The optional argument `port` can be used to specify on which port the server should be opened. If you do not specify this argument, `CreateServer()` will choose a vacant port automatically, and you can use `GetLocalPort()` later to find out the port number of the server. In the first argument, you need to pass an identifier which is needed to refer to this server later on. Alternatively, you can pass `Nil` as the first argument. In that case, `CreateServer()` will select an identifier automatically and return it to you.

The optional argument `ip$` can be used to specify a local IP address the server should be bound to. This defaults to `"*"`, which means that the server will accept connections from the whole network. You can also specify `"127.0.0.1"` (or `"::1"` in IPv6) to allow only connections from the local host. Note that passing `"*"`, which is also the default, might trigger the firewall on some configurations.

The optional `backlog` argument can be used to specify the maximum number of client connections that can be added to the queue. This defaults to 32.

Starting with Hollywood 8.0 there is an optional new `protocol` argument which allows you to specify the Internet protocol that should be used when creating the server. This can be one of the following special constants:

`#IPV4`: Use Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6`: Use Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO`: Let the host operating system determine the Internet protocol to use. You can then use `GetLocalProtocol()` to find out which Internet protocol the host operating system has chosen for this server. See Section 42.14 [GetLocalProtocol], page 835, for details.

The `protocol` argument defaults to the default protocol type set using `SetNetworkProtocol()`. By default, this is `#IPV4` due to historical and portability reasons. See Section 42.23 [SetNetworkProtocol], page 845, for details.

Once the server has been successfully established, you must use the function `InstallEventHandler()` to listen to the events `OnConnect`, `OnDisconnect`, and `OnReceiveData`. These events will inform you whenever a new client tries to connect to your server, or when a client sends new data (i.e. commands that you need to handle) to your server.

INPUTS

`id`	identifier for the new server or `Nil` for auto id selection
`port`	optional: port to open up for this server, or 0 for automatic port selection (defaults to 0)
`ip$`	optional: local IP address to bind server to (defaults to `"*"`)
`backlog`	optional: maximum number of connections that can be queued (defaults to 32)
`protocol`	optional: Internet protocol to use (see above for possible values); defaults to the protocol type set using `SetNetworkProtocol()` (V8.0)

RESULTS

`id`	optional: identifier of the new server; this will only be returned when you pass `Nil` as argument 1 (see above)

42.5 CreateUDPObject

NAME

CreateUDPObject – create a new UDP object (V5.0)

SYNOPSIS

```
[id] = CreateUDPObject(id[, port, ip$, mode, protocol])
```

FUNCTION

This command can be used to create a new UDP object that can receive data and send data to other network participants. The optional argument `port` can be used to specify at which local port the UDP object should be created. If you do not specify this argument, `CreateUDPObject()` will choose a vacant port automatically, and you can use `GetLocalPort()` later to find out the port number of the UDP object. In the first argument, you need to pass an identifier which is needed to refer to this UDP object later on. Alternatively, you can pass `Nil` as the first argument. In that case, `CreateUDPObject()` will select an identifier automatically and return it to you.

Starting with Hollywood 8.0 there is a new `mode` argument which allows you to specify the type of UDP object that should be created for you. This can be one of the following predefined values:

`#UDPSERVER:`

Create an UDP server object. In that case, the optional argument `ip$` can be used to specify a local IP address the UDP object should be bound to. This defaults to `"*"`, which means that the server will accept connections from the whole network. You can also specify `"127.0.0.1"` (or `"::1"` in IPv6) to allow only connections from the local host. Note that passing `"*"`, which is also the default, might trigger the firewall on some configurations. `#UDPSERVER` is the default mode for `CreateUDPObject()`.

`#UDPCLIENT:`

Create a client UDP object. In that case, the optional arguments `ip$` and `port` must be specified to specify the server the client should be connected to.

If your UDP object always sends data to the same server, it is recommended to create it as a `#UDPCLIENT` object because `SendUDPData()` is faster then. (V8.0)

`#UDPNONE:`

Create an unconnected UDP object. In that case, the optional arguments `ip$` and `port` are ignored. Unconnected UDP objects are useful if you need to send data to different servers without being able to receive data on your own. (V8.0)

Additionally, Hollywood 8.0 introduces an optional new `protocol` argument which allows you to specify the Internet protocol that should be used by the UDP object. This can be one of the following special constants:

`#IPV4:` Use Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Use Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` Let the host operating system determine the Internet protocol to use. You can then use `GetLocalProtocol()` or `GetConnectionProtocol()` to find out which Internet protocol the host operating system has chosen for the UDP object

The `protocol` argument defaults to the default protocol type set using `SetNetworkProtocol()`. By default, this is `#IPV4` due to historical and portability reasons. See Section 42.23 [SetNetworkProtocol], page 845, for details.

Once the UDP object is created, you can use the commands `SendUDPData()`, `ReceiveUDPData()`, and the OnReceiveUDPData event handler to communicate with other systems in the network.

Please note that UDP is an unreliable transfer protocol. It is faster than the TCP protocol but data may arrive incompletely or out of order. Thus, it is only suitable for purposes that do not depend on the integrity of the transferred data.

INPUTS

`id` identifier for the new UDP object or `Nil` for auto id selection

`port` optional: port for this UDP object, or 0 for automatic port selection (defaults to 0); must be set for `#UDPCLIENT`

`ip$` optional: IP address to bind server to (for `#UDPSERVER`) or to connect UDP object to (for `#UDPCLIENT`) (defaults to `"*"` for `#UDPSERVER`); must be set for `#UDPCLIENT`

`mode` optional: desired mode for this UDP object (see above for possible values) (V8.0)

`protocol` optional: Internet protocol to use (see above for possible values); defaults to the protocol type set using `SetNetworkProtocol()` (V8.0)

RESULTS

`id` optional: identifier of the new UDP object; this will only be returned when you pass `Nil` as argument 1 (see above)

42.6 DownloadFile

NAME

DownloadFile – download file via HTTP, FTP or other protocol (V5.0)

SYNOPSIS

```
data$, count = DownloadFile(url$[, options, func, userdata])
```

FUNCTION

This command allows you to conveniently download a file from a network server. By default, HTTP and FTP servers are supported but Hollywood plugins may provide support for additional protocols. You have to pass the URL of the file in the `url$` argument. `DownloadFile()` will then download the file and return it as a string. Storing binary data inside strings is possible because Hollywood strings are not limited to printable characters. Instead, they can also contain control characters and the `NULL` character. The second return value indicates the size of the downloaded file in bytes.

Alternatively, you can also set the `File` tag in the optional table argument named `options` to a filename. In that case, the downloaded file won't be returned as a string but it will be saved as the file specified in the `File` tag. This is recommended for bigger files because strings are obviously stored in memory so downloading a large file to a string is generally not a good idea because it will require lots of memory.

The URL passed in `url$` must begin with a protocol prefix like http:// or ftp://, and it must not contain any escaped characters. Escaping will be done by `DownloadFile()` so make sure that you pass only unescaped URLs, e.g. passing `"http://www.mysite.net/cool%20file.html"` will not work. You must specify an URL without escaped characters, so the correct version would be: `"http://www.mysite.net/cool file.html"`. If you want to pass a URL that has already been escaped, you have to set the `Encoded` tag to `True` (see below). In that case, `DownloadFile()` won't do any further escaping on your URL.

`DownloadFile()` also supports authentification for the HTTP and FTP protocols. In that case, username and password have to be passed after the protocol identifier in the form `username:password`, followed by an @ character and the server. Here is an example for user `"joe"` and password `"secret"`: http://joe:secret@www.test.net/private/files.lha. Note that HTTP authentification support was not available before Hollywood 6.0. When downloading from an FTP server, `"anonymous"` is used as the default username and `"anonymous@anonymous.org"` as the default password. If you want to use a different login account, you have to pass the username/password combination in the URL, for example: ftp://joe:secret@ftp.test.net/pub/files.lha.

The URL you pass to this function can also contain a port number. If you want to specify a port number, you have to put it behind the host name and separate it using a colon. Here is an example for using port 1234: http://www.test.com:1234/test/image.jpg. If

no port is specified, `DownloadFile()` will use port 80 for HTTP servers and port 21 for FTP servers.

The second argument can be used to specify further options for the download operation. It is a table that recognizes the following tags:

`File:` If you specify a filename in this table tag, `DownloadFile()` will stream the downloaded data directly to this file instead of returning it as a string. This is useful for very large files on the one hand, but it is also useful for other files because it saves you the hassle of having to save the string data manually to a file and set it to `Nil` afterwards. Therefore, if you are going to save the downloaded string to a file anyway, it is more efficient to use this tag. If you decide not to use this tag, please also read below for some important information when downloading files to strings.

`TransferMode:`
This tag is only supported when downloading files from an FTP source. In that case, you can use this tag to specify whether `DownloadFile()` should transfer the file in ASCII or in binary mode. For ASCII mode, specify `#FTPASCII` here. For binary mode, use `#FTPBINARY`. The default transfer mode is `#FTPBINARY`.

`Proxy:` This tag is only supported when downloading files from an HTTP source. In that case, you can specify a server here that should act as a proxy server for the connection.

`Fail404:` This tag specifies whether or not `DownloadFile()` should fail with a "file not found" error when you pass a URL that points to a non-existent file. Normally, when you request a non-existent file, HTTP servers will generate a special HTML page with a "404 - file not found" error, and send that to you instead. So you will always be getting a file even if you are requesting a non-existent file. If you do not want this behaviour, set this tag to `True`. In that case, `DownloadFile()` will fail when requesting an invalid file and you will not get any 404 error page. By default, this tag is set to `False` which means that an error page is generated. This tag is only supported for downloading files from an HTTP source.

`SilentFail:`
If you set this tag to `True`, `DownloadFile()` will never throw an error but simply exit silently and return an error message in the first return value, and -1 in the second return value to indicate that an error has happened. If it is set to `False`, `DownloadFile()` will throw a system error for all errors that occur. Defaults to `False`.

`Redirect:`
Specifies whether or not the web server is allowed to redirect you to a new URL. This defaults to `True` which means that redirection is allowed. This tag is only supported when downloading files from an HTTP source.

`Post:` This tag is only supported when working with a HTTP server. If this tag is specified, `DownloadFile()` will send a POST request to the HTTP server instead of a GET request. A POST request has the advantage that you can

attach additional data to your request. Thus, it is often used for submitting the contents of web forms, or for uploading files via HTTP. The data that shall be attached to the POST request must be specified in this tag as a string. You can set the type of the data by using the `PostType` tag (see below).

`PostType:`
This tag is only handled when the `Post` tag was also specified. If that is the case, `PostType` specifies the type of data inside the `Post` tag. The type must be passed as a MIME content type string. This tag defaults to `"application/x-www-form-urlencoded"` which is the MIME type used for submitting the contents of web forms to Perl (CGI) scripts.

`UserAgent:`
This tag allows you to change the user agent that `DownloadFile()` sends to the target server. This is useful with servers that refuse to cooperate with unknown user agents. By default, `DownloadFile()` will send `"Hollywood"` in the user agent field of HTTP requests. This tag is only supported for downloading files from an HTTP source. (V5.2)

`CustomHeaders:`
This tag allows you to specify a string of custom headers that should be sent to the HTTP server when making the request. This can be useful for some fine-tuned adjustments for some servers. Keep in mind that the individual header elements have to be terminated by a carriage return and a line feed. This tag is only supported when using the HTTP protocol. (V6.0)

`Encoded:` Set this tag to `True` if the URL you passed to this function has already been correctly escaped. If this tag is set to `True`, `DownloadFile()` won't escape any characters. Instead, it expects you to pass a URL that has already been correctly escaped so that it can be directly used for server requests without any additional escaping. (V6.1)

`Protocol:`
This tag can be used to specify the Internet protocol that should be used when opening the connection. This can be one of the following special constants:

`#IPV4:` Use Internet Protocol version 4 (IPv4).

`#IPV6:` Use Internet Protocol version 6 (IPv6). Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` Let the host operating system determine the Internet protocol to use.

This tag defaults to the default protocol type set using `SetNetworkProtocol()`. By default, this is `#IPV4` due to historical and portability reasons. See Section 42.23 [SetNetworkProtocol], page 845, for details. (V8.0)

`Adapter:` This tag allows you to specify one or more network adapters that should be asked to establish the specified connection. This must be set to a string

containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V8.0)

`SSL:` Set this tag to `True` to request a connection through TLS/SSL encryption. Note that setting this tag when using Hollywood's inbuilt network adapter doesn't have any effect because Hollywood's inbuilt network adapter doesn't support TLS/SSL connections. However, there might be a network adapter provided by a plugin that supports TLS/SSL and if you set this tag to `True` Hollywood will forward your wish to have a TLS/SSL connection to the network adapter provided by the plugin. Do note, though, that you normally don't have to set this tag in case the URL's scheme already indicates an SSL connection by using a prefix such as `"https://"` or `"ftps://"`. (V8.0)

`Async:` If this is set to `True`, `DownloadFile()` will operate in asynchronous mode. This means that it will return immediately, passing an asynchronous operation handle to you. You can then use this asynchronous operation handle to finish the operation by repeatedly calling `ContinueAsyncOperation()` until it returns `True`. This is very useful in case your script needs to do something else while the operation is in progress, e.g. displaying a status animation or something similar. By putting `DownloadFile()` into asynchronous mode, it is easily possible for your script to do something else while the operation is being processed. See Section 19.4 [ContinueAsyncOperation], page 224, for details. Defaults to `False`. (V9.0)

`Verbose:` This tag can be set to `True` to request detailed log information about the connection and the protocol interaction with the server. This is currently only used by Hollywood plugins so if you use Hollywood's internal network adapter, setting this tag to `True` has no effect. Plugins, however, may choose to provide extended connection information when this tag has been set to `True`. Defaults to `False`. (V9.0)

`FileAdapter:`
This tag is only used if the `File` tag is set as well. In that case, `FileAdapter` allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to `default`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file and network adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

The optional parameter `func` can be used to pass a callback function which will be called from time to time by `DownloadFile()` so you can update a progress bar for example. The callback function you specify here will be called with a single argument: A table that contains more information. Here is an overview of the table fields that will be initialized before `DownloadFile()` runs your callback function:

`Action:` `#DOWNLOADFILE_STATUS`

`Count:` Contains the number of bytes that have already been downloaded.

`Total:` Contains the size of the file being downloaded.

`UserData:`
Contains the value you passed in the `userdata` argument.

The callback function of type `#DOWNLOADFILE_STATUS` should normally return `False`. If it returns `True`, the download operation will be aborted.

Finally, there is a fourth optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

If you are downloading to a string, you can use the `StringToFile()` shortcut function to convert the string returned by `DownloadFile()` to a file. Alternatively, you could use the `DefineVirtualFileFromString()` function to create a virtual file from a string source. This can be useful, for example, when you download an image file that you want to load into Hollywood using `LoadBrush()`. By using `DefineVirtualFileFromString()` you can load this file directly into Hollywood without having to save it to a temporary file first.

Important note: Make sure that you set the string returned by this function to `Nil` when you no longer need it. By setting the string to `Nil`, you signal to the Hollywood garbage collector that you no longer need this string and that its memory can be freed. This is especially important for large files. If you download a large file, save it to disk and do not set its string to `Nil`, you will waste a lot of memory. So make sure to be careful with strings returned by `DownloadFile()`.

INPUTS

`url$` URL to file that shall be downloaded

`options` optional: a table containing further options for this download

`func` optional: a callback function that shall be called from time to time

`userdata` optional: user defined data that should be passed to callback function

RESULTS

`data$` the data that was read from the network buffer or an empty string if the `File` tag was specified in the options table

`count` number of bytes successfully transmitted

EXAMPLE

```
DownloadFile("http://www.airsoftsoftwair.de/images/products/" ..
             "hollywood/47_shot1.jpg", {File = "47_shot1.jpg"})
```

The code above downloads the specified file and saves it as "47_shot1.jpg" to the current directory.

```
DownloadFile("http://www.<your server>.com/cgi-bin/formmailer.cgi",
             {Post = "sender=Hollywood&mail=me@hollywood-mal.de" ..
             "&message=Hello from Hollywood!"}))
```

The code above shows to invoke a CGI script using `DownloadFile()`. The data specified in the `Post` tag will be passed to the HTTP server using the POST method.

```
DownloadFile("http://www.hollywood-mal.com/index.html", {
             File = "index.html",
             CustomHeaders = "Accept-Encoding: gzip, deflate\r\n"})
```

The code above downloads the specified file and sends a custom header to tell the server that it can also send the file as a gzip or flate compressed file.

```
@REQUIRE "hurl"
...
DownloadFile("https://www.hollywood-mal.com/index.html", {
             File = "index.html", Adapter = "hurl"})
```

The code above downloads a file using the HTTPS protocol. Since Hollywood doesn't support SSL/TLS by default, this code uses the hURL plugin for the operation because hURL supports SSL/TLS. hURL is activated by passing `hurl` in the `Adapter` tag.

42.7 GetConnectionIP

NAME

GetConnectionIP – get IP address of remote side (V5.0)

SYNOPSIS

```
ip$ = GetConnectionIP(id[, type])
```

FUNCTION

This command returns the IP address of the connection object specified in `id`. This can either be the identifier of a server connection obtained by a call to `OpenConnection()`, the identifier of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events using `InstallEventHandler()`, or it can be the identifier of a UDP object created by `CreateUDPObject()`. The IP address of the remote side is returned as a string by this function.

The optional argument `type` specifies the type of the network object passed in argument 1. The following types are currently supported by this function:

`#NETWORKCONNECTION:`
: Query the IP of a connection obtained by a call to `OpenConnection()`, or the IP of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events that can be installed using the command `InstallEventHandler()`.

`#NETWORKUDP:`
: Query the IP of a UDP object created using the `CreateUDPObject()` call.

If you omit the optional `type` argument, it will default to type `#NETWORKCONNECTION`.

INPUTS

`id` connection object to query

`type` optional: type of network object to query (defaults to `#NETWORKCONNECTION`) (V8.0)

RESULTS

`ip$` IP address of the remote side of the connection as a string

EXAMPLE

```
OpenConnection(1, "www.airsoftsoftwair.de", 80)
DebugPrint(GetConnectionIP(1), GetConnectionPort(80))
CloseConnection(1)
```

The code above connects to www.airsoftsoftwair.de port 80 and then obtains the IP address of this server.

42.8 GetConnectionPort

NAME

GetConnectionPort – get port number of remote side (V5.0)

SYNOPSIS

```
port = GetConnectionPort(id[, type])
```

FUNCTION

This command returns the port number of the connection object specified in `id`. This can either be the identifier of a server connection obtained by a call to `OpenConnection()`, the identifier of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events using `InstallEventHandler()`, or it can be the identifier of a UDP object created by `CreateUDPObject()`.

The optional argument `type` specifies the type of the network object passed in argument 1. The following types are currently supported by this function:

`#NETWORKCONNECTION:`
: Query the port of a connection obtained by a call to `OpenConnection()`, or the IP of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events that can be installed using the command `InstallEventHandler()`.

`#NETWORKUDP:`
: Query the port of a UDP object created using the `CreateUDPObject()` call.

If you omit the optional `type` argument, it will default to type `#NETWORKCONNECTION`.

INPUTS

`id` connection object to query

`type` optional: type of network object to query (defaults to `#NETWORKCONNECTION`) (V8.0)

RESULTS

`port` port number of the remote side of the connection

EXAMPLE

See Section 42.7 [GetConnectionIP], page 830.

42.9 GetConnectionProtocol

NAME

GetConnectionProtocol – get protocol of remote side (V8.0)

SYNOPSIS

```
protocol = GetConnectionProtocol(id[, type])
```

FUNCTION

This command returns the Internet protocol of the connection object specified in `id`. This can either be the identifier of a server connection obtained by a call to `OpenConnection()`, the identifier of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events using `InstallEventHandler()`, or it can be the identifier of a UDP object created by `CreateUDPObject()`.

The optional argument `type` specifies the type of the network object passed in argument 1. The following types are currently supported by this function:

`#NETWORKCONNECTION:`
: Query the protocol of a connection obtained by a call to `OpenConnection()`, or the protocol of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events that can be installed using the command `InstallEventHandler()`.

`#NETWORKUDP:`
: Query the protocol of a UDP object created using the `CreateUDPObject()` call.

If you omit the optional `type` argument, it will default to type `#NETWORKCONNECTION`.

The return value will be one of the following predefined constants:

`#IPV4:` Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` The host system hasn't decided on a protocol for this network object yet.

`#IPUNKNOWN:`
: Network object uses an unknown protocol.

INPUTS

`id` connection object to query

`type` optional: type of network object to query (defaults to `#NETWORKCONNECTION`)

RESULTS

`protocol` protocol of remote side of the connection

42.10 GetHostName

NAME

GetHostName – return standard host name of machine (V5.0)

SYNOPSIS

```
host$ = GetHostName()
```

FUNCTION

This function returns the standard host name of the machine that Hollywood is currently running on.

INPUTS

none

RESULTS

`host$` standard host name of machine

42.11 GetLocalInterfaces

NAME

GetLocalInterfaces – get local interfaces (V9.0)

SYNOPSIS

```
t = GetLocalInterfaces([linklocal])
```

FUNCTION

This function returns a list of all network interfaces that are currently available. This allows you to conveniently determine a system's local IP address. If the optional `linklocal` argument is set to `True`, link-local addresses will be included as well.

`GetLocalInterfaces()` will return a table that contains a number of subtables, each describing a local interface. The following fields will be initialized in each subtable:

`Name:` The name of the interface.

`Address:` The address of the interface. Depending on the setting of the `Protocol` tag (see below), this may be either an IPv4 or IPv6 address.

`Protocol:`

The protocol of the interface. This will be one of the following predefined constants:

`#IPV4:` Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

INPUTS

`linklocal`
optional: `True` to include link-local addresses in the return table, `False` to exclude them (defaults to `False`)

RESULTS

`t` table containing a list of all local interfaces

EXAMPLE

```
t = GetLocalInterfaces()
For Local k = 0 To ListItems(t) - 1
   NPrint(t[k].Name, t[k].Address, t[k].Protocol)
Next
```

The code above prints information about all available network interfaces.

42.12 GetLocalIP

NAME

GetLocalIP – get IP address of local side (V5.0)

SYNOPSIS

```
ip$ = GetLocalIP(id[, type])
```

FUNCTION

This command returns the IP address on the local side of the network object specified in `id`. The optional argument `type` specifies the type of the network object passed in argument 1. The following types are currently supported by this function:

`#NETWORKCONNECTION:`
Query the local IP of a connection obtained by a call to `OpenConnection()`, or the local IP of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events that can be installed using the command `InstallEventHandler()`.

`#NETWORKSERVER:`
Query the local IP of a server established using the `CreateServer()` call.

`#NETWORKUDP:`
Query the local IP of an UDP object created using the `CreateUDPObject()` call.

If you omit the optional `type` argument, it will default to type `#NETWORKCONNECTION`.

INPUTS

`id` network object to query

`type` optional: type of the network object passed in argument 1 (defaults to `#NETWORKCONNECTION`)

RESULTS

`ip$` IP address of the local side of the connection

42.13 GetLocalPort

NAME

GetLocalPort – get port number of local side (V5.0)

SYNOPSIS

```
port = GetLocalPort(id[, type])
```

FUNCTION

This command returns the port number on the local side of the network object specified in `id`. The optional argument `type` specifies the type of the network object passed in argument 1. The following types are currently supported by this function:

`#NETWORKCONNECTION:`
Query the local port of a connection obtained by a call to the `OpenConnection()`, function or the local port of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events using the command `InstallEventHandler()`.

`#NETWORKSERVER:`
Query the local port of a server established using the `CreateServer()` call.

`#NETWORKUDP:`
Query the local port of an UDP object created using the `CreateUDPObject()` call.

If you omit the optional `type` argument, it will default to type `#NETWORKCONNECTION`.

INPUTS

`id` network object to query

`type` optional: type of the network object passed in argument 1 (defaults to `#NETWORKCONNECTION`)

RESULTS

`port` port number of the local side of the connection

42.14 GetLocalProtocol

NAME

GetLocalProtocol – get protocol of local side (V8.0)

SYNOPSIS

```
protocol = GetLocalProtocol(id[, type])
```

FUNCTION

This command returns the Internet protocol on the local side of the network object specified in `id`. The optional argument `type` specifies the type of the network object passed in argument 1. The following types are currently supported by this function:

`#NETWORKCONNECTION:`
: Query the local IP of a connection obtained by a call to `OpenConnection()`, or the local IP of a client connection obtained by listening to the `OnConnect` and `OnReceiveData` events that can be installed using the command `InstallEventHandler()`.

`#NETWORKSERVER:`
: Query the local IP of a server established using the `CreateServer()` call.

`#NETWORKUDP:`
: Query the local IP of an UDP object created using the `CreateUDPObject()` call.

If you omit the optional `type` argument, it will default to type `#NETWORKCONNECTION`.

The return value will be one of the following predefined constants:

`#IPV4:` Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` The host system hasn't decided on a protocol for this network object yet.

`#IPUNKNOWN:`
: Network object uses an unknown protocol.

INPUTS

`id` network object to query

`type` optional: type of the network object passed in argument 1 (defaults to `#NETWORKCONNECTION`)

RESULTS

`protocol` protocol of the local side of the connection (see above for possible return values)

42.15 GetMACAddress

NAME

GetMACAddress – get host system's MAC address (V7.0)

SYNOPSIS

```
addr$ = GetMACAddress()
```

FUNCTION

This function returns the host system's MAC address as 6 octets separated by colons, e.g. `12:34:56:78:9A:BC`.

If the MAC address cannot be obtained, `"Unknown"` is returned.

Note that on AmigaOS 3, the only TCP/IP stack that supports obtaining the MAC address is Roadshow.

INPUTS

none

RESULTS

`addr$` host system's MAC address or `"Unknown"`

42.16 IsOnline

NAME

IsOnline – check if an Internet connection is available (V5.0)

SYNOPSIS

```
bool = IsOnline()
```

FUNCTION

This function can be used to check whether or not an Internet connection is available. It will return `True` if connection to the Internet is possible, and `False` otherwise.

INPUTS

none

RESULTS

`bool` boolean value indicating whether or not Internet is available

42.17 OpenConnection

NAME

OpenConnection – connect to a server (V5.0)

SYNOPSIS

```
[id] = OpenConnection(id, server$, port[, table])
```

FUNCTION

This command can be used to establish a new connection to the server specified in `server$`. This can be either a host name or an IP address directly. The third argument specifies the port at which `OpenConnection()` should try to connect. In the first argument, you need to pass an identifier which is needed to refer to this connection later on. Alternatively, you can pass `Nil` as the first argument. In that case, `OpenConnection()` will select an identifier automatically and return it to you.

Once the connection is successfully established, you can use `SendData()` and `ReceiveData()` to communicate with the server. When you are finished you should call `CloseConnection()` to disconnect from the server.

Please note that scheme prefixes like "http://" or "ftp://" are not part of a server name. These just specify the protocol that is used to communicate with the server. So if you want to connect `http://www.airsoftsoftwair.de` you will have to specify "www.airsoftsoftwair.de" as the server name and 80 as the port because 80 is the standard HTTP port. See below for an example. An exception to this rule might be the case where you use the `Adapter` tag to have a network adapter establish the connection. In that case, the network adapter might ask you to specify a scheme name like "http://" or "ftp://" but it really depends on the network adapter. Hollywood's inbuilt adapter doesn't support scheme prefixes and expects you to specify the IP address or the host name directly.

Starting with Hollywood 8.0, `OpenConnection()` accepts an optional table argument that can be used to specify further options. The following tags are currently recognized by the optional table argument:

`Protocol:`
: This tag allows you to specify the Internet protocol that should be used when opening the connection. This can be one of the following special constants:

  `#IPV4:` Use Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

  `#IPV6:` Use Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

  `#IPAUTO:` Let the host operating system determine the Internet protocol to use. You can then use `GetConnectionProtocol()` to find out which Internet protocol the host operating system has chosen for this server. See Section 42.9 [GetConnectionProtocol], page 832, for details.

  The `Protocol` tag defaults to the default protocol type set using `SetNetworkProtocol()`. By default, this is `#IPV4` due to historical and portability reasons. See Section 42.23 [SetNetworkProtocol], page 845, for details. (V8.0)

`Adapter:` This tag allows you to specify one or more network adapters that should be asked to establish the specified connection. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V8.0)

`SSL:` Set this tag to `True` to request a connection through TLS/SSL encryption. Note that setting this tag when using Hollywood's inbuilt network adapter doesn't have any effect because Hollywood's inbuilt network adapter doesn't support TLS/SSL connections. However, there might be a network adapter provided by a plugin that supports TLS/SSL and if you set this tag to `True`

Hollywood will forward your wish to have a TLS/SSL connection to the network adapter provided by the plugin. (V8.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to network adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`id` identifier for the new connection or `Nil` for auto id selection

`server$` server to connect to

`port` port to connect at

`table` optional: table argument containing further options (see above) (V8.0)

RESULTS

`id` optional: identifier of the new connection; this will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenConnection(1, "www.airsoftsoftwair.de", 80)
SendData(1, "GET http://www.airsoftsoftwair.de/index.html " ..
            "HTTP/1.0\r\n\r\n")
a$ = ReceiveData(1, #RECEIVEALL)
Print(a$)
CloseConnection(1)
```

The code above connects to `http://www.airsoftsoftwair.de` and downloads the index HTML page.

42.18 ReceiveData

NAME

ReceiveData – receive data through the network (V5.0)

SYNOPSIS

```
data$, count, done = ReceiveData(id, mode, ...)
data$, count, done = ReceiveData(id, #RECEIVEBYTES, maxbytes[, callback,
                                 userdata])
data$, count, done = ReceiveData(id, #RECEIVEALL[, untilterm, callback,
                                 userdata])
data$, count, done = ReceiveData(id, #RECEIVELINE[, callback, userdata])
```

FUNCTION

This function can be used to receive data from a server or a client. If you want to receive data from a server, you need to pass an identifier obtained from `OpenConnection()` to this function. If you are a server and want to retrieve data from one of your clients, you need to pass the network identifier of the respective client. You can get the identifiers of

your clients by listening to the `OnConnect` event handler which you can set up by calling `InstallEventHandler()`.

The second argument specifies how much data you want to receive from the sender with this call. Currently, the following modes are supported:

`#RECEIVEBYTES:`
: Receive all available data but not more than the specified number of bytes. If you use this mode, you need to pass the maximum number of bytes you wish to receive in the third argument. `ReceiveData()` will then never obtain more bytes than you specified. However, it can happen that less bytes are returned in case there is not enough data available from the sender. You can find out the number of bytes obtained by looking at the second return value.

`#RECEIVEALL:`
: Receive all data currently available. If the optional argument `untilterm` is set to `True`, `ReceiveData()` will not return before the sender terminates the connection, thus allowing you to receive all data a sender has to offer using just a single call. If `untilterm` is set to `False`, `ReceiveData()` will only return the data that is currently available. It will not wait for additional data to arrive, but it will tell you if there is more data to be retrieved (in the third return value). The default setting for `untilterm` is `True` which means that `ReceiveData()` will read data until the sender terminates the connection.

`#RECEIVELINE:`
: Receive a single line of text from the sender. This mode must only be used when working with non binary data. The carriage return and the newline characters will not be included in the returned string. They are read from the network buffer but they will not be returned by `ReceiveData()` if you use `#RECEIVELINE`.

`ReceiveData()` returns three values: The first return value is a string that contains the data that was received from the network or an empty string if you use a callback to handle the data or no data could be read. Please note that although the data read is returned as a string, it is not limited to text only. It can also contain binary data because Hollywood strings can handle control characters and the `NULL` character just fine. The second return value specifies how many bytes could be read from the network buffer. If this is 0, then there is currently no data available. The third return value is only useful if you use `#RECEIVEALL` transfer mode with `untilterm` set to `False`. In that case, the third return value tells you if more data is available in the network buffer. If there is more data to be read, then `done` is `False`, otherwise it will be `True`.

Starting with Hollywood 6.0 there is an optional `callback` parameter that allows you to pass a callback function that should receive the data read from the server. This can be useful if you need to stream large amounts of data that cannot be efficiently stored inside a Hollywood string. The callback function could simply write the data it receives to a file, for example. Note that if you specify a callback function, `ReceiveData()` will always return an empty string. The callback function you specify will be called with a

single argument: A table that contains more information. Here is an overview of the table fields that will be initialized before `ReceiveData()` runs your callback function:

`Action:` `#RECEIVEDATA_PACKET`

`Data:` The data that has been received from the server. Note that this can contain binary data.

`Count:` Contains the number of bytes in `Data`.

`Total:` Contains the total number of bytes already received.

`UserData:`
Contains the value you passed in the `userdata` argument.

The callback function of type `#RECEIVEDATA_PACKET` should normally return `False`. If it returns `True`, `ReceiveData()` will abort its operations and return immediately.

Finally, there is another optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

INPUTS

`id` identifier of the sender

`mode` the desired transfer mode; see above for a list of currently supported transfer modes

`...` further arguments depend on the specified transfer mode; see above

RESULTS

`data$` the data that was read from the network buffer or an empty string if a callback is specified

`count` number of bytes successfully transmitted

`done` whether or not there is more data in the network buffer (only when `#RECEIVEALL` is used together with `untilterm` set to `False`)

EXAMPLE

See Section 42.17 [OpenConnection], page 837.

42.19 ReceiveUDPData

NAME

ReceiveUDPData – receive data through UDP protocol (V5.0)

SYNOPSIS

```
data$, ip$, port = ReceiveUDPData(id[, size])
```

FUNCTION

This function can be used to receive data from the UDP object specified by `id`. This UDP object must have been created using `CreateUDPObject()` earlier. The optional argument

`size` can be used to specify the maximum number of bytes to receive. By default, this is set to 8192 bytes which is also the maximum number of bytes `ReceiveUDPData()` can handle. So you may set `size` to values less than 8192 bytes but not to more.

`ReceiveUDPData()` returns three values: The first return value is a string containing the data received from the UDP object. The second return value contains the IP address of the sender, and the third return value contains the port number of the sender.

Note that the global network timeout set using `SetNetworkTimeout()` is currently ignored by `ReceiveUDPData()`.

INPUTS

`id`	identifier of the UDP object to use
`size`	optional: maximum number of bytes to receive (defaults to 8192 bytes)

RESULTS

`data$`	the data that was read from the network
`ip$`	IP address of the data sender
`port`	port number of the data sender

42.20 ResolveHostName

NAME

ResolveHostName – convert host name to IP address (V8.0)

SYNOPSIS

```
t = ResolveHostName(host$)
```

FUNCTION

This function can be used to convert the host name specified in `host$` to an IP address. In contrast to the similar function `ToIP()`, `ResolveHostName()` returns all information it can retrieve from the resolver in a table. This is especially useful if you need to get information about the available Internet protocols for the host name. Thus, this function can return multiple IP addresses for the specified host name, e.g. one IPv4 address and one IPv6 address.

`ResolveHostName()` returns a table which contains a number of subtables, one for each IP address successfully resolved. Each of these subtables will have the following fields initialized:

`Address:` The IP address of the specified host name in the format of the respective Internet protocol (see below).

`Protocol:`
The Internet protocol used by this IP address. This can be one of the following special values:

`#IPV4:` Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPUNKNOWN:`
IP address uses an unknown protocol.

INPUTS

`host$` host name to resolve

RESULTS

`t` a table of tables containing all information from the resolver (see above)

EXAMPLE

```
t = ResolveHostName("www.airsoftsoftwair.de")
For Local k = 0 To ListItems(t) - 1
    Print(t[k].address, t[k].protocol)
Next
```

On a Windows 10 system, the code above resolves two IP address for `www.airsoftsoftwair.de`: One IPv4 address and one IPv6 address.

42.21 SendData

NAME

SendData – send data through the network (V5.0)

SYNOPSIS

```
count = SendData(id, data$)
```

FUNCTION

This function can be used to send data to a server or a client. If you want to send data to a server, you need to pass an identifier obtained from `OpenConnection()` to this function. If you are a server and want to send data to one of your clients, you need to pass the identifier of the respective client. You will receive the identifiers of your clients by listening to the `OnConnect` event handler which you can set up by calling `InstallEventHandler()`.

The second argument is a string containing the data that you want to send to the recipient. Please note that although this argument is a string, it is not limited to text only. You can also send raw data with this function because Hollywood strings can handle character control codes as well as the special `NULL` character without problems.

Upon return, `SendData()` will return the number of bytes that it has successfully transmitted to the recipient. This can be less than the number of bytes in `data$`.

INPUTS

`id` identifier of the recipient

`data$` string containing the data that should be sent

RESULTS

`count` number of bytes successfully transmitted

EXAMPLE

See Section 42.17 [OpenConnection], page 837.

42.22 SendUDPData

NAME

SendUDPData – send data through UDP protocol (V5.0)

SYNOPSIS

```
count = SendUDPData(id, data$[, ip$, port])
```

FUNCTION

This function will send the data specified in `data$` to the recipient specified by `ip$` and `port`. The data will be sent through the UDP object specified in the first argument. This UDP object must have been created by `CreateUDPObject()` before. For performance reasons, you must pass an IP address directly to this function. Passing a host name instead is not supported because it would have to be resolved first which would take too much time. `SendUDPData()` will return the number of bytes successfully transferred. This can be less than the number of bytes in `data$`.

Please note that although the data argument is a string, it is not limited to text only. You can also send binary data with this function because Hollywood strings can handle character control codes as well as the special `NULL` character without problems.

Starting with Hollywood 8.0, the `ip$` and `port` arguments are not required in case the UDP object has been created as a UDP object of type `#UDPCLIENT`. In that case, the UDP object is already connected and you don't have to pass `ip$` and `port`. In fact, they are ignored for UDP objects of type `#UDPCLIENT`. See Section 42.5 [CreateUDPObject], page 823, for details.

Note that the global network timeout set using `SetNetworkTimeout()` is currently ignored by `SendUDPData()`.

INPUTS

`id` identifier of the UDP object to use

`data$` string containing the data that should be sent

`ip$` optional: IP address of recipient; must be specified for UDP objects of type `#UDPSERVER` and `#UDPNONE`

`port` optional: port number of recipient; must be specified for UDP objects of type `#UDPSERVER` and `#UDPNONE`

RESULTS

`count` number of bytes successfully transmitted

42.23 SetNetworkProtocol

NAME

SetNetworkProtocol – set default network protocol (V8.0)

SYNOPSIS

```
SetNetworkProtocol(protocol)
```

FUNCTION

This function can be used to set the Internet protocol to be used by functions like `OpenConnection()`, `CreateServer()`, `CreateUDPObject()` and `DownloadFile()` if no explicit protocol has been requested.

You have to pass the desired default Internet protocol in the `protocol` argument. This must be one of the following special values:

`#IPV4:` Use Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Use Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` Let the host operating system determine the Internet protocol to use.

Due to historical and portability reasons, `#IPV4` is the default network protocol Hollywood will use. If you want to change this default, call this function.

INPUTS

`protocol` desired new default Internet protocol to use

42.24 SetNetworkTimeout

NAME

SetNetworkTimeout – set global timeout for network functions (V5.0)

SYNOPSIS

```
SetNetworkTimeout(ms)
```

FUNCTION

This function can be used to define a global timeout setting for all functions of the Hollywood network library. By default, this timeout is set to 10000 milliseconds (10 seconds). This means that if a server takes longer than 10 seconds to respond, the network library will automatically terminate the connection. Defining such a timeout is very important to make sure that network functions cannot block your whole script just because a server is down. Thus, there should always be a reasonable timeout value.

Normally, it should not be necessary to use this function. In rare circumstances, however, it might be useful to modify the global timeout value. If you need to do so, just pass the desired timeout value in milliseconds to this function.

INPUTS

`ms` new desired global timeout in milliseconds (defaults to 10000 milliseconds)

42.25 ToHostName

NAME

ToHostName – convert IP address to host name (V5.0)

SYNOPSIS

```
host$ = ToHostName(ip$)
```

FUNCTION

This function can be used to locate the host name of the specified IP address. The IP address must be specified as a string in the form of four numbers separated by colons, like this: `"10.20.30.40"`. To get the IP address from a host name, use `ToIP()`.

INPUTS

`ip$` IP address to resolve

RESULTS

`host$` resolved host name of the specified IP address

42.26 ToIP

NAME

ToIP – convert host name to IP address (V5.0)

SYNOPSIS

```
ip$ = ToIP(host$[, protocol])
```

FUNCTION

This function can be used to resolve the IP address of the specified host name. The IP address of the host will be returned as a string. To get the host of an IP address, use `ToHostName()`.

Starting with Hollywood 8.0, there is an optional new `protocol` argument which allows you to specify the Internet protocol that should be used for the resulting IP address. This can be one of the following special constants:

`#IPV4:` Use Internet Protocol version 4 (IPv4). IPv4 addresses are limited to 32 bits and are represented using four numbers separated by three dots, e.g. 127.0.0.1.

`#IPV6:` Use Internet Protocol version 6 (IPv6). IPv6 addresses use 128 bits and are represented by eight groups of four hexadecimal digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` Let the host operating system determine the Internet protocol to use.

The `protocol` argument defaults to the default protocol type set using `SetNetworkProtocol()`. By default, this is `#IPV4` due to historical and portability reasons. See Section 42.23 [SetNetworkProtocol], page 845, for details.

To resolve a host name with advanced functionality, take a look at the `ResolveHostName()` function. See Section 42.20 [ResolveHostName], page 842, for details.

INPUTS

`host$` host name to resolve

`protocol` optional: Internet protocol to use (see above for possible values); defaults to the protocol type set using `SetNetworkProtocol()` (V8.0)

RESULTS

`ip$` IP address of the specified host

42.27 UploadFile

NAME

UploadFile – upload file to a server (V5.0)

SYNOPSIS

```
s$, len = UploadFile(url$, options[, func, userdata])
```

FUNCTION

This command allows you to conveniently upload a file to a network server. By default, `UploadFile()` supports the FTP and HTTP protocols but plugins may provide support for additional protocols. Before Hollywood 6.0 `UploadFile()` only supported FTP upload but starting with version 6.0 HTTP upload is supported as well. Since HTTP and FTP upload use two entirely different mechanisms of sending the data to the receiving server, the procedures of uploading a file using FTP and using HTTP are quite different as well.

If you want to upload a file to an FTP server with `UploadFile()`, you will have to pass the URL where the file should be stored on the FTP server as the first argument. This URL must begin with the ftp:// prefix, and it must contain a fully qualified path specification (that is, including the destination filename). The URL must not contain any escaped characters. Escaping will be done by `UploadFile()` so make sure that you pass only unescaped URLs, e.g. passing `"ftp://ftp.site.net/my%20file.zip"` will not work. You must specify an URL without escaped characters, so the correct version would be: `"ftp://ftp.site.net/my file.zip"`. If you want to pass a URL that has already been escaped, you have to set the `Encoded` tag to `True` (see below). In that case, `UploadFile()` won't do any further escaping on your URL.

The file that should be uploaded to the FTP server must be specified in either the `File` or `Data` table element (see below).

If you want to upload a file to an HTTP server, you need to pass the URL of a PHP or CGI script which handles the upload. Note that `UploadFile()` only supports upload using the HTTP POST method. Upload via HTTP PUT is not supported. In addition

to the URL of a PHP or CGI script, you also have to specify the parameters that should be passed to this script. These parameters are passed in the `FormData` table element (see below). The file(s) to be uploaded also have to be passed as parameters in the `FormData` table element.

You can also specify a username and password that shall be used to log into the HTTP or FTP server. When using FTP upload, "anonymous" is used as the default username and "anonymous@anonymous.org" as the default password. If you want to use a different user account, you have to pass the username/password pair in the URL. Here is an example URL for username "johndoe" and password "topsecret": ftp://johndoe:topsecret@ftp.test.net/pub/files.lha for FTP upload or http://johndoe:topsecret@www.test.com/private/upload.php for HTTP upload.

The URL you pass to this function can also contain a port number. If you want to specify a port number, you have to put it after the host name and separate it using a colon. Here is an example for using port 1234: ftp://ftp.test.net:1234/test/image.jpg. If no port is specified, `UploadFile()` will use port 21.

The second argument is a table that recognizes several options. When using FTP upload, the data to be uploaded can be specified by using either the `File` table tag or the `Data` tag. When using HTTP upload, the data to be uploaded must be specified using the `FormData` element. Here is an overview of all tags that are currently recognized by the `options` table:

`File:` For FTP uploads, the file you want to upload must be specified in this tag. In case you want to upload data from a string source, you must use the `Data` tag below. You must use either specify `File` or `Data` in every call to `UploadFile()` for FTP uploads. You must not use this tag for HTTP uploads. Use `FormData` for HTTP uploads (see below).

`Data:` For FTP uploads, this tag allows you to specify a string that will be uploaded to the location specified in argument one. The string is not limited to text only, but it can also contain binary data. If you want to upload data from a file source, you must use the `File` tag instead (see above). You must specify either `Data` or `File` in every call to `UploadFile()` for FTP uploads. You must not use this tag for HTTP uploads. Use `FormData` for HTTP uploads (see below).

`TransferMode:`
This tag can be used to specify whether `UploadFile()` should transfer the file in ASCII or in binary mode. For ASCII mode, specify `#FTPASCII` here. For binary mode, use `#FTPBINARY`. The default transfer mode is `#FTPBINARY`. This tag is only supported for FTP uploads.

`SilentFail:`
If you set this tag to `True`, `UploadFile()` will never throw an error but simply exit silently and return an error message in the first return value, and -1 in the second return value to indicate that an error has happened. If it is set to `False`, `UploadFile()` will throw a system error for all errors that occur. Defaults to `False`.

`FormData:`

This tag is needed for HTTP uploads. It allows you to specify a table of parameters that should be passed to the PHP or CGI script which handles the upload. The file or data to be uploaded has to be passed in this table as well. You have to pass a table of tables to this argument. Every subtable describes a single script parameter. `UploadFile()` will use this table of tables to compose multipart form data request that is then sent to the HTTP server using the POST request type. The following table elements can be used in each subtable:

`Name:` This must be set to the name of the parameter. This table element must always be provided.

`Data:` The data that should be passed as the parameter's value. This must be set to a string. The string can also contain binary data so it is possible to pass the file data to be uploaded in this table element. Alternatively, you can also use the `File` table element to upload data from a file source. Note that you have to specify either the `Data` or the `File` tag for each subtable. If you want to use `Data` to upload file instead of form data, you also have to specify the `MIMEType` and `FileAlias` tags (see below).

`File:` If you don't set the `Data` tag, you need to set this table element to a filename whose contents should be uploaded as part of the parameter passed in `Name`. Alternatively, you can also use the `Data` table element to upload data from a string source. Note that you have to specify either the `File` or the `Data` tag for each subtable.

`MIMEType:`

This tag allows you to set the MIME type of the data to be uploaded. This should be specified whenever the parameter subtable intends to upload a file. It must not be specified in case the parameter subtable merely passes simple form data (i.e. plain text) to the server. This tag defaults to "application/octet-stream" if the `File` tag has been set. If the `Data` tag has been set there is no default for the `MIMEType` tag since the `Data` tag could also contain plain form data. Thus, if you want to use the `Data` tag to upload file data, you always have to explicitly set `MIMEType` to the data's MIME type.

`FileAlias:`

If the parameter subtable intends to upload a file, this tag can be used to set a name for this file. This is usually only needed if the file data you want to upload is specified using the `Data` tag. When using the `File` tag, `UploadFile()` will simply use the name of this file and you don't have to use `FileAlias` at all, though it can be used to override the filename specified in `File`.

It is perfectly allowed to upload more than one file at once. You can use as many subtables as you need with the `FormData` table element. The resulting

HTML page generated by the PHP or CGI script after the upload will be returned as a string by `UploadFile()`. (V6.0)

`CustomHeaders:`
This tag allows you to specify a string of custom headers that should be sent to the HTTP server when making the request. This can be useful for some fine-tuned adjustments for some servers. Keep in mind that the individual header elements have to be terminated by a carriage return and a line feed. This tag is only supported when using the HTTP protocol. (V6.0)

`Encoded:` Set this tag to `True` if the URL you passed to this function has already been correctly escaped. If this tag is set to `True`, `UploadFile()` won't escape any characters. Instead, it expects you to pass a URL that has already been correctly escaped so that it can be directly used for server requests without any additional escaping. (V6.1)

`Protocol:`
This tag can be used to specify the Internet protocol that should be used when opening the connection. This can be one of the following special constants:

`#IPV4:` Use Internet Protocol version 4 (IPv4).

`#IPV6:` Use Internet Protocol version 6 (IPv6). Note that `#IPV6` is currently unsupported on AmigaOS and compatible systems.

`#IPAUTO:` Let the host operating system determine the Internet protocol to use.

This tag defaults to the default protocol type set using `SetNetworkProtocol()`. By default, this is `#IPV4` due to historical and portability reasons. See Section 42.23 [SetNetworkProtocol], page 845, for details. (V8.0)

`Adapter:` This tag allows you to specify one or more network adapters that should be asked to establish the specified connection. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V8.0)

`SSL:` Set this tag to `True` to request a connection through TLS/SSL encryption. Note that setting this tag when using Hollywood's inbuilt network adapter doesn't have any effect because Hollywood's inbuilt network adapter doesn't support TLS/SSL connections. However, there might be a network adapter provided by a plugin that supports TLS/SSL and if you set this tag to `True` Hollywood will forward your wish to have a TLS/SSL connection to the network adapter provided by the plugin. Do note, though, that you normally don't have to set this tag in case the URL's scheme already indicates an SSL connection by using a prefix such as `"https://"` or `"ftps://"`. (V8.0)

`Async:` If this is set to `True`, `UploadFile()` will operate in asynchronous mode. This means that it will return immediately, passing an asynchronous operation

handle to you. You can then use this asynchronous operation handle to finish the operation by repeatedly calling `ContinueAsyncOperation()` until it returns `True`. This is very useful in case your script needs to do something else while the operation is in progress, e.g. displaying a status animation or something similar. By putting `UploadFile()` into asynchronous mode, it is easily possible for your script to do something else while the operation is being processed. See Section 19.4 [ContinueAsyncOperation], page 224, for details. Defaults to `False`. (V9.0)

`Verbose:` This tag can be set to `True` to request detailed log information about the connection and the protocol interaction with the server. This is currently only used by Hollywood plugins so if you use Hollywood's internal network adapter, setting this tag to `True` has no effect. Plugins, however, may choose to provide extended connection information when this tag has been set to `True`. Defaults to `False`. (V9.0)

`FileAdapter:`
This tag is only used if the `File` tag is set as well. In that case, `FileAdapter` allows you to specify one or more file adapters that should be asked if they want to open the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to `default`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to file and network adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

The optional parameter `func` can be used to pass a callback function which will be called from time to time by `UploadFile()` allowing you to update a progress bar for example. The callback function you specify here will be called with a single argument: A table that contains more information. Here is an overview of the table fields that will be initialized before `UploadFile()` runs your callback function:

`Action:` `#UPLOADFILE_STATUS`

`Count:` Contains the number of bytes that have already been uploaded.

`Total:` Contains the size of the file being uploaded.

`UserData:`
Contains the value you passed in the `userdata` argument.

The callback function of type `#UPLOADFILE_STATUS` should normally return `False`. If it returns `True`, the upload operation will be aborted.

Note that when using `UploadFile()` for HTTP uploading, the PHP or CGI script used to handle the upload will also generate a resulting HTML page that is usually shown by the browser after the upload has been completed. This HTML page is returned by `UploadFile()` as a string. The second return value describes the length of this HTML page in bytes. Since `UploadFile()` has to download this resulting HTML page from the server, your callback function will be called while `UploadFile()` is receiving the server's

response so that you can monitor progress. The table that is passed to your callback function will be initialized as follows in that case:

`Action:` `#UPLOADFILE_RESPONSE`

`Count:` Contains the number of bytes that have already been downloaded.

`Total:` Contains the size of the file being downloaded.

`UserData:`
Contains the value you passed in the `userdata` argument.

The callback function of type `#UPLOADFILE_RESPONSE` should normally return `False`. If it returns `True`, the download operation will be aborted.

Finally, there is a fourth optional argument called `userdata`. The value you specify here is passed to your callback function whenever it is called. This is useful if you want to avoid working with global variables. Using the `userdata` argument you can easily pass data to your callback function. You can specify a value of any type in `userdata`. Numbers, strings, tables, and even functions can be passed as user data.

INPUTS

`url$` FTP URL for destination file or URL of a PHP or CGI script on a HTTP server

`options` a table containing the file/data to be uploaded as well as further options

`func` optional: a callback function that shall be called from time to time

`userdata` optional: user defined data that should be passed to callback function

RESULTS

`s$` optional: the resulting HTML page generated by the PHP or CGI script; note that this is only passed when using HTTP upload

`len` optional: the length of the resulting HTML page generated by the PHP or CGI script; note that this is only passed when using HTTP upload

EXAMPLE

```
UploadFile("ftp://ftp.test.net/pub/image.jpg", {File = "image.jpg"})
```

The code above uploads the file "image.jpg" to the FTP server specified in argument 1.

```
UploadFile("http://www.test.com/upload.php", {FormData = {
    {Name = "uploadername", Data = "John Doe"},
    {Name = "uploaderemail", Data = "john@doe.com"},
    {Name = "description", Data = "My profile picture"},
    {Name = "imagefile", File = "image.jpg", MIMEType = "image/jpeg"}}})
```

The code above uploads the file "image.jpg" to a HTTP server. Additionally, it passes the parameters `uploadername`, `uploaderemail`, and `description` to the PHP script.

```
@REQUIRE "hurl"
...
UploadFile("ftp://ftp.test.net/pub/image.jpg",
```

```
    {File = "image.jpg", SSL = True, Adapter = "hurl"})
```

The code above uploads a file using explicit FTPS. Since Hollywood doesn't support SSL/TLS by default, this code uses the hURL plugin for the operation because hURL supports SSL/TLS. hURL is activated by passing `hurl` in the `Adapter` tag.

```
@REQUIRE "hurl"
...
UploadFile("ftps://ftp.test.net/pub/image.jpg",
    {File = "image.jpg", Adapter = "hurl"})
```

The code above uploads a file using implicit FTPS. Since Hollywood doesn't support SSL/TLS by default, this code uses the hURL plugin for the operation because hURL supports SSL/TLS. hURL is activated by passing `hurl` in the `Adapter` tag.

43 Object library

43.1 Overview

This library provides abstract functions to deal with Hollywood objects. Hollywood objects are all objects created and managed by Hollywood, e.g. brushes, anims, background pictures, videos, etc. Those objects will be closed and freed automatically when Hollywood exits. It is recommended, though, that you free objects no longer needed yourself in order to avoid unnecessary memory consumption.

Hollywood objects are addressed either through numeric identifiers or through handles that are returned by all object creation functions when you pass the special value `Nil` as the numeric identifier. See Section 7.8 [Auto id selection], page 91, for details. When using numeric identifiers and passing a numeric identifier that already exists to an object creation function, the existing object will automatically be freed.

Object library functions like `GetAttribute()`, `GetObjectData()`, or `SetObjectData()` require you to pass an object type constant together with the identifier of the object. The following object type constants are currently recognized:

`#ANIM` An animation object created by `@ANIM` or `LoadAnim()`. See Section 17.2 [ANIM], page 177, for details.

`#ANIMSTREAM`
An animation object created by `BeginAnimStream()`. See Section 17.3 [BeginAnimStream], page 180, for details.

`#ASYNCDRAW`
An asynchronous draw object created by `PlayAnim()`, the functions in the move object library, or by the transition effects functions.

`#ASYNCOBJ`
An asynchronous operation handle created by functions like `CopyFile()` or `DownloadFile()`.

`#BGPIC` A background picture object created by `@BGPIC`, `LoadBGPic()` and the like. See Section 20.2 [BGPIC], page 227, for details.

`#BRUSH` A brush object created by `@BRUSH`, `LoadBrush()` and the like. See Section 21.6 [BRUSH], page 251, for details.

`#CLIENT` A client object created by `OpenConnection()` or passed to your `OnConnect` event handler callback. See Section 42.17 [OpenConnection], page 837, for details.

`#CLIPREGION`
A clip region object created by `CreateClipRegion()`. See Section 30.8 [CreateClipRegion], page 596, for details.

`#CONSOLEWINDOW`
A console window object created by `CreateConsoleWindow()`. See Section 23.10 [CreateConsoleWindow], page 327, for details.

`#DIRECTORY`
: A directory object created by `OpenDirectory()`. See Section 26.47 [OpenDirectory], page 458, for details.

`#DISPLAY`
: A display object created by `@DISPLAY` or `CreateDisplay()`. See Section 25.8 [DISPLAY], page 380, for details.

`#FILE`
: A file object created by `@FILE` or `OpenFile()`. See Section 26.18 [FILE], page 433, for details.

`#FONT`
: A font object created by `@FONT` or `OpenFont()`. See Section 54.10 [FONT], page 1125, for details.

`#ICON`
: An icon object created by `@ICON`, `LoadIcon()` and the like. See Section 31.6 [ICON], page 625, for details.

`#INTERVAL`
: An interval object created by `SetInterval()`. See Section 29.26 [SetInterval], page 582, for details.

`#LAYER`
: A Hollywood layer created by one of the commands which draw graphics, e.g. `DisplayBrush()`.

`#MEMORY`
: A memory block object created by `AllocMem()` and the like. See Section 38.1 [AllocMem], page 787, for details.

`#MENU`
: A menu object created by `@MENU` or `CreateMenu()`. See Section 39.8 [MENU], page 804, for details.

`#MOVELIST`
: A move list object created by `AddMove()`. See Section 34.2 [AddMove], page 649, for details.

`#MUSIC`
: A music object created by `@MUSIC`, `OpenMusic()` and the like. See Section 49.28 [MUSIC], page 993, for details.

`#PALETTE:`
: A palette object created by `@PALETTE`, `LoadPalette()` or `CreatePalette()`. See Section 44.17 [PALETTE], page 902, for details.

`#POINTER`
: A mouse pointer object created by `CreatePointer()`. See Section 41.1 [CreatePointer], page 817, for details.

`#SAMPLE`
: A sound sample object created by `@SAMPLE`, `LoadSample()` and the like. See Section 49.39 [SAMPLE], page 1001, for details.

`#SERIAL`
: An serial connection object created by OpenSerialPort. See Section 47.11 [OpenSerialPort], page 952, for details.

`#SERVER`
: A server object created by `CreateServer()`. See Section 42.4 [CreateServer], page 822, for details.

`#SPRITE`
: A sprite object created by `@SPRITE` or `LoadSprite()`. See Section 50.13 [SPRITE], page 1020, for details.

`#TEXTOBJECT`
: A text object created by `CreateTextObject()`. See Section 54.7 [CreateTextObject], page 1121, for details.

`#TIMEOUT` A timeout object created by `SetTimeout()`. See Section 29.27 [SetTimeout], page 583, for details.

`#TIMER` A timer object created by `StartTimer()`. See Section 55.17 [StartTimer], page 1168, for details.

`#UDPOBJECT`
A UDP object created by `CreateUDPObject()`. See Section 42.5 [CreateUDPObject], page 823, for details.

`#VECTORPATH`
A path object created by `StartPath()`. See Section 56.36 [StartPath], page 1197, for details.

`#VIDEO` A video object created by `@VIDEO` or `OpenVideo()`. See Section 57.17 [VIDEO], page 1212, for details.

43.2 ClearObjectData

NAME

ClearObjectData – clear private data key in an object (V5.0)

SYNOPSIS

```
ClearObjectData(type, id[, key$])
```

FUNCTION

This function can be used to remove private data from an object. You have to pass the type and identifier of the object whose private data you would like to modify. If the optional argument `key$` is specified, `ClearObjectData()` will remove the data associated with this key only. If `key$` is omitted, `ClearObjectData()` will remove the private data of all keys in this object.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`type` type of the object

`id` identifier of the object

`key$` optional: key which shall be removed from object; if this argument is omitted, all keys will be removed from this object

43.3 CopyObjectData

NAME

CopyObjectData – copy private data between objects (V5.0)

SYNOPSIS

```
CopyObjectData(srctype, srcid, dsttype, dstid[, overwrite])
```

FUNCTION

This function copies all private data associated with the object specified by `srctype` and `srcid` to the object specified by `dsttype` and `dstid`. The optional `overwrite` argument

specifies whether or not `CopyObjectData()` should overwrite keys in the destination object in case they share the name of keys in the source object. By default, this is set to `True` which means that existing keys in the destination object will be replaced with the keys in the source object in case their names are the same. If you do not want this behaviour, set `overwrite` to `False`.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`srctype` type of source object to use

`srcid` identifier of source object to use

`dsttype` type of destination object to use

`dstid` identifier of destination object to use

`overwrite`
optional: specifies whether or not existing keys in the destination object should be overwritten (defaults to `True`)

EXAMPLE

```
SetObjectData(#BRUSH, 1, "name", "mybrush")
CopyObjectData(#BRUSH, 1, #BRUSH, 2)
DebugPrint(GetObjectData(#BRUSH, 2, "name"))
```

The code above copies all data keys of brush 1 to brush 2. The call to DebugPrint() will print "mybrush" then.

43.4 GetAttribute

NAME

GetAttribute – get information about an object

SYNOPSIS

```
info = GetAttribute(obj, id, attr[, param, param2])
```

FUNCTION

This function can be used to retrieve properties from all different kinds of Hollywood objects. For example, you can query the dimensions of a brush or the length of sound file. Please see below for a complete list of object types and their attributes.

The following attributes can be queried for `#ANIM`:

`#ATTRWIDTH:`
Returns width of the animation.

`#ATTRHEIGHT:`
Returns height of the animation.

`#ATTRTRANSPARENTCOLOR:`
Returns the transparent color of the animation or `#NOTRANSPARENCY`.

`#ATTRNUMFRAMES:`
: Returns the number of frames in this animation. (V2.0)

`#ATTRHASMASK:`
: Returns `True` if animation has a mask. (V2.0)

`#ATTRHASALPHA:`
: Returns `True` if animation has an alpha channel. (V4.5)

`#ATTRFRAMEDELAY:`
: Returns the time anim players should wait after the specified frame in milliseconds. You also need to specify the frame number you want to query in the `param` argument. Frames are counted from 1. If you leave out the `param` argument, the first frame will be used. Please note that not all animation formats support frame delays and that the information might only be available for frames that are already loaded; i.e. if you are querying a random frame of a disk-based anim, it could be that you get a zero return value because the frame has not been loaded yet. (V4.5)

`#ATTRCOUNT:`
: Returns how many animations there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRLOADER:`
: Returns the name of the loader that was used to load this animation. (V6.0)

`#ATTRDEPTH:`
: Returns the depth of the frame specified in the `param` argument. Frames are counted from 1. If the `param` argument is omitted, the first frame will be used. If the depth is less than or equal to 8, the anim is a palette anim. (V9.0)

`#ATTRPALETTE:`
: Returns the palette of the frame specified in the `param` argument. Frames are counted from 1. If the `param` argument is omitted, the first frame will be used. The frame's palette will be returned as a table and will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the frame doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the palette of the frame specified by `param`. Frames are counted from 1. If the `param` argument is omitted, the first frame will be used. If there is no transparent pen or the frame doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRTYPE:`
: Returns the type of the anim. This will be set to either `#ANIMTYPE_RASTER` for a raster anim or `#ANIMTYPE_VECTOR` for a vector anim. (V9.0)

`#ATTRFORMAT:`
: Returns the anim format name as a string. (V10.0)

The following attributes can be queried for `#ANIMSTREAM`:

`#ATTRCOUNT:`
: Returns how many anim stream objects are currently in memory. Useful for keeping track of the resources used by your script. (V5.0)

The following attributes can be queried for `#ASYNCDRAW`:

`#ATTRTYPE:`
: Returns the type of this asynchronous drawing object; will be `#ADF_FX`, `#ADF_MOVEOBJECT` or `#ADF_ANIM`. (V4.5)

`#ATTRNUMFRAMES:`
: Returns the number of frames of this asynchronous drawing object; please note that if you use this value as the base for a loop over `AsyncDrawFrame()`, you must add one loop because the final call to AsyncDrawFrame() does not count as a frame; See Section 19.1 [AsyncDrawFrame], page 221, for details. (V4.5)

`#ATTRCURFRAME:`
: Returns the frame currently on display in this async drawing object. (V4.5)

`#ATTRCOUNT:`
: Returns how many async draw objects there are currently in memory. Useful for tracking memory consumption. (V4.5)

The following attributes can be queried for `#ASYNCOBJ`:

`#ATTRCOUNT:`
: Returns how many asynchronous operation handles there are currently in memory. Useful for tracking memory consumption. (V9.0)

The following attributes can be queried for `#BGPIC`:

`#ATTRWIDTH:`
: Returns width of the BGPic.

`#ATTRHEIGHT:`
: Returns height of the BGPic.

`#ATTRTRANSPARENTCOLOR:`
: Returns the transparent color of the BGPic or `#NOTRANSPARENCY`.

`#ATTRLAYERS:`
: Returns the number of layers attached to this BGPic. (V1.5)

`#ATTRHASMASK:`
: Returns `True` if BGPic has a mask. (V2.0)

`#ATTRHASALPHA:`
: Returns `True` if BGPic has an alpha channel. (V4.5)

`#ATTRCLIPREGION:`
: Returns the identifier of the clip region currently active on this BGPic or -1 if there is no active clip region. (V4.5)

`#ATTRCOUNT:`
: Returns how many BGPics there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRTYPE:`
: Returns the type of the BGPic. This will be set to either `#IMAGETYPE_RASTER` for a raster BGPic or `#IMAGETYPE_VECTOR` for a vector BGPic. (V5.0)

`#ATTRLOADER:`
: Returns the name of the loader that was used to load this BGPic. (V6.0)

`#ATTRSPRITES:`
: Returns the number of sprites currently visible on this BGPic. (V7.0)

`#ATTRDEPTH:`
: Returns the depth of the BGPic. If this is less than or equal to 8, the BGPic is a palette BGPic. (V9.0)

`#ATTRPALETTE:`
: Returns the BGPic's palette as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the BGPic doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the BGPic's palette. If there is no transparent pen or the BGPic doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRCYCLE:`
: If the BGPic has a palette which has color cycling ranges defined, this attribute will return a table containing all color cycling ranges that are defined. In that case, the table returned by `#ATTRCYCLE` will contain a number of subtables which will have the following fields initialized:

  `Low:` The pen index that marks that start of the color range.

  `High:` The pen index that marks the end of the color range.

  `Rate:` The desired speed of the color cycling effect. A value of 16384 indicates 60 frames per second. All other speeds scale linearly from this base, e.g. a value of 8192 indicates 30 frames per second, and so on.

  `Reverse:` If this tag is set to `True`, the colors should be cycled in reverse.

  `Active:` If this tag is set to `True`, this color cycling range is marked as active.

  (V9.0)

`#ATTRFORMAT:`
: Returns the image format name as a string. (V10.0)

The following attributes can be queried for `#BRUSH`:

`#ATTRWIDTH:`
: Returns width of the brush.

`#ATTRHEIGHT:`
: Returns height of the brush.

`#ATTRTRANSPARENTCOLOR:`
: Returns the transparent color of the brush or `#NOTRANSPARENCY`.

`#ATTRHASMASK:`
: Returns `True` if brush has a mask. (V2.0)

`#ATTRHASALPHA:`
: Returns `True` if brush has an alpha channel. (V2.0)

`#ATTRCOUNT:`
: Returns how many brushes there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRTYPE:`
: Returns the type of the brush. This will be set to either `#IMAGETYPE_RASTER` for a raster brush or `#IMAGETYPE_VECTOR` for a vector brush. (V5.0)

`#ATTRHARDWARE:`
: Returns `True` if the specified brush is a hardware brush. See Section 21.37 [hardware brushes], page 280, for details. (V5.0)

`#ATTRDISPLAY:`
: Returns the identifier of the display that this brush belongs to if the brush is a display-dependent hardware brush. Otherwise -1 is returned. See Section 21.37 [hardware brushes], page 280, for details. (V6.0)

`#ATTRLOADER:`
: Returns the name of the loader that was used to load this brush. (V6.0)

`#ATTRDEPTH:`
: Returns the depth of the brush. If this is less than or equal to 8, the brush is a palette brush. (V9.0)

`#ATTRPALETTE:`
: Returns the brush's palette as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the brush doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the brush's palette. If there is no transparent pen or the brush doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRCYCLE:`
: If the brush has a palette which has color cycling ranges defined, this attribute will return a table containing all color cycling ranges that are defined. In that case, the table returned by `#ATTRCYCLE` will contain a number of subtables which will have the following fields initialized:

  `Low:` The pen index that marks that start of the color range.

  `High:` The pen index that marks the end of the color range.

  `Rate:` The desired speed of the color cycling effect. A value of 16384 indicates 60 frames per second. All other speeds scale linearly from this base, e.g. a value of 8192 indicates 30 frames per second, and so on.

  `Reverse:` If this tag is set to `True`, the colors should be cycled in reverse.

  `Active:` If this tag is set to `True`, this color cycling range is marked as active.

  (V9.0)

`#ATTRFORMAT:`
: Returns the image format name as a string. (V10.0)

The following attributes can be queried for `#CLIENT`:

`#ATTRCOUNT:`
: Returns how many network client objects are currently in memory. Useful for keeping track of the resources used by your script. (V5.0)

`#ATTRADAPTER:`
: Returns the name of the adapter that has been used to open this connection or `inbuilt` if the connection has been opened using Hollywood's inbuilt connection handler. (V8.0)

The following attributes can be queried for `#CLIPREGION`:

`#ATTRXPOS:`
: Returns the x-position of the clip region. (V3.0)

`#ATTRYPOS:`
: Returns the y-position of the clip region. (V3.0)

`#ATTRWIDTH:`
: Returns the width of the clip region. (V3.0)

`#ATTRHEIGHT:`
: Returns the height of the clip region. (V3.0)

`#ATTRCOUNT:`
: Returns how many clip regions there are currently in memory. Useful for tracking memory consumption. (V4.5)

The following attributes can be queried for `#CONSOLEWINDOW`:

`#ATTRCOUNT:`
: Returns how many console windows are currently in memory. Useful for tracking memory consumption. (V10.0)

The following attributes can be queried for `#DIRECTORY`:

`#ATTRCOUNT:`
: Returns how many directory handles there are currently open. Useful for tracking resources. (V4.5)

`#ATTRADAPTER:`
: Returns the name of the adapter that has been used to open this directory or `inbuilt` if this directory has been opened using Hollywood's inbuilt directory handler. (V6.0)

`#ATTRFORMAT:`
: Returns the directory format name as a string. This is typically only set if the directory is handled by a plugin because by default, directories don't have a "format" but plugins might implement adapters that map zip files et al. to directories. In that case, `#ATTRFORMAT` allows you to find out the format of the directory source, e.g. "zip archive". (V10.0)

The following attributes can be queried for `#DISPLAY`:

`#ATTRWIDTH:`
: Returns the width of the display; this value does not include the width of the window border.

`#ATTRHEIGHT:`
: Returns the height of the display; this value does not include the height of the window border.

`#ATTRMAXWIDTH:`
: Returns the maximum possible width for this display (i.e. current desktop resolution minus border width).

`#ATTRMAXHEIGHT:`
: Returns the maximum possible height for this display (i.e. current desktop resolution minus border height).

`#ATTRBGPIC:`
: Returns the BGPic associated with this display. (V1.5)

`#ATTRLAYERS:`
: Returns the number of layers in the associated BGPic. (V1.5)

`#ATTRCURSORX:`
: Returns current x-position of the cursor. (V2.5)

`#ATTRCURSORY:`
: Returns current y-position of the cursor. (V2.5)

`#ATTRXPOS:`
: Returns the x-position of the display on the screen. (V3.0)

`#ATTRYPOS:`
: Returns the y-position of the display on the screen. (V3.0)

`#ATTRBORDERLEFT:`
: Returns the width of the left display border or 0 if the display is borderless. (V3.0)

`#ATTRBORDERRIGHT:`
: Returns the width of the right display border or 0 if the display is borderless. (V3.0)

`#ATTRBORDERTOP:`
: Returns the height of the top display border or 0 if the display is borderless. (V3.0)

`#ATTRBORDERBOTTOM:`
: Returns the height of the bottom display border or 0 if the display is borderless. (V3.0)

`#ATTRHOSTWIDTH:`
: Returns the width of the screen on which the display is open (usually the desktop screen). (V3.0)

`#ATTRHOSTHEIGHT:`
: Returns the height of the screen on which the display is open (usually the desktop screen). (V3.0)

`#ATTRFONTASCENDER, #ATTRFONTDESCENDER, #ATTRFONTNAME, #ATTRFONTSIZE, #ATTRFONTSCALABLE, #ATTRFONTAA, #ATTRFONTDEPTH, #ATTRFONTPALETTE, #ATTRFONTTRANSPARENTPEN, #ATTRFONTENGINE, #ATTRFONTTYPE, #ATTRFONTCHARMAP, #ATTRFONTLOADER, #ATTRFONTHEIGHT, #ATTRFONTFORMAT:`
: Information about the font currently selected into this display; please read below in `#TEXTOBJECT` for the info on these attributes. (V3.1)

`#ATTRPOINTER:`
: Returns the identifier of the mouse pointer currently associated with this display. (V4.5)

`#ATTRSTATE:`
: Returns the current state of this display; can be either `#DISPSTATE_OPEN` (if display is open), or `#DISPSTATE_CLOSED` (display is currently closed), or `#DISPSTATE_MINIMIZED` (display is currently minimized). (V4.5)

`#ATTRACTIVE:`
: Returns whether or not this display is currently active; only one display can be active at a time. (V4.5)

`#ATTRMODE:`
: Returns the current display mode; this can be either `#DISPMODE_WINDOWED` for windowed mode or it can be `#DISPMODE_FULLSCREEN` for full screen mode. Note that this is a global setting. You can safely pass 0 for display when querying `#ATTRMODE`. (V4.5)

`#ATTRSCALEMODE:`
: Returns the scale mode currently active in this display. Can be either `#SCALEMODE_NONE`, `#SCALEMODE_AUTO`, or `#SCALEMODE_LAYER`. (V4.5)

`#ATTRSCALEWIDTH:`
: Returns the currently set scaling width for this display. If no scaling is active, this attribute will return the same width as `#ATTRWIDTH`. (V4.5)

`#ATTRSCALEHEIGHT:`
: Returns the currently set scaling height for this display. If no scaling is active, this attribute will return the same height as `#ATTRHEIGHT`. (V4.5)

`#ATTRBORDERLESS:`
: Returns whether or not the display is borderless. (V4.5)

`#ATTRSIZEABLE:`
: Returns whether or not the display is resizeable. (V4.5)

`#ATTRFIXED:`
: Returns whether or not the display is fixed. (V4.5)

`#ATTRNOHIDE:`
: Returns whether or not the display can be iconified by the user. (V4.5)

`#ATTRNOMODESWITCH:`
: Returns `True` if display mode switching via CMD+RETURN (LALT+RETURN on Windows) hotkey is disabled for this display. (V4.5)

`#ATTRTITLE:`
: Returns the display's title string. (V4.5)

`#ATTRMARGINLEFT, #ATTRMARGINRIGHT:`
: Returns the current margin settings for this display as set using `SetMargins()`. (V4.5)

`#ATTRDOUBLEBUFFER:`
: Returns `True` if the specified display is a double- buffered one, `False` if it is not double-buffered. (V4.5)

`#ATTROUTPUTDEVICE:`
: This attribute returns three values containing information about the current output device. The first return value can be either `#DISPLAY`, `#BGPIC`, `#BRUSH`, `#ANIM`, or `#DOUBLEBUFFER`. The second return value specifies the corresponding identifier to the type indicated by the first return value. The third return value, finally, is only used by types `#BRUSH`, `#ANIM` and `#BGPIC` (NB: For `#BGPIC` it is only used when `SelectBGPic()` was called with mode set to either `#SELMODE_NORMAL` or `#SELMODE_COMBO`). In that case, it specifies the graphics of the brush/animation/BGPic that are currently selected:

This can be either `#MASK`, `#ALPHACHANNEL` or `#BRUSH`. Note that the third return value will be set to `#BRUSH` also in case #ANIM/#BGPIC is returned by the first return value. If `#BRUSH` is returned as the third return value, it means that the color channel of the brush/animation is currently selected. If `SelectBGPic()` is in `#SELMODE_LAYERS`, the first and third return value will both be `#BGPIC`. If `SelectBGPic()` is in a different mode, the first return value will be `#BGPIC`, but the third return value will be either `#BRUSH`, `#MASK`, or `#ALPHACHANNEL`. (V4.5)

`#ATTRCOUNT:`
Returns how many display handles there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRMASKMODE:`
Returns the current mask mode. This is a global setting not tied to a specific display. See Section 21.73 [SetMaskMode], page 311, for details. (V4.5)

`#ATTRALPHAINTENSITY:`
Returns the current alpha intensity. This is a global setting not tied to a specific display. See Section 21.67 [SetAlphaIntensity], page 307, for details. (V4.5)

`#ATTRLAYERSON:`
Returns `True` if the specified display has layers enabled, `False` if that is not the case. (V5.0)

`#ATTRORIENTATION:`
Returns the current orientation of the mobile device that Hollywood is running on. The return value will be one of the following orientation modes:

```
#ORIENTATION_PORTRAIT
#ORIENTATION_LANDSCAPE
#ORIENTATION_PORTRAITREV
#ORIENTATION_LANDSCAPEREV
```

Please note that this tag is only supported in the mobile version of Hollywood. In the desktop version it will always return `#ORIENTATION_NONE`. (V5.0)

`#ATTRPUBSCREEN:`
Returns the public screen that this display is currently opened on. This is only supported on AmigaOS compatible operating systems. (V5.2)

`#ATTRDENSITY:`
Returns the logical density of the display. The return value will be one of the following predefined density values:

```
#DENSITY_LOW
#DENSITY_MEDIUM
#DENSITY_HIGH
```

Please note that this tag is only supported in the mobile version of Hollywood. In the desktop version it will always return `#DENSITY_NONE`. (V5.3)

`#ATTRXDPI:`
: Returns the exact physical pixels per inch of the screen on the X axis. Please note that this tag may return a spurious value on older Android devices. (V5.3)

`#ATTRYDPI:`
: Returns the exact physical pixels per inch of the screen on the Y axis. Please note that this tag may return a spurious value on older Android devices. (V5.3)

`#ATTRMENU:`
: Returns the identifier of the menu strip attached to this display or -1 if this display doesn't have a menu strip attached. (V6.0)

`#ATTRMONITOR:`
: Returns the monitor number that this display has been opened on. Monitors are counted from 1 which is the primary monitor. (V6.0)

`#ATTRHOSTMONITORS:`
: Returns the total number of monitors currently available to the system. You can query their dimensions and extended desktop positions using `GetMonitors()`. (V6.0)

`#ATTRXSERVER:`
: Returns the name of the X Server that this display is connected to. This is only supported on Linux. (V6.0)

`#ATTRADAPTER:`
: Returns the name of the display adapter currently in use. If Hollywood's inbuilt display adapter is used, `inbuilt` is returned. (V6.0)

`#ATTRMAXIMIZED:`
: Returns `True` if the display is currently maximized, `False` otherwise. (V7.0)

`#ATTRRAWWIDTH:`
: Returns the raw physical width of the display, regardless of any scaling engine currently active. Use this attribute with care because it can conflict with scaling engines because they always pretend that Hollywood is in running in a different resolution. (V7.0)

`#ATTRRAWHEIGHT:`
: Returns the raw physical height of the display, regardless of any scaling engine currently active. Use this attribute with care because it can conflict with scaling engines because they always pretend that Hollywood is in running in a different resolution. (V7.0)

`#ATTRHOSTTITLEBARHEIGHT:`
: Returns the height of the host screen's title bar. Note that not all systems have a title bar, most notably Windows doesn't have any. In that case, 0 is returned. (V7.0)

`#ATTRHOSTTASKBAR:`
: Returns information about the taskbar on Windows. This tag will return 5 values: The first two values describe the x- and y-position of the taskbar

on the host screen, return values three and four contain the dimensions of the taskbar and the fifth and last return value is a boolean which indicates whether or not the taskbar is currently visible. This tag is currently only supported on Windows. (V7.0)

`#ATTRSPRITES:`
Returns the number of sprites currently active on this display. (V7.0)

`#ATTRHOSTSCALEX:`
Returns the scaling coefficient on the x-axis of the display's monitor. Normally, this is 1 but for high resolution displays (e.g. Retina Macs or 4K monitors on Windows) this can be greater than 1. Note that on Windows this will always be 1 unless you explicitly enable DPI-awareness by setting the `DPIAware` tag in the `@OPTIONS` preprocessor command to `True`. (V7.0)

`#ATTRHOSTSCALEY:`
Returns the scaling coefficient on the y-axis of the display's monitor. Normally, this is 1 but for high resolution displays (e.g. Retina Macs or 4K monitors on Windows) this can be greater than 1. Note that on Windows this will always be 1 unless you explicitly enable DPI-awareness by setting the `DPIAware` tag in the `@OPTIONS` preprocessor command to `True`. (V7.0)

`#ATTRHOSTSCALE:`
Returns the global scaling coefficient of the display's monitor. Normally, this is 1 but for high resolution displays (e.g. Retina Macs or 4K monitors on Windows) this can be greater than 1. Note that on Windows this will always be 1 unless you explicitly enable DPI-awareness by setting the `DPIAware` tag in the `@OPTIONS` preprocessor command to `True`. (V8.0)

`#ATTRIMMERSIVEMODE:`
Returns the immersive mode used by the display. The return value will be one of the following special constants:

```
#IMMERSIVE_NONE
#IMMERSIVE_NORMAL
#IMMERSIVE_LEANBACK
#IMMERSIVE_STICKY
```

See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`#ATTRSYSTEMBARS:`
Returns `True` if the system bars are currently visible, `False` otherwise. This is currently only supported on Android. Note that the system bars can only ever be invisible when a display is in immersive mode. See Section 25.8 [DISPLAY], page 380, for details. (V9.0)

`#ATTRDEPTH:`
Returns the depth of the display. If this is less than or equal to 8, the display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. (V9.0)

`#ATTRPALETTE:`
Returns the display's palette as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned

as RGB colors. If the display doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
Returns the pen that is transparent in the display's palette. If there is no transparent pen or the display doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRPALETTEMODE:`
Returns the current palette mode set using `SetPaletteMode()`. See Section 44.31 [SetPaletteMode], page 914, for details. (V9.0)

`#ATTRDITHERMODE:`
Returns the current dither mode set using `SetDitherMode()`. See Section 44.26 [SetDitherMode], page 910, for details. (V9.0)

`#ATTRPEN:`
Returns the current draw pen set using `SetDrawPen()`. See Section 44.27 [SetDrawPen], page 911, for details. (V9.0)

`#ATTRSHADOWPEN:`
Returns the current shadow pen set using `SetShadowPen()`. See Section 44.35 [SetShadowPen], page 918, for details. (V9.0)

`#ATTRBORDERPEN:`
Returns the current border pen set using `SetBorderPen()`. See Section 44.22 [SetBorderPen], page 907, for details. (V9.0)

`#ATTRBULLETPEN:`
Returns the current bullet pen set using `SetBulletPen()`. See Section 44.23 [SetBulletPen], page 907, for details. (V9.0)

`#ATTRSCALESWITCH:`
Returns if this display will just scale itself to the monitor's current resolution when pressing the `CMD+RETURN` (`LALT+RETURN` on Windows) hotkey or using `#DISPMODE_MODESWITCH` with `ChangeDisplayMode()`. (V9.0)

`#ATTRINTERPOLATE:`
When a scaling engine is active, this will return whether or not interpolated scaling will be used. (V9.0)

The following attributes can be queried for `#EVENTHANDLER`:

`#ATTRFUNCTION:`
Returns the callback function associated with the specified event handler. Note that you need to pass the name of the event handler as a string in id. Just as you would do in `InstallEventHandler()`. (V4.5)

`#ATTRUSERDATA:`
Returns the user data that is associated with the specified event handler. Note that you need to pass the name of the event handler as a string in id. Just as you would do in `InstallEventHandler()`. (V4.5)

The following attributes can be queried for `#FILE`:

`#ATTRMODE:`
: Returns the mode this file was opened in. Can be either `#MODE_READ`, `#MODE_WRITE`, or `#MODE_READWRITE`. (V4.5)

`#ATTRCOUNT:`
: Returns how many file handles there are currently open. Useful for tracking resources. (V4.5)

`#ATTRADAPTER:`
: Returns the name of the adapter that has been used to open this file or `inbuilt` if this file has been opened using Hollywood's inbuilt file handler. (V6.0)

`#ATTRENCODING:`
: Returns the encoding set for the file using `OpenFile()` or `SetFileEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for a list of encodings. (V9.0)

`#ATTRFORMAT:`
: Returns the file format as a string. This is typically only set if the file is handled by a plugin because Hollywood's default file handler doesn't provide any format recognition. Plugins, however, might implement adapters that allow you to open files from zip archives etc. In that case, `#ATTRFORMAT` allows you to find out the format of the file source, e.g. `"zip archive"`.(V10.0)

The following attributes can be queried for `#FONT`:

`#ATTRFONTASCENDER, #ATTRFONTDESCENDER, #ATTRFONTNAME, #ATTRFONTSIZE, #ATTRFONTSCALABLE, #ATTRFONTAA, #ATTRFONTDEPTH, #ATTRFONTPALETTE, #ATTRFONTTRANSPARENTPEN, #ATTRFONTENGINE, #ATTRFONTTYPE, #ATTRFONTCHARMAP, #ATTRFONTLOADER, #ATTRFONTHEIGHT, #ATTRFONTFORMAT:`
: Please see below in `#TEXTOBJECT` for information about the meaning of these attributes. (V4.5)

`#ATTRCOUNT:`
: Returns how many fonts there are currently in memory. Useful for tracking memory consumption. (V4.5)

The following attributes can be queried for `#ICON`:

`#ATTRNUMENTRIES:`
: Returns the number of images in the icon. (V8.0)

`#ATTRSTANDARD:`
: Returns the index of the standard image in the icon or 0 in case there is no standard image in the icon. (V8.0)

`#ATTRWIDTH:`
: Returns the width of the image at the index specified in the optional `param` argument. Indices start at 1. (V8.0)

`#ATTRHEIGHT:`
: Returns the height of the image at the index specified in the optional `param` argument. Indices start at 1. (V8.0)

`#ATTRNUMFRAMES:`
: Returns the number of frames of the image at the index specified in the optional `param` argument. This can be either 1 or 2, depending on whether the image has only a normal state, or a normal and a selected state. Indices start at 1. (V8.0)

`#ATTRCOUNT:`
: Returns how many icons are currently in memory. Useful for tracking memory consumption. (V8.0)

`#ATTRDEPTH:`
: Returns the depth of the image at the index specified in the `param` argument. Indices start at 1. If `param2` is set to `True` the selected image is queried, if it is `False` (also the default) the normal image is queried. If the depth is less than or equal to 8, the image is a palette image. (V9.0)

`#ATTRPALETTE:`
: Returns the palette of the image at the index specified in the `param` argument. Indices start at 1. If `param2` is set to `True` the selected image is queried, if it is `False` (also the default) the normal image is queried. The image's palette will be returned as a table and will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the image doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the palette of the image at the index specified in the `param` argument. Indices start at 1. If `param2` is set to `True` the selected image is queried, if it is `False` (also the default) the normal image is queried. If there is no transparent pen or the image doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRFORMAT:`
: Returns the icon format name as a string. (V10.0)

The following attributes can be queried for `#INTERVAL`:

`#ATTRDURATION:`
: Returns the frequency of this interval object in milliseconds. (V4.5)

`#ATTRFUNCTION:`
: Returns the callback function associated with the specified interval object. (V4.5)

`#ATTRUSERDATA:`
: Returns the user data associated with this interval object. (V4.5)

The following attributes can be queried for `#LAYER`:

`#ATTRTYPE:`
: Returns the type of the layer (e.g. `#PRINT`). (V1.5)

`#ATTRXPOS:`
: Returns the x-position of the layer on the display. (V1.5)

`#ATTRYPOS:`
: Returns the y-position of the layer on the display. (V1.5)

`#ATTRWIDTH:`
: Returns width of the layer. (V1.5)

`#ATTRHEIGHT:`
: Returns height of the layer. (V1.5)

`#ATTRVISIBLE:`
: Returns `True` if the layer is currently visible, `False` if it is hidden. (V1.5)

`#ATTRLAYERID:`
: Returns the identifier of the specified layer; obviously, this makes only sense if you specify a layer name instead of an id as the source. (V2.0)

`#ATTRNUMFRAMES:`
: Returns the number of frames of this layer; only works when used with layers of type `#ANIM` (V2.0) or `#VIDEO` (V6.0).

`#ATTRCURFRAME:`
: Returns the frame which is currently displayed; works only with layers of type `#ANIM`. Note in contrast to most other commands, frames are counted from 0 here so you'll get 0 for the first frame, not 1. (V2.0)

`#ATTRTEXT, #ATTRFONTASCENDER, #ATTRFONTDESCENDER, #ATTRFONTNAME, #ATTRFONTSIZE, #ATTRFONTSCALABLE, #ATTRFONTAA, #ATTRFONTDEPTH, #ATTRFONTPALETTE, #ATTRFONTTRANSPARENTPEN, #ATTRFONTENGINE, #ATTRFONTTYPE, #ATTRFONTCHARMAP, #ATTRFONTLOADER, #ATTRFONTHEIGHT, #ATTRFONTFORMAT:`
: These attributes can be used with layers of type `#PRINT` and `#TEXTOUT` only; to learn more about them read below in the `#TEXTOBJECT` section. (V4.0)

`#ATTRFRAMEDELAY:`
: Returns the time anim players should wait after displaying the current frame in milliseconds; this works only with layers of type `#ANIM`. (V4.5)

`#ATTRCOUNT:`
: Returns how many layers there are currently in memory. Useful for tracking memory consumption. Note that this will return the sum of all layers from all BGPics. If you want to query the number of layers in the current BGPic, use `#ATTRLAYERS` with type `#BGPIC`. (V4.5)

`#ATTRRAWXPOS, #ATTRRAWYPOS, #ATTRRAWWIDTH, #ATTRRAWHEIGHT:`
: These four attributes can be used to find out the real position and size of a layer. The difference between these attributes and the standard `#ATTRXPOS`, `#ATTRWIDTH` etc. attributes is that the standard attributes will always return the position and size of the basic, untransformed layer. The standard

attributes will also not take any under/overhangs into account. Still, you should work with the standard attributes whenever possible because the `#ATTRRAWxxx` attributes operate on a low level in the layers system and could be affected by future changes in the layers system. (V4.7)

`#ATTRZPOS:`
: Returns the z-position of the layer. See Section 34.53 [SetLayerZPos], page 704, for details. (V5.1)

`#ATTRDURATION:`
: Returns the video layer's source duration in milliseconds. This is only supported for layers of type `#VIDEO`. (V6.0)

`#ATTRPOSITION:`
: Returns the current position of a playing or paused video layer in milliseconds. This is only supported for layers of type `#VIDEO`. (V6.0)

`#ATTRFORMAT:`
: Returns the video layer's source format name as a string. This is only supported for layers of type `#VIDEO`. (V6.0)

`#ATTRCANSEEK:`
: Returns whether or not `SeekLayer()` can be used on this video layer. This is only supported for layers of type `#VIDEO`. (V6.0)

`#ATTRPLAYING:`
: Returns `True` if this video layer is currently playing. This is only supported for layers of type `#VIDEO`. (V6.0)

`#ATTRPAUSED:`
: Return `True` if this video is layer currently paused. This is only supported for layers of type `#VIDEO`. (V6.0)

`#ATTRID:` Returns the identifier of this layer's source object. This is only applicable for layers of type `#ANIM`, `#BRUSH`, `#BRUSHPART`, `#BGPICPART`, `#TEXTOBJECT`, `#VECTORPATH` and `#VIDEO`. (V6.0)

`#ATTRDEPTH:`
: Returns the depth of the layer. If this is less than or equal to 8, the layer is a palette layer. (V9.0)

`#ATTRPALETTE:`
: Returns the layer's palette as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the layer doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the layer's palette. If there is no transparent pen or the layer doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRBUTTON:`
: If the layer is attached to a button, this will return the id of that button. Otherwise `Nil` will be returned. (V9.1)

`#ATTRGROUP:`
: Returns the name of the group that this layer is attached to or an empty string if the layer isn't attached to a group. See Section 34.16 [GroupLayer], page 660, for details. (V10.0)

Please note that the position and size values will always refer to the layer in its original, untransformed state. If you rotate or scale a layer, you will still get its original dimensions through `#ATTRWIDTH` and `#ATTRHEIGHT`.

The following attributes can be queried for `#MEMORY`:

`#ATTRSIZE:`
: Returns the size of specified memory block. (V4.5)

`#ATTRCOUNT:`
: Returns how many memory blocks there are currently in memory. Useful for tracking memory consumption. (V4.5)

The following attributes can be queried for `#MENU`:

`#ATTRCOUNT:`
: Returns how many menu strips are currently available. Useful for keeping track of the resources used by your script. (V6.0)

The following attributes can be queried for `#MOVELIST`:

`#ATTRCOUNT:`
: Returns how many move list objects are currently in memory. Useful for keeping track of the resources used by your script. (V5.0)

The following attributes can be queried for `#MUSIC`:

`#ATTRTYPE:`
: Returns the format of the music raw data; this attribute will return one of `#MONO8`, `#MONO16`, `#STEREO8` and `#STEREO16`; not possible with Protracker modules. (V2.0)

`#ATTRDURATION:`
: Returns the music duration in milliseconds. This is unsupported for Protracker modules. If you query `#ATTRDURATION` for Protracker modules, -1 will be returned. (V2.0)

`#ATTRPITCH:`
: Returns the playback pitch (frequency) of the music in Hertz; not possible with Protracker modules. (V2.0)

`#ATTRPOSITION:`
: Returns the position of the music object in milliseconds. (V2.0)

`#ATTRFORMAT:`
: Returns the music format as a string. (V2.0)

`#ATTRBITRATE:`
: Returns the bitrate of the music object; if the music object is currently playing and uses a variable bitrate, you will receive the bitrate of the current frame; not possible with Protracker modules. (V2.0)

`#ATTRCOUNT:`
: Returns how many music objects there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRCANSEEK:`
: Returns whether or not `SeekMusic()` can be used on this music object. (V5.0)

`#ATTRNUMSUBSONGS:`
: Returns the number of subsongs which you can play using `PlaySubsong()`. If this 1, then there is only one song in the music object. (V5.3)

`#ATTRCURSUBSONG:`
: Returns the number of the currently playing subsong. (V5.3)

`#ATTRPLAYING:`
: Returns `True` if this music object is currently playing. (V6.0)

`#ATTRPAUSED:`
: Return `True` if this music object is currently paused. (V6.0)

`#ATTRLOADER:`
: Returns the name of the loader that was used to load this music object. (V6.0)

The following attributes can be queried for `#PALETTE`:

`#ATTRPALETTE:`
: Returns the palette's pens as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the palette. If there is no transparent pen, `#NOPEN` will be returned. (V9.0)

`#ATTRDEPTH:`
: Returns the depth of the palette. This will always be a value between 1 (= 2 colors) and 8 (= 256 colors). (V9.0)

`#ATTRCYCLE:`
: If the palette is a one which has color cycling ranges defined, this attribute will return a table containing all color cycling ranges that are defined. In that case, the table returned by `#ATTRCYCLE` will contain a number of subtables which will have the following fields initialized:

  `Low:` The pen index that marks that start of the color range.

  `High:` The pen index that marks the end of the color range.

`Rate:` The desired speed of the color cycling effect. A value of 16384 indicates 60 frames per second. All other speeds scale linearly from this base, e.g. a value of 8192 indicates 30 frames per second, and so on.

`Reverse:` If this tag is set to `True`, the colors should be cycled in reverse.

`Active:` If this tag is set to `True`, this color cycling range is marked as active.

(V9.0)

`#ATTRLOADER:`
Returns the name of the loader that was used to load this palette. (V9.0)

`#ATTRCOUNT:`
Returns how many palette objects there are currently in memory. Useful for tracking memory consumption. (V9.0)

The following attributes can be queried for `#POINTER`:

`#ATTRWIDTH:`
Returns the width of the pointer image. (V4.5)

`#ATTRHEIGHT:`
Returns the height of the pointer image. (V4.5)

`#ATTRTYPE:`
Returns type of this pointer image; can be `#STDPTR_CUSTOM`, `#STDPTR_SYSTEM` or `#STDPTR_BUSY`; See Section 41.1 [CreatePointer], page 817, for details. (V4.5)

`#ATTRCOUNT:`
Returns how many pointer images there are currently in memory. Useful for tracking memory consumption. (V4.5)

The following attributes can be queried for `#SAMPLE`:

`#ATTRTYPE:`
Returns the format of the samples raw data; this attribute will return one of `#MONO8`, `#MONO16`, `#STEREO8` and `#STEREO16`. (V2.0)

`#ATTRDURATION:`
Returns the sample duration in milliseconds. (V2.0)

`#ATTRPITCH:`
Returns the playback pitch (frequency) of the sample in hertz. (V2.0)

`#ATTRPOSITION:`
Returns how long the sample has been playing and how many times it has looped. The position returned will be a value in milliseconds (1000 milliseconds = 1 second). This value will be reset everytime the sample loops so that the returned position value will never exceed the sample length. The

second return value specifies how many times the sample was played. It will be increased by one every time the sample loops. If you need to find out the total playing time in milliseconds, just multiply the second return value minus 1 by the sample duration (use `#ATTRDURATION`) and add the first return value to it. (V2.0)

`#ATTRCOUNT:`
Returns how many samples there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRNUMFRAMES:`
Returns the number of PCM frames in the sample. (V5.0)

`#ATTRPLAYING:`
Returns `True` if this sample is currently playing. (V6.0)

`#ATTRLOADER:`
Returns the name of the loader that was used to load this sample. (V6.0)

`#ATTRFORMAT:`
Returns the sound format name as a string. (V10.0)

The following attributes can be queried for `#SERIAL`:

`#ATTRCOUNT:`
Returns how many serial connection objects are currently in memory. Useful for keeping track of the resources used by your script. (V8.0)

The following attributes can be queried for `#SERVER`:

`#ATTRCOUNT:`
Returns how many network server objects are currently in memory. Useful for keeping track of the resources used by your script. (V5.0)

The following attributes can be queried for `#SPRITE`:

`#ATTRWIDTH:`
Returns width of the sprite. (V2.0)

`#ATTRHEIGHT:`
Returns height of the sprite. (V2.0)

`#ATTRTRANSPARENTCOLOR:`
Returns the transparent color of the sprite or `#NOTRANSPARENCY`. (V2.0)

`#ATTRHASMASK:`
Returns `True` if sprite has a mask. (V2.0)

`#ATTRHASALPHA:`
Returns `True` if sprite has an alpha channel. (V2.0)

`#ATTRNUMFRAMES:`
Returns the number of frames in this sprite. (V2.0)

`#ATTRCURFRAME:`
: Returns the frame which is currently displayed. (V2.0)

`#ATTRONSCREEN:`
: Returns `True` if the specified sprite is currently on screen. (V2.5)

`#ATTRXPOS:`
: Returns the x-position of the sprite on the screen. (V2.5)

`#ATTRYPOS:`
: Returns the y-position of the sprite on the screen. (V2.5)

`#ATTRCOUNT:`
: Returns how many sprites there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRZPOS:`
: Returns the z-position of the sprite. See Section 50.12 [SetSpriteZPos], page 1019, for details. (V7.0)

`#ATTRDEPTH:`
: Returns the depth of the frame specified in the `param` argument. Frames are counted from 1. If the `param` argument is omitted, the first frame will be used. If the depth is less than or equal to 8, the sprite is a palette sprite. (V9.0)

`#ATTRPALETTE:`
: Returns the palette of the frame specified in the `param` argument. Frames are counted from 1. If the `param` argument is omitted, the first frame will be used. The frame's palette will be returned as a table and will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the frame doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the palette of the frame specified by `param`. Frames are counted from 1. If the `param` argument is omitted, the first frame will be used. If there is no transparent pen or the frame doesn't have a palette, `#NOPEN` will be returned. (V9.0)

The following attributes can be queried for `#TEXTOBJECT`:

`#ATTRWIDTH:`
: Returns width of the text object.

`#ATTRHEIGHT:`
: Returns height of the text object.

`#ATTRFONTASCENDER:`
: Returns the ascender of the current font in pixels; the ascender of a font is the maximum character extent from the baseline to the top of the line; ascender + descender is always equal to the font's pixel height. (V3.1)

`#ATTRFONTDESCENDER:`
: Returns the descender of the current font in pixels; the descender of a font is the maximum character extent from the baseline to the bottom of the line; ascender + descender is always equal to the font's pixel height. (V3.1)

`#ATTRFONTNAME:`
: Returns the name of the currently selected font. (V3.1)

`#ATTRFONTSIZE:`
: Returns the size of the currently selected font. (V3.1)

`#ATTRFONTSCALABLE:`
: Returns `True` if the font is a scalable vector font. (V3.1)

`#ATTRFONTAA:`
: Returns `True` if the font can be anti-aliased. (V3.1)

`#ATTRTEXT:`
: Returns the text string of this text object. (V4.0)

`#ATTRCOUNT:`
: Returns how many text objects there are currently in memory. Useful for tracking memory consumption. (V4.5)

`#ATTRDEPTH:`
: Returns the depth of the text object. If this is less than or equal to 8, the brush is a palette text object. (V9.0)

`#ATTRPALETTE:`
: Returns the text object's palette as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. If the text object doesn't have a palette, an empty table will be returned. (V9.0)

`#ATTRTRANSPARENTPEN:`
: Returns the pen that is transparent in the text object's palette. If there is no transparent pen or the text object doesn't have a palette, `#NOPEN` will be returned. (V9.0)

`#ATTRTYPE:`
: Returns the type of the text object. This will be set to either `#IMAGETYPE_RASTER` for a raster text object or `#IMAGETYPE_VECTOR` for a vector text object. (V10.0)

`#ATTRFONTDEPTH:`
: If the font is an Amiga color font, this attribute will return its depth. (V9.0)

`#ATTRFONTPALETTE:`
: If the font is an Amiga color font, this attribute will return its palette as a table. The table will contain as many items as there are pens in the palette. The individual pens will be returned as RGB colors. (V9.0)

`#ATTRFONTTRANSPARENTPEN:`
: If the font is an Amiga color font, this attribute will return the pen that is transparent in the font's palette. (V9.0)

`#ATTRFONTENGINE:`
: Returns the engine that was used to open this font. This will be either `#FONTENGINE_INBUILT`, `#FONTENGINE_NATIVE` or `#FONTENGINE_PLUGIN`. See Section 54.31 [SetFont], page 1139, for details. (V9.0)

`#ATTRFONTTYPE:`
: Returns the font type. This can be one of the following types:

  `#FONTTYPE_BITMAP:`
  : An Amiga bitmap font.

  `#FONTTYPE_COLOR:`
  : An Amiga color font.

  `#FONTTYPE_VECTOR:`
  : A vector font, e.g. in TrueType or OpenType format.

  `#FONTTYPE_BRUSH:`
  : A custom font created from a brush source using the `CreateFont()` command. See Section 54.6 [CreateFont], page 1119, for details. (V10.0)

  (V9.0)

`#ATTRFONTCHARMAP:`
: Returns the character map used by the font. This is only supported by fonts managed by the inbuilt font engine, i.e. the font must have been opened using `#FONTENGINE_INBUILT`. See Section 54.16 [GetCharMaps], page 1130, for details. (V9.0)

`#ATTRFONTLOADER:`
: Returns the name of the loader that was used to load this font. (V10.0)

`#ATTRFONTHEIGHT:`
: Returns the pixel height of the font. This is often the same as `#ATTRFONTSIZE` but not if the font has been opened in points mode or if the underlying text engine interprets the font size as something different from the font height. In any case, `#ATTRFONTHEIGHT` will always be the same as `#ATTRFONTASCENDER + #ATTRFONTDESCENDER`. (V10.0)

`#ATTRFONTFORMAT:`
: Returns the font format name as a string. (V10.0)

`#ATTRADJUSTX:`
: When drawing text objects using `DisplayTextObject()` Hollywood will position them in a way that they appear as if they had been drawn using `TextOut()` which means that they could be offset to the left and top in case parts of some characters are designed to appear in the area of previous characters. This is often the case with characters like `"j"`. You can query the number of horizontal pixels Hollywood will offset the text object by querying this tag. Adjustment of text objects can be disabled by setting `NoAdjust` to `True` when calling `CreateTextObject()`. (V10.0)

`#ATTRADJUSTY:`
: This does the same as `#ATTRADJUSTX` (see above) but returns the vertical adjustment pixels for this text object. (V10.0)

The following attributes can be queried for `#TIMEOUT`:

`#ATTRDURATION:`
: Returns the timeout duration of this timeout object in milliseconds. (V4.5)

`#ATTRFUNCTION:`
: Returns the callback function associated with the specified timeout object. (V4.5)

`#ATTRUSERDATA:`
: Returns the user data associated with this timeout object. (V4.5)

The following attributes can be queried for `#TIMER`:

`#ATTRCOUNT:`
: Returns how many timer objects are currently in memory. Useful for keeping track of the resources used by your script. (V5.0)

`#ATTRELAPSE:`
: Returns the timer's elapse threshold. (V9.0)

The following attributes can be queried for `#UDPOBJECT`:

`#ATTRCOUNT:`
: Returns how many UDP objects are currently in memory. Useful for keeping track of the resources used by your script. (V5.0)

The following attributes can be queried for `#VECTORPATH`:

`#ATTRCOUNT:`
: Returns how many vector path objects there are currently in memory. Useful for tracking memory consumption. (V5.0)

The following attributes can be queried for `#VIDEO`:

`#ATTRWIDTH:`
: Returns width of the video. (V5.0)

`#ATTRHEIGHT:`
: Returns height of the video. (V5.0)

`#ATTRDURATION:`
: Returns the total video duration in milliseconds. (V5.0)

`#ATTRPOSITION:`
: Returns the current position of a playing or paused video in milliseconds. (V5.0)

`#ATTRFORMAT:`
Returns the video format name as a string. (V5.0)

`#ATTRNUMFRAMES:`
Returns the number of frames of this video. Please note that this is often an approximation because it would take too much time to do a precise calculation of all frames in a video stream. This can also return 0 if the video codec does not support frame calculation. (V5.0)

`#ATTRCOUNT:`
Returns how many videos there are currently in memory. Useful for tracking memory consumption. (V5.0)

`#ATTRCANSEEK:`
Returns whether or not `SeekVideo()` can be used on this video object. (V5.0)

`#ATTRDRIVER:`
Returns the driver used for this video. See Section 57.4 [ForceVideoDriver], page 1202, for details. This is obsolete since Hollywood 6.0. Use `#ATTRLOADER` instead. (V5.1)

`#ATTRPLAYING:`
Returns `True` if this video is currently playing. (V6.0)

`#ATTRPAUSED:`
Return `True` if this video is currently paused. (V6.0)

`#ATTRSCALEWIDTH:`
Returns the current scale width set for the video using `SetVideoSize()`. (V6.0)

`#ATTRSCALEHEIGHT:`
Returns the current scale height set for the video using `SetVideoSize()`. (V6.0)

`#ATTRSCALEMODE:`
Returns the current scale mode set for the video using `SetVideoSize()`. (V6.0)

`#ATTRLOADER:`
Returns the name of the loader that was used to load this video. (V6.0)

INPUTS

`obj`	type of object to query (see list above)
`id`	object identifier
`attr`	which information to return
`param`	optional: additional parameter required by some attributes (see above)

RESULTS

`info`	the information you wanted

EXAMPLE

```
width = GetAttribute(#DISPLAY, 0, #ATTRWIDTH)
```

The above code queries the display for its current width. As there is only one display, you do not have to specify an id.

43.5 GetObjectData

NAME

GetObjectData – retrieve private data from an object (V5.0)

SYNOPSIS

```
data = GetObjectData(type, id, key$)
```

FUNCTION

This function can be used to retrieve private data from an object that has been stored using the `SetObjectData()` function. Just pass the type and the identifier of the object and the `key$` under which the data was stored to this function and it will return the corresponding data.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`type`	type of the object
`id`	identifier of the object
`key$`	key under which the data was stored

RESULTS

`data`	data that has been stored under the specified key

EXAMPLE

See Section 43.10 [SetObjectData], page 886.

43.6 GetObjects

NAME

GetObjects – get all objects of specified type (V5.1)

SYNOPSIS

```
table, count = GetObjects(type)
```

FUNCTION

This function can be used to retrieve a list of all objects of the specified type that are currently in memory. This function will return a table containing the identifiers of the objects as well as the total number of objects of the specified type.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`type`	return objects of this type

RESULTS

`table` a table containing identifiers of all objects of the specified type

`count` number of items in the table

EXAMPLE

```
t, c = GetObjects(#BRUSH)
For Local k = 0 To c - 1 Do DebugPrint(t[k])
```

The code above will list all objects of type `#BRUSH` currently in memory.

43.7 GetObjectType

NAME

GetObjectType – retrieve type of an object handle (V5.0)

SYNOPSIS

```
type = GetObjectType(handle)
```

FUNCTION

This function returns the type of the specified object handle. The object handle passed to this function must have been created using automatic id selection. See Section 7.8 [Auto id selection], page 91, for details.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`handle` handle of object whose type you want to retrieve

RESULTS

`type` type of the object handle

EXAMPLE

```
my_anim = LoadAnim(Nil, "test.gif")
my_brush = LoadBrush(Nil, "test.png")
DebugPrint(GetObjectType(my_anim), GetObjectType(my_brush))
```

The code above will print the values of constants `#ANIM` and `#BRUSH`.

43.8 HaveObject

NAME

HaveObject – check if a certain object is available (V5.2)

SYNOPSIS

```
r = HaveObject(type, id)
```

FUNCTION

This function can be used to check whether a certain object has already been loaded. Just pass the object's type and identifier to this function and it will return `True` or `False` depending on whether or not the object is in memory.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`type` type of the object to check

`id` identifier of the object to check

RESULTS

`r` `True` if object is available, `False` otherwise

EXAMPLE

```
DebugPrint(HaveObject(#BRUSH, 1))
```

The code above prints `True` if brush number 1 is in memory, otherwise `False`.

43.9 HaveObjectData

NAME

HaveObjectData – check if data exists in an object (V6.1)

SYNOPSIS

```
b = HaveObjectData(type, id, key$)
```

FUNCTION

This function can be used to check if private data has been stored in an object under the specified key using the `SetObjectData()` function. Just pass the type and the identifier of the object and the `key$` to check to this function and it will return whether there is data for the key or not.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`type` type of the object

`id` identifier of the object

`key$` key to query

RESULTS

`b` `True` if the key has data, `False` otherwise

43.10 SetObjectData

NAME

SetObjectData – store private data in an object (V5.0)

SYNOPSIS

```
SetObjectData(type, id, key$, value)
```

FUNCTION

This function can be used to associate any kind of private data with an object. You have to pass the type and identifier of the object as well as a key string under which the data should be stored inside the object structure. `value` can be any kind of Hollywood data

value: It can be a string, a table, a number, or even a function. Everything is possible. If the key specified in `key$` is already used inside the object, the old data will be replaced with the new one.

To access the data later, you can use the `GetObjectData()` function.

See Section 43.1 [Object types], page 855, for a list of all object types.

INPUTS

`type`	type of the object to use
`id`	identifier of the object to use
`key$`	key under which the data should be stored
`value`	data to store

EXAMPLE

```
SetObjectData(#BRUSH, 1, "brushgroup", "A")
d$ = GetObjectData(#BRUSH, 1, "brushgroup")
```

The code above stores the value "A" in brush 1 under the key "brushgroup" and then retrieves it again. `d$` will be set to "A".

44 Palette library

44.1 Overview

This library provides functions to deal with palettes. As the name implies, a palette is a palette (= set) of colors. The minimum number of colors in a palette is 2 and the maximum number of colors in a palette is 256. The individual colors in a palette are referred to as pens. Those pens are addressed through their indices within the palette, starting from 0. Thus, the first pen in a palette is pen 0, the second one is pen 1, and so on. There can be one pen in a palette that is marked as a transparent pen. Pixels that use this pen, e.g. in a palette brush, will appear transparent then. Typically, pen 0 is the transparent pen.

The number of colors in a palette is referred to as its depth and it is always expressed as a power of 2 exponent. The following depths are available:

`1-bit:` 2 colors.

`2-bit:` 4 colors.

`3-bit:` 8 colors.

`4-bit:` 16 colors.

`5-bit:` 32 colors.

`6-bit:` 64 colors.

`7-bit:` 128 colors.

`8-bit:` 256 colors.

Because of the limited number of colors that can be stored in a palette, palette graphics are no longer widely used today. However, using palette graphics does have some advantages over using RGB graphics in certain situations, for example:

- Palette graphics make it very easy to implement certain effects like color cycling or fading. Because of the limited number of colors that need to be modified these effects can be computed in very few CPU cycles.
- Memory consumption is much lower than when using RGB graphics. For 32-bit RGB graphics, a single pixel will require 4 bytes of memory whereas in palette mode, a single pixel will just require 1 byte of memory. Thus, a 1920x1080 image will require about 8 megabytes of memory in 32-bit mode but only 2 megabytes of memory in palette mode.
- Because palette graphics require less memory than RGB graphics, they can also be compressed better when saving them to disk, e.g. as PNG images. Saving images that don't use more than 256 colors as RGB pixels in PNG will yield a significantly bigger image file than saving it as a PNG that uses a palette. That is why you might want to save images that don't use many colors as palette images if you care about file sizes. Functions like `SaveBrush()` support palette images as well.

Of course, in order to become useful, a palette always needs to be attached to another object that provides the actual pixel data that should be drawn using the colors taken from the palette. In Hollywood, the following object types support palettes:

- animations

- BGPics
- brushes
- displays
- sprites

For most object types, you can just set the `LoadPalette` tag to `True` in functions like `LoadBrush()` to make Hollywood create a palette brush for you. You can also convert RGB brushes to palette brushes using functions like `QuantizeBrush()` or `RemapBrush()`. Once you have a palette object, you can change its palette using the `SetPalette()` function.

Special care needs to be taken when putting a display in palette mode. This will have several implications that you need to be aware of in order to make the most out of a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details.

44.2 ContrastPalette

NAME

ContrastPalette – enhance or reduce palette contrast (V9.0)

SYNOPSIS

```
ContrastPalette(id, inc[, repeat])
```

FUNCTION

This command can be used to enhance or reduce the color contrast in the specified palette. If the `inc` argument is set to `True`, the contrast is enhanced. If it is set to `False`, the contrast is reduced. The optional argument `repeat` can be used to apply the effect to the palette multiple times. This is useful if you want to create sharper contrasts.

INPUTS

`id`	palette to modify
`inc`	`True` to increase contrast, `False` to decrease contrast
`repeat`	optional: specifies how many times the contrast operation should be repeated (defaults to 1 which means run the effect just once)

44.3 CopyPalette

NAME

CopyPalette – clone a palette (V9.0)

SYNOPSIS

```
[id] = CopyPalette(source, dest)
```

FUNCTION

This function clones the palette specified by `source` and creates a copy of it as palette `dest`. The new palette is independent from the old palette so you can free the source palette after it has been cloned.

If you pass `Nil` as `dest`, `CopyPalette()` will return a handle to the new palette to you. Otherwise the new palette will use the identifier specified in `dest`.

INPUTS

`source` source palette id

`dest` identifier of the palette to be created or `Nil` for auto id selection

RESULTS

`id` optional: handle to the new palette; will only be returned if you specified `Nil` in `dest`

EXAMPLE

```
CopyPalette(1, 10)
FreePalette(1)
```

The above code creates a new palette 10 which contains the same color data as palette 1. Then it frees palette 1 because it is no longer needed.

44.4 CopyPens

NAME

CopyPens – copy pens from one palette to another (V9.0)

SYNOPSIS

```
CopyPens(srcid, dstid, srcidx, n[, dstidx])
```

FUNCTION

This function will copy `n` pens from the palette specified by `srcid` to the palette specified by `dstid`. The pens will be read from index `srcidx` in the source palette and they will be copied to the index `dstidx` in the destination palette. If `dstidx` is omitted, the index specified in `srcidx` will be used as the destination index.

Note that it is allowed to use the same palette identifier for `srcid` and `dstid`. In that case, pens inside a single palette object can be moved around. Overlapping pens are also supported.

INPUTS

`srcid` source palette

`dstid` destination palette; can be the same as the source

`srcidx` index of the first pen to be copied (starts from 0)

`n` number of pens to copy

`dstidx` optional: index to copy pens to in the destination palette; defaults to `srcidx`

EXAMPLE

```
CopyPens(1, 2, 0, 32)
```

The code above copies the first 32 pens from palette 1 to palette 2.

44.5 CreatePalette

NAME

CreatePalette – create new palette (V9.0)

SYNOPSIS

```
[id] = CreatePalette(id[, data, t])
```

FUNCTION

This function creates a new palette and assigns the identifier `id` to it. The `data` argument may either be a table containing a number of colors that should be used to initialize the palette's pens or you may set `data` to one of Hollywood's predefined palette types. See below for all predefined palette types supported by Hollywood. If you pass `Nil` in the `id` argument, `CreatePalette()` will automatically choose an identifier for the new palette and return it to you.

The following predefined palette types are supported by Hollywood:

`#PALETTE_MONOCHROME:`
: Two color, black and white palette.

`#PALETTE_GRAY4:`
: 4 color grayscale palette.

`#PALETTE_GRAY8:`
: 8 color grayscale palette.

`#PALETTE_GRAY16:`
: 16 color grayscale palette.

`#PALETTE_GRAY32:`
: 32 color grayscale palette.

`#PALETTE_GRAY64:`
: 64 color grayscale palette.

`#PALETTE_GRAY128:`
: 128 color grayscale palette.

`#PALETTE_GRAY256:`
: 256 color grayscale palette.

`#PALETTE_CGA:`
: Standard CGA palette (16 colors).

`#PALETTE_OCS:`
: Standard OCS palette (32 colors).

`#PALETTE_EGA:`
: Standard EGA palette (64 colors).

`#PALETTE_AGA:`
: Standard AGA palette (256 colors).

`#PALETTE_WORKBENCH:`
: Standard classic Amiga Workbench palette (256 colors).

`#PALETTE_MACINTOSH:`
Standard classic Macintosh palette (256 colors).

`#PALETTE_WINDOWS:`
Standard classic Windows palette (256 colors).

`#PALETTE_DEFAULT:`
Same as `#PALETTE_AGA`. If you omit the `data` argument, `CreatePalette()` will initialize the new palette using the pens from `#PALETTE_DEFAULT`.

If you pass a table of colors in the `data` argument, make sure that all colors are passed as RGB values. Note that the table can also be a sparse array with only the pens initialized that you actually need. Pens that aren't in the `data` table will be initialized to black. See below for an example.

The optional table argument `t` can be used to specify further options. The following options are currently recognized:

`Depth:` The desired depth for the palette. This must be between 1 (= 2 colors) and 8 (= 256 colors). The default is 8. If `Depth` specifies more colors than you pass in the table in the `data` parameter, the remaining colors will be initialized to black. This tag is only used if you pass a table in the `data` argument. If you pass a predefined palette type in `data`, the predefined palette type's depth overrides the depth specified here.

`TransparentPen:`
This tag can be used to specify the pen that shall be transparent in the palette. This defaults to `#NOPEN` which means that no pen shall be made transparent.

`Cycle:` This tag can be used to define several ranges of colors that can be cycled. When set, you must pass a table of subtables to `Cycle`, each subtable describing a configuration of a color cycling effect. Each subtable supports the following tags:

- `Low:` The pen index that marks that start of the color range.
- `High:` The pen index that marks the end of the color range.
- `Rate:` The desired speed of the color cycling effect. A value of 16384 indicates 60 frames per second. All other speeds scale linearly from this base, e.g. a value of 8192 indicates 30 frames per second, and so on.
- `Reverse:` If this tag is set to `True`, the colors should be cycled in reverse. Defaults to `False`.
- `Active:` If this tag is set to `False`, the color range will be marked as inactive. Defaults to `True`.

INPUTS

`id` id for the new palette or `Nil` for auto id selection

`data` optional: either one of the predefined palette types (see above) or a table containing an array of colors (defaults to `#PALETTE_DEFAULT`)

`t` optional: table for specifying further options (see above)

RESULTS

`id` optional: identifier of the palette; will only be returned if you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreatePalette(1, {#RED, #GREEN, #BLUE}, {Depth = 2})
```

The code above creates a palette with four colors initialized to red, green, blue and black.

```
CreatePalette(1, {[0] = #RED, [127] = #BLUE, [255] = #GREEN})
```

The code above creates a new palette with 256 colors and initializes pen 0 to red, pen 127 to blue, and the last pen to green. All other pens will be initialized to black.

```
CreatePalette(1)
```

Creates a new palette and initializes its colors to those of `#PALETTE_DEFAULT`.

```
CreatePalette(1, #PALETTE_CGA)
```

Creates a new palette using the CGA colors.

44.6 CyclePalette

NAME

CyclePalette – cycle palette colors (V9.0)

SYNOPSIS

```
CyclePalette(id, start, end[, repeat])
```

FUNCTION

This function cycles the palette colors between the pen specified by `start` and the pen specified by `end`. If `end` is greater than `start`, all pens starting at the index `start` will be shifted to the right and wrap at the pen specified by `end`. If `start` is greater than `end`, pens will be cycled in reverse, i.e. they will be shifted to the left, wrapping at the pen index specified by `start`. The `repeat` argument can be used to specify how many times the cycling should be repeated. This defaults to 1 which means that colors should only be cycled once.

INPUTS

`id` identifier of the palette whose pens should be cycled

`start` start pen of cycling range

`end` end pen of cycling range

`repeat` optional: number of times to repeat cycling (defaults to 1)

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_CGA}
SetFont(#SANS, 96)
```

```
SetPaletteMode(#PALETTEMODE_PEN)
t$ = "Hollywood"
For Local k = 0 To StrLen(t$) - 1
    SetDrawPen(k + 2)
    Print(MidStr(t$, k, 1))
Next
ExtractPalette(1, #BGPIC, 1)
StartTimer(1)
Repeat
    CyclePalette(1, 2, 10)
    SetPalette(1)
    WaitTimer(1, 80)
Forever
```

The code above prints the individual of characters of the string "Hollywood" in different colors and then cycles their colors.

44.7 ExtractPalette

NAME

ExtractPalette – extract palette from object (V9.0)

SYNOPSIS

`[id] = ExtractPalette(id, srctype, srcid[, frame])`

FUNCTION

This function extracts the palette from the object specified by `srctype` and `srcid`, creates a new palette and assigns the identifier `id` to that new palette. If you pass `Nil` as `id`, `ExtractPalette()` will return a handle to the new palette to you. Otherwise the new palette will use the identifier specified in `id`.

The following object types can be passed in `srctype`:

`#ANIM:` Extract palette from an anim frame. If you set `srctype` to `#ANIM`, you also need to pass the anim frame whose palette should be extracted in the `frame` parameter. Frames are counted from 1, which is also the default value for `frame`.

`#BGPIC:` Extract palette from a BGPic.

`#BRUSH:` Extract palette from a brush.

`#FONT:` Extract palette from a color font.

`#SPRITE:` Extract palette from a sprite frame. If you set `srctype` to `#SPRITE`, you also need to pass the sprite frame whose palette should be extracted in the `frame` parameter. Frames are counted from 1, which is also the default value for `frame`.

INPUTS

`id` identifier of the palette to be created or `Nil` for auto id selection

`srctype` object type to use as source (see above)

`srcid` id of source object to use

`frame` optional: number of frame to use if object type is `#ANIM` or `#SPRITE` (defaults to 1)

RESULTS

`id` optional: handle to the new palette; will only be returned if you specified `Nil` in `id`

EXAMPLE

```
ExtractPalette(1, #BRUSH, 10)
```

The code above extracts the palette from brush 10 and stores it as palette object 1.

44.8 FreePalette

NAME

FreePalette – free a palette (V9.0)

SYNOPSIS

```
FreePalette(id)
```

FUNCTION

This function frees the palette specified by `id`. To reduce memory consumption, you should free palettes when you do not need them any longer.

INPUTS

`id` identifier of the palette to free

44.9 GammaPalette

NAME

GammaPalette – correct gamma values of palette (V9.0)

SYNOPSIS

```
GammaPalette(id, red, green, blue)
```

FUNCTION

This function can be used to gamma correct the color channels of the specified palette. For each color channel, you have to pass a floating point value that specifies the desired gamma correction. A value of 1.0 means no change, a value smaller than 1.0 darkens the channel, a value greater than 1.0 lightens the channel.

INPUTS

`id` palette to gamma correct

`red` gamma correction for red channel

`green` gamma correction for green channel

`blue` gamma correction for blue channel

EXAMPLE

```
GammaPalette(1, 1.5, 1.0, 0.5)
```

The code above lightens the red channel and darkens the blue channel, while leaving the green color channel untouched.

44.10 GetBestPen

NAME

GetBestPen – get best pen for color (V9.0)

SYNOPSIS

```
pen = GetBestPen(id, color)
```

FUNCTION

This command searches for a pen in the palette specified by `id` whose color is the closest match to the color specified in the `color` argument and returns that pen. The `color` argument must be an RGB color.

INPUTS

`id` identifier of palette

`color` RGB color to find closest matching pen for

RESULTS

`pen` pen that is the closest match for the specified color

EXAMPLE

```
SetDrawPen(GetBestPen(1, #RED))
```

The code above sets the pen that most closely resembles red as the drawing pen.

44.11 GetFreePen

NAME

GetFreePen – find unused pen (V10.0)

SYNOPSIS

```
pen = GetFreePen([t])
```

FUNCTION

This function tries to find an unused pen in the currently active palette image and returns it. If all pens are used, -1 will be returned. By default, this function scans the palette pixel data of the current display so it will only work if the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. If you don't want `GetFreePen()` to use the current display, you can select the active palette image using the `SelectPalette()` command but keep in mind that you need to select a palette object that has pixel data attached, e.g. a palette brush or a palette BGPic. Just

selecting a palette object won't work because free pens can obviously only be determined if there's pixel data.

You can also use the optional table argument `t` to specify the source palette object. The table argument supports exactly the same arguments as the optional table argument of `GetPen()`. See Section 44.13 [GetPen], page 898, for details.

INPUTS

`t` optional: table for specifying further options (see above)

RESULTS

`pen` unused pen index or -1 if all pens are in use

44.12 GetPalettePen

NAME

GetPalettePen – get pen color from palette (V9.0)

SYNOPSIS

```
color = GetPalettePen(id, pen)
```

FUNCTION

This function gets the color of the pen specified by `pen` from the palette specified by `id`. The color will be returned as an RGB color.

INPUTS

`id` identifier of palette to use

`pen` pen you want to get (starting from 0)

RESULTS

`color` color of the pen, specified as an RGB color

EXAMPLE

```
color = GetPalettePen(1, 0)
```

The code gets the color of the first pen in palette 1.

44.13 GetPen

NAME

GetPen – get pen color (V9.0)

SYNOPSIS

```
color = GetPen(pen[, t])
```

FUNCTION

This function gets the color of the pen specified by `pen` from the currently active palette. By default, the current display's palette is the active palette but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays],

page 400, for details. A palette can be made the active one by using the `SelectPalette()` command.

Alternatively, you can also use `GetPen()` to get the pen color from a different palette object. To do so, you need to pass the optional table argument `t` to SetPen() and specify the `Type` and `ID` tags. See below for an example.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object whose palette you want to query. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose palette you want to query. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose palette you want to query. The default is the identifier of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the target type is an animation, sprite, or anim layer, you need to set this tag to specify the frame whose palette you want to query. Frames are counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

INPUTS

`pen` pen you want to get (starting from 0)

`t` optional: table for specifying further options (see above)

RESULTS

`color` color of the pen, specified as an RGB color

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_MONOCHROME}
color0 = GetPen(0)
color1 = GetPen(1)
```

The code above creates a monochrome palette display and queries the colors of the first two pens. `color0` will be black and `color1` will be white.

```
color = GetPen(4, {Type = #BRUSH, ID = 2})
```

The code gets the color of pen 4 in brush 2.

44.14 InvertPalette

NAME

InvertPalette – invert palette colors (V9.0)

SYNOPSIS

```
InvertPalette(id)
```

FUNCTION

This function inverts all colors in the palette specified by `id`, which means that all colors will be replaced with their complements (white will become black, blue will become yellow etc.).

INPUTS

`id` palette to invert

EXAMPLE

```
InvertPalette(1)
```

The code above inverts the colors of palette 1.

44.15 LoadPalette

NAME

LoadPalette – load a palette (V9.0)

SYNOPSIS

```
[id] = LoadPalette(id, filename$[, table])
```

FUNCTION

This function loads the palette specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadPalette()` will automatically choose an identifier and return it.

The palette specified in `filename$` can either be in the IFF ILBM palette format, as established by Deluxe Paint, or, alternatively, `filename$` can also be a normal image file that contains a palette. In that case, `LoadPalette()` will simply extract the palette from the image file.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this palette. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details.

`UserTags:`
: This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

This command is also available from the preprocessor: Use `@PALETTE` to preload palettes.

INPUTS

`id` identifier for the palette or `Nil` for auto id selection

`filename$`
: file to load

`table` optional: further options (see above)

RESULTS

`id` optional: identifier of the palette; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadPalette(1, "DPaint32.pal")
```

This loads "DPaint32.pal" as palette 1.

44.16 ModulatePalette

NAME

ModulatePalette – change brightness, saturation, and hue of palette (V9.0)

SYNOPSIS

```
ModulatePalette(id, brightness, saturation, hue)
```

FUNCTION

This function can be used to change the brightness, saturation, and hue settings of the colors in a palette. For each setting, you need to pass a floating point value that describes the desired change. A value of 1.0 means no change, a value smaller than 1.0 reduces the brightness/saturation/hue, while a value greater than 1.0 enhances it.

INPUTS

`id` palette to modulate

`brightness`
: desired brightness correction

`saturation`
: desired saturation correction

`hue` desired hue correction

EXAMPLE

```
ModulatePalette(1, 1.0, 2.0, 1.0)
```

The code above increases the saturation while leaving brightness and hue untouched. The result is a palette with emphasized colors, just like in a cartoon.

44.17 PALETTE

NAME

PALETTE – define a palette for later use (V9.0)

SYNOPSIS

```
@PALETTE id[, filename$][, table]
```

FUNCTION

This preprocessor command can be used to define a palette for later use. The palette can either be loaded from a file, it can be one of Hollywood's predefined palettes, or you can define your own palette by specifying a collection of colors.

If you pass the `filename$` argument, the palette will be loaded from that file. The file specified in `filename$` can either be in the IFF ILBM palette format, as established by Deluxe Paint, or, alternatively, `filename$` can also be a normal image file that contains a palette. In that case, `@PALETTE` will simply extract the palette from the image file. Be advised, though, that if you pass a full image file to `@PALETTE` and compile your script into an applet or executable, Hollywood will link the whole image file to your applet or executable, which will increase the output file size. Thus, it is advised to only use palette files in the IFF ILBM format with `@PALETTE` because those are really small as they don't contain any image data.

If you don't pass the `filename$` argument, you need to set either the `Type` or `Colors` tag in the optional table argument to define the palette (see below). The optional table argument recognizes the following tags:

`Type:` Set this tag if you want to use one of Hollywood's predefined palettes. In that case, you must set `Type` to the identifier of the desired inbuilt palette. See Section 44.36 [SetStandardPalette], page 918, for a list of inbuilt palettes. If you set this tag, you must not pass `filename$` or `Colors`.

`Colors:` This tag can be used to define a custom palette. If you set `Colors`, you must not pass `filename$` or `Type`. To define a custom palette, set `Colors` to a table containing the desired colors for the palette. The palette depth is calculated automatically from the number of colors in the table. Alternatively, you can also set the `Depth` tag to define the palette depth (see below).

`Depth:` The desired depth for the palette. This must be between 1 (= 2 colors) and 8 (= 256 colors). The default is 8. This must only be specified when also setting the `Colors` tag. If `Depth` specifies more colors than you pass in the table in the `Colors` tag, the remaining colors will be initialized to black.

`TransparentPen:`
This tag can be used to specify the pen that shall be transparent in the palette. This defaults to `#NOPEN` which means that no pen shall be made transparent. This must only be used when the `Type` or `Colors` tag is passed as well.

`Link:` Set this field to `False` if you do not want to have `filename$` linked to your executable/applet when you compile your script. This field defaults to `True` which means that the palette will be linked to your executable/applet when

Hollywood is in compile mode. If you use this tag, you must also pass `filename$`.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load the file specified in `filename$`. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. If you use this tag, you must also pass `filename$`.

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the file specified in `filename$`. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. If you use this tag, you must also pass `filename$`.

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

To load or create palettes at runtime, take a look at the `LoadPalette()` and `CreatePalette()` commands.

INPUTS

`id` a value that is used to identify this palette later in the code

`filename$`
the palette file you want to load

`table` optional: table containing further options

EXAMPLE

```
@PALETTE 1, "DPaint32.pal"
```

Preloads "DPaint32.pal" as palette 1.

```
@PALETTE 1, {Colors = {#WHITE, #BLACK, $7777CC, $BBBBBB}}
```

Defines a custom palette as palette 1.

```
@PALETTE 1, {Type = #PALETTE_CGA}
```

Defines the standard CGA palette as palette 1.

44.18 PaletteToGray

NAME

PaletteToGray – convert palette to gray (V9.0)

SYNOPSIS

```
PaletteToGray(id)
```

FUNCTION

This function converts all colors in the palette specified by `id` to gray.

INPUTS

`id` identifier of the palette to convert

EXAMPLE

```
PaletteToGray(1)
```

Convert all colors in palette 1 to gray.

44.19 ReadPen

NAME

ReadPen – read pen from palette object (V9.0)

SYNOPSIS

```
pen = ReadPen(x, y[, t])
```

FUNCTION

This function reads the pen at the position specified by `x` and `y` from the currently active palette object. By default, the current display is the active palette object but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. You can set the active palette object using the `SelectPalette()` command.

Alternatively, you can also use `ReadPen()` to read a pen from a different palette object. To do so, you need to pass the optional table argument `t` to `ReadPen()` and specify the `Type` and `ID` tags. See below for an example.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object from whose pixel data you want to read the pen. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose pixel data should be used. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette object selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose pixel data should be used. The default is the identifier of the currently active palette object set using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the target type is an animation, sprite, or anim layer, you need to set this tag to specify the frame whose pixel data should be used. Frames are counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

INPUTS

`x` x position to read from

`y` y position to read from

`t` optional: table for specifying further options (see above)

RESULTS

`pen` pen at the specified position

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_MONOCHROME}
pen = ReadPen(0, 0)
```

The code above reads the pen in the top-left corner of the display. This will be 0 because by default, the display background will be filled using pen 0.

```
pen = ReadPen(0, 0, {Type = #BRUSH, ID = 2})
```

The code reads the pen in the top-left corner in brush 2.

44.20 SavePalette

NAME

SavePalette – save palette to a file (V9.0)

SYNOPSIS

```
SavePalette(id, f$[, t])
```

FUNCTION

This function saves the palette specified by `id` to the file specified by `f$`. The palette will be saved in the IFF ILBM palette format, as established by Deluxe Paint.

Starting with Hollywood 10.0, `SavePalette()` accepts an optional table argument that allows you to pass additional arguments to the function. The following tags are currently supported by the optional table argument:

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`

This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

To load a palette back into Hollywood, use the `LoadPalette()` function or the `@PALETTE` preprocessor command.

INPUTS

`id`	identifier of the palette to save
`f$`	destination file
`t`	optional: table containing further options (see above) (V10.0)

44.21 SelectPalette

NAME

SelectPalette – set active palette object (V9.0)

SYNOPSIS

```
SelectPalette(type, id)
```

FUNCTION

This function can be used to set the palette object that should be used by functions like `SetPen()`, `GetPen()` and `SetPalette()` by default. The palette that is used by those functions by default is also called the active palette.

You have to pass the type and identifier of the object whose palette should be made the active one. The following object types can be passed to the `type` argument:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that `SelectPalette()` will neither check if the object passed to it exists nor if it has a palette. For performance reasons, all this won't be verified until you call a function which actually tries to access the currently active palette. Thus, it is also possible to make a palette active which doesn't exist yet.

By default, the current display's palette is the active one. If the current display doesn't have a palette and you call a function that tries to access it, an error will occur. See Section 25.16 [Palette mode displays], page 400, for details.

Note that `EndSelect()` must never be called for `SelectPalette()`. Do not confuse `SelectPalette()` with functions like `SelectBrush()` or `SelectAlphaChannel()` which require you to call `EndSelect()` when you're done with them. This must not be done for `SelectPalette()` as it just sets the default palette for functions like `SetPen()` so you must never call `EndSelect()` for `SelectPalette()`.

INPUTS

`type`	type of object whose palette should be made active
`id`	identifier of object whose palette should be made active

EXAMPLE

```
SelectPalette(#BRUSH, 1)
```

The code above makes the palette of brush 1 the active one.

44.22 SetBorderPen

NAME

SetBorderPen – set border pen (V9.0)

SYNOPSIS

```
SetBorderPen(pen)
```

FUNCTION

This function sets the pen specified by `pen` to the pen that will be used for drawing borders when the palette mode is `#PALETTEMODE_PEN` and the current output device is a palette one. The palette mode can be set using `SetPaletteMode()`.

When the palette mode has been set to `#PALETTEMODE_PEN`, all Hollywood commands that draw a border won't use the border color that has been set using `SetFormStyle()`, `SetFontStyle()` or a `BorderColor` tag but they will use the pen that has been set as the border pen using this function.

See Section 44.31 [SetPaletteMode], page 914, for details.

INPUTS

`pen`	pen to use for drawing borders

44.23 SetBulletPen

NAME

SetBulletPen – set bullet pen (V9.0)

SYNOPSIS

```
SetBulletPen(pen)
```

FUNCTION

If the Hollywood display is currently in palette mode, this function allows you to set the pen to be used for drawing bullets when using `TextOut()` in list mode. By default, bullets are drawn using the current draw pen set using `SetDrawPen()`. If you want them to be drawn with a different pen, you can use this function to do so.

See Section 54.39 [TextOut], page 1149, for more information on bullet lists.

INPUTS

`pen`	pen to draw bullets with (starting from 0)

44.24 SetCycleTable

NAME

SetCycleTable – set color cycling table (V9.0)

SYNOPSIS

```
SetCycleTable(cycle[, t])
```

FUNCTION

This function sets the color cycling table of the currently active palette to the one specified in `cycle`. You must pass a table of subtables in `cycle`, each subtable describing a configuration of a color cycling effect. Each subtable supports the following tags:

`Low:` The pen index that marks that start of the color range.

`High:` The pen index that marks the end of the color range.

`Rate:` The desired speed of the color cycling effect. A value of 16384 indicates 60 frames per second. All other speeds scale linearly from this base, e.g. a value of 8192 indicates 30 frames per second, and so on.

`Reverse:` If this tag is set to `True`, the colors should be cycled in reverse. Defaults to `False`.

`Active:` If this tag is set to `False`, the color range will be marked as inactive. Defaults to `True`.

By default, `SetCycleTable()` will copy the color cycling table to the current display's palette which is the default active palette but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. You can select a different palette as the active one by using the `SelectPalette()` command.

Alternatively, the specified color cycling table can also be set to a different palette object. To do so, you need to pass the optional table argument to SetCyclingTable() and specify the `Type` and `ID` tags.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object whose color cycling table you want to set. This can be one of the following object types:

```
#BGPIC
#BRUSH
#PALETTE
```

`Type` defaults to the type of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose cycle table you want to set. The default is the identifier of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

INPUTS

`cycle` table containing color cycling ranges (see above)

`t` optional: table for specifying further options (see above)

44.25 SetDepth

NAME

SetDepth – set palette depth (V9.0)

SYNOPSIS

```
SetDepth(depth[, t])
```

FUNCTION

This function sets the depth of the currently active palette to the one specified in `depth`. `depth` must be a bit depth ranging from 1 (= 2 colors) to 8 (= 256 colors). See Section 44.1 [Palette overview], page 889, for details. Note that if the specified depth is less than that of the pixel data attached to the palette, the pixel data will be remapped to match the new depth.

By default, the current display's palette is the active palette but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. A palette can be made the active one by using the `SelectPalette()` command.

Alternatively, the specified depth can also be set to a different palette object. To do so, you need to pass the optional table argument to `SetDepth()` and specify the `Type` and `ID` tags. See below for an example.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object whose depth you want to set. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose depth you want to modify. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose depth you want to set. The default is the identifier of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the target type is an animation, sprite, or anim layer, you need to set this tag to specify the frame whose depth should be set. Frames are counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

`Remap:` If this tag is set to `False`, out-of-range pens will not be remapped to existing pens but instead they will simply be set to the pen specified in the `ClipPen`

tag (see below), i.e. no remapping will take place. Note that `Remap` is only effective when reducing colors. If the new depth has more pens than the old depth, `Remap` won't do anything. (V10.0)

`ClipPen:` This is only used in case the `Remap` tag is set to `False` (see above). In that case, out-of-range pens will not be remapped to existing pens but will simply be set to the pen specified in the `ClipPen` tag, i.e. no remapping will take place. Note that `ClipPen` is only effective when reducing colors. If the new depth has more pens than the old depth, `ClipPen` won't do anything. (V10.0)

INPUTS

`depth` desired new palette depth (ranging from 1 to 8)

`t` optional: table for specifying further options (see above)

EXAMPLE

```
SetDepth(4, {Type = #BRUSH, ID = 2})
```

The code above sets the palette depth of brush 2 to 4 (= 16 colors).

44.26 SetDitherMode

NAME

SetDitherMode – set dither mode (V9.0)

SYNOPSIS

```
SetDitherMode(mode)
```

FUNCTION

When palette mode is set to `#PALETTEMODE_REMAP`, which is also the default, you can use this command to configure the dithering mode to use. The desired dithering mode has to be passed in the `mode` argument. Dithering can increase the quality of the remapped graphics but it is slower than remapping graphics without dithering.

The following dithering modes are currently available:

`#DITHERMODE_NONE:`
No dithering. This is the default dithering mode.

`#DITHERMODE_FLOYDSTEINBERG:`
Use Floyd-Steinberg dithering.

See Section 44.31 [SetPaletteMode], page 914, for details.

INPUTS

`mode` desired dithering mode (see above)

44.27 SetDrawPen

NAME

SetDrawPen – set draw pen (V9.0)

SYNOPSIS

```
SetDrawPen(pen)
```

FUNCTION

This function sets the pen specified by `pen` to the pen that will be used for drawing to palette output devices when the palette mode is `#PALETTEMODE_PEN`. The palette mode can be set using `SetPaletteMode()`. See Section 44.31 [SetPaletteMode], page 914, for details.

When the palette mode has been set to `#PALETTEMODE_PEN`, the following functions will use the pen set using `SetDrawPen()` instead of the color that is passed to them:

- `Arc()`
- `Box()`
- `Circle()`
- `Cls()`
- `CreateTextObject()`
- `DrawPath()`
- `Ellipse()`
- `Line()`
- `Plot()`
- `Polygon()`
- `Print()`
- `TextOut()`

Note that in case the palette mode is `#PALETTEMODE_PEN`, any shadow or border effect also won't be drawn in the color that was set using `SetFormStyle()`, `SetFontStyle()` or `ShadowColor` or `BorderColor` tags. Instead, shadow and border will be drawn using the pen set via `SetShadowPen()` and `SetBorderPen()`, respectively.

INPUTS

`pen`	desired drawing pen; pens start at 0

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_DEFAULT}
SetFillStyle(#FILLCOLOR)
SetPaletteMode(#PALETTEMODE_PEN)
SetDrawPen(10)
Box(#CENTER, #CENTER, 320, 240)
```

The code above will create a palette mode display and then draw a filled rectangle to the center of the screen using palette pen 10.

44.28 SetGradientPalette

NAME

SetGradientPalette – create color gradient in palette (V9.0)

SYNOPSIS

```
SetGradientPalette(id, startcolor, endcolor)
```

FUNCTION

This function creates a color gradient in the palette specified by `id`. The first pen will be initialized to the color specified in `startcolor` and the last pen will be initialized to the color specified in `endcolor`. All pens between the first and the last pen will be filled with intermediary colors so that the result will be a smooth gradient between `startcolor` and `firstcolor`. Obviously, the more colors the palette has, the smoother the resulting gradient will be so it's recommended to set the palette depth to 8 (= 256 colors) for the best result.

INPUTS

`id` identifier of palette to use

`startcolor`
start color of the gradient

`endcolor` end color of the gradient

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_DEFAULT, Height = 512}
CreatePalette(1)
SetGradientPalette(1, #BLACK, #BLUE)
SetPaletteMode(#PALETTEMODE_PEN)
SetPalette(1)
SetFillStyle(#FILLCOLOR)
For Local y = 0 To 255
    SetDrawPen(y)
    Box(0, y * 2, 640, 2)
Next
```

The code above creates gradient between black and blue in palette 1 and draws it.

44.29 SetPalette

NAME

SetPalette – change palette (V9.0)

SYNOPSIS

```
SetPalette(id[, t])
```

FUNCTION

This function replaces all pens in the currently active palette with the pens from palette specified by `id`. By default, the current display's palette is the active palette but of course only in case the current display is a palette mode display. See Section 25.16

[Palette mode displays], page 400, for details. A palette can be made the active one by using the `SelectPalette()` command.

Alternatively, the palette specified by `id` can also be copied to other objects. To do so, you need to pass the optional table argument to `SetPalette()` and set the destination object type in the `Type` table tag and the object's identifier in the `ID` table tag. For example, to assign palette 1 to brush 2, do the following:

```
SetPalette(1, {Type = #BRUSH, ID = 2})
```

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object you want to copy the palette to. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose palette you want to modify. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object you want to copy the palette to. The default is the identifier of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the palette is to be copied to an animation, sprite, or anim layer, you need to set this tag to specify the frame the palette should be copied to. Frames are counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

`Remap:` If this is set to `True`, the colors of the target object will be remapped to match the colors in the source palette as closely as possible. By default, there will be no remapping and the actual pixel data of the target object will remain untouched. If you want remapping, set this tag to `True` but be warned that remapping all pixels will of course take much more time than just setting a new palette without remapping. Defaults to `False`.

`Dither:` If the `Remap` tag (see above) has been set to `True`, you can use the `Dither` tag to specify whether or not dithering should be used. Defaults to `True` which means dithering should be used.

`CopyCycleTable:`
Palettes can have a table containing color cycling information. If you set this tag to `True`, this cycle table will be copied to the target object as well. Defaults to `False`.

INPUTS

`id`	id of the palette to set
`t`	optional: table for specifying further options (see above)

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_MONOCHROME}
SetFillStyle(#FILLCOLOR)
SetPaletteMode(#PALETTEMODE_PEN)
SetDrawPen(1)
Box(#CENTER, #CENTER, 320, 240)
WaitLeftMouse
CreatePalette(1, {#WHITE, #BLACK}, {Depth = 1})
SetPalette(1)
```

The code above creates a monochrome palette display with a black background and white rectangle in the center. After a mouse-click the colors of the background and the white rectangle will be reversed by setting a new palette which uses white instead of black in pen 0 and black instead of white in pen 1.

44.30 SetPaletteDepth

NAME

SetPaletteDepth – set palette depth (V9.0)

SYNOPSIS

```
SetPaletteDepth(id, depth)
```

FUNCTION

This function sets the depth of the palette specified by `id` to the depth specified in `depth`. `depth` must be a bit depth ranging from 1 (= 2 colors) to 8 (= 256 colors). See Section 44.1 [Palette overview], page 889, for details.

INPUTS

`id`	identifier of palette to modify
`depth`	desired new palette depth (ranging from 1 to 8)

EXAMPLE

```
SetPaletteDepth(1, 8)
```

The code above changes the depth of palette 1 to 8 (= 256 colors).

44.31 SetPaletteMode

NAME

SetPaletteMode – set palette drawing mode (V9.0)

SYNOPSIS

```
SetPaletteMode(mode)
```

FUNCTION

This function sets the palette drawing mode to the mode specified in the `mode` argument. This mode will be used whenever the output device is palette-based, for example a palette mode display or a palette brush.

The following modes are currently supported:

`#PALETTEMODE_REMAP:`

All graphics that are drawn will be remapped to the output device's palette. This is the default palette mode but be warned that this can become very slow because Hollywood has to find the closest pen match for every single pixel it draws. To get the best drawing performance in palette mode, you should use `#PALETTEMODE_PEN` instead (see below). The way graphics data is remapped to a a palette output device can be configured by calling `SetDitherMode()`. This allows you to enable or disable dithering and you can also specify the dithering algorithm to use. Note that when using `#PALETTEMODE_REMAP` single-color drawing functions like `Box()`, `Circle()` or `TextOut()` won't draw using the drawing pen set via `SetDrawPen()` but using the RGB color that is passed to the function.

`#PALETTEMODE_PEN:`

When using `#PALETTEMODE_PEN`, all palette graphics will be copied to the output device without any pixel remapping. This is very fast but of course, it requires the palette of the graphics object that should be drawn and the palette of the output device to be the same or the result will have messed up colors. So if you use `#PALETTEMODE_PEN`, you should make sure that all your graphics objects share the same palette. Furthermore, when `#PALETTEMODE_PEN` is active, all single-color drawing functions like `Box()`, `Circle()` and `TextOut()` won't draw in the RGB color that you pass to them but they will all use the drawing pen set using `SetDrawPen()`.

The same is true for the shadow and border color: When palette mode is set to `#PALETTEMODE_PEN`, all graphics functions that support shadows and borders won't use the color specified in `SetFormStyle()`, `SetFontStyle()` or in standard draw tags like `ShadowColor` but they will use the pens that were specified using functions like `SetShadowPen()`, `SetBorderPen()` or draw tags like `ShadowPen` and `BorderPen`.

Furthermore, antialiasing of text and graphics primitives will be disabled when `#PALETTEMODE_PEN` is active because in most cases palettes don't have enough colors for satisfactorily anti-aliasing edges.

Note, however, that even if `#PALETTEMODE_PEN` is active, RGB graphics, of course, still have to be remapped because it's obviously impossible to draw RGB graphics to a palette output device without remapping the RGB colors to palette pens. Thus, drawing 32-bit true color graphics to palette output devices should be avoided because it will always be slow because remapping needs to be done for those graphics and there is no way around this.

The default drawing mode is `#PALETTEMODE_REMAP` but it is recommended to use `#PALETTEMODE_PEN` for performance reasons. See Section 25.16 [Palette displays], page 400, for details.

INPUTS

`mode` desired palette drawing mode (see above)

44.32 SetPalettePen

NAME

SetPalettePen – change palette pen (V9.0)

SYNOPSIS

```
SetPalettePen(id, pen, color)
```

FUNCTION

This function sets the color of the pen specified by `pen` to the color specified by `color` in the palette specified by `id`.

INPUTS

`id` identifier of palette

`pen` pen you want to modify (starting from 0)

`color` new color for the pen, must be specified as an RGB color

EXAMPLE

```
SetPalettePen(1, 0, #RED)
```

The code above sets pen 0 to red in palette 1.

44.33 SetPaletteTransparentPen

NAME

SetPaletteTransparentPen – set transparent pen of palette (V9.0)

SYNOPSIS

```
SetPaletteTransparentPen(id, pen)
```

FUNCTION

This function sets the transparent pen of the palette specified by `id` to the pen specified in `pen`. Pens are counted from 0.

INPUTS

`id` identifier of palette to use

`pen` desired transparent pen (starting from 0)

EXAMPLE

```
SetPaletteTransparentPen(1, 4)
```

The code makes pen 4 in palette 1 transparent.

44.34 SetPen

NAME

SetPen – change pen color (V9.0)

SYNOPSIS

```
SetPen(pen, color[, t])
```

FUNCTION

This function sets the color of the pen specified by `pen` to the color specified by `color`. The change will be done in the currently active palette. By default, the current display's palette is the active palette but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. A palette can be made the active one by using the `SelectPalette()` command.

Alternatively, you can also make `SetPen()` change pens in a different palette object. To do so, you need to pass the optional table argument `t` to `SetPen()` and specify the `Type` and `ID` tags. See below for an example.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object whose pen you want to modify. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose pen you want to modify. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose pen you want to modify. The default is the identifier of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the target type is an animation, sprite, or anim layer, you need to set this tag to specify the frame whose pen you want to modify. Frames are counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

INPUTS

`pen` pen you want to modify (starting from 0)

`color` new color for the pen, must be specified as an RGB color

`t` optional: table for specifying further options (see above)

EXAMPLE

```
@DISPLAY {Palette = #PALETTE_MONOCHROME}
SetFillStyle(#FILLCOLOR)
SetPaletteMode(#PALETTEMODE_PEN)
SetDrawPen(1)
Box(#CENTER, #CENTER, 320, 240)
WaitLeftMouse
SetPen(0, #WHITE)
SetPen(1, #BLACK)
```

The code above creates a monochrome palette display with a black background and white rectangle in the center. After a mouse-click the colors of the background and the white rectangle will be reversed by setting pen 0 to white and pen 1 to black.

```
SetPen(4, #RED, {Type = #BRUSH, ID = 2})
```

The code changes pen 4 in brush 2 to red.

44.35 SetShadowPen

NAME

SetShadowPen – set shadow pen (V9.0)

SYNOPSIS

```
SetShadowPen(pen)
```

FUNCTION

This function sets the pen specified by `pen` to the pen that will be used for drawing shadows when the palette mode is `#PALETTEMODE_PEN` and the current output device is a palette one. The palette mode can be set using `SetPaletteMode()`.

When the palette mode has been set to `#PALETTEMODE_PEN`, all Hollywood commands that draw a shadow won't use the shadow color that has been set using `SetFormStyle()`, `SetFontStyle()` or a `ShadowColor` tag but they will use the pen that has been set as the shadow pen using this function.

See Section 44.31 [SetPaletteMode], page 914, for details.

INPUTS

`pen`	pen to use for drawing shadows

44.36 SetStandardPalette

NAME

SetStandardPalette – copy colors from standard palette (V9.0)

SYNOPSIS

```
SetStandardPalette(id, type)
```

FUNCTION

This function can be used to copy the colors from the standard palette specified by `type` to the palette specified by `id`. Note that this might change the depth of the palette specified by `id` because the standard palette's depth is copied to the palette specified by `id` as well.

The following standard palettes are currently available:

`#PALETTE_MONOCHROME:`
: Two color, black and white palette.

`#PALETTE_GRAY4:`
: 4 color grayscale palette.

`#PALETTE_GRAY8:`
: 8 color grayscale palette.

`#PALETTE_GRAY16:`
: 16 color grayscale palette.

`#PALETTE_GRAY32:`
: 32 color grayscale palette.

`#PALETTE_GRAY64:`
: 64 color grayscale palette.

`#PALETTE_GRAY128:`
: 128 color grayscale palette.

`#PALETTE_GRAY256:`
: 256 color grayscale palette.

`#PALETTE_CGA:`
: Standard CGA palette (16 colors).

`#PALETTE_OCS:`
: Standard OCS palette (32 colors).

`#PALETTE_EGA:`
: Standard EGA palette (64 colors).

`#PALETTE_AGA:`
: Standard AGA palette (256 colors).

`#PALETTE_WORKBENCH:`
: Standard classic Amiga Workbench palette (256 colors).

`#PALETTE_MACINTOSH:`
: Standard classic Macintosh palette (256 colors).

`#PALETTE_WINDOWS:`
: Standard classic Windows palette (256 colors).

`#PALETTE_DEFAULT:`
: Same as `#PALETTE_AGA`.

INPUTS

`id` identifier of palette to use

`type` desired standard palette to copy to target palette

EXAMPLE

```
SetStandardPalette(1, #PALETTE_EGA)
```

The code above copies the standard EGA palette to palette 1. The new depth of palette 1 will be 6 (= 64 colors) after the operation.

44.37 SetTransparentPen

NAME

SetTransparentPen – set transparent pen (V9.0)

SYNOPSIS

```
SetTransparentPen(pen[, t])
```

FUNCTION

This function sets the transparent pen in the currently active palette to the one specified in `pen`. Pens are counted from 0. By default, the current display's palette is the active palette but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. A palette can be made the active one by using the `SelectPalette()` command.

Alternatively, the specified transparent pen can also be set to a different palette object. To do so, you need to pass the optional table argument to `SetTransparentPen()` and specify the `Type` and `ID` tags. See below for an example.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object whose transparent pen you want to set. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose transparent pen you want to modify. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose transparent pen you want to set. The default is the identifier of the currently active palette selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the target type is an animation, sprite, or anim layer, you need to set this tag to specify the frame whose transparent pen should be set. Frames are

counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

INPUTS

`pen` desired transparent pen (starting from 0)

`t` optional: table for specifying further options (see above)

EXAMPLE

```
SetTransparentPen(4, {Type = #BRUSH, ID = 2})
```

The code makes pen 4 in brush 2 transparent.

44.38 SetTransparentThreshold

NAME

SetTransparentThreshold – set alpha mapping threshold (V9.0)

SYNOPSIS

```
SetTransparentThreshold(threshold)
```

FUNCTION

This function can be used to specify a threshold value between 0 and 255 that should be used when quantizing alpha transparency to monochrome transparency. For example, when drawing an RGB brush with alpha transparency to a palette display, Hollywood needs to decide which pixels to remap and draw to the display and which pixels to ignore because they are (partly) transparent.

All pixels whose alpha value is less than or equal to the threshold value specified in `threshold` will be considered transparent. This defaults to 0 which means that only completely invisible pixels will be considered transparent. Depending on the actual image data you want to quantize, it might be necessary to choose a different threshold value here, however.

There is no "one size, fits all" best threshold value here. It all depends on the source image data you want to quantize. Sometimes you might want to have partially transparent pixels in the destination image, sometimes not. That's why it might be necessary to call this function with different threshold values depending on the actual image that needs to be quantized.

INPUTS

`threshold`

desired transparent pixel threshold (must be between 0 and 255); the default is 0

44.39 SolarizePalette

NAME

SolarizePalette – apply solarization effect to palette (V9.0)

SYNOPSIS

```
SolarizePalette(id, level)
```

FUNCTION

This command can be used to apply a solarization effect to the specified palette. The solarization effect tries to simulate the look of photographic film exposed to light. The second argument controls the intensity of the solarization effect and can be any value between 0 and 255, or a percentage specification inside a string.

INPUTS

`id`	palette to solarize
`level`	desired solarization level (0 to 255, or a string containing a percentage specification)

44.40 TintPalette

NAME

TintPalette – tint palette (V9.0)

SYNOPSIS

```
TintPalette(id, color, level)
```

FUNCTION

This function tints all colors in the palette specified by `id` with the RGB color specified in `color` using the tint level specified in `level`. The `level` argument must be between 0 (no tinting) to 255 (full tinting). Alternatively, `level` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id`	identifier of the palette to tint
`color`	a RGB color to use for tinting
`level`	tint level (0 to 255 or percent specification)

EXAMPLE

```
TintPalette(1, #RED, 128)
```

The code above adds some red to all colors in palette 1.

44.41 WritePen

NAME

WritePen – write pen to palette object (V9.0)

SYNOPSIS

```
WritePen(x, y, pen[, t])
```

FUNCTION

This function writes the pen specified by `pen` to the position specified by `x` and `y` in the currently active palette object. By default, the current display is the active palette object

but of course only in case the current display is a palette mode display. See Section 25.16 [Palette mode displays], page 400, for details. You can set the active palette object using the `SelectPalette()` command.

Alternatively, you can also use `WritePen()` to write a pen to a different palette object. To do so, you need to pass the optional table argument `t` to `WritePen()` and specify the `Type` and `ID` tags. See below for an example.

The following tags are supported by the optional table argument `t`:

`Type:` Set this to the type identifier of the object from whose pixel data you want to write to. This can be one of the following object types:

```
#ANIM
#BGPIC
#BRUSH
#DISPLAY
#LAYER
#PALETTE
#SPRITE
```

Note that if you use types `#ANIM` or `#SPRITE`, you also need to set the `Frame` tag (see below) to indicate the frame whose pixel data should be used. If you use `#LAYER` and the specified layer is an anim layer, you also need to set the `Frame` tag.

`Type` defaults to the type of the currently active palette object selected using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`ID:` Set this tag to the identifier of the object whose pixel data should be used. The default is the identifier of the currently active palette object set using `SelectPalette()`. See Section 44.21 [SelectPalette], page 906, for details.

`Frame:` If the target type is an animation, sprite, or anim layer, you need to set this tag to specify the frame whose pixel data should be used. Frames are counted from 1. Defaults to 1 when used with anims and sprites and to the current frame when used with anim layers.

INPUTS

`x` x position to write to

`y` y position to write to

`pen` the pen to write

`t` optional: table for specifying further options (see above)

EXAMPLE

```
WritePen(0, 0, 10, {Type = #BRUSH, ID = 2})
```

The code plots a pixel that uses pen 10 to the top-left corner in brush 2.

45 Plugin library

45.1 DisablePlugin

NAME
DisablePlugin – disable a plugin (V6.0)

SYNOPSIS
`DisablePlugin(name$)`

FUNCTION
This function can be used to disable the specified plugin. Please note that not all plugins can be disabled. Disabling plugins is only supported for plugins that provide loaders and savers for additional formats. It is not supported for plugins that replace complete core components inside Hollywood, e.g. by providing a custom display adapter. These plugins cannot be disabled.

To enable a plugin again later, call the `EnablePlugin()` function. See Section 45.2 [EnablePlugin], page 925, for details.

INPUTS
`name$` name of the plugin to disable

45.2 EnablePlugin

NAME
EnablePlugin – enable a plugin (V6.0)

SYNOPSIS
`EnablePlugin(name$)`

FUNCTION
This function can be used to enable the specified plugin. This is only necessary if you disabled it previously using `DisablePlugin()`.

INPUTS
`name$` name of the plugin to enable

45.3 GetPlugins

NAME
GetPlugins – get information about available plugins (V5.3)

SYNOPSIS
`t = GetPlugins()`

FUNCTION
This function returns a table that contains information about all available plugins. You will get one subtable for each plugin that has been loaded. Use `ListItems()` to find out

how many plugin subtables are in the table. Each subtable will contain the following fields:

`Name:` Name of the plugin.

`Version:` Version number of plugin.

`Revision:`
Revision number of plugin.

`Capabilities:`
Bitmask describing the Hollywood capabilities that this plugin extends. This will be a combination of the following capabilities:

```
#PLUGINCAPS_CONVERT
#PLUGINCAPS_LIBRARY
#PLUGINCAPS_IMAGE
#PLUGINCAPS_ANIM
#PLUGINCAPS_SOUND
#PLUGINCAPS_VECTOR
#PLUGINCAPS_VIDEO
#PLUGINCAPS_SAVEIMAGE
#PLUGINCAPS_SAVEANIM
#PLUGINCAPS_SAVESAMPLE
#PLUGINCAPS_REQUIRE
#PLUGINCAPS_DISPLAYADAPTER
#PLUGINCAPS_TIMERADAPTER
#PLUGINCAPS_REQUESTERADAPTER
#PLUGINCAPS_FILEADAPTER
#PLUGINCAPS_DIRADAPTER
#PLUGINCAPS_AUDIOADAPTER
#PLUGINCAPS_EXTENSION
#PLUGINCAPS_NETWORKADAPTER
#PLUGINCAPS_SERIALIZE
#PLUGINCAPS_ICON
#PLUGINCAPS_SAVEICON
#PLUGINCAPS_IPCADAPTER
#PLUGINCAPS_FONT
```

`Author:` Author of the plugin.

`Description:`
Description of the plugin.

`Copyright:`
Copyright string of the plugin.

`URL:` URL of the plugin's homepage (where to get updates, etc.)

`Date:` Compilation date of plugin.

`Settings:`
Fully qualified path to settings tool of the plugin.

`HelpFile:`
Fully qualified path to the help file of the plugin.

`Path:` Fully qualified path to the plugin's binary code.

`FileTypes:`
This item contains a subtable that contains tables describing all file formats that this plugin makes available. This is for example useful for adapting your file requesters to contain additional extensions that are supported by plugins. For every new file type there will be a table with the following fields initialized:

`Type:` Set to the type of the file format. This will be one of the following constants:

```
#FILETYPE_IMAGE
#FILETYPE_ANIM
#FILETYPE_SOUND
#FILETYPE_VIDEO
#FILETYPE_ICON
#FILETYPE_FONT
```

`Name:` Set to a string describing the name of the file format, e.g. `"TIFF"`.

`Extensions:`
Set to a string containing all extensions used by this file format. The extensions are separated by the `"|"` character and do not contain a dot, e.g. `"tif|tiff"`.

`MIMEType:`
Set to a string that describes the MIME of the file format. This can also be empty.

`Flags:` Set to a bitmask combination that describes the capabilities of this file type. The following flags are currently defined:

`#FILETYPEFLAGS_SAVE:`
If this flag is set, the entry describes a file type that this plugin can save. The `FormatID` tag will contain the constant used to refer to this plugin file type saver in that case.

`#FILETYPEFLAGS_ALPHA:`
Indicates that this file type supports alpha channel loading or saving (depending on whether `#FILETYPEFLAGS_SAVE` is set).

`#FILETYPEFLAGS_QUALITY:`
Only used for `#FILETYPE_IMAGE` or `#FILETYPE_ANIM` with save mode enabled. In that case this flag indicates that the image/anim saver supports different quality levels (ranging from 0 to 100).

`#FILETYPEFLAGS_FPS:`
Only used for `#FILETYPE_ANIM` with save mode enabled. In that case this flag indicates that the anim saver supports different frames per second settings.

`FormatID:`
If `#FILETYPEFLAGS_SAVE` is set in `Flags`, this tag will contain the constant identifier that has to be passed to the respective save function in order to use this file type saver. For example, for files of type `#FILETYPE_IMAGE` `FormatID` contains the identifier that has to be passed to `SaveBrush()`.

`ModuleName:`
Contains the plugin's module name. This is equal to the plugin's file name minus the file extension. The module name of the plugin is unique among all loaded plugins. Hollywood will never load two plugins with the same module name. (V6.0)

`Disabled:`
Tells you whether or not this plugin is currently disabled because of a call to `DisablePlugin()`. (V6.0)

INPUTS
none

RESULTS
`t` table containing information about the plugins loaded

45.4 HavePlugin

NAME
HavePlugin – check if a plugin is available (V6.0)

SYNOPSIS
```
ok, loaded = HavePlugin(name$[, version, revision])
```

FUNCTION
This function can be used to check if the plugin specified in `name$` is currently available. In that case `HavePlugin()` will return `True` in the first return value. The second return value indicates whether the plugin has been auto-loaded at startup or not. By default, Hollywood will auto-load all plugins at startup but this behaviour can be changed either by prefixing plugin filenames with an underscore character ('_') or by using the `-skipplugins` console argument. If the plugin has not been loaded, you can call `LoadPlugin()` on it to load it manually.

`HavePlugin()` accepts two optional arguments that can be used to check if a certain version of the plugin is available. Note that this can only be checked if the plugin has been loaded already. If the plugin hasn't been loaded yet, `HavePlugin()` won't be able to check its version.

INPUTS

`name$` plugin to check

`version` optional: version number to look for (defaults to 0 which means any version is acceptable)

`revision` optional: revision number to look for (defaults to 0 which means any revision is acceptable)

RESULTS

`ok` `True` if the plugin is available in the specified version

`loaded` `True` if the plugin has already been loaded

45.5 LoadPlugin

NAME

LoadPlugin – load a plugin at runtime (V6.0)

SYNOPSIS

```
LoadPlugin(name$[, table])
```

FUNCTION

This function can be used to load and initialize the specified plugin at runtime. `LoadPlugin()` does basically the same as `@REQUIRE` but can be called while your script is already running while `@REQUIRE` is executed by the preprocessor. As `LoadPlugin()` is a runtime function you cannot load certain plugins which require lowlevel initialization from this function, e.g. it is not possible to runtime-load plugins which install display adapters because Hollywood has already setup its inbuilt display driver by then. Runtime loading plugins which just install loaders or savers of additional file formats works fine, though.

`LoadPlugin()` accepts an optional table argument which can contain the following tags:

`Version:` Minimum plugin version required. Hollywood will fail if the installed plugin does not have at least this version number. This defaults to 0 which means that any version will do.

`Revision:`
Minimum plugin revision required. Hollywood will fail if the installed plugin does not have at least this revision number. This defaults to 0 which means that any revision will do.

`SkipRequire:`
Set this tag to `True` if you want Hollywood to skip calling the plugin's require initialization code. This is only useful for some advanced debugging purposes and should normally not be touched. Defaults to `False`.

Additionally, the optional table argument can contain an unlimited number of additional tags to be passed directly to the plugin's initialization routine exactly in the same way as done by the `@REQUIRE` preprocessor command. The additional argument acceptable

here depend on the respective plugin. Please consult the documentation of the plugin to find out if it accepts any additional parameters that can be passed here.

See Section 45.6 [REQUIRE], page 930, for details.

INPUTS

`name$` plugin to load

`table` optional: table containing further parameters (see above)

45.6 REQUIRE

NAME

REQUIRE – declare a plugin dependency (V5.0)

SYNOPSIS

```
@REQUIRE plugin$[, table]
```

FUNCTION

This preprocessor command allows you to declare an external plugin dependency which your script requires to run. If you use this preprocessor command, Hollywood will make sure that the specified plugin is installed before running your script. Optionally, you can pass additional parameters to the plugin which allows you to control how the plugin is initialized.

Please note that although Hollywood loads all plugins automatically on startup, many plugins require you to call `@REQUIRE` before they can be used. This is because these plugins need custom initialization code which is only run if you explicitly call `@REQUIRE` on them. For example, plugins which install a display adapter will not be activated unless you call `@REQUIRE` on them. Plugins which just add a loader or saver for additional file formats, however, will be automatically activated even if you don't call `@REQUIRE` on them.

This preprocessor command can also be used to load plugins which have been exempt from automatic loading at startup. Plugins can be exempt from auto-loading by prefixing their filename with an underscore character ('_') or by using the `-skipplugins` console argument. If you want to load a plugin that has been skipped by the auto loader, just call `@REQUIRE` on it from your script and it will be loaded by the preprocessor. Alternatively, you can also load these plugins using the `LoadPlugin()` function.

Starting with Hollywood 6.0 this preprocessor command accepts an optional table argument which allows you to pass additional parameters to the plugin's initialization routine. The parameters accepted here vary from plugin to plugin. Please consult the documentation of the plugin to find out if it accepts any additional parameters that can be passed to `@REQUIRE`. The following two parameters are supported for every plugin:

`Version:` Minimum plugin version required. Hollywood will fail if the installed plugin does not have at least this version number. This defaults to 0 which means that any version will do. (V6.0)

`Revision:`
: Minimum plugin revision required. Hollywood will fail if the installed plugin does not have at least this revision number. This defaults to 0 which means that any revision will do. (V6.0)

`Link:`
: If this tag is set to `True`, the specified plugin will be linked into your executable when compiling your script. This will only work if you've set up the plugin linker infrastructure correctly. See Section 4.5 [Linking plugins], page 61, for details. Make sure to carefully read the licenses of all plugins you link to your executable because licenses like LGPL affect your project if you statically link against LGPL software. This tag defaults to `False` which means that the plugin won't be linked. (V7.0)

Please note that you must not specify an absolute path in `plugin$`. Just pass the name of the plugin.

See Section 5.1 [Plugins], page 65, for more information on plugins.

INPUTS

`plugin$` name of the required plugin

`table` optional: table containing further options to be passed to the plugin (V6.0)

EXAMPLE

```
@REQUIRE "xml"
```

Declares that your script requires the "xml.hwp" plugin to be installed. Any version will be accepted.

```
@REQUIRE "myplugin", {Version = 2, Revision = 1, User = "John"}
```

The code above checks for version 2.1 of "myplugin.hwp" and also passes the additional argument "User=John" to the plugin's initialization code.

46 Requester library

46.1 ColorRequest

NAME

ColorRequest – prompt the user to select a color (V5.0)

SYNOPSIS

```
ret = ColorRequest(title$[, t])
```

DEPRECATED SYNTAX

```
ret = ColorRequest(title$[, color])
```

FUNCTION

This command opens a color requester prompting the user to select a color from a palette of predefined colors. Additionally, the user can also mix a custom color. The `title$` argument can be used to set the title for the color requester's dialog window. If an empty string (`""`) is passed as `title$`, the requester will use the title specified in the `@APPTITLE` preprocessor command.

`ColorRequest()` supports several optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `ColorRequest()`.

The following table fields are recognized by this function:

`Color:` This table tag can be used to specify a color that is initially selected. If you do not set this table tag, a random color will be preselected.

`X:` Initial x-position for the color requester on the screen. Not all platforms support this. (V9.0)

`Y:` Initial y-position for the color requester on the screen. Not all platforms support this. (V9.0)

`Width:` Initial width for the color requester dialog. Not all platforms support this. (V9.0)

`Height:` Initial height for the color requester dialog. Not all platforms support this. (V9.0)

Please note that this command requires reqtools.library to be installed on AmigaOS and compatibles.

INPUTS

`title$` title for the requester

`t` optional: table containing further arguments (see above) (V9.0)

RESULTS

`ret` the user's selection or -1 if the user cancelled the requester

EXAMPLE

```
r = ColorRequest("Select a color")
If r = -1
  Print("Requester cancelled!")
Else
  SetFillStyle(#FILLCOLOR)
  Box(0, 0, 640, 480, r)
EndIf
```

Ask the user for a color and then draws a filled rectangle using the selected color.

46.2 FileRequest

NAME

FileRequest – open a file requester

SYNOPSIS

```
f$ = FileRequest(title$[, t])
```

DEPRECATED SYNTAX

```
f$ = FileRequest(title$[, filter$, mode, defdir$, deffile$])
```

FUNCTION

This function opens a file requester that allows the user to select a file. You can specify the title of the requester by setting the `title$` argument. This can also be an empty string (`""`) to use the default title. The file that the user has selected will be returned in `f$` including the path where it resides. If the user cancels the requester, the string `f$` will be empty.

`FileRequest()` supports many optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `FileRequest()`.

The following table fields are recognized by this function:

`Mode:` This table tag allows you to put the requester into save or multiselect mode. For save mode, pass `#REQ_SAVEMODE` and for multiselect mode pass `#REQ_MULTISELECT`. If you use multiselect mode, this function will not return a string but a table that contains all the files the user selected terminated by an empty string. Starting with Hollywood 6.0 you can also set the flag `#REQ_HIDEICONS` if you want to have `*.info` files hidden on AmigaOS. Note that `#REQ_HIDEICONS` is a flag that can be combined with the other modes by ORing it into a bitmask. `#REQ_HIDEICONS` is only supported on AmigaOS. (V2.0)

`Path:` This table tag can be used to specify the initial path for the file requester. (V3.0)

`File:` This table tag can be used to specify the initial file for the file requester. (V3.0)

`Filters:` This table tag can be used to specify filters that define which files should be shown in the requester. This can either be a string or a table.

If it is a string, it must contain the extensions of the files that should be shown in the requester. These extensions must be separated by '`|`' characters. For example: `"voc|wav|8svx|16sv|iff|aiff"` will only show files which have one of those extensions. Make sure not to include the `.` before the file extension but just the actual extension. The default is `"*"` which means that all files should be shown.

Starting with Hollywood 9.0, you can also set `Filters` to a table to define individual groups of files and a description for each group. To do this, set `Filters` to a table that contains an arbitrary number of subtables, each describing a single group of files. Each subtable must contain the following items:

`Filter:` A string containing the file extensions of this group. This string must be in the same format as described above, i.e. the individual extensions must be separated by the vertical bar character (`|`), for example `"jpg|jpeg"`.

`Description:`
A string describing the filter group, e.g. `"JPEG images"`.

`HideFilter:`
This table item is optional. If you set it to `True`, `FileRequest()` won't show the individual file extensions that belong to the filter group but just its description. Note that not all platforms support this. Defaults to `False`.

`X:` Initial x-position for the file requester on the screen. Not all platforms support this. (V9.0)

`Y:` Initial y-position for the file requester on the screen. Not all platforms support this. (V9.0)

`Width:` Initial width for the file requester dialog. Not all platforms support this. (V9.0)

`Height:` Initial height for the file requester dialog. Not all platforms support this. (V9.0)

INPUTS

`title$` title for the requester; pass an empty string (`""`) to use the default title

`t` optional: table containing further arguments (see above) (V9.0)

RESULTS

`f$` the user's selection or an empty string if the requester was cancelled; if the requester was opened in multi-select mode, a table containing all files will be returned

EXAMPLE

```
f$ = FileRequest("Select a picture", {Filters = "png|jpg|jpeg|bmp"})
```

```
If f$ = ""
  Print("Requester cancelled!")
Else
  Print("Your selection:", f$)
EndIf
```

Ask the user for a file and print the result.

```
files = FileRequest("Select some files", {Mode = #REQ_MULTISELECT})
If files[0] = ""
  Print("Requester cancelled!")
Else
  NPrint("Path:", PathPart(files[0]))
  NPrint("Files selected:", ListItems(files) - 1)
  While files[c] <> ""
    NPrint(FilePart(files[c]))
    c = c + 1
  Wend
EndIf
```

The code above opens a multi-select file requester and prints all the files which the user selected.

```
f$ = FileRequest("Select file", {
  {Description = "Image files", Filter = "png|jpg|jpeg|bmp"},
  {Description = "Audio files", Filter = "wav|mp3|mp4"},
  {Description = "All files", Filter = "*"}
})
```

The code above shows how to use multiple filter groups with descriptions.

46.3 FontRequest

NAME

FontRequest – ask user to select a font (V5.0)

SYNOPSIS

```
t = FontRequest(title$[, t])
```

DEPRECATED SYNTAX

```
t = FontRequest(title$[, font$, size])
```

FUNCTION

This command will open a requester that will list all fonts currently available in the system. The user is then prompted to select a font from this list. The user can also choose an output size for the font, as well as the font style and font color. Note that the color selection is not supported on every platform. The `title$` argument specifies the title text for the requester's dialog window.

`FontRequest()` supports several additional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is

recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `FontRequest()`.

The following table fields are recognized by this function:

`Font:` Use this table tag to specify the name of a font that shall be initially selected.

`Size:` Use this table tag to specify the font size that shall be initially selected.

`X:` Initial x-position for the font requester on the screen. Not all platforms support this. (V9.0)

`Y:` Initial y-position for the font requester on the screen. Not all platforms support this. (V9.0)

`Width:` Initial width for the font requester dialog. Not all platforms support this. (V9.0)

`Height:` Initial height for the font requester dialog. Not all platforms support this. (V9.0)

Upon return, `FontRequest()` initializes a table containing all parameters selected by the user and returns this table to the script. The return table will have the following fields initialized:

`Name:` The complete font name (i.e. family name plus style). For example, `"Arial Bold Italic"`. This is a string you could pass directly to `SetFont()` or `OpenFont()`.

`Family:` The family name of this font, e.g. `"Arial"`.

`Size:` Contains the selected font size (e.g. 36).

`Weight:` The weight of the font. This will be set to one of the following weight constants:

```
#FONTWEIGHT_THIN
#FONTWEIGHT_EXTRALIGHT
#FONTWEIGHT_ULTRALIGHT
#FONTWEIGHT_LIGHT
#FONTWEIGHT_BOOK
#FONTWEIGHT_NORMAL
#FONTWEIGHT_REGULAR
#FONTWEIGHT_MEDIUM
#FONTWEIGHT_SEMIBOLD
#FONTWEIGHT_DEMIBOLD
#FONTWEIGHT_BOLD
#FONTWEIGHT_EXTRABOLD
#FONTWEIGHT_ULTRABOLD
#FONTWEIGHT_HEAVY
#FONTWEIGHT_BLACK
#FONTWEIGHT_EXTRABLACK
#FONTWEIGHT_ULTRABLACK
```

`Slant:` The slant style of the font. This will be set to one of the following slant constants:

```
#FONTSLANT_ROMAN
#FONTSLANT_ITALIC
#FONTSLANT_OBLIQUE
```

`Bold:` `True` if the user chose a bold font style.

`Italic:` `True` if the user chose an italic font style.

`Underline:` `True` if the user chose an underlined font style.

`StrikeOut:` `True` if the user chose a striked out font style.

`Color:` The font color chosen by the user in RGB format.

Please note that the `Underline`, `StrikeOut`, and `Color` fields are not supported on all platforms. If the host operating system's font dialog does not support them, they will all be set to `False`.

INPUTS

`title$` title for the requester

`t` optional: table containing further arguments (see above) (V9.0)

RESULTS

`t` a table containing all parameters chosen by the user (see above for a description of the table fields)

EXAMPLE

```
t = FontRequest("Select a font")
NPrint("Font:", t.name)
NPrint("Family:", t.family)
NPrint("Size:", t.size)
NPrint("Weight:", t.weight)
NPrint("Slant:", t.slant)
NPrint("Underline:", t.underline)
NPrint("Strike:", t.strikeout)
NPrint("Color:", HexStr(t.color))
```

The code above pops up a font requester and then prints out all information gathered from the user.

46.4 ImageRequest

NAME

ImageRequest – prompt the user to select an image (V8.0)

SYNOPSIS

```
[id] = ImageRequest(id[, type])
```

PLATFORMS

Android only

FUNCTION

This function can be used to prompt the user to select an image. The image will then be stored as the brush specified in `id`. If you specify `Nil` in the `id` argument, `ImageRequest()` will automatically choose an identifier for the brush and return the identifier to you.

The optional type argument allows you to specify the image source to use when prompting the user for an image. This can currently be set to one of the following predefined constants:

`#REQ_GALLERY:`
: Open the device's gallery and prompt the user to select an image that will then be returned as a Hollywood brush to your script.

`#REQ_CAMERA:`
: Open the device's camera and prompt the user to take a picture that will then be returned as a Hollywood brush to your script.

The default mode is `#REQ_GALLERY`, i.e. `ImageRequest()` will prompt the user to select an image from the device's gallery.

To find out if this function has failed because the user cancelled the image requester, just use `HaveObject()` to see if the brush object exists after `ImageRequest()` returns. If it doesn't exist, the user has cancelled the image requester. See Section 43.8 [HaveObject], page 885, for details.

INPUTS

`id` — id for the brush or `Nil` for auto id selection

`type` — optional: image source to use; see above for possible modes; defaults to `#REQ_GALLERY`

RESULTS

`id` — optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
ImageRequest(1, #REQ_CAMERA)
If HaveObject(#BRUSH, 1)
   DisplayBrush(1, #CENTER, #CENTER)
Else
   NPrint("Requester cancelled!")
EndIf
```

The code above prompts the user to take a picture with the camera and then displays this picture. It also checks if the requester has been cancelled.

46.5 ListRequest

NAME

ListRequest – prompt choice from a list of options (V5.0)

SYNOPSIS

```
choice = ListRequest(title$, body$, choices[, t])
```

DEPRECATED SYNTAX

```
choice = ListRequest(title$, body$, choices[, active])
```

FUNCTION

This command can be used to present a list of choices to the user and ask him to select one of the list entries. The first argument specifies the title text for the requester's dialog window. The second argument specifies the body text that shall appear above the list of choices. The third argument must be a table containing an arbitrary number of strings from which the user shall be able to choose.

When `ListRequest()` returns, you will receive the index of the list entry that the user has selected as the return value. If the user hasn't selected an item or cancelled the requester, -1 will be returned.

`ListRequest()` supports several additional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `ListRequest()`.

The following table fields are recognized by this function:

`Active:` This table tag can be used to preselect one of the choices in the list. Simply pass the index of the entry to preselect in the `Active` table tag. Indices start at 0 for the first entry and run through number of entries minus 1. If `Active` is omitted or out of range, nothing will be preselected.

`X:` Initial x-position for the list requester on the screen. Not all platforms support this. (V9.0)

`Y:` Initial y-position for the list requester on the screen. Not all platforms support this. (V9.0)

`Width:` Initial width for the list requester dialog. Not all platforms support this. (V9.0)

`Height:` Initial height for the list requester dialog. Not all platforms support this. (V9.0)

Starting with Hollywood 6.0 you can pass an empty string (`""`) as `title$`. In that case, the requester will use the title specified in the `@APPTITLE` preprocessor command.

INPUTS

`title$` title for the requester

`body$` body text to display above the list view widget

`choices` table containing a number of string entries that constitute the available choices

`t` optional: table containing further arguments (see above) (V9.0)

RESULTS

`choice` index of the user's selection or -1 if the user cancelled the requester; indices start at 0 for the first entry and run through the number of entries minus 1

EXAMPLE

```
r = ListRequest("User prompt", "Which of these is not an island?",
{"Australia", "Fiji", "New Zealand", "Easter Island", "Hawaii",
"Goa", "Madagascar", "Maldives", "Seychelles"})
If r = -1
  Print("You chose the chicken exit!")
ElseIf r = 5
  Print("That's right, congratulations!")
Else
  Print("Sorry, but that is an island...")
EndIf
```

The code above shows how to use `ListRequest()` for a little quiz.

46.6 PathRequest

NAME

PathRequest – pop up a path requester (V2.0)

SYNOPSIS

```
p$ = PathRequest(title$[, t])
```

DEPRECATED SYNTAX

```
p$ = PathRequest(title$[, mode, defdir$])
```

FUNCTION

This function opens a path requester that allows the user to select a path. You can specify the title of the requester by setting the `title$` argument. This can also be an empty string (`""`) to use the default title.

`PathRequest()` returns the user's path selection or `""` if the user has cancelled the requester.

`PathRequest()` supports several optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `PathRequest()`.

The following table fields are recognized by this function:

`Mode:` The `Mode` table tag allows you to specify the mode of the path requester. This can either be `#REQ_SAVEMODE` for save mode or `#REQ_NORMAL` for normal mode. Defaults to `#REQ_NORMAL`.

`Path:` This table tag can be used to specify the initial path for the path requester. (V3.0) By default, this will be the current directory.

`X:` Initial x-position for the path requester on the screen. Not all platforms support this. (V9.0)

`Y:` Initial y-position for the path requester on the screen. Not all platforms support this. (V9.0)

`Width:` Initial width for the path requester dialog. Not all platforms support this. (V9.0)

`Height:` Initial height for the path requester dialog. Not all platforms support this. (V9.0)

INPUTS

`title$` title for the requester; pass an empty string (`""`) to use the default title

`t` optional: table containing further arguments (see above) (V9.0)

RESULTS

`p$` the user's selection or an empty string if he cancelled the requester

EXAMPLE

```
p$ = PathRequest("Select a path")
If p$ = ""
  Print("Requester cancelled!")
Else
  Print("Your selection: ")
  Print(p$)
EndIf
```

Ask the user for a path and print its name.

46.7 PermissionRequest

NAME

PermissionRequest – request permission from user (V8.0)

SYNOPSIS

```
ok = PermissionRequest(perms)
```

PLATFORMS

Android only

FUNCTION

This function can be used to request certain permissions from the user. Due to security reasons, Android apps need first ask for user permission before they will be able to execute certain actions. This function can be used to request such permissions from the user. Android will then show a dialog box in which the user can either accept or decline the permissions. If he declines, `PermissionRequest()` will return `False`, otherwise `True` will be returned.

The permissions you want to request have to be passed in the `perms` argument. This can be set to one or more of the following permission flags:

`#PERMREQ_READEXTERNAL:`
: If your app has this permission, it will be able to read files from the external storage device. The external storage device can be accessed through the `SDCard` item in the table returned by `GetSystemInfo()`. By default, Android apps are not allowed to read from the external storage device.

`#PERMREQ_WRITEEXTERNAL:`
: If your app has this permission, it will be able to write and read files to/from the external storage device. The external storage device can be accessed through the `SDCard` item in the table returned by `GetSystemInfo()`. By default, Android apps are not allowed to write to the external storage device. Note that `#PERMREQ_WRITEEXTERNAL` implies `#PERMREQ_READEXTERNAL` so you don't have to set `#PERMREQ_READEXTERNAL` when using this flag.

To ask for multiple permissions at once, simply combine them using the bitwise Or operator.

Note that this function is only needed when compiling stand-alone APKs using the Hollywood APK Compiler. When using the Hollywood Player, the Hollywood Player will automatically request the `#PERMREQ_WRITEEXTERNAL` permission for you so you don't have to do that manually.

INPUTS

`perms` one or more permissions to request (see above for possible values)

RESULTS

`ok` `True` if user granted permission, `False` if he declined them

EXAMPLE

```
If PermissionRequest(#PERMREQ_WRITEEXTERNAL)
  t = GetSystemInfo()
  StringToFile("Hello World", FullPath(t.SDCard, "test.txt"))
Else
  NPrint("Sorry, no permission!")
EndIf
```

The code above tries to get a permission from the user to write to the external storage device. If the user grants this permission, the code will write a file named `test.txt` to the external storage device.

46.8 StringRequest

NAME

StringRequest – ask the user to enter a string (V2.0)

SYNOPSIS

```
s$, ok = StringRequest(title$, body$[, t])
```

DEPRECATED SYNTAX

```
s$, ok = StringRequest(title$, body$[, def$, type, maxchars, password])
```

FUNCTION

This function opens a requester prompting the user to enter a string. You can specify the title for the requester window in `title$` and the body text in `body$`. If you pass an empty string (`""`) `title$`, the requester will use the title specified in the `@APPTITLE` preprocessor command.

`StringRequest()` will return the string the user has entered if the user acknowledges the requester. If the user cancels this requester, an empty string will be returned. The second return value allows you identify whether or not the user pressed the 'OK' button. This is normally only needed if your application allows an empty string on 'OK'. In that case you need to check the second return value, too.

`StringRequest()` supports many optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `StringRequest()`.

The following table fields are recognized by this function:

`Type:` This table tag can be used to specify which characters the user is allowed to enter. This can currently be `#NUMERICAL` for numbers only or `#ALL` for no restriction on characters that can be entered. Defaults to `#ALL`.

`Password:` Set this table tag to `True` to put the requester in password mode. In that case, the user's input will be hidden. Defaults to `False`.

`MaxLength:` This table tag can be used to specify the number of characters the user will be allowed to enter. This defaults to 0, which means that there is no limit concerning the number of characters the user may enter.

`Text:` This table tag can be used to specify the default text for the string requester. The text you specify here will be initially shown in the string requester's text entry widget.

`X:` Initial x-position for the string requester on the screen. Not all platforms support this. (V9.0)

`Y:` Initial y-position for the string requester on the screen. Not all platforms support this. (V9.0)

`Width:` Initial width for the string requester dialog. Not all platforms support this. (V9.0)

`Height:` Initial height for the string requester dialog. Not all platforms support this. (V9.0)

Please note that this command requires reqtools.library to be installed on AmigaOS 3, MorphOS, and AROS. Under AmigaOS 4 the `StringRequest()` function works without reqtools.library.

INPUTS

`title$` title for the requester window

`body$` text for the requester body

`t` optional: table containing further arguments (see above) (V9.0)

RESULTS

`s$` the string entered by the user or `""` if requester was cancelled

`ok` `True` if the user pressed the 'OK' button, `False` otherwise (V4.5)

EXAMPLE

```
a$ = StringRequest("My Program", "Please enter your name!")
Print("Hello,", a$, "!")
```

Ask the user for his name and print it out.

46.9 SystemRequest

NAME

SystemRequest – pop up a choice requester

SYNOPSIS

```
res = SystemRequest(title$, body$, buttons$[, icon])
```

FUNCTION

This function pops up a standard system requester that displays a message (body$) and also allows the user to make a selection using one of the buttons specified by `buttons$`. Separate the buttons specified in `buttons$` by a `"|"`. The return value tells you which button the user has pressed. Please note that the right most button always has the value of `False` (0) because it is typically used as the "Cancel" button. For example, if you have three buttons "One|Two|Three", button "Three" has a return value of 0, "Two" returns 2, and "One" returns 1.

New in Hollywood 4.0: You can use the optional argument icon to add a little icon to the requester. The following icons are possible:

`#REQICON_NONE:`
No icon

`#REQICON_INFORMATION:`
An information sign

`#REQICON_ERROR:`
An error sign

`#REQICON_WARNING:`
A warning sign

`#REQICON_QUESTION:`
A question mark

Please note that currently requester icons are not supported on every platform that Hollywood runs on.

Starting with Hollywood 6.0, `SystemRequest()` might map the following button specifications to certain system-specific buttons: "OK", "OK|Cancel", "Yes|No" and

"Yes|No|Cancel". This means that it could happen that the buttons suddenly appear in the user's language instead of the English versions you passed to `SystemRequest()`. It can also mean that the order of the buttons is changed, e.g. on macOS the "OK" button is typically placed to the right of the "Cancel" button whereas on other systems it is the other way round. Nevertheless, the return values will always be consistent, i.e. "OK" or "Yes" will always have a return value of 1 whereas "Cancel" or "No" has a return value of 0 the only exception being "Yes|No|Cancel" where "No" has a return value of 2 because there is also a "Cancel" button which has a return value of 0.

Also, it is possible to pass an empty string ("") in the first parameter since Hollywood 6.0. In that case, the requester will use the title specified in the `@APPTITLE` preprocessor command.

INPUTS

`title$`	title for the requester
`body$`	text to appear in the body of the requester
`buttons$`	one or more buttons that the user can press
`icon`	optional: icon to show in the requester (defaults to `#REQICON_NONE`) (V4.0)

RESULTS

`res`	the button that was pressed by the user

EXAMPLE

```
sel = SystemRequest("Pizza Service", "Select your pizza!",
                    "Prosciutto e funghi|Calzone|Margerita|Hawaii")
Switch sel
Case 1:
  Print("1x Prosciutto e funghi = 8 Euro")
Case 2:
  Print("1x Calzone = 10 Euro")
Case 3:
  Print("1x Margerita = 9 Euro")
Case 0:
  Print("1x Hawaii = 12 Euro")
EndSwitch
```

The above code asks the user for a pizza and displays the price of that pizza.

47 Serial port library

47.1 ClearSerialQueue

NAME

ClearSerialQueue – clear serial port read buffer (V8.0)

SYNOPSIS

```
ClearSerialQueue(id)
```

FUNCTION

This function can be used to clear the read buffer of the serial port connection specified by `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

To poll the number of bytes currently in the read buffer, use the `PollSerialQueue()` function. See Section 47.12 [PollSerialQueue], page 954, for details.

Note that this function is currently unsupported on Android.

INPUTS

`id`	identifier of the serial port whose read buffer shall be cleared

47.2 CloseSerialPort

NAME

CloseSerialPort – close serial port connection (V8.0)

SYNOPSIS

```
CloseSerialPort(id)
```

FUNCTION

This function can be used to close the serial port connection specified by `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You should always close serial port connections as soon as you are done with them.

INPUTS

`id`	identifier of the serial port connection to close

47.3 FlushSerialPort

NAME

FlushSerialPort – flush serial port connection (V8.0)

SYNOPSIS

```
FlushSerialPort(id)
```

FUNCTION

This function can be used to flush the serial port connection specified by `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

INPUTS

`id` identifier of the serial port connection to flush

47.4 GetBaudRate

NAME

GetBaudRate – get baud rate for serial port connection (V8.0)

SYNOPSIS

```
baud = GetBaudRate(id)
```

FUNCTION

This command can be used to get the baud rate for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#BAUD_300:`
300 bits per second.

`#BAUD_600:`
600 bits per second. (V9.0)

`#BAUD_1200:`
1200 bits per second. (V9.0)

`#BAUD_2400:`
2400 bits per second.

`#BAUD_4800:`
4800 bits per second.

`#BAUD_9600:`
9600 bits per second.

`#BAUD_19200:`
19200 bits per second.

`#BAUD_38400:`
38400 bits per second.

`#BAUD_57600:`
57600 bits per second.

`#BAUD_115200:`
115200 bits per second.

`#BAUD_460800:`
460800 bits per second.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`baud` current baud rate as a special constant (see above)

47.5 GetDataBits

NAME

GetDataBits – get data bits for serial port connection (V8.0)

SYNOPSIS

```
bits = GetDataBits(id)
```

FUNCTION

This command can be used to get the number of data bits for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#DATA_5:` Use 5 data bits.

`#DATA_6:` Use 6 data bits.

`#DATA_7:` Use 7 data bits.

`#DATE_8:` Use 8 data bits.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`bits` current data bits as a special constant (see above)

47.6 GetDTR

NAME

GetDTR – get DTR pin state for serial port connection (V8.0)

SYNOPSIS

```
state = GetDTR(id)
```

FUNCTION

This command can be used to get the DTR pin state for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#DTR_ON:` DTR pin is set.

`#DTR_OFF:`
DTR pin is cleared.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`state` state of DTR pin as a special constant (see above)

47.7 GetFlowControl

NAME

GetFlowControl – get flow control for serial port connection (V8.0)

SYNOPSIS

```
flow = GetFlowControl(id)
```

FUNCTION

This command can be used to get the flow control for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#FLOW_OFF:`
: No flow control.

`#FLOW_HARDWARE:`
: Hardware flow control using CTS/RTS.

`#FLOW_XON_XOFF:`
: Software flow control using XON/XOFF handshaking.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`flow` flow control as a special constant (see above)

47.8 GetParity

NAME

GetParity – get parity mode for serial port connection (V8.0)

SYNOPSIS

```
parity = GetParity(id)
```

FUNCTION

This command can be used to get the parity mode for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#PARITY_NONE:`
: Do not use any parity bit.

`#PARITY_EVEN:`
: Use 1 bit of even parity.

`#PARITY_ODD:`
: Use 1 bit of odd parity.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`parity` parity bit as a special constant (see above)

47.9 GetRTS

NAME

GetRTS – get RTS pin state for serial port connection (V8.0)

SYNOPSIS

```
state = GetRTS(id)
```

FUNCTION

This command can be used to get the RTS pin state for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#RTS_ON:` RTS pin is set.

`#RTS_OFF:`
RTS pin is cleared.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`state` state of RTS pin as a special constant (see above)

47.10 GetStopBits

NAME

GetStopBits – get stop bits for serial port connection (V8.0)

SYNOPSIS

```
bits = GetStopBits(id)
```

FUNCTION

This command can be used to get the number of stop bits for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

The return value will be one of the following special constants:

`#STOP_1:` Use 1 stop bit.

`#STOP_2:` Use 2 stop bits.

INPUTS

`id` identifier of the serial port connection to use

RESULTS

`bits` current stop bits as a special constant (see above)

47.11 OpenSerialPort

NAME

OpenSerialPort – open serial port connection (V8.0)

SYNOPSIS

```
[id] = OpenSerialPort(id, portname$[, table])
```

FUNCTION

This function can be used to open a connection to the serial port specified in `portname$` and assign the identifier `id` to the connection. If you pass `Nil` in `id`, `OpenSerialPort()` will automatically choose an identifier and return it.

The name you pass in `portname$` depends on the platform your script is running on. On Windows it could be `COM1`, on Linux and macOS it could be `/dev/ttyS0` or `/dev/ttyUSB0` in case you're using a USB adapter. On AmigaOS you have to pass the serial.device unit you want to open in `portname$` and on Android it is assumed that there is only one port so `portname$` is ignored.

Starting with Hollywood 9.0, `portname$` can also be a string in the format "<device-name>:<port>" now on AmigaOS and compatibles. This is useful in case you want `OpenSerialPort()` to open an alternative serial device instead of AmigaOS's standard `serial.device`. For example, passing "serialpl2303.device:0" in `portname$` will try to open `serialpl2303.device` on port 0.

Additionally, you can pass an optional table argument allowing you to set the parameters for the serial port connection. The following fields are currently recognized:

`BaudRate:`

The desired baud rate for the connection. This can be one of the following special constants:

`#BAUD_300:`

300 bits per second.

`#BAUD_600:`

600 bits per second. (V9.0)

`#BAUD_1200:`

1200 bits per second. (V9.0)

`#BAUD_2400:`

2400 bits per second.

`#BAUD_4800:`

4800 bits per second.

`#BAUD_9600:`

9600 bits per second. This is the default.

`#BAUD_19200:`
: 19200 bits per second.

`#BAUD_38400:`
: 38400 bits per second.

`#BAUD_57600:`
: 57600 bits per second.

`#BAUD_115200:`
: 115200 bits per second.

`#BAUD_460800:`
: 460800 bits per second.

`DataBits:`
: The desired data bits for the connection. This can be set to one of the following special constants:

`#DATA_5:` Use 5 data bits.

`#DATA_6:` Use 6 data bits.

`#DATA_7:` Use 7 data bits.

`#DATE_8:` Use 8 data bits. This is the default.

`StopBits:`
: The desired stop bits for the connection. This can be set to one of the following special constants:

`#STOP_1:` Use 1 stop bit. This is the default.

`#STOP_2:` Use 2 stop bits.

`Parity:` The desired parity mode. This can be set to one of the following special constants:

`#PARITY_NONE:`
: Do not use any parity bit. This is the default.

`#PARITY_EVEN:`
: Use 1 bit of even parity.

`#PARITY_ODD:`
: Use 1 bit of odd parity.

`FlowControl:`
: The desired type of flow control to use. This can be set to one of the following special constants:

`#FLOW_OFF:`
: Do not use any flow control. This is the default.

`#FLOW_HARDWARE:`
: Use hardware flow control using CTS/RTS.

`#FLOW_XON_XOFF:`
: Use software flow control using XON/XOFF handshaking.

`RTS:` The desired state of the RTS pin. Note that manually setting the RTS pin isn't supported on every platform. Where supported, it can be set to one of the following special constants:

`#RTS_ON:` Set the RTS pin.

`#RTS_OFF:` Clear the RTS pin.

`DTR:` The desired state of the DTR pin. Note that manually setting the DTR pin isn't supported on every platform. Where supported, it can be set to one of the following special constants:

`#DTR_ON:` Set the DTR pin.

`#DTR_OFF:` Clear the DTR pin.

As you can see above, the default configuration used by `OpenSerialPort()` is 9600/8-N-1, i.e. 9600 bps, 8 data bits, no parity bit, 1 stop bit. This is the most common configuration and should work on every platform.

INPUTS

`id` identifier for the new serial connection or `Nil` for auto id selection

`portname$` serial port to open

`table` optional: further options (see above)

RESULTS

`id` optional: identifier of the serial port connection; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenSerialPort(1, "COM1")
WriteSerialData(1, "Hello World!")
CloseSerialPort(1)
```

The code above opens the serial port COM1 on Windows, sends the string "Hello World!" to the receiver and closes the serial port connection. Note that there is no guarantee that all 12 bytes could be sent to the serial port. In stable code, you would have to check the return value of `WriteSerialData()` and call it again if necessary to send the remaining bytes.

47.12 PollSerialQueue

NAME

PollSerialQueue – poll number of bytes in read buffer (V8.0)

SYNOPSIS

```
n = PollSerialQueue(id)
```

FUNCTION

This function can be used to poll the number of bytes currently in the read buffer of the serial port connection specified by `id`. This serial port connection must have been opened using `OpenSerialPort()` before.

To clear a serial port connection's read buffer, use the `ClearSerialQueue()` command. See Section 47.1 [ClearSerialQueue], page 947, for details.

Note that this function is currently unsupported on Android.

INPUTS

`id`	identifier of the serial port whose read buffer you want to poll

RESULTS

`n`	number of bytes in read buffer

47.13 ReadSerialData

NAME

ReadSerialData – read data from serial port connection (V8.0)

SYNOPSIS

```
data$, count = ReadSerialData(id, len[, timeout])
```

FUNCTION

This command can be used to read `len` bytes of data from the serial port connection specified in `id`. The serial port connection must have been opened using `OpenSerialPort()` before. Additionally, you can pass a duration in milliseconds in the `timeout` argument to set a timeout for the read operation. If the `timeout` parameter is specified, `ReadSerialData()` will never block for longer than the specified duration. Otherwise it will wait forever for data to arrive.

`ReadSerialData()` will return the data it has read from the serial port and the length of the data in bytes. Note that this can be less than the length specified in `len`. If `ReadSerialData()` returns less bytes than you requested in `len`, you have to call `ReadSerialData()` again and again until you have received all the data you need.

Note that the value returned in `count` will always be the same as the `ByteLen()` for `data$`. The only reason for the `count` return value is a performance gain because in that way you don't have to call `ByteLen()` to calculate the length of `data$`.

To poll the number of bytes currently in the read buffer, use the `PollSerialQueue()` function. See Section 47.12 [PollSerialQueue], page 954, for details.

INPUTS

`id`	identifier of the serial port connection to use
`len`	number of bytes to read from the serial port
`timeout`	optional: number of milliseconds after which to abort the operation (defaults to 0 which means to block forever until data arrives)

RESULTS

`data$`	the data read from the serial port

`count` number of bytes read from the serial port

EXAMPLE

```
OpenSerialPort(1, "COM1")
Print(ReadSerialData(1, 256))
```

The code above will wait forever for data to arrive from the serial port. As soon as something arrives, it will return and print it. This can be less than 256 bytes. The only thing that is guaranteed is that it will never be more than 256 bytes.

47.14 SetBaudRate

NAME

SetBaudRate – set baud rate for serial port connection (V8.0)

SYNOPSIS

```
SetBaudRate(id, baud)
```

FUNCTION

This command can be used to set the baud rate for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired baud rate in the `baud` parameter. This must be one of the following special constants:

`#BAUD_300:`
300 bits per second.

`#BAUD_600:`
600 bits per second. (V9.0)

`#BAUD_1200:`
1200 bits per second. (V9.0)

`#BAUD_2400:`
2400 bits per second.

`#BAUD_4800:`
4800 bits per second.

`#BAUD_9600:`
9600 bits per second.

`#BAUD_19200:`
19200 bits per second.

`#BAUD_38400:`
38400 bits per second.

`#BAUD_57600:`
57600 bits per second.

`#BAUD_115200:`
115200 bits per second.

`#BAUD_460800:`
460800 bits per second.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired baud rate

47.15 SetDataBits

NAME

SetDataBits – set data bits for serial port connection (V8.0)

SYNOPSIS

```
SetDataBits(id, bits)
```

FUNCTION

This command can be used to set the number of data bits for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired data bits in the `bits` parameter. This must be one of the following special constants:

`#DATA_5:` Use 5 data bits.

`#DATA_6:` Use 6 data bits.

`#DATA_7:` Use 7 data bits.

`#DATE_8:` Use 8 data bits.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired data bits

47.16 SetDTR

NAME

SetDTR – set DTR pin state for serial port connection (V8.0)

SYNOPSIS

```
SetDTR(id, state)
```

FUNCTION

This command can be used to set the DTR pin state for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired state in the `state` parameter. This must be one of the following special constants:

`#DTR_ON:` Set the DTR pin.

`#DTR_OFF:`
Clear the DTR pin.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired DTR pin state

47.17 SetFlowControl

NAME

SetFlowControl – set flow control for serial port connection (V8.0)

SYNOPSIS

```
SetFlowControl(id, flow)
```

FUNCTION

This command can be used to set the flow control for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired flow control mode in the `flow` parameter. This must be one of the following special constants:

`#FLOW_OFF:`
Do not use any flow control.

`#FLOW_HARDWARE:`
Use hardware flow control using CTS/RTS.

`#FLOW_XON_XOFF:`
Use software flow control using XON/XOFF handshaking.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired flow control type

47.18 SetParity

NAME

SetParity – set parity mode for serial port connection (V8.0)

SYNOPSIS

```
SetParity(id, parity)
```

FUNCTION

This command can be used to set the parity mode for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired parity mode in the `parity` parameter. This must be one of the following special constants:

`#PARITY_NONE:`
Do not use any parity bit.

`#PARITY_EVEN:`
Use 1 bit of even parity.

`#PARITY_ODD:`
Use 1 bit of odd parity.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired parity mode

47.19 SetRTS

NAME
SetRTS – set RTS pin state for serial port connection (V8.0)

SYNOPSIS
`SetRTS(id, state)`

FUNCTION
This command can be used to set the RTS pin state for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired state in the `state` parameter. This must be one of the following special constants:

`#RTS_ON:` Set the RTS pin.

`#RTS_OFF:`
Clear the RTS pin.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired RTS pin state

47.20 SetStopBits

NAME
SetStopBits – set stop bits for serial port connection (V8.0)

SYNOPSIS
`SetStopBits(id, bits)`

FUNCTION
This command can be used to set the number of stop bits for the serial port connection specified in `id`. This serial port connection must have been opened using `OpenSerialPort()` before. You have to pass the desired stop bits in the `bits` parameter. This must be one of the following special constants:

`#STOP_1:` Use 1 stop bit.

`#STOP_2:` Use 2 stop bits.

INPUTS

`id` identifier of the serial port connection to use

`baud` desired stop bits

47.21 WriteSerialData

NAME

WriteSerialData – write data to serial port connection (V8.0)

SYNOPSIS

```
count = WriteSerialData(id, data$[, timeout])
```

FUNCTION

This command can be used to write the data specified in `data$` to the serial port connection specified in `id`. The serial port connection must have been opened using `OpenSerialPort()` before. Additionally, you can pass a duration in milliseconds in the `timeout` argument to set a timeout for the write operation. If the `timeout` parameter is specified, `WriteSerialData()` will never block for longer than the specified duration.

`WriteSerialData()` returns the number of bytes written to the serial port. Note that this can be less than the bytes in `data$`. If only parts of `data$` have been sent to the serial port, you need to call `WriteSerialData()` again to send the rest.

INPUTS

`id` identifier of the serial port connection to use

`data$` the data to write to the serial port

`timeout` optional: number of milliseconds after which to abort the operation (defaults to 0 which means to block forever until data can be sent)

RESULTS

`count` number of bytes successfully written

EXAMPLE

See Section 47.11 [OpenSerialPort], page 952.

48 Serializer library

48.1 DeserializeTable

NAME

DeserializeTable – deserialize string to table (V9.0)

SYNOPSIS

```
table = DeserializeTable(s$[, t])
```

DEPRECATED SYNTAX

```
table = DeserializeTable(s$[, adapter])
```

FUNCTION

This function deserializes the string specified by `s$` to a table and returns it.

The optional table argument `t` can be used to specify the following additional options:

`Adapter:` The table will be deserialized using the deserializer that is specified in the `Adapter` tag. This can be the name of an external deserializer plugin (e.g. `xml`) or it can be one of the following inbuilt deserializers:

- `Default:` Use Hollywood's default deserializer. This will deserialize data from the JSON format to a Hollywood table. This is also the default if no other default has been set using `SetDefaultAdapter()`.
- `Inbuilt:` Use Hollywood's legacy deserializer. Using this deserializer is not recommended any longer as the data is in a proprietary, non-human-readable format. Using JSON is a much better choice.

`UserTags:`
This tag can be used to specify additional data that should be passed to serializer plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Mode:` This tag can be used to set the serialization mode to use for the operation. It defaults to the serialization mode set using `SetSerializeMode()`. See Section 48.5 [SetSerializeMode], page 965, for details. (V10.0)

`Options:` This tag can be used to set the serialization options to use for the operation. It defaults to the serialization options set using `SetSerializeOptions()`. See Section 48.6 [SetSerializeOptions], page 968, for details. (V10.0)

`SrcEncoding:`
This tag can be used to specify the source character encoding. This defaults to the string library's default character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

`DstEncoding:`
This tag can be used to specify the destination character encoding. This defaults to the string library's default character encoding as set

by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

If the `Adapter` tag isn't specified, it will default to the default set using `SetDefaultAdapter()`.

Tables can be serialized to strings using the `SerializeTable()` function. See Section 48.4 [SerializeTable], page 964, for details.

INPUTS

`s$` string to deserialize

`t` optional: table specifying further options (see above)

RESULTS

`table` table deserialized from string

EXAMPLE

See Section 48.4 [SerializeTable], page 964.

48.2 GetSerializeMode

NAME

GetSerializeMode – get serialization mode (V10.0)

SYNOPSIS

```
mode = GetSerializeMode()
```

FUNCTION

This function returns the current serialization mode that has been set using `SetSerializeMode()`. See Section 48.5 [SetSerializeMode], page 965, for details.

INPUTS

none

RESULTS

`mode` current serialization mode

48.3 ReadTable

NAME

ReadTable – read table from file (V4.0)

SYNOPSIS

```
table = ReadTable(id[, t])
```

FUNCTION

This function reads a Hollywood table from the file specified by `id` and returns it. Reading starts from the current file cursor position which you can modify using the `Seek()` command.

Starting with Hollywood 9.0, the data will be deserialized using the deserializer that can be specified in the `Adapter` tag in the optional table argument. Before version 9.0, `ReadTable()` always used Hollywood's legacy deserializer which uses a proprietary, non-human-readable format.

The following tags are currently recognized in the optional table argument:

`Adapter:` This table tag can be used to specify the deserializer that should be used to import the data into a Hollywood table. This can be the name of an external deserializer plugin (e.g. `xml`) or it can be one of the following inbuilt deserializers:

`Default:` Use Hollywood's default deserializer. This will deserialize data from the JSON format to a Hollywood table. Note that even though the name of this deserializer claims to be the default one, it is actually not. For compatibility reasons, `ReadTable()` will use the `Inbuilt` deserializer by default (see below). If you want `ReadTable()` to use the JSON deserializer, you explicitly have to request it by setting `Adapter` to `Default`.

`Inbuilt:` Use Hollywood's legacy deserializer. The only data this deserializer will accept is data written by Hollywood's legacy serializer during the `WriteTable()` call. Note that for compatibility reasons, this is still the default deserializer. However, it is not recommended any longer as the data is in a proprietary, non-human-readable format. Using JSON is a much better choice.

If the `Adapter` tag isn't specified, it defaults to the default set using `SetDefaultAdapter()`. Note that for compatibility reasons, this default isn't `Default` but `Inbuilt`. See above for an explanation.

`UserTags:`
This tag can be used to specify additional data that should be passed to serializer plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Mode:` This tag can be used to set the serialization mode to use for the operation. It defaults to the serialization mode set using `SetSerializeMode()`. See Section 48.5 [SetSerializeMode], page 965, for details. (V10.0)

`Options:` This tag can be used to set the serialization options to use for the operation. It defaults to the serialization options set using `SetSerializeOptions()`. See Section 48.6 [SetSerializeOptions], page 968, for details. (V10.0)

`SrcEncoding:`
This tag can be used to specify the source character encoding. This defaults to the string library's default character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

`DstEncoding:`
This tag can be used to specify the destination character encoding. This defaults to the string library's default character encoding as set

by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

INPUTS

`id` file to read from

`t` optional: table containing further options (V9.0)

RESULTS

`table` the table read from the file

EXAMPLE

See Section 48.7 [WriteTable], page 969.

48.4 SerializeTable

NAME

SerializeTable – serialize table to string (V9.0)

SYNOPSIS

```
s$ = SerializeTable(table[, t])
```

DEPRECATED SYNTAX

```
s$ = SerializeTable(table[, t])
```

FUNCTION

This function serializes the table specified by `table` to a string and returns it.

The optional table argument `t` can be used to specify additional options. The following tags are currently recognized in the optional table argument:

`Adapter:` The table will be serialized using the serializer that is specified in the `Adapter` tag. This can be the name of an external serializer plugin (e.g. `xml`) or it can be one of the following inbuilt serializers:

- `Default:` Use Hollywood's default serializer. This will serialize the table to the JSON format. This is also the default if no other default has been set using `SetDefaultAdapter()`.
- `Inbuilt:` Use Hollywood's legacy serializer. This will serialize the table to a custom, proprietary format.

`UserTags:` This tag can be used to specify additional data that should be passed to serializer plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Mode:` This tag can be used to set the serialization mode to use for the operation. It defaults to the serialization mode set using `SetSerializeMode()`. See Section 48.5 [SetSerializeMode], page 965, for details. (V10.0)

`Options:` This tag can be used to set the serialization options to use for the operation. It defaults to the serialization options set using `SetSerializeOptions()`. See Section 48.6 [SetSerializeOptions], page 968, for details. (V10.0)

`SrcEncoding:`
: This tag can be used to specify the source character encoding. This defaults to the string library's default character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

`DstEncoding:`
: This tag can be used to specify the destination character encoding. This defaults to the string library's default character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

If the `Adapter` tag isn't specified, it will default to the default set using `SetDefaultAdapter()`.

The string can later be deserialized back to a table by using the `DeserializeTable()` function. See Section 48.1 [DeserializeTable], page 961, for details.

INPUTS

`table`	table to serialize
`t`	optional: table specifying further options (see above)

EXAMPLE

```
mytable = {1, 2, 3, 4, 5,
  "Hello World",
  x = 100, y = 150,
  subtable = {10, 9, 8, 7}
}

s$ = SerializeTable(mytable)
mytable2 = DeserializeTable(s$)
```

The code above serializes `mytable` to a string in the JSON format and then deserializes that string back to a table. In the end, `mytable` and `mytable2` will be two independent tables but with identical contents.

48.5 SetSerializeMode

NAME

SetSerializeMode – set serialization mode (V10.0)

SYNOPSIS

`SetSerializeMode(mode)`

FUNCTION

This function sets the serialization mode to the one specified by `mode`. Currently, the following serialization modes are available: `#SERIALIZEMODE_HOLLYWOOD`, `#SERIALIZEMODE_LIST`, and `#SERIALIZEMODE_NAMED`. How the individual serialization mode are interpreted depends on the serializer.

Hollywood's legacy serializer, which serializes to a proprietary format, only supports `#SERIALIZEMODE_HOLLYWOOD`. Hollywood's JSON serializer supports all three serialization modes. Here is how the JSON serializer interprets the different serialization modes:

`#SERIALIZEMODE_HOLLYWOOD:`
: This is the default serialization mode. All JSON elements will be serialized to table fields. On top of that, the Hollywood serializer can also serialize binary data and even complete Hollywood functions. Hollywood functions and binary data will be serialized as Base64 data. The Hollywood serializer also supports sparse arrays, i.e. tables whose indices aren't strictly sequential but have gaps between the individual indices. A disadvantage of the Hollywood serializer is that it sometimes uses some special markers in the JSON file to tell Hollywood about the data stored in a JSON element, e.g. whether the data is to be interpreted as a string, as binary data or as a Hollywood function. Consequently, you might not be able to deserialize any arbitrary JSON file with the Hollywood serializer because some things in the JSON might be wrongly interpreted as one of Hollywood's special markers. As long as you only deserialize data written by the Hollywood serializer, you will of course never run into any problems. For example, consider the following table:

```
t = {foo = "bar", seqarray = {1,2,3,4,5}, sparsearray =
    {1,[2]=2,[4]=3,[6]=4,[8]=5}}
```

When serializing this to JSON using the Hollywood serializer, the result will look like this:

```
{
   "seqarray": [1,2,3,4,5],
   "sparsearray": {"0": 1, "2": 2, "4": 3, "6": 4, "8": 5},
   "foo": "bar"
}
```

You can see that sparse arrays are serialized by using named JSON indices. This, on the other hand, means that when deserializing JSON files using the Hollywood serializer named elements that consist of nothing but numbers are interpreted as sparse array fields which is why the Hollywood serializer can't be used to deserialize any arbitrary JSON file but should only be used with JSONs that the Hollywood serializer created.

Another disadvantage of the Hollywood serializer is that the position of elements in the JSON file can be completely random because they are serialized from Hollywood table fields which don't have any particular order. If you want the JSON elements to keep a fixed order, you'll have to use `#SERIALIZEMODE_LIST` instead.

By default, `#SERIALIZEMODE_HOLLYWOOD` will convert all JSON key names to lower-case. If you don't want that, you can change the behaviour by setting the `NoLowerCase` tag to `True` in `SetSerializeOptions()`. See Section 48.6 [SetSerializeOptions], page 968, for details.

`#SERIALIZEMODE_NAMED:`

This is like `#SERIALIZEMODE_HOLLYWOOD` except that it doesn't support any Hollywood extensions. This means that you can only serialize numbers, strings, and tables but no binary data or Hollywood functions. Also, this serializer imposes some restrictions on tables, namely that they must either use string indices or numeric indices but not both. If numeric indices are used, those indices must also be strictly sequential, i.e. table indices must be sequential within a certain range [0..n]. There must not be any gaps like in the example above so it's not possible to serialize sparse arrays with the named serializer. The advantage of `#SERIALIZEMODE_NAMED` over `#SERIALIZEMODE_HOLLYWOOD` is that since the named serializer doesn't support any Hollywood extensions you can use it to deserialize any arbitrary JSON file without issues. This is because the named serializers doesn't use any markers because it doesn't support any Hollywood extensions so there is no risk of clashes between JSON data and Hollywood markers. Just like `#SERIALIZEMODE_HOLLYWOOD`, however, the position of elements in the JSON file can be completely random because they are serialized from Hollywood table fields which don't have any particular order. If you want the JSON elements to keep a fixed order, you'll have to use `#SERIALIZEMODE_LIST` instead.

By default, `#SERIALIZEMODE_NAMED` will convert all JSON key names to lower-case. If you don't want that, you can change the behaviour by setting the `NoLowerCase` tag to `True` in `SetSerializeOptions()`. See Section 48.6 [SetSerializeOptions], page 968, for details.

`#SERIALIZEMODE_LIST:`

This mode will serialize JSON elements using lists instead of named table fields. This has the advantage that the order of all JSON elements will be preserved. Also, the spelling of the individual JSON keys will be preserved and you could even use the same key several times. A disadvantage is that it's a bit more difficult to access the JSON data because it is stored in key-value pair tables. For example, consider the following JSON data:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  }
}
```

The list serializer will store each key-value in its own table using `key` and `value` as named table indices. So to access data from the JSON above, you'd have to use this code:

```
Print(t[3].key, t[3].value)                    ; "age 27"
Print(t[4].value[2].key, t[4].value[2].value) ; "state NY"
```

With the named or Hollywood serializer you could access the JSON data by simply using the key name:

```
Print(t.age)                  ; prints "27"
Print(t.address.state)        ; prints "NY"
```

As you can see, using the named or Hollywood serializer leads to code that is more readable but the downside is that the element order in the JSON won't be preserved so that the JSON might be more difficult to read.

Note that if you use an external serializer (e.g. a plugin) the interpretation of the different serialization modes could be completely different. The documentation above is only valid for Hollywood's inbuilt JSON serializer.

Also note that this function will globally change the serialization mode. You can also change the serialization mode locally by setting the `Mode` tag in the optional table arguments of functions like `SerializeTable()`. See Section 48.4 [SerializeTable], page 964, for details.

INPUTS

`mode` desired serialization mode

48.6 SetSerializeOptions

NAME

SetSerializeOptions – set serialization options (V10.0)

SYNOPSIS

```
SetSerializeOptions(t)
```

FUNCTION

This function can be used to set some options for the serializer. You have to pass a table in `t`. The table can contain the following tags:

`NoLowerCase:`
: Set this to `True` if the serializer shouldn't automatically convert all key names to lower case. Hollywood's inbuilt JSON serializer currently converts all key names to lower case when using the serialization modes `#SERIALIZEMODE_HOLLYWOOD` and `#SERIALIZEMODE_NAMED`. External serializers like plugins might interpret this option differently. Defaults to `False`.

Note that this function will globally change the serialization options. You can also change the serialization options locally by setting the `Options` tag in the optional table arguments of functions like `SerializeTable()`. See Section 48.4 [SerializeTable], page 964, for details.

INPUTS

`t` table containing serialization options (see above)

48.7 WriteTable

NAME

WriteTable – write table to file (V4.0)

SYNOPSIS

```
WriteTable(id, table[, t])
```

DEPRECATED SYNTAX

```
WriteTable(id, table[, txtmode, nobrk])
```

FUNCTION

This function writes the Hollywood table specified by `table` to the file specified by `id`. The table will be serialized using the serializer that can be specified in the optional arguments. It will be written to the file at the current cursor position which you can modify by using the `Seek()` command. Tables written to files can later be loaded back into Hollywood tables by using the `ReadTable()` command.

This function is fully recursive. Your table can contain as many subtables as you need. Additionally, the table can even contain Hollywood functions. See below for an example.

`WriteTable()` supports several optional arguments. Before Hollywood 9.0, those had to be passed as optional parameters (see above). Since Hollywood 9.0, however, it is recommended to use the new syntax, which has a single optional table argument that can be used to pass one or more optional arguments to `WriteTable()`.

The following table fields are recognized by this function:

`Adapter:` This table tag can be used to specify the serializer that should be used to export the Hollywood table. This can be the name of an external serializer plugin (e.g. `xml`) or it can be one of the following inbuilt serializers:

`Default:` Use Hollywood's default serializer. This will serialize the table data to the JSON format. Note that even though the name of this serializer claims to be the default one, it is actually not. For compatibility reasons, `WriteTable()` will use the `Inbuilt` serializer by default (see below). If you want `WriteTable()` to use the JSON serializer, you explicitly have to request it by setting `Adapter` to `Default`.

`Inbuilt:` Use Hollywood's legacy serializer. This will serialize the table into a custom, proprietary format. This is the format `WriteTable()` has used since Hollywood 4.0 and for compatibility reasons, it is still the default serializer. However, it is not recommended any longer as this serializer will output data that is not in a human-readable format. Using the JSON serializer is a much better choice.

If `Adapter` isn't specified, it defaults to the default set using `SetDefaultAdapter()`. Note that for compatibility reasons, this default isn't `Default` but `Inbuilt`. See above for an explanation.

`TextMode:`

When using Hollywood's legacy serializer, which is still the default, this argument can be set to `True` to tell `WriteTable()` to export binary data as

text. Note that even if you set this tag to `True`, the text won't be in a human-readable format. If you want to serialize the table into human-readable text, use the JSON serializer (see above). Defaults to `False`.

`NoLineBreak:`
: If the `TextMode` tag has been set to `True`, `WriteTable()` will automatically insert line breaks after every 72 characters for better readability. If you don't want that, set `NoLineBreak` to `True`. In that case, no line breaks will be inserted. Note that this tag only affects Hollywood's legacy serializer. It doesn't have any effect on other serializers. Defaults to `False`. (V6.1)

`UserTags:`
: This tag can be used to specify additional data that should be passed to serializer plugins. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

`Mode:`
: This tag can be used to set the serialization mode to use for the operation. It defaults to the serialization mode set using `SetSerializeMode()`. See Section 48.5 [SetSerializeMode], page 965, for details. (V10.0)

`Options:`
: This tag can be used to set the serialization options to use for the operation. It defaults to the serialization options set using `SetSerializeOptions()`. See Section 48.6 [SetSerializeOptions], page 968, for details. (V10.0)

`SrcEncoding:`
: This tag can be used to specify the source character encoding. This defaults to the string library's default character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

`DstEncoding:`
: This tag can be used to specify the destination character encoding. This defaults to the string library's default character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

INPUTS

`id` file to write to

`table` table to write to the file

`t` optional: table containing further arguments (see above) (V9.0)

EXAMPLE

```
mytable = {1, 2, 3, 4, 5,
  "Hello World",
  x = 100, y = 150,
  subtable = {10, 9, 8, 7},
  mulfunc = Function(a, b) Return(a*b) EndFunction
}
```

```
OpenFile(1, "table.json", #MODE_WRITE)
WriteTable(1, mytable, {Adapter = "default"})
CloseFile(1)

OpenFile(1, "table.json", #MODE_READ)
newtable = ReadTable(1, {Adapter = "default"})
CloseFile(1)

Print(newtable[0], newtable[5], newtable.x, newtable.y,
      newtable.subtable[0], newtable.mulfunc(9, 9))
```

The code above writes the table mytable to file "table.json". After that, it opens file "table.json" again and reads the table back into Hollywood. The imported table will be stored in the variable `newtable`. Finally, we will access the newly imported table and print some of its data to the screen. The output of the code above will be "1 Hello World 100 150 10 81".

49 Sound library

49.1 Overview

Hollywood's sound library offers two basic object types: Samples and music objects. Samples are typically short sounds like game or feedback effects whereas music objects are used for longer sounds playing in the background. The biggest difference between samples and music objects is that samples are loaded entirely into memory whereas music objects are streamed from disk. Thus, you should only use samples for short sounds because loading a 4 minute music track as a sample will easily occupy about 40 megabytes of memory. Samples are optimized for immediate playback which is why they are often uploaded to audio hardware memory when they are loaded so that they can be played with very low latency. Music streams, however, might need a little bit longer to start playing.

Samples can be loaded via the `@SAMPLE` and `LoadSample()` commands. You can also create your own samples using the `CreateSample()` command. To play a sample, use the `PlaySample()` command.

Music objects can be loaded via the `@MUSIC` and `OpenMusic()` commands. You can also create your own music objects using the `CreateMusic()` command. To play a music object, use the `PlayMusic()` command.

By default, Hollywood's sound library allocates 8 audio channels for sound playback. This means that Hollywood will run out of channels in case you try to play more than 8 different samples, music objects, or video streams at a time. If your script needs more than 8 channels for some particular reasons, you need to increase the number of channels using the `-numchannels` console argument.

49.2 CloseAudio

NAME

CloseAudio – close audio hardware (V8.0)

SYNOPSIS

```
CloseAudio()
```

FUNCTION

This function can be used to close the audio hardware. It is normally not necessary to call this command because Hollywood will automatically close the audio hardware when it no longer needs it. On AmigaOS and compatibles, however, there are situations where you might need fine-tuned control over the audio hardware, for example because another program tries to get exclusive access to the audio hardware, which means that your script has to release it first. In those situations you might want to call `CloseAudio()` manually. Apart from that particular situation, there is no need to call this function at all.

Note that calling `CloseAudio()` will not only stop all audio output but it will also free all samples because samples are usually uploaded to sound card memory when they are loaded so it is impossible to keep them in memory while the audio hardware is closed. Music and video objects, however, aren't freed by `CloseAudio()`, though. `CloseAudio()` will only stop their playback.

INPUTS
none

EXAMPLE
See Section 49.29 [OpenAudio], page 995.

49.3 CloseMusic

NAME
CloseMusic – close a music object (V2.0)

SYNOPSIS
```
CloseMusic(id)
```

FUNCTION
This function frees any memory occupied by the music object specified by `id` and closes the file. This is normally not necessary because Hollywood frees all memory when it quits. However, if you are running out of memory and want to free the music object by yourself, use this function.

INPUTS

`id`	identifier of the music object to close

49.4 CopySample

NAME
CopySample – clone a sample (V5.0)

SYNOPSIS
```
[id] = CopySample(source, dest)
```

FUNCTION
This function clones the samples specified by `source` and creates a copy of it as sample `dest`. The new sample is independent from the old one so you could free the source sample after it has been cloned.

If you pass `Nil` as `dest`, `CopySample()` will return a handle to the new sample to you. Otherwise the new sample will use the identifier specified in `dest`.

INPUTS

`source`	source sample id
`dest`	identifier of the sample to be created or `Nil` for auto id selection

RESULTS

`id`	optional: handle to the new sample; will only be returned if you specified `Nil` in `dest`

EXAMPLE
```
CopySample(1, 10)
```

```
FreeSample(1)
```

The above code creates a new sample 10 which contains the same audio data as sample 1. Then it frees sample 1 because it is no longer needed.

49.5 CreateMusic

NAME

CreateMusic – create dynamic music stream (V5.0)

SYNOPSIS

```
[id] = CreateMusic(id, pitch, fmt)
```

FUNCTION

This function can be used to create a dynamic music stream that has to be fed constantly with new PCM data through a user defined callback function. This allows you to play gapless audio using PCM data generated on the fly by a callback function. The music object will be added to Hollywood's music list and can be accessed through the specified id. If you pass `Nil` in `id`, `CreateMusic()` will automatically select an identifier and return it to you. You also have to specify the desired playback frequency for the music in the `pitch` argument as well as the encoding of the PCM data in the `fmt` argument. Currently, the following formats are supported: `#MONO8`, `#STEREO8`, `#MONO16`, and `#STEREO16`.

Before you call this function, you have to install a callback function of type `FillMusicBuffer` using the `InstallEventHandler()` function. This callback will then be called whenever the audio server needs new PCM data. To deliver the new PCM data to the audio server, your callback has to call the `FillMusicBuffer()` function. See Section 49.7 [FillMusicBuffer], page 978, for details.

Once you have created the music object using `CreateMusic()`, you can then use all the regular commands from the music library to work with the new music. For instance, you can use `PlayMusic()` to start playback and `PauseMusic()` to pause the music object.

Make sure that you always use a main loop that calls `WaitEvent()` when you use this function because the callback function of `CreateMusic()` will always be called by `WaitEvent()`! If you do not use a `WaitEvent()` loop, your callback will never get called and thus no sound will ever play!

Please note that this is a lowlevel function that runs pretty close on the hardware level. Thus, your callback function should never block your script for a longer time. It should return as soon as possible. Never call any functions that could block the script in `CreateMusic()` callback functions. For instance, calling `Wait()` or `SystemRequest()` in a music callback is a very bad idea.

INPUTS

`id`	identifier for the new music or `Nil` for auto selection
`pitch`	desired playback frequency for the music object
`fmt`	desired format for the music object

RESULTS

`id`	optional: identifier of new music object; this is only used if `Nil` is passed in the first argument

49.6 CreateSample

NAME

CreateSample – create a sample (V2.0)

SYNOPSIS

```
[id] = CreateSample(id, table, pitch[, fmt, length])
```

FUNCTION

This function can be used to create a new sample from custom PCM data. The sample will be added to Hollywood's sample list and can be accessed by the specified id. If you pass `Nil` in `id`, `CreateSample()` will automatically select an identifier and return it to you. You also have to specify the desired playback frequency for this sample in the pitch argument. The optional argument `fmt` allows you to specify the format of the PCM data you are about to pass to this function. Currently, the following formats are supported: `#MONO8` (which is the default), `#STEREO8`, `#MONO16`, and `#STEREO16`. The optional argument `length` specifies the desired length in PCM frames for the new samples.

The sample data must be passed as signed integers. For 8-bit samples the valid sample range runs from -128 to 127, and for 16-bit samples the valid sample range runs from -32768 to 32767.

If you want to create a stereo sample, you must pass interleaved PCM data, i.e. left channel sample is followed by right channel sample is followed by left channel sample, and so on.

Starting with Hollywood 5.0, the PCM data can be passed to this function in a number of different ways. `CreateSample()` can use an array of PCM samples, an identifier of an open file, or a memory block as the source for the new sample. Which source is used depends on the setting in the table argument which accepts the following tags:

`Source:` This tag specifies from which source `CreateSample()` should fetch the audio data for the sample. It must be set to a string that identifies the audio data source. The following sources are possible:

- `PCM` Fetch audio data directly from an array of PCM samples. If you specify this field, you must pass the PCM data in the same table starting at index 0. `CreateSample()` will then read `length` PCM frames from this table. If `length` is not specified, then `CreateSample()` will read all PCM frames from the table.
- `File` Fetch audio data from an open file. If you use this source type, you also need to specify a valid file identifier in the `ID` tag. It is also necessary to specify the optional `length` argument so that `CreateSample()` knows how many frames it should fetch from the specified file.
- `Memory` Fetch audio data from a memory block. If you use this source type, you also need to specify a valid memory block identifier in the `ID` tag. It is also necessary to specify the optional `length`

argument so that `CreateSample()` knows how many frames it should fetch from the specified memory block.

The default value for the `Source` tag is `PCM` which means fetch the audio data from an array of PCM samples stored in the same table as these options.

`ID:` This tag is only required for source types `File` and `Memory`. In that case, you need to pass a valid file / memory block identifier here.

`Offset:` This tag can only be used in conjunction with source type `Memory`. In that case, it specifies an offset into the memory block at which `CreateSample()` should start fetching audio data. The offset is specified in bytes.

`Swap:` This tag can only be used in conjunction with source types `File` and `Memory` and a sample depth of 16 bits. In that case, the `Swap` tag can be used to specify whether or not `CreateSample()` should swap the two bytes making up a 16 bit sample. This is required if the sample data in the file or memory block is encoded in little endian format (LSB first). `CreateSample()`, however, requires 16-bit sample data to be in big endian format (MSB first). So if your source can only provide sample data in LSB format, simply set the `Swap` tag to `True` and everything should be fine. This tag defaults to `False` which means do not swap anything.

Please note that the new sample should use at least 1000 PCM frames. If you use less frames, the playback in loop mode will become very CPU intensive. Even if your sample has only 32 different wave forms, you should concatenate them until your sample has at least 1000 frames for performance reasons.

If you pass large sample tables to this function, please do not forget to set these tables to `Nil` when you no longer need them. Otherwise you will waste great amounts of memory.

Starting with Hollywood 5.0, `CreateSample()` can also create empty samples if you pass an empty table or specify a length of zero. In that case, you can use functions like `InsertSample()` to fill the sample with audio data later.

INPUTS

`id` identifier for the new sample or `Nil` for auto selection

`table` table containing parameters for the new sample

`pitch` desired playback frequency for the sample

`fmt` optional: format of the samples passed in argument 2 (defaults to `#MONO8`) (V5.0)

`length` optional: desired length of the new sample in PCM frames (V5.0)

RESULTS

`id` optional: identifier of new sample; this is only used if `Nil` is passed in the first argument

EXAMPLE

```
smpdata = {}
slen = 32
For k = 0 To 30
```

```
  For i = 0 To (slen\2)-1
    smpdata[k*slen+i] = -128
    smpdata[k*slen+i+(slen\2)] = 127
  Next
Next
CreateSample(1, smpdata, 6982)
PlaySample(1)
```

The code above creates a simple beep sound.

49.7 FillMusicBuffer

NAME

FillMusicBuffer – feed sound server with new audio data (V5.0)

SYNOPSIS

`FillMusicBuffer(id, type$, samples[, table])`

FUNCTION

This function is used in connection with dynamic music streams that have been initialized using `CreateMusic()`. These dynamic music streams need to be constantly fed with new audio data. This is handled by `FillMusicBuffer()`. `FillMusicBuffer()` will send the specified audio data to Hollywood's sound server which will in turn send it to the audio device. This chain of processors makes it possible to play gapless, dynamically generated audio data from your script.

Hollywood's sound server decides when it needs more audio data and thus the sound server is also the one that decides when you have to call `FillMusicBuffer()`. Hollywood will notify you when it needs more audio data by raising a `FillMusicBuffer` event so that the callback function you provided using the `InstallEventHandler()` function will get called. Inside this callback function you will now have to call `FillMusicBuffer()` to feed new data to the sound server. It is not allowed to call `FillMusicBuffer()` at other times! You must only call it inside of a callback function of type `FillMusicBuffer`.

This function takes four arguments: The first one specifies the music object to use. Your callback will receive this information in the `ID` tag of the event message. The third argument specifies how many samples (in PCM frames) you are providing to the audio server. This must be set to exactly the same number of frames that are requested from you by the callback handler. You get the number of frames requested from you in the `Samples` tag of the event message. The table argument is optional and must only be used for certain types (see below). The `type$` argument specifies how you will provide the new PCM data to the audio server. This can be one of the following strings:

`PCM` You will provide the new PCM data directly. In this case, you have to put an array that contains the new PCM frames into the `Data` tag of the optional table argument (see below).

`Sample` You will provide the new PCM data in the form of a sample. In this case, you have to put the identifier of the sample in the `ID` tag of the optional table argument (see below). Furthermore, you can use the `Start`, `End`, `Offset`,

and `Loop` tags to fine-tune the method that the audio server should use to fetch new PCM data. See below for more information.

`Mute` If you specify this type, the audio server will mute audio output for the duration of the number of PCM frames requested. If you pass `Mute` in the `Type` tag, you do not have to specify anything else.

`End` Specify this type if you want playback of your music to stop. Once the audio server receives an `End` packet, it will wait until all queued packets have been played and will stop the playback of your music thereafter.

The tags in the optional table argument depend on the type specified in `type$`. The following tags are recognized:

`Data:` If `type$` is set to `PCM`, you need to return an array of PCM data in this tag. The array should contain as many frames as requested by the audio server in the `Samples` tag. The PCM data must be passed as signed integers. For 8-bit data the valid sample range runs from -128 to 127, and for 16-bit data the valid sample range runs from -32768 to 32767. If you are using stereo mode, you must pass interleaved PCM data, i.e. left channel sample is followed by right channel sample is followed by left channel sample, and so on.

`ID:` If `type$` is set to `Sample`, you need to return the identifier of a sample from which the audio server should fetch the PCM data in this tag. You can fine-tune the way the audio server fetches the PCM data from this sample using the `Start`, `End`, `Offset`, and `Loop` tags. See below for more information.

`Start, End:`
If `type$` is set to `Sample`, these two tags allow you to specify the range in the source sample that the audio server should use for fetching samples. This is useful if you want the audio server to fetch data from only a part of the sample. Both values have to be specified in PCM frames. These tags default to 0 for `Start` and length of the specified sample for `End`. This means that by default the whole sample will be used for fetching PCM data.

`Offset:` Specifies an offset into the sample at which the audio server should start fetching PCM data. This offset must be specified in PCM frames and is relative to the position specified in `Start`. For instance, if you pass 10000 in `Start` and 100 in `Offset`, then the audio server will start fetching PCM data from offset 10100. This tag defaults to 0 which means start fetching PCM data from the beginning of the sample.

`Loop:` Specifies whether or not the audio server should continue fetching PCM data at the beginning of the source sample once its end has been reached. This defaults to `True` which means that the audio server will automatically revert to the beginning of the sample if its end has been reached and more PCM data is required. The beginning of the sample is defined by the value specified in the `Start` tag.

Please note that you have to use `FlushMusicBuffer()` if you need to update the audio data with a very low latency. `FillMusicBuffer()` will always buffer about 1 second of music data. This means that it will take about 1 second from the call to

`FillMusicBuffer()` until you can actually hear the audio data that you've just sent. If you need to update the audio data in real-time, e.g. when seeking to a new position in the stream, you will have to flush the music buffer first. See Section 49.8 [FlushMusicBuffer], page 980, for details.

INPUTS

`id`	identifier of the music object to use
`type$`	desired way for providing new audio data to the device (see above)
`samples`	number of samples that you are providing in PCM frames; must be identical to the number of samples requested by the `FillMusicBuffer` event
`table`	optional: table containing further options (see above)

49.8 FlushMusicBuffer

NAME

FlushMusicBuffer – flush buffer of dynamic music stream (V6.0)

SYNOPSIS

```
FlushMusicBuffer(id)
```

FUNCTION

This function is used in connection with dynamic music streams that have been initialized using `CreateMusic()`. These dynamic music streams need to be constantly fed with new audio data which is done by repeatedly calling `FillMusicBuffer()`. `FlushMusicBuffer()` can be used to empty all music buffers and refill them with new data. This can be useful if you need to immediately update the audio data that is being played, for example because the user has seeked the music stream to a new position. After the call to `FlushMusicBuffer()`, Hollywood will immediately trigger a `FillMusicBuffer` event so that your script gets a chance to refill the audio buffers after they have been flushed.

By default, there will always be a lag of about 1 second between the call to `FillMusicBuffer()` and the time you can hear the audio data on your sound device. If you call `FlushMusicBuffer()` first, the data can be sent to the sound device with a lower latency.

See Section 49.7 [FillMusicBuffer], page 978, for details.

INPUTS

`id`	identifier of the music object to flush

49.9 ForceSound

NAME

ForceSound – fail if audio hardware cannot be allocated (V8.0)

SYNOPSIS

```
ForceSound(fail)
```

FUNCTION

Normally, when a script tries to play a sound and the audio hardware cannot be allocated, Hollywood will continue running normally, just without sound. If you don't want that, i.e. if you want Hollywood to fail in case the audio hardware cannot be allocated, call this function and pass `True` in the `fail` parameter. In that case Hollywood will throw an error in case the audio hardware cannot be allocated.

Note that alternatively, you can also check the result of `IsSound()` to see if the audio hardware can be allocated. See Section 49.24 [IsSound], page 989, for details.

INPUTS

`fail`	specifies whether or not Hollywood should fail if the audio hardware cannot be allocated (the default setting is `False`, which means that Hollywood won't fail)

49.10 FreeModule

NAME

FreeModule – free a module / OBSOLETE

SYNOPSIS

```
FreeModule(id)
```

IMPORTANT NOTE

This command is obsolete. Please use `CloseMusic()` instead.

FUNCTION

This function frees the memory of the module specified by `id`. This is normally not necessary because Hollywood frees all memory when it quits. However, if you are running out of memory and want to free the sample by yourself, use this function.

INPUTS

`id`	identifier of the module

49.11 FreeSample

NAME

FreeSample – free a sample

SYNOPSIS

```
FreeSample(id)
```

FUNCTION

This function frees the memory of the sample specified by `id`. This is normally not necessary because Hollywood frees all memory when it quits. However, if you are running out of memory and want to free the sample by yourself, use this function.

INPUTS

`id`	identifier of the sample

EXAMPLE
See Section 49.26 [LoadSample], page 990.

49.12 GetChannels

NAME
GetChannels – get number of available channels (V6.1)

SYNOPSIS
```
n = GetChannels()
```

FUNCTION
This function returns the total number of available channels for audio output. This defaults to 8 but can be changed by using the `-numchannels` console argument. See Section 3.2 [Console arguments], page 33, for details.

Note that this argument doesn't return the free audio channels but the total audio channels. To check if there is a channel that can be used for audio output, use the `HaveFreeChannel()` function instead. See Section 49.16 [HaveFreeChannel], page 984, for details.

Also note that if the legacy audio driver is active on AmigaOS (it is by default on AmigaOS 3.x for performance reasons) the first four channels will be reserved for Protracker playback. The console argument `-nolegacyaudio` can be used to disable the legacy audio driver on AmigaOS 3.x. See Section 3.2 [console arguments], page 33, for details.

INPUTS
none

RESULTS

`n`	total number of available channels

49.13 GetPatternPosition

NAME
GetPatternPosition – get current pattern position (V1.9)

SYNOPSIS
```
pos = GetPatternPosition()
```

FUNCTION
This function returns the pattern position of the currently playing Protracker module. If no module is playing, -1 is returned. You can time your script to the music with this function.

You can also use `WaitPatternPosition()` which halts the program flow until a certain pattern position is reached.

INPUTS
none

RESULTS

`pos` current pattern position or -1

49.14 GetSampleData

NAME

GetSampleData – retrieve sample's raw data (V5.0)

SYNOPSIS

```
table, count = GetSampleData(id)
```

FUNCTION

This function can be used to retrieve the raw PCM samples of the sample specified in `id`. The PCM samples will be returned inside a table. The format of the individual samples will be either 8-bit signed (ranging from -128 to +127) or 16-bit signed (ranging from -32768 to 32767). You can find out the sample format by querying the `#ATTRTYPE` of the sample using the `GetAttribute()` function. If the sample uses two channels (i.e. stereo), the PCM data will be returned in interleaved order, i.e. left channel sample is followed by right channel sample is followed by left channel sample, and so on.

The second return value of this function is a counter value that indicates the number of sample frames in the table. Be warned that this value does not return the actual total array elements but the number of sample frames. For stereo samples, the left and right channel samples together form a single sample frame. Thus, if you get stereo data, there will be twice as many samples in the table than indicated by `count` because the latter counts in sample frames instead of raw samples.

If you get large sample tables from this function, please do not forget to set these tables to `Nil` when you no longer need them. Otherwise you will waste great amounts of memory.

To convert a table of PCM data back into a sample, you can use the `CreateSample()` command.

INPUTS

`id` identifier of sample to use

RESULTS

`table` a table containing the raw PCM data of the specified sample

`count` number of sample frames inside the table

49.15 GetSongPosition

NAME

GetSongPosition – get current song position (V1.9)

SYNOPSIS

```
pos = GetSongPosition()
```

FUNCTION

This function returns the song position of the currently playing Protracker module. If no module is playing, -1 is returned. You can time your script to the music with this function.

You can also `WaitSongPosition()` which halts the program flow until a certain song position is reached.

INPUTS

none

RESULTS

`pos`	current song position or -1

49.16 HaveFreeChannel

NAME

HaveFreeChannel – check if a free channel is available (V6.1)

SYNOPSIS

```
n = HaveFreeChannel()
```

FUNCTION

This function checks if there is a free channel for audio output. If there is, `HaveFreeChannel()` will return the index of this channel, otherwise 0 is returned.

Note that if the legacy audio driver is active on AmigaOS (it is by default on AmigaOS 3.x for performance reasons) the first four channels will be reserved for Protracker playback. The console argument `-nolegacyaudio` can be used to disable the legacy audio driver on AmigaOS 3.x. See Section 3.2 [console arguments], page 33, for details.

INPUTS

none

RESULTS

`n`	index of free channel or 0 if all channels are currently occupied

49.17 InsertSample

NAME

InsertSample – insert one sample into another one (V5.0)

SYNOPSIS

```
InsertSample(src, dst, pos[, len, table])
```

FUNCTION

This function can be used to insert `len` PCM frames of the sample specified in `src` into PCM frame position `pos` of the sample specified in `dst`. If the optional argument `len` is not specified, the whole sample will be inserted into the position specified in `pos`. If the two samples do not use the same format, this function will automatically perform

an appropriate conversion of the audio data so that sample depth, channel layout, and sampling rate of the two samples match.

The optional table argument allows you to configure advanced options for the insert operation. The following tags are currently recognized by the optional table argument:

`Start, End`:
: These two tags allow you to specify a range in the source sample that should be inserted into the destination one. This is useful if you want to insert only a part of the source sample into the destination. Both values have to be specified in PCM frames. These tags default to 0 for `Start` and length of the source sample for `End`. This means that by default the whole sample will be inserted.

`Offset`:
: Specifies an offset into the source sample at which `InsertSample()` should start fetching PCM data for the destination sample. This offset must be specified in PCM frames and is relative to the position specified in `Start`. For instance, if you pass 10000 in `Start` and 100 in `Offset`, then `InsertSample()` will start fetching PCM data at offset 10100. This tag defaults to 0 which means start fetching PCM data from the beginning of the source sample.

`Loop`:
: Specifies whether or not `InsertSample()` should continue to fetch PCM data at the beginning of the source sample once its end has been reached. This defaults to `True` which means `InsertSample()` will automatically revert to the beginning of the sample if its end has been reached and more PCM data is required. The beginning of the sample is defined by the value specified in the `Start` tag.

Please note that this command will extend the length of the destination sample. Existing audio data will not be overwritten. It will just be shifted forward by the insert operation.

INPUTS

`src` identifier of the source sample

`dst` identifier of the sample that shall be modified

`pos` position in PCM frames where `src` should be inserted into `dst`

`len` optional: number of PCM frames to insert into `src` (defaults to the length of `dst`)

`table` optional: table containing further parameters (see above)

EXAMPLE

```
InsertSample(1, 2, 44100, 44100, {Start = 25000, End = 30000})
```

The code above inserts one second of audio data from sample 1 into sample 2. The sample will be inserted at offset 44100. Audio data will be fetched from sample 1 but only in the range of PCM frames 25000 to 30000 in a loop.

49.18 IsChannelPlaying

NAME
IsChannelPlaying – check if a channel is playing (V6.1)

SYNOPSIS
```
playing[, type, id] = IsChannelPlaying(n)
```

FUNCTION
This function checks if the channel specified by `n` is currently playing and returns `True` if it is, `False` otherwise. If the channel is currently playing, `IsChannelPlaying()` will also return type and id of the object that is currently playing on this channel. This can be `#MUSIC`, `#SAMPLE`, or `#VIDEO`.

INPUTS

`n`	channel index to check; channel indices start from 1 to the number of available channels

RESULTS

`playing`	`True` if the channel is playing, `False` otherwise
`type`	optional: object type currently playing on this channel; only returned if the channel is currently playing
`id`	optional: object id currently playing on this channel; only returned if the channel is currently playing

49.19 IsModule

NAME
IsModule – determine if a module is in a supported format / OBSOLETE

SYNOPSIS
```
ret = IsModule(file$)
```

IMPORTANT NOTE
This command is obsolete. Please use `IsMusic()` instead.

FUNCTION
This function will check if the file specified `file$` is in a supported module format. If it is, this function will return `True`, otherwise `False`. If this function returns `True`, you can load the module by calling `LoadModule()`.

INPUTS

`file$`	file to check

RESULTS

`ret`	`True` if the module is in a supported format, `False` otherwise

49.20 IsMusicPlaying

NAME

IsMusicPlaying – check if music is currently playing (V4.5)

SYNOPSIS

```
playing = IsMusicPlaying(id)
```

FUNCTION

This function checks if the music object specified by `id` is currently playing. If it is, `True` is returned, `False` otherwise.

INPUTS

`id`	identifier of music object to check

RESULTS

`playing`	`True` if music object is currently playing; `False` otherwise

49.21 IsMusic

NAME

IsMusic – determine if a file is in a supported music format (V2.0)

SYNOPSIS

```
ret, fmt$ = IsMusic(file$[, table])
```

FUNCTION

This function will check if the file specified in `file$` is in a supported music format. If it is, this function will return `True` in the first return value, otherwise `False`. If this function returns `True`, you can open the music file using `OpenMusic()`.

The second return value is a string containing the music format of the file.

Starting with Hollywood 6.0 this function accepts an optional table argument which allows you to configure further options:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this music object. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`

This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

See Section 49.30 [OpenMusic], page 995, for a list of supported music formats.

INPUTS

`file$`	file to check
`table`	optional: table configuring further options (V6.0)

RESULTS

`ret`	`True` if the music object is in a supported format, `False` otherwise
`fmt$`	format of the music file

49.22 IsSamplePlaying

NAME

IsSamplePlaying – check if a sample is playing

SYNOPSIS

```
playing = IsSamplePlaying(id)
```

FUNCTION

This function checks if the sample specified by `id` is currently playing and returns `True` if it is, `False` otherwise.

INPUTS

`id`	identifier of a sample

RESULTS

`playing`	`True` if the sample specified by `id` is playing, `False` otherwise

EXAMPLE

```
LoadSample(1, "Sound/Samples/ChurchOrgan.wav")
PlaySample(1)
Repeat
  Wait(2)
Until IsSamplePlaying(1) = False
FreeSample(1)
```

The above code loads the sample "Sound/Samples/ChurchOrgan.wav", plays it and then waits for it to finish. After that, the sample is freed. If you just want to do something like above, it is easier for you to use the `WaitSampleEnd()` command. But if you want to do some things during the sample is playing, you will have to do it this way (using `IsSamplePlaying()` and a loop).

49.23 IsSample

NAME

IsSample – determine if a sample is in a supported format

SYNOPSIS

```
ret = IsSample(file$[, table])
```

FUNCTION

This function will check if the file specified `file$` is in a supported sample format. If it is, this function will return `True`, otherwise `False`. If this function returns `True`, you can load the sample by calling `LoadSample()`.

Starting with Hollywood 6.0 this function accepts an optional table argument which allows you to configure further options:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this sample. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`file$` file to check

`table` optional: table configuring further options (V6.0)

RESULTS

`ret` `True` if the sample is in a supported format, `False` otherwise

49.24 IsSound

NAME

IsSound – determine if Hollywood can output audio

SYNOPSIS

```
ret = IsSound()
```

FUNCTION

This function checks if Hollywood can output audio. You can use this function if your application cannot run without audio being output. Normally, if Hollywood cannot output audio it will just skip all audio related code and still execute the script. If you do not want that, use `IsSound()` to determine if audio can be output.

Starting with Hollywood 8.0, you can also use the `ForceSound()` function to make Hollywood fail if the audio hardware cannot be allocated. See Section 49.9 [ForceSound], page 980, for details.

INPUTS
none

RESULTS
`ret` `True` if sound can be played, `False` otherwise

EXAMPLE
```
If IsSound() = False
  SystemRequest("My App", "Sorry, sound is required!", "OK")
  End
EndIf
```
The above code checks if it can output sound and quits with a error message if this is not possible.

49.25 LoadModule

NAME
LoadModule – load a module / OBSOLETE

SYNOPSIS
`LoadModule(id, filename$)`

IMPORTANT NOTE
This command is obsolete. Please use `OpenMusic()` instead.

FUNCTION
This function loads the module specified by `filename$` into memory and gives it the identifier `id`. The module must be in Protracker format.

INPUTS
`id` identifier for the module

`filename$`
file to load

EXAMPLE
```
LoadModule(5, "Modules/StardustMemories.mod")
```
The above declaration assigns module number 5 to the module "StardustMemories.mod" located in a subdrawer named "Modules".

49.26 LoadSample

NAME
LoadSample – load a sample

SYNOPSIS
`[id] = LoadSample(id, filename$[, table])`

FUNCTION

This function loads the sample specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadSample()` will automatically choose an identifier and return it.

Sample formats that are supported on all platforms are RIFF WAVE, IFF 8SVX, IFF 16SV, and sample formats you have a plugin for. Depending on the platform Hollywood is running on, more sample formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all sample formats you have datatypes for as well.

Starting with Hollywood 6.0, this function accepts an optional table argument which allows you to pass additional parameters:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this sample. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

This command is also available from the preprocessor: Use `@SAMPLE` to preload samples!

Please note that this function loads sample data completely into memory. If you plan to play long samples, you should better use `OpenMusic()` which buffers only small portions of the sound data in memory.

INPUTS

`id` identifier for the sample or `Nil` for auto id selection

`filename$`
file to load

`table` optional: table configuring further options (see above) (V6.0)

RESULTS

`id` optional: identifier of the sample; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadSample(1, "Sound/Samples/WahWah.wav")
PlaySample(1)
WaitSampleEnd(1)
```

```
FreeSample(1)
```

The above code loads the sample "Sound/Samples/WahWah.wav", plays it, waits for it to end and frees it.

49.27 MixSample

NAME

MixSample – mix existing sample(s) into a new sample (V5.0)

SYNOPSIS

```
[id] = MixSample(id, len, pitch, fmt, smp1, opt1, ...)
```

FUNCTION

This function can be used to mix one or more existing samples into a new sample. The new sample will be added to Hollywood's sample list and can be accessed by the specified id. If you pass `Nil` in `id`, `MixSample()` will automatically select an identifier and return it to you. The second argument `len` specifies the desired length of the new sample in PCM frames. The third argument `pitch` specifies how many frames per second should be sent to the audio device. For CD quality, you would pass 44100 as the `pitch` argument but in many cases 22050 is also sufficient. The `fmt` argument specifies the desired sample format for the new sample. Currently, the following formats are supported: `#MONO8`, `#STEREO8`, `#MONO16`, and `#STEREO16`. The former two specify 8-bit PCM encodings while the latter two use 16-bits per channel.

The samples that should be mixed into the new sample are passed to `MixSample()` from argument 5 onwards. For each sample that should be mixed into the new sample you have to pass its identifier as well as a table that contains further parameters for the mixing operation. You can repeat this pattern as many times as you like. The options table that has to be passed for each sample supports the following tags:

`Pitch:` Specifies the frequency that should be used when mixing this sample. This tag defaults to the sample's current frequency defined using `SetPitch()`.

`Offset:` Specifies the offset inside the sample from which PCM data should be fetched for mixing. This must be specified in PCM frames. This tag defaults to 0 which means start fetching data from the beginning.

`Length:` Specifies the maximum number of PCM frames that should be mixed. Defaults to -1 which means mix as many frames as available into the new sample.

`Loop:` Specifies whether or not the mixer should continue to fetch PCM data at the beginning of the sample once its end has been reached. This defaults to `True` which means the mixer will automatically revert to the beginning of the sample if its end has been reached and more PCM data is required.

`Scale:` Specifies a scaling factor that should be applied to the mixing operation. If you are mixing many samples together, you are likely to get some noise artefacts you do not want. You can reduce these artefacts by reducing the volume level. This can be achieved using this tag. Every PCM frame will

be multiplied with the scaling factor you pass here. Thus, to reduce the volume by 50%, simply pass 0.5 here. This tag defaults to 1.0 which means no scaling should be applied.

`Threshold:`
: This tag allows you to specify a threshold at which this sample should be mixed into the destination sample. For example, if you would like this sample to kick in after 10,000 PCM frames have been mixed, you would specify 10000 here. The value specified here must be passed in PCM frames. This tag defaults to 0 which means that this sample should be mixed into the destination sample right from the start.

This function is powerful. It will perform automatic conversion between different sample encodings, sampling rates, and channel layouts. Also, you can mix as many samples into the new sample as you like. The samples to be mixed can also be the same, i.e. you can mix the same sample into the new samples multiple times using different mixing parameters like varying pitch speed or thresholds. If you get unwanted noise artefacts, try reducing the volume of single samples using the `Scale` tag (see above).

INPUTS

`id`	identifier for the new sample or `Nil` for auto selection
`len`	desired length for the new sample in PCM frames
`pitch`	desired playback frequency for the new sample
`fmt`	desired format for the new sample
`smp1`	first sample to mix
`opt1`	options table for first sample to mix
`...`	optional: you can repeat the id/options sequence as often as you want so you can mix as many samples together as you like

RESULTS

`id`	optional: identifier of new sample; this is only used if `Nil` is passed in the first argument

EXAMPLE

```
MixSample(1, 10 * 44100, 44100, #STEREO16, 2, {}, 3,
          {Threshold = 3 * 44100}, 4, {6 * 44100})
```

The code above creates a new sample in 44.1 format, using 16 bits per PCM frame and two channels. The sample's length will be exactly 10 seconds. The sample will start with sample 2. After three seconds sample 3 will kick in and after six seconds sample 4 will start to play.

49.28 MUSIC

NAME

MUSIC – preload a music file for later use (V2.0)

SYNOPSIS

```
@MUSIC id, filename$[, table]
```

FUNCTION

Use this preprocessor command to preload a music object which you want to play later using `PlayMusic()`. The music file can be in any format supported by Hollywood. Please have a look at the `OpenMusic()` documentation for information on supported music formats. If the music file is in a streaming format, this preprocessor command will only initialize the music object for later playback. It will not load large music objects completely into memory but they will be played as a sound stream buffered from disk.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Link:` Set this field to `False` if you do not want to have this music file linked to your executable/applet when you compile your script. This field defaults to `True` which means that the music file is linked to your executable/applet when Hollywood is in compile mode.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this music file. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Music formats that are supported on all platforms are RIFF WAVE, IFF 8SVX, IFF 16SV, Protracker modules, and formats you have a plugin for. Depending on the platform Hollywood is running on, more music formats might be supported. For example, on Windows, Hollywood supports all formats that DirectShow can load, and on macOS, all formats recognized by Apple's AudioFile interface are supported.

If you want to open the music file manually, please use the `OpenMusic()` command.

INPUTS

`id` a value that is used to identify this music object later in the code

`filename$`
the file you want to have loaded

`table` optional: a table configuring further options (see above)

EXAMPLE

```
@MUSIC 1, "TurricanII_Remix.mod"
```

The code above opens "TurricanII_Remix.mod" so that it can be played later using `PlayMusic()`.

49.29 OpenAudio

NAME

OpenAudio – open audio hardware (V8.0)

SYNOPSIS

```
OpenAudio()
```

FUNCTION

This function can be used to open the audio hardware. It is normally not necessary to call this command because Hollywood will automatically open the audio hardware as soon as it needs it. On AmigaOS and compatibles, however, there are situations where you might need fine-tuned control over the audio hardware, for example because another program tries to get exclusive access to the audio hardware, which means that your script has to release it first. In those situations you might want to call `OpenAudio()` and `CloseAudio()` manually. Apart from that particular situation, there is no need to call these functions at all.

INPUTS

none

EXAMPLE

```
OpenAudio()
OpenMusic(1, "Turrican2_Remix.mod")
PlayMusic(1)
WaitLeftMouse
StopMusic(1)
CloseAudio()
```

The code above plays "Turrican2_Remix.mod" and then closes the audio hardware, making it possible for other programs on AmigaOS aiming for exclusive audio hardware access to reserve it.

49.30 OpenMusic

NAME

OpenMusic – open a music file (V2.0)

SYNOPSIS

```
[id] = OpenMusic(id, filename$[, table])
```

FUNCTION

This function opens the music file specified by `filename$` and assigns the `id` to it. If you pass `Nil` in `id`, `OpenMusic()` will automatically choose an identifier and return it. The

file specified in `filename$` will be opened and prepared for playback. Please note that files opened using `OpenMusic()` will be played using audio streaming. `LoadSample()` on the other hand, will load the entire sound file into memory first. Thus, you should use `LoadSample()` for playing short sounds and `OpenMusic()` for longer sounds and background music.

Music formats that are supported on all platforms are RIFF WAVE, IFF 8SVX, IFF 16SV, Protracker modules, and formats you have a plugin for. Depending on the platform Hollywood is running on, more music formats might be supported. For example, on Windows, Hollywood supports all formats that DirectShow can load, and on macOS, all formats recognized by Apple's AudioFile interface are supported.

Starting with Hollywood 6.0, this function accepts an optional table argument which allows you to pass additional parameters:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this music file. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

This command is also available from the preprocessor: Use `@MUSIC` to preload music objects!

INPUTS

`id` identifier for the music object or `Nil` for auto id selection

`filename$`
file to load

`table` optional: table configuring further options (see above) (V6.0)

RESULTS

`id` optional: identifier of the music object; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenMusic(1, "Turrican2_Remix.mod")
PlayMusic(1)
```

The code above plays "Turrican2_Remix.mod".

49.31 PauseModule

NAME
PauseModule – pause the currently playing module / OBSOLETE

SYNOPSIS
`PauseModule()`

IMPORTANT NOTE
This command is obsolete. Please use `PauseMusic()` instead.

FUNCTION
This function pauses the currently playing module. It can be resumed with the `ResumeModule()`.

INPUTS
none

49.32 PauseMusic

NAME
PauseMusic – pause a playing music (V2.0)

SYNOPSIS
`PauseMusic(id)`

FUNCTION
This function pauses the music object with the identifier `id`. This music object must be playing when you call this command. You can resume the playback by using the `ResumeMusic()` command.

INPUTS

`id`	identifier of the music object to pause

49.33 PlayModule

NAME
PlayModule – start playing a music module / OBSOLETE

SYNOPSIS
`PlayModule(number)`

IMPORTANT NOTE
This command is obsolete. Please use `PlayMusic()` instead.

FUNCTION
Starts playing the music module with the number `number`. This music module must have been loaded with `LoadModule()`. If there is already a music module playing, it will be stopped automatically.

INPUTS

`number`	number of the music module to start

EXAMPLE

```
LoadModule(1,"StardustMemories.mod")
PlayModule(1)
```

49.34 PlayMusic

NAME

PlayMusic – start playback of a music object (V2.0)

SYNOPSIS

```
PlayMusic(id[, table])
```

FUNCTION

This command starts the playback of the music object specified by `id`. This music object must have been opened with either the `@MUSIC` preprocessor command or the `OpenMusic()` command.

Please note that before Hollywood 6.0 only one music could be played at a time. Starting with Hollywood 6.0 this limit is no longer there, but it is still enforced in order to be compatible with older scripts. Thus, if a music object is already playing and you call this command, that music will be stopped first before playback of the new music starts. If you need to play multiple music objects at the same time, you will have to explicitly disable this behaviour by calling `LegacyControl()` and setting the `SingleMusic` tag to `False`. `PlayMusic()` will no longer stop any playing music. See Section 52.22 [LegacyControl], page 1085, for details.

Prior to Hollywood 4.5, the second argument was optional and specified how many times the music object should be played. Starting with Hollywood 4.5, the second argument is now an optional table argument. Of course, the old syntax is still supported for compatibility. New scripts should use the new syntax though. The optional table argument recognizes the following tags:

`Times:` This tag can be used to specify how many times the music object shall be played. This tag defaults to 1 which means that the music object is only played once. If you want your music object to loop infinitely, pass 0 as the second argument.

`Volume:` Set this to the desired playback volume. This field can range from 0 to 64. If not specified, the music object's default volume will be used (you can modify the default volume of a music object by calling `SetMusicVolume()`).

`Channel:` Channel to use for playback of this music object. By default, `PlayMusic()` will automatically choose a vacant channel and will fail if there is no vacant channel. To override this behaviour, you can use this field. When specified, it will always enforce playback on the very channel specified here. If the channel is already playing, it will be stopped first. (V6.1)

INPUTS

`id` identifier of the music object to start

`table` optional: table argument specifying further options (V4.5)

EXAMPLE
See Section 49.30 [OpenMusic], page 995.

49.35 PlaySample

NAME
PlaySample – start playing a sample

SYNOPSIS
`PlaySample(id[, table], ...)`

FUNCTION
Starts playing the sample specified by `id`. You can load samples either using the `LoadSample()` command or by using the `@SAMPLE` preprocessor command (the sample will be preloaded then).

Starting with Hollywood 2.0, you can also pass a table as the second argument that defines parameters for the sample playback. The table can contain the following fields:

`Times:` This field can be used to specify how many times the sample shall be played. This defaults to 1 which means that the sample will be played once. If you want the sample to loop infinitely, set `Times` to 0.

`Volume:` Set this to the desired playback volume. This field can range from 0 to 64. If not specified, the sample's default volume will be used (you can modify the default volume of a sample by calling `SetVolume()`).

`Pitch:` Set this to the desired playback pitch. This value has to be passed in hertz. If not specified, the sample's default pitch will be used (you can modify the default pitch of a sample by calling `SetPitch()`).

`Time:` This field can be used to define how long the sample shall be played. Hollywood will loop the sample until the given time has elapsed. Time must be specified in milliseconds. This tag is mutually exclusive with the `Times` tag.

`Panning:` This field allows you to set the channel panning for this sample. This must be in the range of 0 to 255. 0 means left speaker playback only, 128 means both speakers, and 255 means right speaker only. If not specified, the sample's default pan setting will be used (you can modify the default panning of a sample by calling `SetPanning()`). (V4.5)

`Channel:` Channel to use for playback of this sample. By default, `PlaySample()` will automatically choose a vacant channel and will fail if there is no vacant channel. To override this behaviour, you can use this field. When specified, it will always enforce playback on the very channel specified here. If the channel is already playing, it will be stopped first. (V6.1)

Also new in Hollywood 2.0 is the possibility to play multiple samples at once with one call to `PlaySample()`. Simply repeat the argument list as many times as you like and `PlaySample()` will play all specified samples together - perfectly synchronized. Please note that for each additional sample there is also an additional optional argument, that

either specifies the number of times the sample shall be played or it is a table that contains further attributes for the sample playback. See above for all possibilities.

INPUTS

`id`	identifier of sample to play
`table`	a table that contains playback parameters for the sample (V2.0)
`...`	the argument list can be repeated as many times as you like; `PlaySample()` will start all samples at the same time if you specify more than one

EXAMPLE

```
PlaySample(1)
```

The above code starts playing sample 1. The sample will not be looped.

```
PlaySample(1, {Time = 10000})
```

The code above plays sample 1 for exactly 10 seconds (= 10000 milliseconds).

```
PlaySample(1, {Times = 2}, 2, {Times = 4}, 3, {Time=5000})
```

The code above plays sample 1 two times, sample 2 four times and sample 3 is played for 5 seconds. All three samples are started at once.

49.36 PlaySubsong

NAME

PlaySubsong – play subsong of music object

SYNOPSIS

```
PlaySubsong(number[, id, table])
```

FUNCTION

This command can be used to play the specified subsong of a music object. If you omit the optional argument `id` the currently playing music is used.

The optional table argument can be used to specify further options. This table argument supports the same fields like the `PlayMusic()` command. See Section 49.34 [PlayMusic], page 998, for details.

Please note that only some music formats support subsongs. For example, old tracker module formats can often contain multiple subsongs. If a Protracker module is used, this command will jump to the specified song position.

INPUTS

`number`	number of subsong to play
`id`	optional: identifier of the music object to use (defaults to currently playing music) (V5.3)
`table`	optional: table argument specifying further options (V5.3)

EXAMPLE

```
PlaySubsong(5, 1)
```

The above code starts playing Protracker module number 1, starting at song position 5.

49.37 ResumeModule

NAME

ResumeModule – resume the paused module / OBSOLETE

SYNOPSIS

```
ResumeModule()
```

IMPORTANT NOTE

This command is obsolete. Please use `ResumeMusic()` instead.

FUNCTION

This function resumes the currently paused module. It can be paused with the `PauseModule()`.

INPUTS

none

49.38 ResumeMusic

NAME

ResumeMusic – resume a paused music object (V2.0)

SYNOPSIS

```
ResumeMusic(id)
```

FUNCTION

This function resumes the playback of the paused music object with the identifier `id`. You can pause the playback of a music object with the `PauseMusic()`.

INPUTS

`id`	identifier of the music object to be resumed

49.39 SAMPLE

NAME

SAMPLE – preload a sound sample for later use (V2.0)

SYNOPSIS

```
@SAMPLE id, filename$[, table]
```

FUNCTION

Use this preprocessor command to preload a sound sample which you want to play later using `PlaySample()`.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Link:` Set this field to `False` if you do not want to have this sample linked to your executable/applet when you compile your script. This field defaults to `True` which means that the sample is linked to your executable/applet when Hollywood is in compile mode.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this sample. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Sample formats that are supported on all platforms are RIFF WAVE, IFF 8SVX, IFF 16SV, and sample formats you have a plugin for. Depending on the platform Hollywood is running on, more sample formats might be supported. For example, on Amiga compatible systems Hollywood will be able to open all sample formats you have datatypes for as well.

If you want to load the sample manually, please use the `LoadSample()` command.

INPUTS

`id` a value that is used to identify this sample later in the code

`filename$`
the sound sample you want to have loaded

`table` optional: a table configuring further options (see above)

EXAMPLE

```
@SAMPLE 1, "Gunshot.8svx"
```

The above declaration assigns sample number 5 to the sample "Gunshot.8svx".

```
@SAMPLE 1, "Sound/Samples/Gunshot.wav", {Link=False}
```

Does the same as above but the sample will not be linked when the script is compiled.

49.40 SaveSample

NAME

SaveSample – save sample to disk (V5.0)

SYNOPSIS

```
SaveSample(id, f$[, fmt, t])
```

FUNCTION

This command saves the sample specified in `id` to the file specified in `f$`. The optional argument `fmt` specifies the format in which the sample should be exported. Currently, only `#SMPFMT_WAVE` is supported here. This will save the sample in the RIFF WAVE format.

Starting with Hollywood 10.0, `SaveSample()` accepts an optional table argument that allows you to pass additional arguments to the function. The following tags are currently supported by the optional table argument:

`Adapter:` This tag allows you to specify one or more file adapters that should be asked if they want to save the specified file. If you use this tag, you must set it to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

INPUTS

`id`	identifier of sample to save
`f$`	path to save location
`fmt`	optional: format in which to export the sample (defaults to `#SMPFMT_WAVE`)
`t`	optional: table containing further options (see above) (V10.0)

EXAMPLE

```
@SAMPLE 1, "ouch.8svx"

SaveSample(1, "ouch.wav")
```

The code above loads a sample in the IFF 8SVX format and saves it as a RIFF WAVE sample.

49.41 SeekMusic

NAME

SeekMusic – seek to a certain position in a music object (V2.0)

SYNOPSIS

```
SeekMusic(id, pos)
```

FUNCTION

You can use this function to seek to the specified position in the music specified by `id`. The music object does not have to be playing. If the music is playing and you call `SeekMusic()`, it will immediately skip to the specified position. The position is specified in milliseconds. Thus, if you want to skip to the position 3:24, you would have to pass the value 204000 because 3 * 60 * 1000 + 24 * 1000 = 204000.

Please note that this function does not work with Protracker modules.

INPUTS

`id`	identifier of the music object to use
`pos`	new position for the music

49.42 SetChannelVolume

NAME

SetChannelVolume – set channel volume (V6.1)

SYNOPSIS

```
SetChannelVolume(n, volume)
```

FUNCTION

This function modifies the volume of the channel specified by `n`. Channel indices range from 1 to the number of channels. `volume` can be a number ranging from 0 (mute) to 64 (full) or a string containing a percent specification, e.g. `"50%"`.

INPUTS

`n`	channel index
`volume`	new volume for the channel

49.43 SetMasterVolume

NAME

SetMasterVolume – set the master volume (V1.5)

SYNOPSIS

```
SetMasterVolume(vol)
```

FUNCTION

Please note: This command is somewhat obsolete and subject to removal. You should better use `SetVolume()` for samples and `SetMusicVolume()` for music files/Protracker modules.

This function sets the master volume to the specified value `vol`. Using this command you can realise sound fade outs or ins.

Starting with Hollywood 2.0, `vol` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`vol` new master volume (range 0 to 64 or percent specification)

EXAMPLE

```
For k = 64 To 0 Step -5
   SetMasterVolume(k)
Next
```

The code above fades out all playing sounds by modifying the master volume.

49.44 SetMusicVolume

NAME

SetMusicVolume – modify volume of a music object (V2.0)

SYNOPSIS

```
SetMusicVolume(id, volume)
```

FUNCTION

This function modifies the volume of the music object specified by `id`. If the music object is currently playing, the volume will be modified on-the-fly which can be used for sound fades etc.

Starting with Hollywood 2.0, `volume` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id` identifier of the music object

`volume` new volume for the music object (range: 0=mute until 64=full volume or percent specification)

49.45 SetPanning

NAME

SetPanning – set sample panning (V1.9)

SYNOPSIS

```
SetPanning(id, pan)
```

FUNCTION

This function allows you to specify where the sample with the identifier `id` shall be played. The parameter `pan` ranges from 0 to 255 where 0 means that the sample will only be played through the left speaker and 255 means that the sample will only be played through the right speaker. If you want to play the sample through both speakers at the same level, use 128 which is also the default.

You can also specify the special constances `#LEFT`, `#CENTER`, and `#RIGHT` which correspond to 0, 128, and 255, respectively.

If the sample is currently playing, the panning will be modified on-the-fly which can be used for some nice effects.

INPUTS

`id` identifier of the sample to use

`pan` new pan value (ranges from 0 to 255)

EXAMPLE

```
SetPanning(1, 255)
PlaySample(1)
```

The code above will play the sample only through the right speaker.

49.46 SetPitch

NAME

SetPitch – modify pitch of a sample

SYNOPSIS

```
SetPitch(id,pitch)
```

FUNCTION

This function modifies the pitch of the sample specified by `id`. If the sample is currently playing, the pitch is modified on-the-fly which can be used for some nice sound effects. Pitch is specified in hertz.

INPUTS

`id` identifier of a sample

`pitch` new pitch for the sample in hertz

49.47 SetVolume

NAME

SetVolume – modify volume of a sample

SYNOPSIS

```
SetVolume(id,volume)
```

FUNCTION

This function modifies the volume of the sample specified by `id`. If the sample is currently playing, the volume will be modified on-the-fly which can be used for sound fades etc.

Starting with Hollywood 2.0, `volume` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id` identifier of a sample

`volume` new volume for the sample (range: 0=mute until 64=full volume or percent specification)

EXAMPLE

```
LoadSample(1, "Sound/Samples/GroovyLoop.wav")
```

```
PlaySample(1)
Wait(100)
For k = 64 To 0 Step -1
   SetVolume(1,vol)
Next
```

The above code loads the sample "Sound/Samples/GroovyLoop.wav", plays it, waits 2 seconds and then does a fade out.

49.48 StopChannel

NAME

StopChannel – stop channel playback (V6.1)

SYNOPSIS

`StopChannel(n)`

FUNCTION

Stops playback on the channel specified by `n`. Channel indices range from 1 to the number of available channels.

INPUTS

`n`	index of channel to stop

49.49 StopModule

NAME

StopModule – stop the currently playing module / OBSOLETE

SYNOPSIS

`StopModule()`

IMPORTANT NOTE

This command is obsolete. Please use `StopMusic()` instead.

FUNCTION

Stops the module that is currently playing and frees all used audio channels.

INPUTS

none

49.50 StopMusic

NAME

StopMusic – stop a currently playing music (V2.0)

SYNOPSIS

`StopMusic(id)`

FUNCTION
This function stops the music object specified by `id`. This won't fail if the specified music isn't currently playing.

INPUTS

`id`	identifier of the music object to be stopped

49.51 StopSample

NAME
StopSample – stop playing a sample

SYNOPSIS
```
StopSample(id)
```

FUNCTION
Stops playback of the sample with the specified id. This won't fail if the specified sample isn't currently playing.

INPUTS

`id`	identifier of sample to stop

49.52 WaitMusicEnd

NAME
WaitMusicEnd – halt until music has finished playing (V10.0)

SYNOPSIS
```
WaitMusicEnd(id)
```

FUNCTION
This function halts the script execution until the music specified by `id` has finished playing. After that, the execution of your script is continued. If you need to do something while your music is playing, use the `IsMusicPlaying()` command in conjunction with a loop.

INPUTS

`id`	identifier of a music that is currently playing

49.53 WaitPatternPosition

NAME
WaitPatternPosition – halt program until module reaches pattern position

SYNOPSIS
```
WaitPatternPosition(pos)
```

FUNCTION

This function halts the program flow until the currently playing module reaches the specified pattern position `pos`. You have to call `PlayModule()` before using this command. This is useful for timing your applications with the music.

INPUTS

`pos` pattern position to wait for

EXAMPLE

```
PlayModule(1)
WaitPatternPosition(63)
```

The above code starts playing module number 1 and waits then for reaching the end of the first pattern.

49.54 WaitSampleEnd

NAME

WaitSampleEnd – halt until sample has finished playing

SYNOPSIS

```
WaitSampleEnd(id)
```

FUNCTION

This function halts the program flow until the sample specified by `id` has finished playing. After that, the execution of your script is continued. If you need to do something while your sample is playing, use the `IsSamplePlaying()` command in conjunction with a loop.

INPUTS

`id` identifier of a sample that is currently playing

EXAMPLE

See Section 49.26 [LoadSample], page 990.

49.55 WaitSongPosition

NAME

WaitSongPosition – halt program until module reaches song position

SYNOPSIS

```
WaitSongPosition(pos)
```

FUNCTION

This function halts the program flow until the currently playing module reaches the specified song position `pos`. You have to call `PlayModule()` before using this command. This is useful for timing your applications with the music.

INPUTS

`pos` song position to wait for

EXAMPLE

```
PlayModule(1)
WaitSongPosition(2)
```

The above code starts playing module number 1 and waits then for song position 2.

50 Sprite library

50.1 Overview

Sprites are elementary parts of many applications. They can be used for many different purposes and Hollywood is very flexible with them because they are implemented fully in software. Thus, there are no restrictions on sprite size, colors, transparency and so on. Traditionally, sprites are used for player and enemy graphics in games but they also come handy in many other situations.

Generally spoken, sprites have three distinctive attributes which distinguishes them from brushes:

1. Sprites are always on the front of the display.
2. Every sprite can only be once on the screen.
3. A Sprite can have multiple frames.

A more detailed explanation of sprite features follows below.

1. Sprites are always on the front of the display. Look at the following code:

   ```
   LoadBrush(1, "test.iff")
   LoadSprite(1, "test2.iff")
   DisplaySprite(1, 0, 0)
   DisplayBrush(1, 0, 0)
   ```

   You see that we load two images: Brush 1 and sprite 1. Now we draw sprite 1 and after that we draw brush 1. Normally, brush 1 should be drawn over sprite 1 because it `DisplayBrush()` is called after `DisplaySprite()`. Sprites, however, are always on the front of the display and that is why in this special case, brush 1 is drawn behind sprite 1.

   This applies to all normal graphics commands of Hollywood: You can never draw graphics over a sprite! Sprites are always on the front and normal functions, i.e. non-sprite functions, can never paint sprites over. You can only draw sprites over with new sprites.

2. Every sprite can only be on the screen once. Look at the following code:

   ```
   LoadSprite(1, "test.iff")
   DisplaySprite(1, 0, 0)
   DisplaySprite(1, 100, 100)
   ```

   You see that we display sprite 1 two times. First, it is displayed at 0:0 and second, it is displayed at 100:100. Because every sprite can only be on the screen once, the second call to `DisplaySprite()` will not draw sprite 1 again but move it from 0:0 to 100:100 instead. `DisplaySprite()` checks if sprite 1 is on the screen, and if it is, it is picked up and moved to the new position. Thus, you can easily move your sprites around on the screen.

3. A sprite can have multiple frames. Because sprites are often used for animation, each sprite can carry multiple frames just like a Hollywood animation object. The `DisplaySprite()` command accepts an optional argument which allows you to specify which frame it shall show.

Additional information on the sprite implementation in Hollywood:

- Sprites are tied to clip regions: When you display a sprite for the first time, it will be tied to the clip region that is currently active. The sprite will stay in that clip region even if you deactivate the clip region later. To let a sprite out of a clip region, you can either free the whole clip region using `FreeClipRegion()` or remove the sprite using `RemoveSprite()` and then display it again when no clip region is active.
- If you display a new background picture, all sprites on the old background picture are automatically removed.
- Sprites can only be drawn to your display. You cannot draw sprites to brushes, masks or alpha channels.
- Layers cannot be used together with sprites. These two concepts don't go together.
- The doublebuffering functions cannot be used together with sprites. If you use a doublebuffer, you normally do not need sprites anyway.

50.2 CopySprite

NAME

CopySprite – clone a sprite (V2.0)

SYNOPSIS

```
[id] = CopySprite(source, dest)
```

FUNCTION

This function clones the sprite specified by `source` and creates a copy of it as sprite `dest`. If you specify `Nil` in the `dest` argument, this function will choose an identifier for the new sprite automatically and return it to you. The new sprite is independent from the old sprite so you could free the source sprite after it has been cloned.

If you just want to have a new sprite with the same graphics as your old sprite, you should use `CreateSprite()` instead; it can create sprite links which are very memory-friendly, i.e. they consume very little memory and thus should be preferred to `CopySprite()` whenever possible.

INPUTS

`source`	source sprite id
`dest`	id for the sprite to be created or `Nil` for auto ID select

RESULTS

`id`	optional: identifier of the cloned sprite; will only be returned when you pass `Nil` as argument 2 (see above)

50.3 CreateSprite

NAME

CreateSprite – create a sprite (V2.0)

SYNOPSIS

```
[id] = CreateSprite(id, type, ...)
```

```
[id] = CreateSprite(id, #ANIM, source_id)
[id] = CreateSprite(id, #BRUSH, source_id[, width, height, frames,
                                fr_per_row, sourcex, sourcey])
[id] = CreateSprite(id, #SPRITE, source_id1, source_id2, ...)
[id] = CreateSprite(id, #TEXTOBJECT, source_id)
```

FUNCTION

This function creates a new sprite from the specified source. The sprite source can be an animation, a brush, a sprite or a text object. The new sprite will be stored under the specified `id`. If you specify `Nil` in the `id` argument, this function will choose an identifier for the new sprite automatically and return it to you. The arguments of `CreateSprite()` depend on what source type you specify.

If type is `#ANIM`, simply pass the identifier of the animation object to use in argument 3.

If type is `#BRUSH`, you have to specify the identifier of the source brush in argument 3. You can also make `CreateSprite()` extract several frames out of a brush. If you want that, you will have to pass at least the arguments `width`, `height` and `frames`. `Width` and `height` define the dimensions for the sprite to be created and `frames` specifies how many frames `CreateSprite()` shall read from the source brush. If the frames are aligned in multiple rows in the source brush, you will also have to pass the argument `fr_per_row` to tell `CreateSprite()` how many frames there are in every row. Finally, you can tell `CreateSprite()` where in the brush it should start scanning by specifying the arguments `sourcex` and `sourcey` (they both default to 0). `CreateSprite()` will then start off at position `sourcex` and `sourcey` and read `frames` number of images with the dimensions of `width` by `height` from the brush specified in `source_id`. After it has read `fr_per_row` images, it will advance to the next row. If you specify only three arguments, `CreateSprite()` will simply convert the brush specified in `source_id` to a sprite.

If type is `#SPRITE`, `CreateSprite()` will create a new sprite from an unlimited number of source sprites. You can specify as many source sprites as you want. Of course, each of the source sprites can also have multiple frames. When using `#SPRITE`, `CreateSprite()` will never copy the graphics data of the specified source sprites. For performance reasons, the source sprites will only be referenced and thus your new sprite will depend on them. By using only references to existing sprites, `CreateSprite()` executes very fast and with very low memory footprint. This is very convenient if you want to create various different animation sequences from always the same source sprites. Please note though that because sprites created using `#SPRITE` source type depend of their sub sprites, they will automatically be freed if one of the sub sprites is freed. So you should not free the sub sprites before you are done with the newly created sprite.

If type is `#TEXTOBJECT`, `CreateSprite()` will create a sprite from the specified text object. You only have to pass the identifier of the source text object.

INPUTS

`id`	identifier for the new sprite or `Nil` for auto ID select
`type`	type of source object
`...`	further arguments depend on the type specified (see above)

RESULTS

`id`	optional: identifier of the new sprite; will only be returned when you pass `Nil` as argument 1 (see above)

50.4 DisplaySprite

NAME

DisplaySprite – display a sprite (V2.0)

SYNOPSIS

```
DisplaySprite(id, x, y[, frame])
```

FUNCTION

This function displays the sprite specified in id at the specified position. If the sprite is already on the screen, it will be moved to the new position. The optional argument can be used to specify which frame shall be displayed. If it is omitted, `DisplaySprite()` will display the next frame of the sprite (if the sprite has multiple frames).

INPUTS

`id`	identifier of the sprite to display
`x`	desired x-position
`y`	desired y-position
`frame`	optional: frame to display (defaults to 0 which means show the next frame of the sprite)

50.5 FlipSprite

NAME

FlipSprite – flip a sprite (V2.0)

SYNOPSIS

```
FlipSprite(id, xflip)
```

FUNCTION

This function flips (mirrors) the sprite specified by `id`. If xflip is set to `True`, the sprite will be flipped in x-direction otherwise it will be flipped in y-direction.

This function can only be used on sprites that are not referenced by any other sprites. It also cannot be used on sprite links created using `CreateSprite()` with source type set to `#SPRITE`.

INPUTS

`id`	sprite to flip
`xflip`	`True` for horizontal (x) flip, `False` for vertical (y) flip

50.6 FreeSprite

NAME

FreeSprite – free a sprite (V2.0)

SYNOPSIS

```
FreeSprite(id)
```

FUNCTION

This function frees the memory occupied by sprite `id`. To reduce memory consumption, you should free sprites when you do not need them any longer.

If the sprite is still on the screen and you call `FreeSprite()`, it will be removed before it is freed.

INPUTS

`id` identifier of the sprite to be freed

50.7 LoadSprite

NAME

LoadSprite – load a sprite (V2.0)

SYNOPSIS

```
[id] = LoadSprite(id, filename$[, args])
```

FUNCTION

This function loads the sprite specified by `filename$` into memory and assigns the identifier `id` to it. If you pass `Nil` in `id`, `LoadSprite()` will automatically choose an identifier and return it.

Supported image formats are PNG, JPEG, BMP, IFF ILBM, and some more depending on the platform Hollywood is running on. Starting with Hollywood 4.5, `LoadSprite()` can also open animation formats (IFF ANIM, GIF ANIM, uncompressed AVIs or AVIs using Motion JPEG compression) and convert these animations into a sprite directly.

The optional argument `args` accepts a table which can contain further options for the loading operation. The following fields can be set in the args table:

`Transparency:`
: Here you can specify a color that shall appear transparent in the sprite. The color you specify here will be masked out then.

`LoadAlpha:`
: Set this field to `True` if the image contains an alpha channel that shall be loaded.

`X, Y, Width, Height, Frames, FPR:`
: This lot of fields can be used to fine-tune the loading operation. You can use these fields to make `LoadSprite()` create a sprite with multiple frames from a single picture. `Width` and `Height` define the dimensions for the sprite and `Frames` specifies how many frames `LoadSprite()` shall read from the source image. If the frames are aligned in multiple rows in the source image, you

will also have to pass the argument `FPR` (stands for frames per row) to tell `LoadSprite()` how many frames there are in each row. Finally, you can tell `LoadSprite()` where in the image it should start scanning by specifying the fields `X` and `Y` (they both default to 0). `LoadSprite()` will then start off at position `X` and `Y` and read `Frames` number of images with the dimensions of `Width` by `Height` from the picture specified by `filename$`. After it has read `FPR` number of images, it will advance to the next row. All of these fields can only be used if you specify an image file in `filename$`. If you specify an anim file, they are ignored.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this sprite. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
If this tag is set to `True`, the monochrome transparency of the sprite will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based sprite. If you want to load the alphachannel of a sprite, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
If this tag is set to `True`, Hollywood will load the sprite as a palette sprite. This means that you can get and modify the sprite's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette sprites also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`. (V9.0)

`TransparentPen:`
If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the file (if the file format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. A sprite cannot have a mask and an alpha channel!

This command is also available from the preprocessor: Use `@SPRITE` to preload sprites!

INPUTS

`id` identifier for the sprite or `Nil` for auto id selection

`filename$`
file to load

`args` optional: table that specifies further options for the loading operation

RESULTS

`id` optional: identifier of the sprite; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
LoadSprite(1, "MySprite.png", {Transparency = #RED})
```

This loads "MySprite.png" as sprite 1 with the color red being transparent.

```
LoadSprite(1, "PlayerSprites.png", {Width = 32, Height = 32,
     Frames = 16, FPR = 8, Transparency = #BLACK})
```

The code above creates sprite 1 from the file "PlayerSprites.png". Sprite 1 will be of the dimensions 32x32 and will contain 16 different frames. The single frames are aligned with 8 frames per row in the image "PlayerSprites.png". Thus, `LoadSprite()` needs to scan two rows to read the full 16 frames.

50.8 MoveSprite

NAME

MoveSprite – move a sprite from a to b (V2.0)

SYNOPSIS

```
MoveSprite(id, xa, ya, xb, yb[, table])
```

FUNCTION

This function moves (scrolls) the sprite specified by `id` softly from the location specified by xa,ya to the location specified by xb,yb.

Further configuration options are possible using the optional argument table. You can specify the move speed, special effect, and whether or not the move shall be asynchronous. See Section 21.46 [MoveBrush], page 287, for more information on the optional table argument. Besides the table elements mentioned in the `MoveBrush()` documentation, `MoveSprite()` accepts one additional table element named `AnimSpeed`: The anim speed value defines after how many draws the frame number should be increased; therefore a higher number means a lower playback speed of the animation.

INPUTS

`id` identifier of the sprite to use

`xa`	source x position
`ya`	source y position
`xb`	destination x position
`yb`	destination y position
`table`	optional: further configuration options

EXAMPLE

```
MoveSprite(1, 100, 50, 0, 50, {Speed = 5, AnimSpeed = 4})
```

Moves the sprite 1 from 100:50 to 0:50 with move speed 5 and anim playback speed 4.

50.9 RemoveSprite

NAME

RemoveSprite – remove a sprite from the display (V2.0)

SYNOPSIS

```
RemoveSprite(id)
```

FUNCTION

This function removes the sprite specified by `id` from the display. Note that the sprite will not be freed, so you can display it again at any time you like.

INPUTS

`id`	identifier of the sprite to remove

50.10 RemoveSprites

NAME

RemoveSprites – remove all sprites from the display (V2.0)

SYNOPSIS

```
RemoveSprites([keep])
```

FUNCTION

This function will remove all sprites from the display. If you set the optional argument `keep` to `True`, the sprites are still removed but will additionally be rendered as normal graphics to the display. This means that you could now paint them over with other graphics (e.g. a `"Game Over"` brush).

If `keep` is set to `True`, you will not see that this command does anything. That impression is, however, wrong. The sprites are indeed removed but you do not see a difference because they are immediately rendered as normal graphics to the display, so that you can paint them over.

INPUTS

`keep`	optional: `True` if the sprites shall be kept as normal graphics (defaults to `False`)

50.11 ScaleSprite

NAME

ScaleSprite – scale a sprite (V2.0)

SYNOPSIS

```
ScaleSprite(id, width, height)
```

FUNCTION

This command scales the sprite specified by `id` to the specified width and height.

This function can only be used on sprites that are not referenced by any other sprites. It also cannot be used on sprite links created using `CreateSprite()` with source type set to `#SPRITE`.

Please note: You should always do scale operations with the original sprite. For instance, if you scale sprite 1 to 12x8 and then scale it back to 640x480, you will get a messed image. Therefore you should always keep the original brush and scale only copies of it.

You can also pass `#KEEPASPRAT` as either `width` or `height`. Hollywood will calculate the size then automatically by taking the aspect-ratio of the sprite into account.

Alternatively, `width` and `height` can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id`	identifier of the sprite to scale
`width`	desired new width for the sprite
`height`	desired new height for the sprite

50.12 SetSpriteZPos

NAME

SetSpriteZPos – change the z-position of a layer (V7.0)

SYNOPSIS

```
SetSpriteZPos(id, zpos)
```

FUNCTION

This command can be used to change a sprite's z-position. The z-position of a sprite is its position in the hierarchy of sprites. The first (i.e. backmost) sprite has a z-position of 1, the last (i.e. frontmost) sprite's z-position is equal to the number of sprites currently present. You need to pass the new desired z-position for the specified sprite to this function. The sprite will then assume exactly this z-position, existing sprites that are on or after this z-position will be shifted down. To move a sprite all the way to the front (i.e. highest z-position), you can pass the special value 0 for the `zpos` argument. To move a sprite all the way to the back, specify 1 in the `zpos` argument.

INPUTS

`id`	identifier of the sprite whose z position shall be changed
`zpos`	new z position for the sprite or 0 to move the sprite to the highest z position

50.13 SPRITE

NAME

SPRITE – preload a sprite for later use (V2.0)

SYNOPSIS

```
@SPRITE id, filename$[, table]
```

FUNCTION

This preprocessor command preloads the sprite specified in `filename$` and assigns the identifier `id` to it.

Supported image formats are PNG, JPEG, BMP, IFF ILBM, and some more depending on the platform Hollywood is running on. Starting with Hollywood 4.5, `@SPRITE` can also open animation formats (IFF ANIM, GIF ANIM, uncompressed AVIs or AVIs using Motion JPEG compression) and convert these animations into a sprite directly.

The third argument is optional. It is a table that can be used to set further options for the loading operation. The following fields of the table can be used:

`Transparency:`
: This field can be used to specify a color in RGB notation that shall be made transparent in the sprite.

`LoadAlpha:`
: Set this field to `True` if the alpha channel of the image shall be loaded, too. Please note that not all pictures have an alpha channel and that not all picture formats are capable of storing alpha channel information. It is suggested that you use the PNG format if you need alpha channel data. This field defaults to `False`.

`Link:`
: Set this field to `False` if you do not want to have this sprite linked to your executable/applet when you compile your script. This field defaults to `True` which means that the sprite is linked to your to your executable/applet when Hollywood is in compile mode.

`X, Y, Width, Height, Frames, FPR:`
: This lot of fields can be used to fine-tune the loading operation. You can use these fields to make `@SPRITE` create a sprite with multiple frames from a single picture. `Width` and `Height` define the dimensions for the sprite and `Frames` specifies how many frames `@SPRITE` shall read from the source image. If the frames are aligned in multiple rows in the source image, you will also have to pass the argument `FPR` (stands for frames per row) to tell `@SPRITE` how many frames there are in each row. Finally, you can tell `@SPRITE` where in the image it should start scanning by specifying the fields `X` and `Y` (they both default to 0). `@SPRITE` will then start off at position `X` and `Y` and read `Frames` number of images with the dimensions of `Width` by `Height` from the picture specified by `filename$`. After it has read `FPR` number of images, it will advance to the next row. All of these fields can only be used if you specify an image file in `filename$`. If you specify an anim file, they are ignored.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this sprite. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`LoadTransparency:`
If this tag is set to `True`, the monochrome transparency of the image will be loaded. Please note that this tag is specifically designed for monochrome transparency channels, i.e. a transparent pen in a palette-based image. If you want to load the alphachannel of an image, set the `LoadAlpha` tag to `True`. This tag defaults to `False`. (V6.0)

`LoadPalette:`
If this tag is set to `True`, Hollywood will load the sprite as a palette sprite. This means that you can get and modify the sprite's palette which is useful for certain effects like color cycling. You can also make pens transparent using the `TransparentPen` tag (see below) or the `LoadTransparency` tag (see above). Palette sprites also have the advantage of requiring less memory because 1 pixel just needs 1 byte of memory instead of 4 bytes for 32-bit images. This tag defaults to `False`. (V9.0)

`TransparentPen:`
If the `LoadPalette` tag has been set to `True` (see above), the `TransparentPen` tag can be used to define a pen that should be made transparent. Pens are counted from 0. Alternatively, you can also set the `LoadTransparency` tag to `True` to force Hollywood to use the transparent pen that is stored in the file (if the file format supports the storage of transparent pens). This tag defaults to `#NOPEN`. (V9.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Please note that the `Transparency`, `LoadTransparency` and `LoadAlpha` fields are mutually exclusive. A sprite can only have one transparency setting!

If you want to load sprites manually, please use the `LoadSprite()` command.

INPUTS

`id` a value that is used to identify this sprite later in the code

`filename$`
the picture file you want to load

`table` optional argument specifying further options

EXAMPLE

```
@SPRITE 1, "MySprite.png", {Transparency = #RED}
```

This loads "MySprite.png" as sprite 1 with the color red being transparent.

```
@SPRITE 1, "PlayerSprites.png", {Width = 32, Height = 32,
     Frames = 16, FPR = 8, Transparency = #BLACK}
```

The code above creates sprite 1 from the file "PlayerSprites.png". Sprite 1 will be of the dimensions 32x32 and will contain 16 different frames. The single frames are aligned with 8 frames per row in the image "PlayerSprites.png". Thus, `@SPRITE` needs to scan two rows to read the full 16 frames.

51 String library

51.1 AddStr

NAME

AddStr – append substring to a string

SYNOPSIS

```
var$ = AddStr(string1$, string2$)
```

FUNCTION

Appends `string2$` to `string1$` and returns the new string.

This function is obsolete and only here for compatibility reasons. Starting with Hollywood 2.0 you should use the string concatenation operator .. for concatenating two strings.

INPUTS

`string1$` source string

`string2$` string to append

RESULTS

`var$` resulting string

EXAMPLE

```
test$ = AddStr("Hello", " World!")
Print(test$)
```

This will print "Hello World!"

51.2 ArrayToStr

NAME

ArrayToStr – convert code point array to string (V6.0)

SYNOPSIS

```
s$ = ArrayToStr(t[, encoding])
```

FUNCTION

This function reads all code points contained in the table `t`, appends them to a string and returns this string. `ArrayToStr()` will stop reading values from `t` once it encounters a code point of 0 (string terminator) or the end of the table.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

To convert strings into arrays, you can use the function `StrToArray()`. See Section 51.58 [StrToArray], page 1062, for details.

INPUTS

`t` a table containing an arbitrary number of code points

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`s$` string made up of the code points in the table

EXAMPLE

```
s$ = ArrayToStr({'H', 'e', 'l', 'l', 'o'})
DebugPrint(s$)
```

Prints "Hello".

51.3 Asc

NAME

Asc – return code point of character in string

SYNOPSIS

```
var = Asc(string$[, pos, encoding])
```

FUNCTION

Returns the code point value of the character at index `pos` in `string$`. The `pos` index must be in characters, not in bytes. If the `pos` argument is omitted, the code point of the first character will be returned.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

If you want to work with raw bytes instead of code points, you can either set the `encoding` parameter to `#ENCODING_RAW` or use the `ByteAsc()` function. See Section 51.6 [ByteAsc], page 1026, for details.

INPUTS

`string$` input string

`pos` optional: index of character whose code point should be returned (defaults to 0) (V7.0)

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var` code point value

EXAMPLE

```
result = Asc("A")
Print(result)
```

This will print "65" which is the code point value of "A".

51.4 Base64Str

NAME

Base64Str – encode or decode Base64 data (V6.0)

SYNOPSIS

```
data$ = Base64Str(s$, decode)
```

FUNCTION

This function can be used to encode arbitrary data into the Base64 format or decode a Base64 formatted string back into its original binary data. The second argument specifies whether this function should encode or decode the data.

INPUTS

`s$`	data to encode or decode
`decode`	`True` to decode Base64 data, `False` to encode the specified data into Base64

RESULTS

`data$`	encoded or decoded data

51.5 BinStr

NAME

BinStr – convert value to a binary formatted string (V2.0)

SYNOPSIS

```
bin$ = BinStr(val[, length])
```

FUNCTION

This function converts the value specified by `val` into binary format (base 2) and returns it as a string. The optional argument `length` allows you to specify how many bits shall be put into the string. This can be `#INTEGER` for 32 bits, `#SHORT` for 16 bits, and `#BYTE` for 8 bits. By default, `#INTEGER` will be used.

INPUTS

`val`	value to convert
`length`	optional: how many bits shall be converted (must be `#INTEGER`, `#SHORT`, or `#BYTE`) (V3.0)

RESULTS

`bin$`	binary notation of `val`

EXAMPLE

```
a$ = BinStr(255, #BYTE)
```

This returns the string `"11111111"`.

51.6 ByteAsc

NAME

ByteAsc – get single byte from string (V7.1)

SYNOPSIS

```
v = ByteAsc(string$[, pos])
```

FUNCTION

Returns the value of the byte at the string index specified by `pos`. The return value `v` will be in the range of 0 to 255.

Since Hollywood strings cannot only contain text but also raw binary data, this function is suitable for accessing raw string bytes at specified indices. The normal `Asc()` function, on the other hand, is more appropriate for dealing with text strings because it operates in Unicode mode by default which means that the input string must be in valid UTF-8.

INPUTS

`string$`	input string
`pos`	optional: index of byte to return (defaults to 0)

RESULTS

`v`	string byte at specified index

51.7 ByteChr

NAME

ByteChr – convert single byte to string (V7.1)

SYNOPSIS

```
s$ = ByteChr(v)
```

FUNCTION

Converts the byte value specified by `v` into a string. `v` must be in the range of 0 to 255.

Since Hollywood strings cannot only contain text but also raw binary data, this function is suitable for composing strings using non-encoded byte data. The normal `Chr()` function, on the other hand, is more appropriate for dealing with text strings because it operates in Unicode mode by default which means that it composes UTF-8 strings by default, i.e. passing 255 will result in a string that has two bytes because of the UTF-8 encoding rules.

INPUTS

`v`	byte to convert into string (in the range of 0 to 255)

RESULTS

`s$`	resulting string

51.8 ByteLen

NAME

ByteLen – return string length in bytes (V7.0)

SYNOPSIS

```
len = ByteLen(s$)
```

FUNCTION

This function returns the length of string `s$` in bytes. If you need to know the string length in characters, use `StrLen()` instead. See Section 51.56 [StrLen], page 1061, for details.

In the UTF-8 character encoding a single character may need a storage space of up to 4 bytes. In the ISO 8859-1 character encoding there is no difference between byte and character sizes.

INPUTS

`s$` input string

RESULTS

`len` length of input string in bytes

EXAMPLE

```
len = ByteLen("äöü")
Print(len)
```

If Hollywood is in Unicode mode, this will return 6 because each of the characters needs two bytes in the UTF-8 character encoding. In ISO 8859-1 mode, there is no difference between characters and bytes which means the code above will return 3.

51.9 ByteOffset

NAME

ByteOffset – convert character to byte offset (V7.0)

SYNOPSIS

```
boff = ByteOffset(s$, coff[, encoding])
```

FUNCTION

This function returns the byte offset of the character at the offset specified by `coff` inside string `s$`. This offset is in characters, starting from 0.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

In the UTF-8 character encoding a single character may need a storage space of up to 4 bytes. In the ISO 8859-1 character encoding there is no difference between byte and character sizes. Hence, it doesn't really make sense to call this function with the character encoding set to `#ENCODING_ISO8859_1`.

To convert a byte offset into a character offset use the `CharOffset()` function. See Section 51.12 [CharOffset], page 1030, for details.

INPUTS

`s$` input string

`coff` character offset to be mapped to a byte offset (starting from 0)

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`boff` byte offset of the specified character

EXAMPLE

```
boff = ByteOffset("äöü", 2)
Print(boff)
```

If Hollywood is in Unicode mode, this will return 4 because the two characters before the "ü" take up 2 bytes each in UTF-8 code space. In ISO 8859-1 there is no difference between characters and bytes, so 2 will be returned in that case.

51.10 ByteStrStr

NAME

ByteStrStr – convert value to raw bytes (V8.0)

SYNOPSIS

```
r$ = ByteStrStr(v[, type, le])
```

FUNCTION

This function can be used to convert a numeric value to raw bytes that are returned as a string. The number of bytes that will be written to the return string depend on the type that you pass in the `type` argument. The following types are currently supported:

`#BYTE:` Stores an 8-bit value (1 byte) in the return string.

`#SHORT:` Stores a 16-bit value (2 bytes) in the return string.

`#INTEGER:`
Stores a 32-bit value (4 bytes) in the return string. This is the default.

`#FLOAT:` Stores a 32-bit floating point value (4 bytes) in the return string.

`#DOUBLE:` Stores a 64-bit floating point value (8 bytes) in the return string.

For all multi-byte types, i.e. all types except `#BYTE`, you can use the additional `le` argument to specify the order in which the bytes should be stored in the return string. If you set `le` to `True`, the bytes will be stored in little endian order (LSB first), otherwise the bytes will be stored in big endian order (MSB first). Big endian is also the default.

If you need to convert raw bytes to a value, you can use the `ByteVal()` function. See Section 51.11 [ByteVal], page 1029, for details.

INPUTS

`v` numeric value to convert to binary data

`type` optional: type of value to store in string (defaults to `#INTEGER`)

`le` optional: whether or not to use little endian byte order (defaults to `False`)

RESULTS

`r$` resulting string

51.11 ByteVal

NAME

ByteVal – convert raw bytes to value (V8.0)

SYNOPSIS

```
v = ByteVal(s$[, type, le])
```

FUNCTION

This function can be used to convert raw bytes from the string passed in `s$` to a numeric value. The number of bytes that will be read from the string `s$` depend on the type that you pass in the `type` argument. The following types are currently supported:

`#BYTE:` Reads an 8-bit value (1 byte) from the string and returns it.

`#SHORT:` Reads a 16-bit value (2 bytes) from the string and returns it.

`#INTEGER:`
Reads a 32-bit value (4 bytes) from the string and returns it. This is the default.

`#FLOAT:` Reads a 32-bit floating point value (4 bytes) from the string and returns it.

`#DOUBLE:` Reads a 64-bit floating point value (8 bytes) from the string and returns it.

For all multi-byte types, i.e. all types except `#BYTE`, you can use the additional `le` argument to specify the order in which the bytes should be read from `s$`. If you set `le` to `True`, the bytes will be read in little endian order (LSB first), otherwise the bytes will be read in big endian order (MSB first). Big endian is also the default.

Note that for all integer types the result will always be unsigned. You can use the `Cast()` function if you need to cast the result to a signed type. See Section 37.12 [Cast], page 763, for details.

If you need to convert a value to raw bytes, you can use the `ByteStrStr()` function. See Section 51.10 [ByteStrStr], page 1028, for details.

INPUTS

`s$` string to read data from

`type` optional: type of value to read (defaults to `#INTEGER`)

`le` optional: whether or not to use little endian byte order (defaults to `False`)

RESULTS

`v` resulting value

51.12 CharOffset

NAME

CharOffset – convert byte to character offset (V7.0)

SYNOPSIS

```
coff = CharOffset(s$, boff[, encoding])
```

FUNCTION

This function returns the offset, in characters, of the character at the offset specified by `boff`, in bytes and starting from 0, inside string `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

In the UTF-8 character encoding a single character may need a storage space of up to 4 bytes. In the ISO 8859-1 character encoding there is no difference between byte and character sizes. Hence, it doesn't really make sense to call this function with the character encoding set to `#ENCODING_ISO8859_1`.

To convert a character offset into a byte offset use the `ByteOffset()` function. See Section 51.9 [ByteOffset], page 1027, for details.

INPUTS

`s$`	input string
`boff`	byte offset to be mapped to a character offset (starting from 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`coff`	character offset of the specified character

EXAMPLE

```
coff = CharOffset("äöü", 2)
Print(coff)
```

If Hollywood is in Unicode mode, this will return 1 because the "ä" character takes up 2 bytes in UTF-8 code space. In ISO 8859-1 there is no difference between characters and bytes, so 1 will be returned in that case.

51.13 CharWidth

NAME

CharWidth – return byte width of character (V7.0)

SYNOPSIS

```
w = CharWidth(s$[, pos, encoding])
```

FUNCTION

Calculates the byte width of the character at position `pos` inside `s$`. The position must be specified in characters, not in bytes. The `pos` argument is optional and defaults to 0, i.e. the beginning of the string, if omitted.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

In the UTF-8 character encoding a single character may need a storage space of up to 4 bytes. In the ISO 8859-1 character encoding there is no difference between byte and character sizes. Hence, it doesn't really make sense to call this function with the character encoding set to `#ENCODING_ISO8859_1`.

INPUTS

`s$`	input string
`pos`	optional: index, in characters, of the character whose width should be calculated
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`w`	byte width of character at the specified position

EXAMPLE

```
w = CharWidth("ä")
Print(w)
```

If Hollywood is in Unicode mode, this will return 2 because the "ä" character takes up 2 bytes in UTF-8 code space. In ISO 8859-1 there is no difference between characters and bytes, so 1 will be returned in that case.

51.14 Chr

NAME

Chr – convert code point value to string

SYNOPSIS

```
var$ = Chr(value[, encoding])
```

FUNCTION

Converts the code point value specified in `value` into a string.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

If you want to work with raw bytes instead of code points, you can either set the `encoding` parameter to `#ENCODING_RAW` or use the `ByteChr()` function. See Section 51.7 [ByteChr], page 1026, for details.

INPUTS

`value`	code point value
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$` resulting string

EXAMPLE

```
test$ = Chr(65)
Print(test$)
```

This will print `"A"` to the screen because 65 is the code point value for `"A"`, in both ASCII and Unicode character encodings.

51.15 CompareStr

NAME

CompareStr – compare two strings (V7.0)

SYNOPSIS

```
r = CompareStr(s1$, s2$[, casesen, encoding])
```

FUNCTION

This function compares `s1$` and `s2$` and returns how the two strings are related. If `s1$` is lexically less than `s2$`, -1 is returned. If `s1$` is lexically greater than `s2$`, 1 is returned, otherwise, i.e. if the strings are equal, the return value is 0.

The optional argument `casesen` can be used to specify whether or not the strings should be compared in a case-sensitive manner. This defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s1$` first string to compare

`s2$` second string to compare

`casesen` optional: `True` for a case-sensitive comparison, otherwise FALSE; the default is `True` or whatever default has been set using the `IgnoreCase()` command

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`r` relation of the two strings

EXAMPLE

```
DebugPrint(CompareStr("z", "a"))
```

The code above prints 1 because `"z"` is lexically greater than `"a"`.

51.16 ConvertStr

NAME

ConvertStr – convert between character encodings (V7.0)

SYNOPSIS

```
c$ = ConvertStr(s$, inencoding, outencoding)
```

FUNCTION

Converts `s$` from the character encoding specified by `inencoding` to the character encoding specified by `outencoding` and returns it. See Section 13.2 [Character encodings], page 149, for a list of valid encodings.

INPUTS

`s$` input string

`inencoding`
source character encoding

`outencoding`
destination character encoding

RESULTS

`c$` converted string in new character encoding

51.17 CountStr

NAME

CountStr – count number of substring occurrences (V4.5)

SYNOPSIS

```
n = CountStr(s$, sub$[, casesen, startpos, encoding])
```

FUNCTION

This function counts the number of occurrences of `sub$` inside `s$`. The optional argument `casesen` specifies whether or not the strings shall be compared in a case sensitive manner. Furthermore, you can use the `startpos` argument to specify an offset into `s$` at which `CountStr()` should start counting. This offset is in characters, not in bytes. Position 0 means the beginning of the string.

The `casesen` parameter defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` source string

`sub$` the string to search in `s$`

`casesen` optional: set this to `True` if the case in `s$` must match the case setting in `sub$`; the default is `True` or whatever default has been set using the `IgnoreCase()` command

`startpos` optional: character offset inside `s$` to start the search (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`n` number of occurrences of `sub$` inside `s$`

EXAMPLE

```
ret = CountStr("What is that on your head? Is that a new hat? " ..
    "You have not had that on our last chat!", "hat")
```

The above code should return 6 because "hat" occurs 6 times in the source string.

51.18 CRC32Str

NAME

CRC32Str – calculate CRC32 checksum of string (V5.0)

SYNOPSIS

```
sum = CRC32Str(s$)
```

FUNCTION

This function calculates the CRC32 checksum of the string specified in `s$` and returns it. Note that Hollywood strings can also contain binary data so that you can also use this function with non-text strings.

If you want to compute the CRC32 checksum of a file, use the `CRC32()` function instead.

INPUTS

`s$` string whose checksum you want to have calculated

RESULTS

`sum` CRC32 checksum of string

51.19 EmptyStr

NAME

EmptyStr – check if string is empty (V7.1)

SYNOPSIS

```
bool = EmptyStr(s$)
```

FUNCTION

This function returns `True` if the string is empty, i.e. if it contains only whitespace characters. The following characters are whitespace characters: space, form-feed ("\f"), newline ("\n"), carriage return ("\r"), horizontal tab ("\t"), and vertical tab ("\v").

INPUTS

`s$`	input string

RESULTS

`bool`	`True` if input string contains only whitespace characters

51.20 EndsWith

NAME

EndsWith – check if string ends on substring (V7.1)

SYNOPSIS

```
bool = EndsWith(s$, substr$[, casesen, encoding])
```

FUNCTION

This function can be used to check if `s$` ends on the substring specified by `substr$`. If it does, `True` is returned, `False` otherwise. If the optional argument `casesen` is set to `False`, the strings do not have to match in case. `casesen` defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	input string
`substr$`	string to compare against `s$`
`casesen`	optional: whether or not a case sensitive comparison should be activated; the default is `True` or whatever default has been set using the `IgnoreCase()` command
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	boolean value indicating success or failure

51.21 Eval

NAME

Eval – evaluate string expression (V5.0)

SYNOPSIS

```
val = Eval(expr$[, table])
```

FUNCTION

`Eval()` evaluates the numeric expression passed to it in `expr$` and returns its result as a number. The string passed in `expr$` can use all operators that are recognized

by Hollywood except the string concatenation operator because this operator requires a string while the `Eval()` function only works with numbers. You can also prioritize certain subexpressions by using parentheses. The operators defined by `Eval()` use the same priorities as in Hollywood itself. See Section 9.7 [Operator priorities], page 110, for a list of all Hollywood operators and their priorities.

Numbers inside `expr$` can be specified in decimal or hexadecimal format. If you use hexadecimal format, you have to prefix the number using the $ dollar sign.

It is also possible to use variables in the expression. Your script variables, however, are not automatically available to `Eval()`. To make your script variables available to `Eval()`, you need to set the `MapVariables` tag in the optional table argument to `True`. Alternatively, you can also choose to define private variables for `Eval()` by setting the `Variables` tag in the optional table argument. If both script variables and private variables are used, the private ones will take precedence.

Here is a list of all tags currently supported by the optional table argument:

`Variables:`
: This tag can be used to pass an array of variables that you want your expression to be able to access to `Eval()`. You have to pass an array here that consists of a number of `Name` and `Value` pairs. `Name` must contain the desired name for the variable and `Value` must contain the value the variable should be initialized to. The semantics of the variable name are the same as in Hollywood, i.e. it is only allowed to use alphanumerical characters plus the special characters `"$"`, `"!"`, and `"_"`. The first character must not be a number. See below for an example.

`MapVariables:`
: If this tag is set to `True`, `Eval()` will also take normal Hollywood variables into account when evaluating the expression. Note that the variables must be numeric variables only and the ones declared in the `Variables` tag (see above) will take precedence. Defaults to `False`. (V9.0)

`NoDeclare:`
: If this tag is set to `True`, all undeclared variables in the expression will be treated as 0. By default, `Eval()` will throw an error when trying to access an undeclared variable. Defaults to `False`. (V9.0)

INPUTS

`expr$` numeric expression to evaluate

`table` optional: table containing further arguments

RESULTS

`val` evaluation result

EXAMPLE

```
v = Eval("5*(6+1)")
```

The call above returns 35.

```
v = Eval("var1*(var2+1)", {Variables = {{Name = "var1", Value = 5},
```

```
        {Name = "var2", Value = 6}})
```

The code above does the same as the first example but uses variables instead of direct numbers.

```
var1 = 5
var2 = 6
v = Eval("var1*(var2+1)", {MapVariables = True})
```

The code above does the same as the first two examples but now maps the Hollywood variables `var1` and `var2` to `Eval()`'s variable space.

51.22 FindStr

NAME

FindStr – find a substring in a string

SYNOPSIS

```
pos = FindStr(string$, substring$[, casesensitive, startpos, encoding])
```

FUNCTION

Searches for `substring$` in `string$` and returns the position of the substring. The position is returned in characters, not in bytes, starting at position 0 for the first character. If `substring$` cannot be found, -1 is returned. The optional argument `casesensitive` allows you to specify if the search should be case sensitive. This defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

Starting with Hollywood 4.5, you can also specify a starting position for the search in the optional argument `startpos`. This position needs to be specified in characters, not in bytes. Specifying 0 as the starting position means start searching from the beginning of the string. This is also the default.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$` string to search in

`substring$`
: string to find in `string$`

`casesensitive`
: `True` for a case sensitive search or `False` for a case insensitive search; the default is `True` or whatever default has been set using the `IgnoreCase()` command

`startpos` optional: starting position of the search operation in characters (defaults to 0) (V4.5)

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`pos` position of `substring$` in `string$` in characters or -1 if not found

EXAMPLE

```
result = FindStr("Hello World!", "World")
Print(result)
```

This will print "6" which is the position of the first char "W".

51.23 FormatNumber

NAME

FormatNumber – convert number to string with digit separation (V10.0)

SYNOPSIS

```
s$ = FormatNumber(n[, decimals, point$, thousands$])
```

FUNCTION

This function converts the number specified by `n` to a string, separating the digits by the thousands and clipping the decimal places to the number passed in the `decimals` argument. You can also pass the character that should be used as a decimal separator in `point$` and the character that should be used as a thousands separator in `thousands$`. This makes `FormatNumber()` suitable for locale-sensitive number formatting. You can use the `GetLocaleInfo()` function to get the decimal point and thousand separator character for the current locale. See Section 36.8 [GetLocaleInfo], page 744, for details.

INPUTS

`n` number to convert to string

`decimals` optional: number of decimal places to use (defaults to 0)

`point$` optional: character to use as decimal point (defaults to ".")

`thousands$`
optional: character to use as thousands separator (defaults to ",")

RESULTS

`s$` formatted string

EXAMPLE

```
t = GetLocaleInfo()
s$ = FormatNumber(1234567.89, 2, t.DecimalPoint, t.ThousandSeparator)
DebugPrint(s$)
```

The code above will convert 1234567.89 to a string using the current locale's decimal point and thousand separator.

51.24 FormatStr

NAME

FormatStr – compose a C-style formatted string (V5.0)

SYNOPSIS

```
s$ = FormatStr(fmt$, ...)
```

FUNCTION

This function can be used to compose a C-style formatted string with Hollywood. You have to pass the formatting template in the first argument and you have to pass one additional argument for every token in the template. Hollywood supports most of the tokens of C's `printf` specification. Here is a list of all tokens that are currently supported:

`%c`	ASCII character
`%d`	Signed decimal integer
`%i`	Same as `%d`
`%o`	Unsigned octal integer
`%u`	Unsigned decimal integer
`%x`	Unsigned hexadecimal integer (lower case notation)
`%X`	Unsigned hexadecimal integer (upper case notation)
`%e`	Floating point number in exponential notation
`%E`	Same as `%e` but in upper case exponential notation
`%f`	Floating point number in normal notation
`%g`	Floating point number in `%e` or `%f` format (whichever is more compact)
`%G`	Same as `%g` but uses upper case if exponential notation is used
`%s`	String

You can also specify a width field before the token to limit the number of characters that the specific token should add to the string. For example, if you use the token `%.6x`, the hexadecimal number generated by this function will always have 6 digits.

As the percent sign is used for tokens by this function, you need to escape it if you just want to append a percent sign to the string. In that case simply use a double percent sign (`%%`).

INPUTS

`fmt$`	formatting template containing one or more tokens (see above for supported tokens)
`...`	additional arguments (one for each token present in `fmt$`)

RESULTS

`s$`	resulting string

EXAMPLE

```
a = 128
s$ = FormatStr("The number " .. a .. " is $%x in hexadecimal notation", a)
```

The code above converts the number 128 to hexadecimal notation.

```
a = 255
s$ = FormatStr("The number " .. a .. " is $%.6x in RGB notation", a)
```

The code above converts the number 255 into a 6 digit hexadecimal value which is often used to specify RGB colors.

51.25 HexStr

NAME

HexStr – convert value to a hex string (V1.5)

SYNOPSIS

```
hex$ = HexStr(val)
```

FUNCTION

This function converts the value specified by `val` into hexadecimal digits and returns it as a string. The returned string will be prefixed with a dollar sign ($) and all alphabetical hexadecimal digits will be in upper case.

INPUTS

`val`	value to convert

RESULTS

`hex$`	hexadecimal representation of `val`

EXAMPLE

```
a$ = HexStr(255)
```

This will return the string `"$FF"`.

51.26 IgnoreCase

NAME

IgnoreCase – define default case sensitive setting (V9.0)

SYNOPSIS

```
IgnoreCase(casesen)
```

FUNCTION

This function allows you to specify whether functions like `FindStr()` and `ReplaceStr()` should be case sensitive by default. If you set the `casesen` argument to `True`, those functions will be case sensitive. If you set it to `False`, they won't be case sensitive. By default, they are case sensitive.

`IgnoreCase()` affects the following Hollywood functions:

- `CompareStr()`

- `CountStr()`
- `EndsWith()`
- `FindStr()`
- `ReplaceStr()`
- `ReverseFindStr()`
- `StartsWith()`

INPUTS

`casesen` `True` if functions should case sensitive by default, `False` otherwise

51.27 InsertStr

NAME

InsertStr – insert substring into string with optional overwriting (V4.5)

SYNOPSIS

```
var$ = InsertStr(s$, sub$, pos[, overwrite, encoding])
```

FUNCTION

This function inserts `sub$` into `s$` at the position specified by `pos` (0 means beginning of the string). The position needs to be specified in characters, not in bytes. If the optional argument `overwrite` is set to `True`, `sub$` will overwrite any characters that were previously there on the insert position.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` string to insert `sub$` into

`sub$` substring to insert

`pos` position at which to insert `sub$` in characters (0 means beginning)

`overwrite`
optional: whether the `sub$` should overwrite characters in `s$` or shift them to the right (defaults to `False` which means no overwriting)

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$` resulting string

EXAMPLE

```
Print(InsertStr("Hollywood is a very cool program!", " very", 19))
```

The above code prints "Hollywood is a very very cool program!"

```
Print(InsertStr("Hollywood is a very cool program!", "good", 20, True))
```

The above code prints "Hollywood is a very good program!"

51.28 IsAlNum

NAME

IsAlNum – check if character is alphanumeric (V7.0)

SYNOPSIS

```
bool = IsAlNum(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is either a decimal digit or an upper or lower case letter and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.29 IsAlpha

NAME

IsAlpha – check if character is alphabetic (V7.0)

SYNOPSIS

```
bool = IsAlpha(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is an alphabetic letter and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.30 IsCntrl

NAME

IsCntrl – check if character is a control character (V7.0)

SYNOPSIS

```
bool = IsCntrl(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is a control character and returns `True` if it is, otherwise `False`. A control character is a character that is not printable, e.g. backspace, tab, line feed, escape, etc. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.31 IsDigit

NAME

IsDigit – check if character is a decimal digit (V7.0)

SYNOPSIS

```
bool = IsDigit(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is a decimal digit and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.32 IsGraph

NAME

IsGraph – check if character has a graphical representation (V7.0)

SYNOPSIS

```
bool = IsGraph(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` has a graphical representation and returns `True` if it has, otherwise `False`. This function does the same as `IsPrint()` except that it does not return `True` for the space character. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.33 IsLower

NAME

IsLower – check if character is a lowercase letter (V7.0)

SYNOPSIS

```
bool = IsLower(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is a lowercase letter and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.34 IsPrint

NAME

IsPrint – check if character is printable (V7.0)

SYNOPSIS

```
bool = IsPrint(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is printable and returns `True` if it is, `False` otherwise. Printable characters are all characters that have a graphical representation, including the space character. Printable characters are the opposite to control characters. `IsPrint()` basically does the same as `IsGraph()`, the only difference being that `IsPrint()` returns `True` for the space character as well whereas `IsGraph()` doesn't. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool`	`True` or `False`, depending on the test's result

51.35 IsPunct

NAME

IsPunct – check if character is a punctuation character (V7.0)

SYNOPSIS

```
bool = IsPunct(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is a punctuation character and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	source string
`pos`	optional: index of character to test (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool` `True` or `False`, depending on the test's result

51.36 IsSpace

NAME

IsSpace – check if character is a white-space character (V7.0)

SYNOPSIS

```
bool = IsSpace(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is a white-space character and returns `True` if it is, otherwise `False`. White-space characters are characters such as space, tab, newline, carriage return, etc. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` source string

`pos` optional: index of character to test (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool` `True` or `False`, depending on the test's result

51.37 IsUpper

NAME

IsUpper – check if character is an uppercase letter (V7.0)

SYNOPSIS

```
bool = IsUpper(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is an uppercase letter and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` source string

`pos` optional: index of character to test (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool` `True` or `False`, depending on the test's result

51.38 IsXDigit

NAME

IsXDigit – check if character is a hexadecimal digit (V7.0)

SYNOPSIS

```
bool = IsXDigit(s$[, pos, encoding])
```

FUNCTION

Checks if the character at index `pos` inside string `s$` is a hexadecimal digit and returns `True` if it is, otherwise `False`. The optional `pos` parameter must be in characters, not in bytes. It defaults to 0 which will test the first character in `s$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` source string

`pos` optional: index of character to test (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool` `True` or `False`, depending on the test's result

51.39 LeftStr

NAME

LeftStr – return the n leftmost characters of a string

SYNOPSIS

```
var$ = LeftStr(string$, len[, encoding])
```

FUNCTION

This function returns the `len` leftmost characters of `string$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$` source string

`len` number of characters to return

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$` resulting string

EXAMPLE

```
test$ = LeftStr("Hello World!", 5)
Print(test$)
```

This will print "Hello" to the screen.

51.40 LowerStr

NAME

LowerStr – convert all characters in a string to lower case

SYNOPSIS

```
var$ = LowerStr(string$[, encoding])
```

FUNCTION

Converts all characters in `string$` to lower case and returns the new string.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$` string to convert

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$` converted string

EXAMPLE

```
test$ = LowerStr("Hello World!")
Print(test$)
```

This will print "hello world!" to the screen.

51.41 MD5Str

NAME

MD5Str – calculate MD5 checksum of string (V5.0)

SYNOPSIS

```
sum$ = MD5Str(s$)
```

FUNCTION

This function calculates the MD5 checksum of the string specified in `s$` and returns it. The 128-bit checksum is returned as a string containing 16 hex digits. Note that Hollywood strings can also contain binary data so that you can also use this function with non-text strings.

If you want to compute the MD5 checksum of a file, use the `MD5()` function instead.

INPUTS

`s$` string whose checksum you want to have calculated

RESULTS

`sum$` MD5 checksum of string

51.42 MidStr

NAME

MidStr – extract characters from a string

SYNOPSIS

```
var$ = MidStr(string$, startpos[, len, encoding])
```

FUNCTION

Returns `len` characters from `string$` (starting at position `startpos`) as a new string. The starting position `startpos` needs to be specified in characters, not in bytes.

Starting with Hollywood 6.1 `len` may be omitted or set to -1. In that case, the remaining characters in `string$` will be returned.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$` source string

`startpos` starting offset in characters (first character is at position 0)

`len` optional: how many characters shall be returned; if you omit this argument or set it to -1, all remaining characters will be returned

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$` resulting string

EXAMPLE

```
test$ = MidStr("Hello World!", 4, 3)
Print(test$)
```

This will print "o W" to the screen.

51.43 PadNum

NAME

PadNum – convert number to string with padding (V2.0)

SYNOPSIS

```
s$ = PadNum(num, len)
```

FUNCTION

This function converts the integer specified in `num` to a string. Additionally, it adds leading zeros until the string is of the length specified in `len`. The number must not be negative and must not contain any decimal places.

INPUTS

`num`	number to convert to string
`len`	desired string length

RESULTS

`s$`	padded string

EXAMPLE

```
DebugPrint(PadNum(9, 2))
```

Prints "09".

51.44 PatternFindStr

NAME

PatternFindStr – parse a string using pattern matching iterator function (V5.0)

SYNOPSIS

```
func, state, val = PatternFindStr(s$, pat$[, encoding])
```

FUNCTION

This function can be used in conjunction with the generic `For` statement to parse the string specified in `s$` according to the pattern specified in `pat$`. As required by the generic `For` statement, `PatternFindStr()` will return three values: An iterator function, a private state information, and an initial value for the traversal. Each time it is called, the iterator function will return the next captures from pattern `pat$` over string `s$`. If `pat$` specifies no captures, then the whole match is produced in each call.

The pattern specified in `pat$` must adhere to the pattern syntax as described in the documentation of the `PatternReplaceStr()` function. See Section 51.47 [PatternReplaceStr], page 1053, for details.

See Section 11.4 [Generic For statement], page 127, for details.

Starting with Hollywood 6.0, this function is also available in a version that can be used without a generic `For` statement. So if you only care about the first occurrence of `pat$` in `s$`, then you might want to use `PatternFindStrDirect()` instead. See Section 51.45 [PatternFindStrDirect], page 1051, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	string to parse
`pat$`	pattern according to which the string should be parsed
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`func`	iterator function
`state`	private state information
`val`	initial traversal value

EXAMPLE

```
s$ = "Hello World from Hollywood"
For w$ In PatternFindStr(s$, "%a+") Do DebugPrint(w$)
```

The above code will iterate over all the words from string `s$`, printing one per line.

```
t = {}
s$ = "Name=Andreas, Sex=Male, Nationality=German"
For k, v in PatternFindStr(s$, "(%w+)=(%w+)") Do t[k] = v
```

The above example collects all pairs key=value from the given string into a table.

51.45 PatternFindStrDirect

NAME

PatternFindStrDirect – parse a string using pattern matching (V6.0)

SYNOPSIS

```
start, end, ... = PatternFindStrDirect(s$, pat$[, start, encoding])
```

FUNCTION

This function parses the string specified in `s$` according to the pattern specified in `pat$`. If there is a match, the indices of where the match starts and ends in the source string are returned together with all strings that have been captured. If there is no match, `PatternFindStrDirect()` will return -1. The optional argument `start` can be used to specify a character index inside `s$` where searching should begin. This defaults to 0 which means `PatternFindStrDirect()` should start at the beginning of `s$`.

This function does pretty much the same as `PatternFindStr()` but does not require you to use a generic `For` statement. Instead, all captures are returned along with the start and end indices. Keep in mind, though, that `PatternFindStrDirect()` does not operate inside a generic `For` loop, so only the first occurrence of `pat$` inside `s$` will be handled of course.

If you do not need the start and end indices, you can also use the `PatternFindStrShort()` function instead. See Section 51.46 [PatternFindStrShort], page 1052, for details.

The pattern specified in `pat$` must adhere to the pattern syntax as described in the documentation of the `PatternReplaceStr()` function. See Section 51.47 [PatternReplaceStr], page 1053, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	string to parse
`pat$`	pattern according to which the string should be parsed
`start`	optional: position where search should start (defaults to 0)
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`start`	position of the first match inside `s$` or -1 for no match
`end`	position of the last match inside `s$`
`...`	individual strings with all captures

EXAMPLE

```
DebugPrint(PatternFindStrDirect("Name=Andreas", "(%w+)=(%w+)"))
```

The above example returns the strings next to the equal sign and the range 0 to 11 which describes the complete source string.

51.46 PatternFindStrShort

NAME

PatternFindStrShort – parse a string using pattern matching (V6.0)

SYNOPSIS

```
... = PatternFindStrShort(s$, pat$[, start, encoding])
```

FUNCTION

This function does the same as `PatternFindStrDirect()` but does not return the start and end indices. See Section 51.45 [PatternFindStrDirect], page 1051, for details.

The pattern specified in `pat$` must adhere to the pattern syntax as described in the documentation of the `PatternReplaceStr()` function. See Section 51.47 [PatternReplaceStr], page 1053, for details.

INPUTS

`s$`	string to parse
`pat$`	pattern according to which the string should be parsed

`start` optional: position where search should start (defaults to 0)

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`...` individual strings with all captures

EXAMPLE

```
DebugPrint(PatternFindStrShort("Name=Andreas", "(%w+)=(%w+)"))
```

The above example returns the strings next to the equal sign.

51.47 PatternReplaceStr

NAME

PatternReplaceStr – modify string contents using pattern matching (V5.0)

SYNOPSIS

```
r$, n = PatternReplaceStr(s$, pat$, repl[, n, encoding])
```

FUNCTION

`PatternReplaceStr()` can be used to modify a string's contents using pattern matching. This function is powerful and can be used for all kinds of string operations. It returns a copy of `s$` in which all occurrences of the pattern `pat$` have been replaced by a replacement string specified by `repl`. `PatternReplaceStr()` also returns, as a second value, the total number of substitutions made.

The third argument `repl` can be either a string or a callback function. If `repl` is a string, then its value is used for replacement. Any sequence in `repl` of the form `%n`, with `n` between 1 and 9, stands for the value of the n-th captured substring (see below). If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument. If the value returned by this function is a string, then it is used as the replacement string; otherwise, the replacement string is the empty string.

The optional argument `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of `pat$` is replaced.

The pattern specified in `pat$` is made up of a sequence of pattern items. A pattern item is usually a character class which in turn represents a set of characters. The following combinations are allowed in describing a character class:

`x (where x is not one of the magic characters ^$()%.[]*+-?)`
Represents the character x itself.

`. (a dot)` Represents all characters.

`%a` Represents all letters.

`%c` Represents all control characters.

`%d` Represents all digits.

`%g` Represents all characters that have a graphical representation. (V7.0)

`%l` Represents all lowercase letters.

`%p` Represents all punctuation characters.

`%s` Represents all space characters.

`%u` Represents all uppercase letters.

`%w` Represents all alphanumeric characters.

`%x` Represents all hexadecimal digits.

`%z` Represents the character with representation 0.

`%x (where x is any non-alphanumeric character)`
Represents the character `x`. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a `%` when used to represent itself in a pattern.

`[set]` Represents the class which is the union of all characters in set. A range of characters may be specified by separating the end characters of the range with a `-`. All classes `%x` described above may also be used as components in set. All other characters in set represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the `-` character. The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.

`[^set]` Represents the complement of set, where set is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The following items are valid pattern items:

- a single character class, which matches any single character in the class
- a single character class followed by `*`, which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence
- a single character class followed by `+`, which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence
- a single character class followed by `-`, which also matches 0 or more repetitions of characters in the class. Unlike `*`, these repetition items will always match the shortest possible sequence
- a single character class followed by `?`, which matches 0 or 1 occurrence of a character in the class
- `%n`, for `n` between 1 and 9; such item matches a substring equal to the n-th captured string (see below)

– `%bxy`, where `x` and `y` are two distinct characters; such item matches strings that start with `x`, end with `y`, and where the `x` and `y` are balanced. This means that, if one reads the string from left to right, counting +1 for an `x` and -1 for a `y`, the ending `y` is the first `y` where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

A pattern is a sequence of pattern items. A `^` at the beginning of a pattern anchors the match at the beginning of the subject string. A `$` at the end of a pattern anchors the match at the end of the subject string. At other positions, `^` and `$` have no special meaning and represent themselves.

A pattern may contain sub-patterns enclosed in parentheses; they describe captures. When a match succeeds, the substrings of the subject string that match captures are stored (captured) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `(a*(.)%w(%s*))`, the part of the string matching `a*(.)%w(%s*)` is stored as the first capture (and therefore has number 1); the character matching `.` is captured with number 2, and the part matching `%s*` has number 3.

As a special case, the empty capture () captures the current string position (a number). For instance, if we apply the pattern `()aa()` on the string "flaaap", there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	string to modify
`pat$`	pattern according to which the string should be modified
`repl`	replacement string or callback function to handle replacements (see above)
`n`	optional: maximum number of substitutions to make (defaults to the length of s$ plus 1)
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`r$`	resulting string
`n`	number of substitutions made

EXAMPLE

```
s$ = PatternReplaceStr("Hello World", "(%w+)", "%1 %1")
```

The code above returns "Hello Hello World World"

```
s$ = PatternReplaceStr("Hello World from Hollywood", "(%w+)%s*(%w+)",
     "%2 %1")
```

The code above returns "World Hello Hollywood from"

```
s$ = PatternReplaceStr("home = $HOME, user = $USER", "%$(%w+)", GetEnv)
```

The code above returns "home = /home/andreas, user = andreas" (on Linux).

```
Local t = {name = "Hollywood", version="5.0"}
s$ = PatternReplaceStr("$name_$version.jpg", "%$(%w+)", Function(v)
     Return(t[v]) EndFunction)
```

The code above returns "Hollywood_5.0.jpg"

51.48 RepeatStr

NAME

RepeatStr – repeat string multiple times (V5.0)

SYNOPSIS

```
var$ = RepeatStr(s$, n)
```

FUNCTION

This function takes `s$`, repeats it `n` times, and returns the resulting string.

INPUTS

`s$`	the string to repeat
`n`	number of times to repeat string; must be > 0

RESULTS

`var$`	the resulting string

EXAMPLE

```
Print(RepeatStr("Hollywood!", 5))
```

The above code prints "Hollywood!Hollywood!Hollywood!Hollywood!Hollywood!"

51.49 ReplaceStr

NAME

ReplaceStr – replaces a substring with another string

SYNOPSIS

```
var$ = ReplaceStr(s$, search$, replace$[, cs, startpos, encoding])
```

FUNCTION

Searches for `search$` in `s$` and replaces all occurrences of `search$` with `replace$`. The optional argument `cs` turns case sensitivity on (`True`) or off (`False`). This defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

Starting with Hollywood 4.5, you can also specify a starting position for the search in the optional argument `startpos`. This position needs to be specified in characters, not in bytes. Specifying 0 as the starting position means search and replace from the beginning of the string. This is also the default.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	string to search for `search$`
`search$`	string to search for
`replace$`	string replacement for `search$`
`cs`	`True` for case sensitivity, `False` for no case sensitivity; the default is `True` or whatever default has been set using the `IgnoreCase()` command
`startpos`	optional: starting position of the search and replace operation in characters (defaults to 0) (V4.5)
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$`	resulting string

EXAMPLE

```
test$ = "Hello World!"
test$ = ReplaceStr(test$, "World", "People")
Print(test$)
```

This will print "Hello People!" to the screen.

51.50 ReverseFindStr

NAME

ReverseFindStr – find a substring in a string in reverse (V9.0)

SYNOPSIS

```
pos = ReverseFindStr(string$, substring$[, casesen, startpos, encoding])
```

FUNCTION

Searches for `substring$` in `string$` and returns the position of the substring. In contrast to `FindStr()`, searching is done in reverse, i.e. from the string's end to its start.

The position is returned in characters, not in bytes, starting at position 0 for the first character. If `substring$` cannot be found, -1 is returned. The optional argument `casesen` allows you to specify if the search should be case sensitive. This defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

You can also specify a starting position for the search in the optional argument `startpos`. This position needs to be specified in characters, not in bytes. By default, `startpos` is set to the length of `string$` in characters minus 1.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$` string to search in

`substring$`
string to find in `string$`

`casesen` `True` for a case sensitive search or `False` for a case insensitive search; the default is `True` or whatever default has been set using the `IgnoreCase()` command

`startpos` optional: starting position of the search operation in characters (defaults to the length of `string$` minus 1)

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`pos` position of `substring$` in `string$` in characters or -1 if not found

EXAMPLE

```
result = ReverseFindStr("Hello, Hello!", "Hello")
Print(result)
```

This will print `"7"` because searching is done in reverse, which is why the position of the second `"Hello"` is returned.

51.51 ReverseStr

NAME

ReverseStr – reverse order of characters in string (V7.0)

SYNOPSIS

```
r$ = ReverseStr(s$[, encoding])
```

FUNCTION

Reverses the order of characters in string `s$` and returns the new string.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` input string

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`r$` reversed string

EXAMPLE

```
r$ = ReverseStr("Hello")
Print(r$)
```

This prints `"olleH"`.

51.52 RightStr

NAME

RightStr – return rightmost characters of a string

SYNOPSIS

```
var$ = RightStr(string$, len[, encoding])
```

FUNCTION

Returns the rightmost `len` characters from `string$`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$`	source string
`len`	number of characters to return
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$`	resulting string

EXAMPLE

```
test$ = RightStr("Hello World!", 6)
Print(test$)
```

Prints "World!" to the screen.

51.53 SplitStr

NAME

SplitStr – split a string in several pieces (V2.0)

SYNOPSIS

```
table, count = SplitStr(src$, token$[, multiple])
```

FUNCTION

This function splits the string specified in `src$` into several pieces by looking for the separator `token$` in `src$`. `token$` must be a string containing at least one character that shall act as a separator in `src$`. `SplitStr()` will return a table containing all the pieces and the number of pieces in the string as the second return value.

If the specified token does not appear in the source string, `src$` is returned.

Starting with Hollywood 7.1 there is a new optional argument named `multiple`. If this is set to `True`, multiple occurrences of `token$` next to each other will be considered a single occurrence. This can be useful when using the space character as `token$` and you want this function to work with an arbitrary number of spaces between the different parts.

Note that before Hollywood 8.0, `token$` was limited to a string using only one character. This limit has been lifted for Hollywood 8.0 and the string can now be of arbitrary length.

INPUTS

`src$` string to split

`token$` one character string containing a separator token

`multiple` optional: whether or not to treat multiple occurrences of `token$` next to each other as a single token (defaults to `False`) (V7.1)

RESULTS

`table` table where the new substrings are stored

`count` how many substrings this function created

EXAMPLE

```
array, c = SplitStr("AmigaOS3|MorphOS|AmigaOS4|WarpOS|AROS", "|")
For k = 1 To c Do NPrint(array[k - 1])
```

The above code will print

```
AmigaOS3
MorphOS
AmigaOS4
WarpOS
AROS
```

The variable `c` will be set to 5 because `SplitStr()` finds five substrings and places them in the table specified.

51.54 StartsWith

NAME

StartsWith – check if string starts with substring (V7.1)

SYNOPSIS

```
bool = StartsWith(s$, substr$[, casesen, encoding])
```

FUNCTION

This function can be used to check if `s$` starts with the substring specified by `substr$`. If it does, `True` is returned, `False` otherwise. If the optional argument `casesen` is set to `False`, the strings do not have to match in case. `casesen` defaults to the global case sensitive default mode set using `IgnoreCase()`. See Section 51.26 [IgnoreCase], page 1040, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` input string

`substr$` string to compare against `s$`

`casesen` optional: whether or not a case sensitive comparison should be activated; the default is `True` or whatever default has been set using the `IgnoreCase()` command

`encoding` optional: character encoding to use (defaults to default string encoding)

RESULTS

`bool` boolean value indicating success or failure

51.55 StripStr

NAME

StripStr – strip whitespace from string (V7.1)

SYNOPSIS

```
r$ = StripStr(s$)
```

FUNCTION

This function strips all whitespace characters from both ends of a string and returns the stripped string. The following characters are whitespace characters: space, form-feed ("\f"), newline ("\n"), carriage return ("\r"), horizontal tab ("\t"), and vertical tab ("\v").

INPUTS

`s$` input string

RESULTS

`r$` stripped string

EXAMPLE

```
s$ = StripStr("    Hello World    ")
```

This will return "Hello World".

51.56 StrLen

NAME

StrLen – return character length of a string

SYNOPSIS

```
len = StrLen(str$[, encoding])
```

FUNCTION

This function returns the character length of `str$`. Note that for Unicode strings this isn't necessarily the same as the byte length of `str$`. To find out the byte length of a string, use the `ByteLen()` function or pass a non-Unicode encoding in `encoding`.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`str$` input string

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`len` character length of string

EXAMPLE

```
len = StrLen("Hello")
```

This will return 5.

51.57 StrStr

NAME

StrStr – convert a number to a string

SYNOPSIS

```
var$ = StrStr(value[, digits])
```

FUNCTION

Converts the numerical value `value` to a string and returns it. The optional argument `digits` allows you to define how many decimal places shall be used if value is a real number. It defaults to 2.

INPUTS

`value` number to convert to string

`digits` optional: how many decimal places to use if a float value is specified (defaults to 2)

RESULTS

`var$` string representing the numeric value

EXAMPLE

```
test$ = StrStr(256)
Print(test$)
```

Prints "256" to the screen.

51.58 StrToArray

NAME

StrToArray – convert a string to an array of code points (V2.0)

SYNOPSIS

```
t = StrToArray(s$[, encoding])
```

FUNCTION

This function extracts the code point values from `s$` and returns them in a table. The table will have as many elements as the string has characters plus a terminating zero.

To convert the array back to a string, you can use the function `ArrayToStr()`. See Section 51.2 [ArrayToStr], page 1023, for details.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$` string to convert

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`t` a table containing the code point values of the string's characters

EXAMPLE

```
t = StrToArray("Hello World")
DebugPrint(Chr(t[6]))
```

Prints "W".

51.59 ToNumber

NAME

ToNumber – convert a string to a number (V2.0)

SYNOPSIS

```
n = ToNumber(s$[, base])
```

FUNCTION

This function tries to convert the string specified in `s$` to a number. The optional argument `base` can be used to convert binary, octal, and hexadecimal numbers or any other bases ranging from 2 to 36. It defaults to 10 (decimal). In bases from 11 to 36 the letters of the English alphabet are used as the additional digits (10=A, 35=Z). Case sensitivity is not required.

Starting with Hollywood 6.0 this function can also convert a variable of type `#LIGHTUSERDATA` to a number. Since this variable type is meant to store pointers, it is only interesting for expert users or debugging purposes.

INPUTS

`s$` string to convert

`base` optional: base of the conversion (defaults to 10)

RESULTS

`number` converted number

EXAMPLE

```
r = ToNumber("10000")          ; returns 10000
r = ToNumber("10110111", 2)    ; returns 183
r = ToNumber("523", 8)         ; returns 339
r = ToNumber("FFFF", 16)       ; returns 65535
```

51.60 ToString

NAME

ToString – convert any data type to a string (V2.0)

SYNOPSIS

```
s$ = ToString(data)
```

FUNCTION

This function can convert any type to a string. You can pass in tables, functions, strings, numbers and `Nil`. `ToString()` is also used by the `Print()` and `DebugPrint()` commands so they can print any types, too.

Additionally, if you pass in a table which metatable has a `__tostring` field, this metamethod is called.

INPUTS

`data` value to convert to string

RESULTS

`s$` a string

EXAMPLE

```
s$ = ToString(DisplayBrush)   ; returns "Function: 74cd2456"
s$ = ToString({1,2,3,4,5})    ; returns "Table: 74ab1344"
s$ = ToString(Nil)            ; returns "Nil"
s$ = ToString(5)              ; returns "5"
s$ = ToString("Hello")        ; returns "Hello"
```

51.61 ToUserData

NAME

ToUserData – convert number to a userdata pointer (V6.0)

SYNOPSIS

```
ptr = ToUserData(val)
```

FUNCTION

This function can be used to turn an arbitrary number into a variable of type `#LIGHTUSERDATA` which is used to store memory pointers. You could then pass this variable to a function which expects a `#LIGHTUSERDATA` parameter.

Note that this function is dangerous and should only be used by people who know what they are doing. Using pointers that point to unallocated address space can easily crash your script.

To convert a pointer of type `#LIGHTUSERDATA` back into a number, use the `ToNumber()` function. See Section 51.59 [ToNumber], page 1063, for details.

INPUTS

`val` numeric value to be converted into a `#LIGHTUSERDATA` pointer

RESULTS

`ptr`	pointer of type `#LIGHTUSERDATA`

51.62 TrimStr

NAME

TrimStr – strip leading or tailing characters (V2.0)

SYNOPSIS

```
s$ = TrimStr(src$, chr$, tail[, encoding])
```

FUNCTION

This function can be used to strip all characters that match `chr$` from the head or the tail of `src$`. The string `chr$` must only contain one character. `tail` must be `True` to start stripping from the right or `False` to start from the left side. The stripped string will be returned.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`src$`	string to strip
`chr$`	a single character string
`tail`	`True` or `False` indicating where to begin
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`s$`	a stripped string

EXAMPLE

```
a$ = TrimStr("aaaaHello World", "a", False)
DebugPrint(a$)
```

Prints "Hello World".

```
a$ = TrimStr("aaaaHello Worldaaaa", "a", True)
DebugPrint(a$)
```

Prints "aaaaHello World".

51.63 UnleftStr

NAME

UnleftStr – remove rightmost characters from a string

SYNOPSIS

```
var$ = UnleftStr(string$, len[, encoding])
```

FUNCTION

Removes `len` rightmost characters from `string$` and returns the new string.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$`	string to remove characters from
`len`	number of characters to remove
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$`	resulting string

EXAMPLE

```
test$ = UnleftStr("Hello World!", 7)
Print(test$)
```

This will print "Hello".

51.64 UnmidStr

NAME

UnmidStr – remove characters from the middle of a string (V4.5)

SYNOPSIS

```
var$ = UnmidStr(s$, pos, len[, encoding])
```

FUNCTION

This function removes `len` characters from `s$` starting at position `pos` in the string. The position needs to be specified in characters, not in bytes. Position 0 indicates the start of the string. The truncated string is returned.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`s$`	string to truncate
`pos`	position (in characters) at which to start removing characters
`len`	number of characters to remove
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$`	truncated string

EXAMPLE

```
Print(UnmidStr("This is definitely not a funny example", 19, 4))
```

The task of figuring out what the mysterious call above might do is left to the reader.

51.65 UnrightStr

NAME

UnrightStr – remove leftmost characters from a string

SYNOPSIS

```
var$ = UnrightStr(string$, len[, encoding])
```

FUNCTION

Removes the `len` leftmost characters from `string$` and returns the new string.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$`	string to remove characters from
`len`	number of characters to remove
`encoding`	optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$`	resulting string

EXAMPLE

```
test$ = UnrightStr("Hello World!", 6)
Print(test$)
```

This will print "World!".

51.66 UpperStr

NAME

UpperStr – convert string to upper case

SYNOPSIS

```
var$ = UpperStr(string$[, encoding])
```

FUNCTION

Converts all characters in `string$` to upper case.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

INPUTS

`string$`	string to convert to upper case

`encoding` optional: character encoding to use (defaults to default string encoding) (V7.0)

RESULTS

`var$` resulting string

EXAMPLE

```
Print(UpperStr("Hello World!"))
```

Prints "HELLO WORLD!" to the screen.

51.67 Val

NAME

Val – convert a string to a number

SYNOPSIS

```
var, chrs = Val(string$)
```

FUNCTION

Converts the specified string into a number. If the string does not start with a number, 0 will be returned. The string can also contain hexadecimal numbers starting with a "$".

Starting with Hollywood 2.0 you can also pass a string which contains a binary number to this function now. Simply prefix it with a '%' character. Additionally, a second value is returned now which specifies how many characters `Val()` has read from the string. This allows you to determine the length of the number.

INPUTS

`string$` string to convert to number

RESULTS

`var` variable that receives the converted number

`chrs` number of characters converted

EXAMPLE

```
result, chrs = Val("500 people were on the train.")
Print(result, "-", chrs)
```

This will print "500-3" to the screen.

51.68 ValidateStr

NAME

ValidateStr – validate string (V7.0)

SYNOPSIS

```
ok, n = ValidateStr(s$[, encoding])
```

FUNCTION

Validates the string specified by `s$` and returns `True` if the string contains only valid characters, `False` otherwise. The second return value contains the number of valid characters in the string. If validation succeeds, this will be the same as the result of `StrLen()`. Otherwise this will tell you the offset of the first invalid character in the string.

The optional `encoding` parameter can be used to set the character encoding to use. This defaults to the default string encoding set using `SetDefaultEncoding()`. See Section 13.2 [Character encodings], page 149, for details.

This function is only useful in case `#ENCODING_UTF8` is used. If the encoding is set to `#ENCODING_ISO8859_1`, this function will always return `True`.

INPUTS

`s$`	input string
`encoding`	optional: character encoding to use (defaults to default string encoding)

RESULTS

`ok`	boolean value indicating success or failure
`n`	number of valid characters in the string (V7.1)

52 System library

52.1 Beep

NAME

Beep – play the system beep (V8.0)

SYNOPSIS

```
Beep([type])
```

FUNCTION

This function can be used to play the system's default beep sound. For systems which support different beep sounds for different contexts, you can pass the optional `type` argument to specify which beep sound should be played.

The following beep types are currently supported:

`#BEEPSYSTEM:`
: System beep sound. This is the default.

`#BEEPERROR:`
: Error beep sound.

`#BEEPWARNING:`
: Warning beep sound.

`#BEEPQUESTION:`
: Question beep sound.

`#BEEPINFORMATION:`
: Information beep sound.

Note that not all operating systems support all kinds of beeps. Only `#BEEPSYSTEM` is supported on every platform.

INPUTS

`type` optional: type of beep sound to play (defaults to `#BEEPSYSTEM`)

EXAMPLE

```
Beep()
```

The code above plays the default beep.

52.2 CollectGarbage

NAME

CollectGarbage – force a garbage collection or set new threshold (V2.0)

SYNOPSIS

```
CollectGarbage([threshold])
```

FUNCTION

This function can be used to force a garbage collection. To do that, you have to call `CollectGarbage()` without specifying the optional argument `threshold`. If you pass the

optional threshold argument, however, then this value will be set as the new garbage collector threshold. This means that whenever the garbage size is bigger than the specified threshold in kilobytes, Hollywood will automatically run the garbage collector.

If you leave out the optional argument, Hollywood will immediately run the garbage collector and set a new threshold that is twice the size of garbage just collected.

To get information about the state of the garbage collector, call the `GCInfo()` function.

INPUTS

`threshold`
optional: threshold in kilobytes that specifies when Hollywood should run the garbage collector (defaults to 0 which means run the garbage collector immediately)

52.3 DisableLineHook

NAME
DisableLineHook – disable the line hook (V6.0)

SYNOPSIS
`DisableLineHook()`

FUNCTION
This function can be used to disable Hollywood's line hook. The line hook is a function which is automatically called by Hollywood after every code line. It is responsible for controlling several tasks that Hollywood needs for its housekeeping, for example event handling and video playback. Thus, it is very important that the line hook gets called several times per second at least, otherwise your application will become unresponsive and video playback will stutter. Disabling the line hook for a brief period of time, however, can increase the raw performance of Hollywood's virtual machine significantly since all the code in the linehook isn't executed at all. Hence, you may want to disable it temporarily if you need to do some complex calculations as fast as possible. But make sure to enable the line hook again as soon as possible to prevent your application from becoming unresponsive.

To enable the line hook again, you will have to call `EnableLineHook()`.

INPUTS
none

52.4 ELSE

NAME
ELSE – block to enter if all conditions failed (V7.0)

SYNOPSIS
`@ELSE`

FUNCTION
This preprocessor command signals the beginning of a block that should be entered if all previous `@IF` and `@ELSEIF` preprocessor command of the same scope didn't trigger.

See Section 52.17 [IF], page 1080, for details and an example.

INPUTS
none

EXAMPLE
See Section 52.17 [IF], page 1080.

52.5 ELSEIF

NAME
ELSEIF – test for another condition (V7.0)

SYNOPSIS
```
@ELSEIF val
```

FUNCTION
This preprocessor command tests for the condition specified in `val`. If it is `True`, the preprocessor will continue parsing your script. If `val` is `False`, however, `@ELSEIF` will branch to the next `@ELSEIF`, `@ELSE`, or `@ENDIF` statement. Note that the condition must be a constant expression since `@ELSEIF` operates at preprocessor level, i.e. script variables are not available at that time.

See Section 52.17 [IF], page 1080, for details and an example.

INPUTS

`val`	condition to test; must be a constant expression

EXAMPLE
See Section 52.17 [IF], page 1080.

52.6 EnableLineHook

NAME
EnableLineHook – enable the line hook (V6.0)

SYNOPSIS
```
EnableLineHook()
```

FUNCTION
This function can be used to enable Hollywood's line hook. The line hook is a function which is automatically called by Hollywood after every code line. It is responsible for controlling several tasks that Hollywood needs for its housekeeping, for example event handling and video playback. Thus, it is very important that the line hook gets called several times per second at least, otherwise your application will become unresponsive and video playback will stutter. Disabling the line hook for a brief period of time,

however, can increase the raw performance of Hollywood's virtual machine significantly since all the code in the linehook isn't executed at all. Hence, you may want to disable it temporarily if you need to do some complex calculations as fast as possible. But make sure to enable the line hook again as soon as possible to prevent your application from becoming unresponsive.

To disable the line hook, you will have to call `DisableLineHook()`.

INPUTS

none

52.7 End

NAME

End – terminate Hollywood

SYNOPSIS

```
End([code])
```

FUNCTION

This function will immediately terminate your script, shut down Hollywood, free all memory, and close all libraries.

Starting with Hollywood 7.0 you can pass the optional `code` parameter to set the return code that Hollywood's process should pass to its parent process. This is useful when running Hollywood programs from a console.

INPUTS

`code`	optional: return code to pass to parent process (defaults to 0) (V7.0)

52.8 ENDIF

NAME

ENDIF – declare end of conditional block (V7.0)

SYNOPSIS

```
@ENDIF
```

FUNCTION

This preprocessor command signals the end of a conditional block that was previously started by an `@IF` preprocessor command.

See Section 52.17 [IF], page 1080, for details and an example.

INPUTS

none

EXAMPLE

See Section 52.17 [IF], page 1080.

52.9 GCInfo

NAME

GCInfo – query garbage collector status (V2.0)

SYNOPSIS

```
count, threshold = GCInfo()
```

FUNCTION

This function returns information about the current status of the garbage collector. The first return value tells you how many kilobytes of memory is currently occupied by Hollywood's VM whereas the second return value indicates the threshold in kilobytes that should trigger the garbage collector. Whenever memory consumption exceeds the specified threshold, Hollywood will automatically run the garbage collector.

You can also run the garbage collector manually or change the garbage collector threshold by calling the `CollectGarbage()` function.

INPUTS

none

RESULTS

`count` amount of memory in kilobytes currently used by the Hollywood VM

`threshold`
current garbage collector threshold

52.10 GetConstant

NAME

GetConstant – get constant's value from string (V4.5)

SYNOPSIS

```
val = GetConstant(c$)
```

FUNCTION

This function returns the value of a constant that is passed in as a string. It is probably of not much use.

INPUTS

`c$` constant inside string whose value is to be retrieved

RESULTS

`val` value of the constant

EXAMPLE

```
DebugPrint(#HOLLYWOOD, "=", GetConstant("#HOLLYWOOD"))
```

Obtains the constant `#HOLLYWOOD` first via direct specification and then via `GetConstant()`. The result should be the same.

52.11 GetDefaultAdapter

NAME

GetDefaultAdapter – get default adapter for type (V10.0)

SYNOPSIS

```
adapter$ = GetDefaultAdapter(type)
```

FUNCTION

This function gets the default adapter for the type specified by `type` and returns it. The adapter types that can be passed in `type` are described in the manual for the `SetDefaultAdapter()` function. See Section 52.26 [SetDefaultAdapter], page 1091, for details.

Note that if no default adapter has been set using `SetDefaultAdapter()` for a specific type, an empty string will be returned.

INPUTS

`type` adapter type to query

52.12 GetDefaultLoader

NAME

GetDefaultLoader – get default loader for type (V10.0)

SYNOPSIS

```
loader$ = GetDefaultLoader(type)
```

FUNCTION

This function gets the default loader for the type specified by `type` and returns it. The loader types that can be passed in `type` are described in the manual for the `SetDefaultLoader()` function. See Section 52.27 [SetDefaultLoader], page 1092, for details.

Note that if no default loader has been set using `SetDefaultLoader()` for a specific type, an empty string will be returned.

INPUTS

`type` loader type to query

52.13 GetMemoryInfo

NAME

GetMemoryInfo – get memory information

SYNOPSIS

```
space = GetMemoryInfo(type)
```

FUNCTION

This function returns information about the amount of memory in your system. The following constants can be specified as `type`:

`#CHIPMEMORY:`
Returns the amount of chip memory

`#FASTMEMORY:`
Returns the amount of fast memory

INPUTS

`type` one of the constants as listed above

RESULTS

`space` memory space

EXAMPLE

```
chip=GetMemoryInfo(#CHIPMEMORY)
fast=GetMemoryInfo(#FASTMEMORY)
Print("You have", chip, "bytes of chip memory and", fast,
      "bytes of fast memory!")
```

The above code prints out the chip and fast memory.

52.14 GetSystemInfo

NAME

GetSystemInfo – get OS-specific information (V4.5)

SYNOPSIS

```
t = GetSystemInfo()
```

FUNCTION

This function can be used to query certain information from the operating system that Hollywood is currently running on. `GetSystemInfo()` returns a table that contains several fields which are different depending on the operating system Hollywood is currently running on.

The following fields will be initialized in the return table:

`UserHome:`
Path to the user's home directory. Supported on Windows, macOS, and Linux. (V5.3)

`UserName:`
Name of the current user. Supported on Windows, macOS, and Linux. (V5.3)

`ProgramFiles:`
Path to the program files directory on this computer. Supported on Windows since V4.5 and on macOS since V5.3.

`AppData:` Path to the application data folder for the current user on this computer. Supported on Windows since V4.5, on macOS since V5.3, and on iOS since V7.0.

`CommonAppData:`
: Path to the application data folder for all users on this computer. Supported on Windows and macOS. (V6.1)

`LocalAppData:`
: Path to the local, non-roaming application data folder for the current user on this computer. Supported on Windows and macOS. On macOS, this will be set to the same path as `CommonAppData`. (V9.0)

`MyDocuments:`
: Path to the "My documents" folder on this computer. Supported on Windows since V4.5, on macOS since V5.3, and on iOS since V7.0.

`Windows:`
: Path to the Windows directory on this computer. Supported only on Windows. (V4.5)

`SDCard:`
: Path to the external storage device. This is called `SDCard` for legacy reasons because in the early days of Android, the external storage device typically was an SD card. This tag is only supported on Android. Also note that starting from Android 6.0, apps are no longer allowed to read from and write to this folder without explicit user permission. When using the Hollywood Player, it will automatically request such permission from the user on startup. When compiling stand-alone APKs using the Hollywood APK Compiler, though, you have to manually request read and/or write permission for this folder by using the `PermissionRequest()` function. See Section 46.7 [PermissionRequest], page 942, for details. (V5.1)

`ExternalStorage:`
: Path to the external storage folder on this device. Supported only on Android. (V5.1)

`InternalStorage:`
: Path to the internal storage folder on this device. Supported only on Android. (V5.1)

`AppBundle:`
: Path to the application bundle of the current program. Supported only on macOS (V6.1) and iOS (V7.0).

`Preferences:`
: The path that Hollywood uses to store preferences managed using `SavePrefs()`. (V7.1)

`TempFiles:`
: A path that you can write temporary files to. (V7.1)

INPUTS

none

RESULTS

`t` a table containing the fields described above

52.15 GetType

NAME

GetType – examine a variable (V2.0)

SYNOPSIS

```
type = GetType(var)
```

FUNCTION

This function can be used to find out the type of a variable or value. Possible return values are `#NUMBER`, `#STRING`, `#TABLE`, `#FUNCTION`, `#USERDATA`, `#LIGHTUSERDATA`, `#THREAD` and `#NIL`.

This function is often used to find out if a variable is `Nil`. Starting with Hollywood 6.0, however, there is also a new convenience function named `IsNil()` which can also be used to check variables against `Nil`.

See Section 8.1 [Data types], page 97, for details.

INPUTS

`var` variable to examine

EXAMPLE

```
type = GetType("Hello World")
```

This will return `#STRING`.

```
type = GetType({1, 2, 3, 4})
```

This will return `#TABLE`.

```
type = GetType(Function() DebugPrint("Hello") EndFunction)
```

This will return `#FUNCTION`.

52.16 GetVersion

NAME

GetVersion – get information about the Hollywood version in use (V3.0)

SYNOPSIS

```
t = GetVersion()
```

FUNCTION

This function can be used to obtain some information about the Hollywood version currently running your script or applet. You can also retrieve information about the platform that your script is running on.

This function will return a table with the following fields initialized:

`Application:`
: This will be `"Hollywood"` or `"HollywoodPlayer"` depending on which version is in use. If you save your script as an executable, this will always be `"HollywoodPlayer"`.

`Version:` A string containing the version of Hollywood, e.g. `"3.0"`.

`Version_Date:`
A string containing the build date of this Hollywood version.

`Kernel:` A string containing the kernel version of Hollywood, e.g. `"3.0"`.

`Kernel_Date:`
A string containing the build date of the Hollywood kernel.

`Beta:` This field will be set to `True` if a beta version of Hollywood is used, `False` otherwise.

`Platform:`
This is probably the most useful field because it contains the platform on which Hollywood is currently running. This field can be `"AmigaOS3"`, `"MorphOS"`, `"WarpOS"`, `"AmigaOS4"`, `"AROS"`, `"Win32"`, `"MacOS"`, `"Linux"`, `"iOS"`, or `"Android"`. Note that `"Win32"` is also returned for 64-bit Windows. It's called `"Win32"` for historical reasons.

`Demo:` This field will be set to `True` if the user is running a demo version of Hollywood. (V4.71)

`Plugins:` This field will be set to a table that contains information about all plugins that have been loaded by Hollywood for this script. See Section 45.3 [GetPlugins], page 925, for details. (V5.1)

`CPU:` This field will be set to the CPU architecture that Hollywood is currently running on. This field can be `"m68k"`, `"m68k/ppc"` (for WarpOS), `"ppc"`, `"arm"`, `"arm64"`, `"i386"`, or `"x64"`. (V5.2)

`BigEndian:`
This field will be set to `True` if Hollywood is running on a big endian CPU (i.e. 68000 or PowerPC) or `False` for a little endian CPU (x86, x64 and ARM). (V6.0)

INPUTS

none

RESULTS

`t` a table containing information about the Hollywood version in use

EXAMPLE

```
t = GetVersion()
If t.platform = "Win32" Then Error("Sorry, Win32 is not supported yet!")
```

The code above checks on which version we are running and exits with an error if Hollywood is running on Windows.

52.17 IF

NAME

IF – test for condition (V7.0)

SYNOPSIS

`@IF val`

FUNCTION

This preprocessor command tests for the condition specified in `val`. If it is `True`, the preprocessor will continue parsing your script. If `val` is `False`, however, `@IF` will branch to the next `@ELSEIF`, `@ELSE`, or `@ENDIF` statement, allowing you to skip certain portions of the script if certain conditions aren't met.

Note that the condition must be a constant expression since `@IF` operates at preprocessor level, i.e. script variables are not available at that time. You can use numeric constants or Hollywood constants defined using either the Const statement or the `-setconstants` console argument. There are also some inbuilt constants that allow you to test for the platform Hollywood is running on (or compiling for) and the Hollywood version. Please see below for a list.

Also note that the `@IF` and `@ELSEIF` preprocessor commands won't complain if you use a constant that hasn't been declared at all. You won't get a "Constant not found!" error in this case. Instead, undeclared constants will simply evaluate to 0. The reason behind this design is that the platform-specific constants (see below) are only defined when running a Hollywood script on the respective platform or compiling for it. Thus, when testing for a certain platform on another platform, you won't get an error if the platform constant hasn't been defined. Instead, the non-existing constant will simply evaluate to 0.

In contrast to the normal If statement, `@IF` operates at preprocessor level. This means that you can use it to force the preprocessor to take certain routes or ignore certain portions of your code. This isn't possible with the normal If statement at all because once that executes, the preprocessor has already finished its job.

For example, you could force the preprocessor to parse different code depending on which platform Hollywood is running. You can also tell the preprocessor to ignore certain portions of the code. Those portions aren't even checked for syntactical correctness. They are completely skipped, just like comments, so anything could be in those blocks.

Here is an example which uses a different background picture and window title for the individual platforms supported by Hollywood:

```
@IF #HW_AMIGA
   @BGPIC 1, "bg_amiga.png"
   @DISPLAY {Title = "My project (Amiga)"}
@ELSEIF #HW_MACOS
   @BGPIC 1, "bg_macos.png"
   @DISPLAY {Title = "My project (macOS)"}
@ELSEIF #HW_LINUX
   @BGPIC 1, "bg_linux.png"
   @DISPLAY {Title = "My project (Linux)"}
@ELSEIF #HW_WINDOWS
   @BGPIC 1, "bg_windows.png"
   @DISPLAY {Title = "My project (Windows)"}
@ELSE
   @BGPIC 1, "bg_default.png"
```

```
    @DISPLAY {Title = "My project (Default)"}
@ENDIF
```

You wouldn't be able to use the normal If statement for this purpose because, as its very name implies, the preprocessor parses the script before it is run. Hence, if you used the normal If statement all preprocessor commands in the code above would be parsed and executed because the preprocessor command would just ignore runtime instructions like the normal If statement.

As you can see above, there are some inbuilt constants that allow you to test for the platform Hollywood is currently running or compiling for. The following inbuilt constants are available:

`#HW_AMIGA`
: Defined on AmigaOS or compatible platforms, i.e. AmigaOS, MorphOS, AROS, WarpOS.

`#HW_AMIGAOS3`
: Defined on AmigaOS 3.

`#HW_AMIGAOS4`
: Defined on AmigaOS 4.

`#HW_ANDROID`
: Defined on Android.

`#HW_AROS` Defined on AROS.

`#HW_IOS` Defined on iOS.

`#HW_LINUX`
: Defined on Linux.

`#HW_MACOS`
: Defined on macOS.

`#HW_MORPHOS`
: Defined on MorphOS.

`#HW_WARPOS`
: Defined on WarpOS.

`#HW_WINDOWS`
: Defined on Windows.

`#HW_LITTLE_ENDIAN`
: Defined on little endian systems.

`#HW_64BIT`
: Defined on 64-bit systems.

`#HW_VERSION`
: Contains the integer part of Hollywood's version number.

`#HW_REVISION`
: Contains the fractional part of Hollywood's version number.

Note that when compiling your script into an executable, Hollywood will automatically set the constants of the target architecture instead of the build architecture. For example, if you compile your script on Windows for AmigaOS 3, Hollywood will set the constants `#HW_AMIGA` and `#HW_AMIGAOS3`. The constants `#HW_WINDOWS` and `#HW_LITTLE_ENDIAN` won't be set. They will only be set if you compile for Windows or run your script on a Windows machine.

Also note that when compiling applets, none of the architectural constants above will be set. Applets are completely platform-agnostic so if you compile an applet, none of the architectural constants will be set. An exception is if you compile an applet explicitly for Android or iOS by passing `"android"` or `"ios"` to the `-exetype` console argument. In that case, `#HW_ANDROID` and `#HW_IOS`, respectively, will indeed be set. If you compile a platform-independent applet by passing `"applet"` to the `-exetype` argument, though, none of the architectural constants will be set.

INPUTS

`val` condition to test; must be a constant expression

EXAMPLE

```
@IF #HW_VERSION >= 7
   ...
@ENDIF
```

The code above tells the preprocessor only to parse the following code if we have at least version 7 of Hollywood.

52.18 IIf

NAME

IIf – returns value depending on the condition (V2.0)

SYNOPSIS

```
ret = IIf(expr, true_expr, false_expr)
```

FUNCTION

This function checks if the expression `expr` is `True` (non-zero). If it is, the function returns the expression specified in `true_expr`, else `false_expr` is returned. It is important to note that both true and false expressions are evaluated in every case, no matter if the expression is `True` or `False`. So things like `IIf(a <> 0, 5 / a, 0)` will not work for `a=0` because Hollywood will try evaluating `5/0` which obviously does not work.

INPUTS

`expr` source expression

`true_expr`
expression to return if expr is `True`

`false_expr`
expression to return if expr is `False`

RESULTS

`ret` result

52.19 INCLUDE

NAME

INCLUDE – import code from external script files or applets (V2.0)

SYNOPSIS

```
@INCLUDE file$
```

FUNCTION

This preprocessor command will import all code from the file specified by `file$` into the current project. The code will be inserted into the current project at the position where `@INCLUDE` is defined. Normally, you will want to use `@INCLUDE` at the beginning of your project.

This preprocessor command is useful if you have larger projects and want to spread them over multiple files. The included files usually contain functions that you can call then from your main script. Included files can also contain preprocessor commands.

Starting from Hollywood 4.0, you can also include Hollywood applets using this command. This is useful for importing code from libraries.

See Section 7.6 [Include files], page 88, for details.

INPUTS

`file$` script file to include

EXAMPLE

```
;---File: script2.hws---
Function p_Print(t$)
Print(t$)
EndFunction
;EOF script2.hws

;---File: mainscript.hws
@INCLUDE "script2.hws"

p_Print("Hello World!")
WaitLeftMouse
End
;EOF mainscript.hws
```

The code above consists of two scripts: script2.hws contains the function p_Print() which simply calls Hollywood's `Print()` function. script2.hws is then included in the main script which also calls the p_Print() function then.

52.20 IsNil

NAME

IsNil – check if a variable is `Nil` (V6.0)

SYNOPSIS

```
bool = IsNil(var)
```

FUNCTION

This function checks if the specified variable is `Nil`. Along with `GetType()` `IsNil()` is the only reliable way to find out if a variable is `Nil` or not. Checking the variable against the `Nil` identifier is not reliable because this will also result in `True` if the variable is zero instead of `Nil`. Example:

```
a = 0
b = Nil
DebugPrint(IsNil(a), a = Nil)  ; prints "0 1"
DebugPrint(IsNil(b), b = Nil)  ; prints "1 1"
```

You see that "a = Nil" returns `True` although `a` is zero and not `Nil`. That is because `Nil` is always regarded as zero when used in expressions. Thus, if you want to find out whether a variable really is `Nil`, always use `IsNil()` or `GetType()`.

See Section 8.1 [Data types], page 97, for details.

INPUTS

`var`	variable to examine

RESULTS

`bool`	`True` if the variable is `Nil`, `False` otherwise

52.21 IsUnicode

NAME

IsUnicode – determine if Hollywood is in Unicode mode (V7.0)

SYNOPSIS

```
bool = IsUnicode()
```

FUNCTION

This function returns `True` if Hollywood is currently in Unicode mode, `False` otherwise. Since scripts should always run in Unicode mode, this function is probably of not much use.

INPUTS

none

RESULTS

`bool`	`True` if Hollywood is in Unicode mode, `False` otherwise

52.22 LegacyControl

NAME

LegacyControl – enable or disable certain legacy features (V6.0)

SYNOPSIS

```
LegacyControl(feature$, enable)
```

FUNCTION

This function can be used to enable or disable certain legacy functionality that Hollywood still supports for compatibility reasons. You have to pass the name of the feature that you want to address as well as `True` to enable this feature or `False` to disable it.

The following strings can currently be passed in `feature$`:

`SingleMusic:`
: Before Hollywood 6.0 only one music could be playing at a time. This limitation has been removed with Hollywood 6.0 but by default, PlayMusic() will still stop any playing music in order to be fully compatible with previous versions. If you don't want that, you have to call `LegacyControl()` and set `SingleMusic` to `False`. Hollywood will then be able to play multiple music objects at the same time. This tag defaults to `True`.

`LineBasedShapes:`
: Before Hollywood 6.0 all round shapes drawn by the functions `Arc()`, `Circle()`, `Ellipse()` and `Box()` (when using the optional parameter to draw a box with round corners) were drawn as polygons which made them look rather square. Starting with Hollywood 6.0 these round shapes are now drawn as real Bézier splines if antialiasing is turned on. This will be a little slower than the polyline-based approach from previous versions, but it will look much better. If you want Hollywood to keep using the polygon-based approach from previous versions, you can set this tag to `True`. In that case, the functions listed above will draw shapes that look exactly the same as the ones drawn by previous versions. This tag defaults to `False` except when running an applet compiled by Hollywood versions older than 6.0. In that case it defaults to `True`.

INPUTS

`feature$` name of the feature to enable or disable (see above)

`enable` `True` to enable the feature, `False` to disable it

52.23 LINKER

NAME

LINKER – pass options to linker (V8.0)

SYNOPSIS

```
@LINKER table
```

FUNCTION

This preprocessor command can be used to pass options to Hollywood's linker. Thus, it is only used when compiling scripts to applets or executables. When just running scripts with the Hollywood interpreter it is ignored.

You have to pass a table to the `@LINKER` preprocessor command. The following tags are currently recognized:

`Files:` This tag allows you to pass a list of files that shall be linked to the applet or executable to the linker. This is similar to the `-linkfiles` console argument,

but it allows you to store the files to be linked directly in your script which might be more convenient than having to maintain an external database file for this purpose. You need to pass a table containing a list of files to be linked to this tag. Note that it is very important that the file name specifications passed to `Files` must be exactly identical to the ones passed to Hollywood functions which should then load the linked files instead. If you don't use identical file paths, Hollywood won't be able to map the files linked to the applet or executable to the corresponding commands. See Section 4.3 [Linking data files], page 58, for details.

`Fonts:` This tag allows you to pass a list of fonts that shall be linked to the applet or executable to the linker. This is similar to the `-linkfonts` console argument, but it allows you to store the fonts to be linked directly in your script which might be more convenient than having to maintain an external database file for this purpose. Your script will then automatically load the linked fonts from your applet or executable when you call `SetFont()`. Using the `Fonts` tag of the `@LINKER` preprocessor command to link fonts into applets or executables is an alternative to using the `@FONT` preprocessor command. Normally, however, using `@FONT` should be much easier than using `Fonts` so you should use `Fonts` only with good reasons. See Section 4.4 [Linking fonts], page 60, for details.

INPUTS

`table` table containing options to pass to the linker

EXAMPLE

```
@LINKER {Files = {"test.jpg", "title.png"},
    {Fonts = {"Arial", "Times New Roman"}}

LoadBGPic(1, "test.jpg")
LoadBrush(1, "title.png")
SetFont("Arial", 36)
NPrint("Hello World")
SetFont("Times New Roman", 72)
NPrint("Hello Hollywood")
```

The code above links the files `test.jpg` and `title.png` to the applet or executable. `LoadBGPic()` and `LoadBrush()` will then the load the files directly from the applet or the executable instead of an external source. Additionally, the fonts `Arial` and `Times New Roman` will be linked to the applet or executable, and `SetFont()` will also open those fonts directly from the applet or executable then.

52.24 OpenURL

NAME

OpenURL – open URL in default web browser (V4.5)

SYNOPSIS

```
OpenURL(url$)
```

FUNCTION

This function opens the specified in the default web browser.

INPUTS

`url$` URL to open

EXAMPLE

```
OpenURL("http://www.airsoftsoftwair.com/")
```

The code above takes you directly to the Airsoft Softwair headquarters.

52.25 OPTIONS

NAME

OPTIONS – configure miscellaneous options (V4.5)

SYNOPSIS

```
@OPTIONS table
```

FUNCTION

This preprocessor command allows you configure miscellaneous general options. You have to pass a table to this command that specifies which things you want to configure.

The following tags are currently recognized by `@OPTIONS`:

`LockSettings:`
: This tag has the same function as the console argument with the same name. If you set `LockSettings` to `True`, compiled Hollywood programs will not accept any arguments from the console or from the pseudo-console. The only difference to the console argument is that if you use `LockSettings` in the preprocessor commands, it will also forbid any user changes when running Hollywood scripts. I.e. if you use `LockSettings` here, your script will always use the style as defined in the preprocessor commands. You cannot change the style by passing arguments like `Borderless` or `Sizeable` to the script.

`SoftTimer:`
: If you set this tag to `True`, Hollywood will use a low resolution software timer instead of the high resolution hardware timer. This is sometimes necessary because with certain older Windows XP hardware, the timer may occassionally leap which can cause unexpected behaviour. This tag is only supported on the Windows platform. (V5.3)

`NoCommodity:`
: If you set this tag to `True`, Hollywood will not add itself to the system's list of commodities on AmigaOS. This tag is only supported on AmigaOS and compatibles. Defaults to `False`. (V6.0)

`RegisterApplication:`
: If you set this tag to `True`, Hollywood will register itself as an AmigaOS 4 application on startup through application.library. This is necessary if you want to call functions that deal with AmigaOS 4 application objects like

`SendApplicationMessage()` or if you want your application to appear in AmiDock. To change the icon that is shown in AmiDock, use the `@APPICON` preprocessor command. This tag is only available on AmigaOS 4. Defaults to `False`. (V6.0)

`DockyContextMenu:`
This tag allows you to specify the identifier of a menu strip that should be used as a context menu for your application's docky in AmigaOS 4's AmiDock system. The menu strip you specify here must only contain a single menu tree without any sub menus. Hotkeys inside the menu strip are not supported either because they don't make sense in a context menu that isn't always visible. Please note that setting this tag will automatically make your application appear as an app docky in AmiDock. See Section 16.1 [AmiDock information], page 165, for details on the difference between app and standard dockies. This tag is only recognized if `RegisterApplication` has been set to `True` and it is obviously only supported on AmigaOS 4. (V6.0)

`DockyBrush:`
This tag allows you to specify the identifier of a brush that should be shown as your application's icon in AmiDock on AmigaOS 4. Normally, you would use the `@APPICON` preprocessor command to configure your application's AmiDock icon but the `DockyBrush` tag can come in handy in one of the following situations: First, `DockyBrush` allows you to specify an arbitrary brush and thus you are not restricted to the predefined sizes made available by `@APPICON`. Instead, your application's docky icon can be of any size you want. Second, if you use `DockyBrush` Hollywood will automatically create an app docky for you whereas using `@APPICON` would create a standard docky (as long as your docky does not have a context menu attached). See Section 16.1 [AmiDock information], page 165, for more details on the difference between app and standard dockies. This tag is only recognized if `RegisterApplication` has been set to `True` and it is obviously only supported on AmigaOS 4. (V6.0)

`NoDocky:` If this tag is set to `True`, Hollywood will not show your application in AmiDock. This tag is useful if you would like to have an invisible application that can use all the application functionality like the message mechanism and Ringhio but doesn't appear in AmiDock. This tag is only recognized if `RegisterApplication` has been set to `True` and it is obviously only supported on AmigaOS 4. (V6.0)

`Encoding:`
This tag can be used to set the script's character encoding. Note that you have to put this statement at the very beginning of your script or there will be problems. The following character encodings are currently supported:

`#ENCODING_UTF8:`
Script's character encoding is UTF-8 (with or without BOM). This is also the default and should be used whenever and whereever possible.

`#ENCODING_ISO8859_1`:
: Script's character encoding is ISO 8859-1. Note that due to historical reasons Hollywood will not use ISO 8859-1 character encoding on AmigaOS and compatibles but whatever is the system's default character encoding. `#ENCODING_ISO8859_1` will put Hollywood in legacy mode and should make your script fully compatible with Hollywood versions older than 7.0. However, since ISO 8859-1 mode has several drawbacks, it isn't recommended to use this legacy mode permanently. Instead, you should adapt your scripts to work correctly in Unicode mode.

Note that it isn't recommended to use `#ENCODING_ISO8859_1` because Hollywood will only run correctly on locales compatible with Western European languages then. You should always use `#ENCODING_UTF8` because this will put Hollywood in Unicode mode and make sure that Hollywood runs correctly on all locales. Since `#ENCODING_UTF8` is also the default, you normally don't have to use the `Encoding` tag at all.

The encoding you specify here is automatically set as the default encoding for both the text and string library using `SetDefaultEncoding()`. This means that all functions of the string and text libraries will default to this encoding. (V7.0)

`NoChDir`: By default, Hollywood will always change the current directory to the directory of the script or applet it is currently running. Pass this argument if you don't want this behaviour. In that case, Hollywood won't change the current directory when running a script. (V7.1)

`EnableDebug`:
: If this tag is set to `False`, the commands `DebugPrint()`, `DebugPrintNR()`, `Assert()`, `DebugOutput()` and `@WARNING` will be ignored. This allows you to globally disable debugging functions with just a single call. When compiling scripts Hollywood will set `EnableDebug` to `False` by default. This is the recommended setting because it will prevent people from reverse engineering your projects because they won't be able to activate debug output by specifying the `-debugoutput` console argument. When running scripts, `EnableDebug` defaults to `True` to allow you to debug your scripts. (V7.1)

`GlobalPlugins`:
: On AmigaOS and compatibles, plugins can also be globally installed in `LIBS:Hollywood`. Executables compiled by Hollywood, however, will only load the plugins that are stored next to the executable in its directory. If you want your executable to load all plugins in `LIBS:Hollywood` as well, you have to set the `GlobalPlugins` tag to `True`. Obviously, this tag is only supported on AmigaOS and compatible platforms. (V9.0)

`DPIAware`:
: This tag is only supported on Windows. If you set it to `True`, Hollywood will start in DPI-aware mode. This means that it will not ask the OS to automatically scale Hollywood to fit to the monitor's DPI. If `DPIAware` is set to `False` (which is also the default), Hollywood will automatically apply

scaling on high-DPI monitors so that its display doesn't appear too small on them. For example, a display of 640x480 pixels will appear really tiny on a high-DPI monitor. By automatically adapting displays to the monitor's DPI, Hollywood will try to avoid this. However, that scaling can make displays appear blurry on high-DPI monitors. So if you don't want that, set `DPIAware` to `True`. Note, however, that you'll need to take care of making sure that your display appears correctly on high-DPI monitors then. You can do this by setting the `SystemScale` tag in the `@DISPLAY` preprocessor command, for example. Note that `DPIAware` is only supported on Windows. On all other platforms Hollywood is always DPI-aware. (V9.0)

`ConsoleMode:`
: If you set this tag to `True`, Hollywood will compile an executable that runs in console mode on Windows. On Windows, there is a distinction between console and Windows programs so if you want to compile a program for the console, you will explicitly have to tell Hollywood to do so. You can do that by setting this tag to `True`. Note that this tag is obviously only handled when compiling executables for Windows with Hollywood. Otherwise it is simply ignored. Defaults to `False`. (V9.0)

`Quiet:`
: If you set this tag to `True`, Hollywood won't show its traditional startup output but will start quietly. When running Hollywood applets that have `Quiet` set to `True` using the Hollywood Player for Amiga and compatibles, the player also won't open its startup window. Defaults to `False`. (V9.0)

INPUTS

`table`
: table specifying desired options (see above)

52.26 SetDefaultAdapter

NAME

SetDefaultAdapter – set default adapter (V10.0)

SYNOPSIS

```
SetDefaultAdapter(type, adapter$)
```

FUNCTION

This function can be used to set the default adapter for the type specified by `type` to the adapter specified by `adapter$`. All Hollywood functions that support adapters of the specified type will then default to the adapter set using this function.

The following adapter types are currently supported by Hollywood and can be passed in `type`:

`#ADAPTER_FILE:`
: File, directory, and filesystem adapters. They are used to open files, directories and handle filesystem operation. They can be passed to all functions that deal with files.

`#ADAPTER_NETWORK:`
: Network adapters. They can be used to replace Hollywood's inbuilt network handler. Functions that support network adapters include `OpenConnection()` and `DownloadFile()`.

`#ADAPTER_SERIALIZE:`
: Serializer adapters. They are used to serialize and deserialize data. Functions that support serializer adapters are `SerializeTable()` and `DeserializeTable()` among others.

The default adapter for all the adapter types listed above is `default` which means that plugin adapters will always be asked before Hollywood's inbuilt handlers.

The string you pass in `adapter$` can contain one or more adapters. If several adapters are specified, they must be separated by `|` characters. See Section 7.9 [Loaders and adapters], page 92, for details.

INPUTS

`type` adapter type to use (see above for possible values)

`adapter$` new default for the specified adapter

EXAMPLE

```
SetDefaultAdapter(#ADAPTER_FILE, "inbuilt")
OpenFile(1, "test.txt")
```

The code above will set the default file adapter to `inbuilt`. This means that all Hollywood functions that deal with files will no longer support any plugin file adapters because you've told Hollywood to only use the inbuilt file adapter.

52.27 SetDefaultLoader

NAME

SetDefaultLoader – set default loader (V10.0)

SYNOPSIS

```
SetDefaultLoader(type, loader$)
```

FUNCTION

This function can be used to set the default loader for the type specified by `type` to the loader specified by `loader$`. All Hollywood functions that support loaders of the specified type will then default to the loader set using this function.

The following loader types are currently supported by Hollywood and can be passed in `type`:

`#LOADER_IMAGE:`
: Image loaders, used for example by functions like `LoadBrush()`.

`#LOADER_ANIM:`
: Animation loaders, used for example by functions like `OpenAnim()`.

`#LOADER_SOUND:`
: Sound loaders, used for example by functions like `OpenMusic()`.

`#LOADER_VIDEO:`
: Video loaders, used for example by functions like `OpenVideo()`.

`#LOADER_ICON:`
: Icon loaders, used for example by functions like `LoadIcon()`.

`#LOADER_FONT:`
: Font loaders, used for example by functions like `OpenFont()`.

The default loader for all the loader types listed above is `default` which means that Hollywood will first ask plugin loaders, then inbuilt loaders, then native loaders to open a file. An exception is `#LOADER_FONT` where, due to compatibility reasons, only native loaders will be asked to open the font.

The string you pass in `loader$` can contain one or more loaders. If several loaders are specified, they must be separated by `|` characters. See Section 7.9 [Loaders and adapters], page 92, for details.

INPUTS

`type` loader type to use (see above for possible values)

`loader$` new default for the specified loader

EXAMPLE

```
SetDefaultLoader(#LOADER_IMAGE, "plugin")
LoadBrush(1, "test.jpg")
```

The code above will set the default image loader to `plugin`. This means that functions like `LoadBrush()` will no longer use Hollywood's inbuilt image loader or the image loader provided by the OS. Only plugin image loaders will be asked to load image files.

52.28 SetVarType

NAME

SetVarType – declare a variable / OBSOLETE

SYNOPSIS

```
SetVarType(var, type[, arraysize])
```

FUNCTION

As of Hollywood 2.0, this function is obsolete and only included for compatibility reasons.

This function can be used to declare a variable. You only have to declare variables that are different from the default type (`#LONG`) and which do not have an identifier in their name like the `"$"` for strings or the `"!"` for floats. Arrays always have to be declared by specifying the optional argument arraysize.

INPUTS

`var` variable name to declare

`type` desired type (can be `#LONG`, `#FLOAT`, `#STRING`)

`arraysize`
: optional: if you specify this argument, an array of the specified size will be allocated for you instead of a single variable

52.29 ShowNotification

NAME

ShowNotification – show system notification (V8.0)

SYNOPSIS

```
ShowNotification(title$, msg$[, table])
```

FUNCTION

This function can be used to pop up a system notification box that shows the message passed in `msg$` to the user. The notification box will use the title specified in `title$`. If you pass an empty string (`""`) in the `title$` parameter, the notification will use the title specified in the `@APPTITLE` preprocessor command.

The optional table argument can be used to configure further parameters for the system notification. The following tags are currently recognized by the `table` argument:

`Icon:` This tag can be used to set the icon that should be shown inside the notification box if the host system supports it. The following icon types can currently be specified here:

`#REQICON_NONE:`
: No icon

`#REQICON_INFORMATION:`
: An information sign

`#REQICON_ERROR:`
: An error sign

`#REQICON_WARNING:`
: A warning sign

`#REQICON_QUESTION:`
: A question mark

Note that not all platforms support this tag.

`Timeout:` This tag can be used to specify a timeout in milliseconds for the notification. If this tag is set, the notification will disappear after the specified time (in milliseconds) has elapsed. Otherwise the notification will use the host system's default timeout value for notifications. You can also pass one of the following special constants here:

`#DURATION_SHORT:`
: Use this for a short notification.

`#DURATION_LONG:`
: Use this for a long notification.

Note that on Android, you must pass one of the special constants listed above. Passing a millisecond value directly isn't supported on Android. Also note that not all platforms support the `Timeout` tag.

`X:` Desired horizontal position for the notification. This can also be one of Hollywood's special coordinate constants like `#CENTER`. This tag is only supported on Android and iOS.

`Y:` Desired vertical position for the notification. This can also be one of Hollywood's special coordinate constants like `#CENTER`. This tag is only supported on Android and iOS.

`NoSound:` Set this tag to `True` if you don't want any sound to accompany the notification. This tag is currently only supported on macOS.

Note that on Windows notifications can only be shown if your application has a tray icon. Thus, `ShowNotification()` will automatically add a tray icon for your application if there is none yet. If you would like to manually install a tray icon for your application on Windows, call `SetTrayIcon()` before calling `ShowNotification()`. See Section 31.12 [SetTrayIcon], page 636, for details.

Also note that on AmigaOS 4 `ShowNotification()` is implemented through application.library's Ringhio system and thus can only be used if your script has registered itself as an AmigaOS 4 application using the `RegisterApplication` tag in `@OPTIONS`. If you need fine-tuned control over Ringhio messages and you don't need to be portable, you can also use the `ShowRinghioMessage()` function on AmigaOS 4 instead because this offers even more configuration options. See Section 16.13 [ShowRinghioMessage], page 173, for details.

On MorphOS `ShowNotification()` uses MagicBeacon's notification system. On AmigaOS 3 Ranchero is used for showing notifications. Note that since apps need to register before they can show notifications with Ranchero it's recommended that if you intend to use `ShowNotification()` on AmigaOS 3 with Ranchero you should also use `@APPIDENTIFIER` to specify a unique ID for your app. Otherwise the generic app name `"Hollywood"` will be used.

INPUTS

`title$` desired title for the notification or empty string for default title

`msg$` desired message text for the notification

`table` optional: table containing further configuration parameters (see above)

52.30 Sleep

NAME

Sleep – halt script execution for a certain amount of time (V10.0)

SYNOPSIS

```
Sleep(time)
```

FUNCTION

This function halts the script execution for the time interval specified in `time`. This time interval must be specified in milliseconds.

`Sleep()` does the same as `Wait()` except that `Sleep()` always operates in milliseconds whereas `Wait()` uses ticks by default.

INPUTS

`time` time interval in milliseconds

EXAMPLE

```
Sleep(4000)
```

The code above halts the script execution for 4 seconds.

52.31 VERSION

NAME

VERSION – define which Hollywood version is required (V2.0)

SYNOPSIS

```
@VERSION version, revision
```

FUNCTION

This preprocessor command allows you to define the version of Hollywood that your script needs to run. You should always use this preprocessor command as the first thing in your script.

INPUTS

`version` required Hollywood version

`revision` required Hollywood revision

EXAMPLE

```
@VERSION 2,0
```

Defines that this script requires at least Hollywood 2.0.

52.32 Wait

NAME

Wait – wait for a certain amount of time

SYNOPSIS

```
Wait(time[, unit])
```

FUNCTION

This function waits for the time specified by `time` and then continues the script execution. The default unit for `time` is ticks. A tick is 1/50 second. So if you want to wait one second, you will have to set ticks to 50.

Starting with Hollywood 1.9 you can specify different units to make the wait more precise. The following unit types are possible:

`#MILLISECONDS:`
Time is in milliseconds (1/1000 second)

`#SECONDS:`
Time is in seconds

`#TICKS:` Time is in ticks (default) (1/50 second)

You can also use the `Sleep()` function which will do the same as `Wait()` but operates in milliseconds so you save some typing.

INPUTS

`time` time to wait

`unit` optional: specify a different time unit as listed above (defaults to `#TICKS`) (V1.9)

EXAMPLE

```
Wait(200)
```

Wait for 4 seconds.

53 Table library

53.1 Concat

NAME

Concat – concatenate table strings into a single string (V5.0)

SYNOPSIS

```
s$ = Concat(table[, sep$, start, end])
```

FUNCTION

This function can be used to concatenate the strings from table indices `start` to `end` to a single string that is optionally delimited by the string specified in `sep$`. If `start` and `end` are not specified, they default to 0 and number of table items minus one respectively, which means that by default all table strings are concatenated. If `sep$` is not specified, it will default to the empty string which means that no separator will be put between the strings.

Please note that this function will only take strings at integer indices into account. Strings at non-integer indices will not be concatenated.

INPUTS

`table`	table that should be used as the source
`sep$`	optional: separator string to use (defaults to `""`)
`start`	optional: table index of first string to concatenate (defaults to 0, which means start at index 0 of the table)
`end`	optional: table index of the last string to concatenate (defaults to number of table items minus one, which means end at the end of the table)

EXAMPLE

```
t = {"Hello", "this", "is", "a", "test!"}
DebugPrint(Concat(t, " "))
```

The code above concatenates all strings of the table and separates them by inserting a space character.

53.2 CopyTable

NAME

CopyTable – make independent copy of a table (V4.6)

SYNOPSIS

```
t = CopyTable(src[, shallow])
```

FUNCTION

This function can be used to make an independent copy of the specified source table. As you have probably noticed, when assigning a table to a new variable using the equal (=) operator, only a reference to the table will be assigned to the new variable. This is done

due to efficiency reasons because making complete copies of the table is not necessary in most cases. In some cases, however, you need to have a fully independent copy of the table. This can be done using this function.

Starting with Hollywood 6.0 this function accepts an optional argument named `shallow`. If you set this argument to `True`, `CopyTable()` will do a shallow copy of the table which means that instead of making an independent copy, all sub-tables will only be copied by reference. A copy by reference means that if the source table is modified, all copies by reference will be modified as well. Shallow copies of a table have the advantage that they save resources and they can also come in handy in case of self-referential tables which would lead to a stack overflow during a deep copy.

See Section 8.4 [Tables], page 100, for details.

INPUTS

`src`	table to copy
`shallow`	optional: specifies whether or not a shallow copy of the table shall be made (defaults to `False`) (V6.0)

RESULTS

`t`	deep or shallow copy of table

EXAMPLE

```
t1 = {1, 2, 3, 4, 5}
t2 = t1
t2[0] = 10
DebugPrint(t1[0]) ; -> prints 10 because t2 is only a reference to t1
t3 = CopyTable(t1)
t3[0] = 20
DebugPrint(t1[0]) ; -> prints 10 now!
```

This code demonstrates first the copy-by-reference default behaviour of Hollywood which only creates a reference to an existing table. Afterwards, a deep copy is made using `CopyTable()`.

53.3 CreateList

NAME

CreateList – create optimized list (V9.0)

SYNOPSIS

```
list = CreateList()
```

FUNCTION

This function creates an optimized list and returns it as an empty table.

The advantage when using optimized lists instead of normal Hollywood tables with functions like `InsertItem()`, `RemoveItem()`, `ListItems()` and `GetItem()` is that all those functions will be much faster.

The disadvantage is that adding or removing items may only be done via `InsertItem()` and `RemoveItem()`. You must not add or remove items from optimized lists by modifying the table directly. It's necessary to use the functions mentioned above.

To convert an existing Hollywood table to an optimized list, you can use the `SetListItems()` function. See Section 53.20 [SetListItems], page 1112, for details.

INPUTS

none

RESULTS

`list` an empty optimized list

EXAMPLE

```
t = CreateList()
;t = {}
StartTimer(1)
For Local k = 1 To 10000
   InsertItem(t, k)
Next
NPrint(ListItems(t))
NPrint("This took", GetTimer(1), "ms")
```

The code above creates an empty optimized list, adds 10000 items to it and prints the time this took. Disable the first line and uncomment the second line to see how much faster optimized lists are in comparison to normal Hollywood tables.

53.4 ForEach

NAME

ForEach – iterate over all elements of a table (V5.0)

SYNOPSIS

```
[v] = ForEach(table, func[, userdata])
```

FUNCTION

This function can be used to iterate over all elements of the table specified in the first argument. For each table element this command will call the user function specified in `func`. The user function will receive two arguments: The first argument will contain the index of the table element, whereas the second argument will contain the value at that index. If the user function returns a value the loop is broken, and this value is returned as the result from `ForEach()`.

Please note that this function will traverse the whole table. If you would just like to iterate over integer indices, use the `ForEachI()` command instead. See Section 53.5 [ForEachI], page 1102, for details.

Starting with Hollywood 6.1 this function accepts an optional `userdata` parameter. The value you pass here will be forwarded to your callback as the third function parameter. The value can be of any type.

INPUTS

`table` table that should be traversed

`func` user function to call for each table element

`userdata` optional: user data to pass to callback function (V6.1)

RESULTS

`v` optional: return value if iteration is broken by user function (see above)

EXAMPLE

```
t = {1, 2, 3, 4, Test$="Hello", Value=9.2}
ForEach(t, DebugPrint)
```

The code above dumps the contents of table 't' using `ForEach()`.

53.5 ForEachI

NAME

ForEachI – iterate over all integer indices of a table (V5.0)

SYNOPSIS

```
[v] = ForEachI(table, func[, userdata])
```

FUNCTION

This function can be used to iterate over all integer indices of the table specified in the first argument. For each table element at an integer index this command will call the user function specified in `func`. The user function will receive two arguments: The first argument will contain the index of the table element, whereas the second argument will contain the value at that index. If the user function returns a value the loop is broken, and this value is returned as the result from `ForEachI()`.

Please note that this function will traverse only integer indices. If you would like to iterate over the whole table, use the `ForEach()` command instead. See Section 53.4 [ForEach], page 1101, for details.

Starting with Hollywood 6.1 this function accepts an optional `userdata` parameter. The value you pass here will be forwarded to your callback as the third function parameter. The value can be of any type.

INPUTS

`table` table that should be traversed

`func` user function to call for each integer index

`userdata` optional: user data to pass to callback function (V6.1)

RESULTS

`v` optional: return value if iteration is broken by user function (see above)

EXAMPLE

```
t = {1, 2, 3, 4, Test$="Hello", Value=9.2}
ForEachI(t, DebugPrint)
```

The code above dumps only the integer indices of table 't'. This means that the indices `Test$` and `Value` won't be respected.

53.6 GetItem

NAME

GetItem – get list item (V9.0)

SYNOPSIS

```
item = GetItem(list, idx)
```

FUNCTION

This returns the item at index `idx` in the list specified by `list`. If `idx` is -1, the last list item will be returned. If `idx` is out of range, `Nil` will be returned.

Note that you normally shouldn't use this function because you can access list elements quicker by just using the []-operator. The only case where using this function has a real speed advantage is when you want to get the last item of an optimized list and you don't know the number of items in the list. In that case, using `GetItem()` is faster than calling `ListItems()` first.

INPUTS

`list`	list to use
`idx`	index of the list item to get; pass -1 to get the last item

RESULTS

`item`	item at the specified index

53.7 GetMetaTable

NAME

GetMetaTable – retrieve a table's metatable (V2.0)

SYNOPSIS

```
mt = GetMetaTable(t)
```

FUNCTION

This function retrieves the metatable of the specified table and returns it. If the specified table does not have a metatable, `Nil` is returned. See Section 9.8 [Metamethods], page 110, for more information on metatables and metamethods.

INPUTS

`t`	table whose metatable you want to retrieve

RESULTS

`mt`	metatable of the specified table or `Nil` if table does not have a metatable

53.8 HaveItem

NAME

HaveItem – check if a table item exists (V6.0)

SYNOPSIS

```
bool = HaveItem(t, key)
bool = HasItem(t, key)
```

FUNCTION

This function checks whether the specified table has an item at index `key` or not. If there is an item at this index, `HaveItem()` will return `True`. This is a convenience function for `RawGet()`.

Note that if you pass a string in the `key` parameter, it will be converted to lower case automatically. If you don't want that, use `RawGet()` instead.

Starting with Hollywood 9.1, this function has a synonym called `HasItem()` which does the same but is more grammatical.

INPUTS

`t`	table to query
`key`	key to check

RESULTS

`bool`	`True` or `False` depending on whether the item exists

EXAMPLE

```
t = {x = 10, y = 20}
NPrint(HaveItem(t, "x"), HaveItem(t, "y"), HaveItem(t, "z"))
```

The code above will print 1 / 1 / 0 (= `True`, `True`, `False`).

53.9 InsertItem

NAME

InsertItem – insert item into a list (V2.0)

SYNOPSIS

```
InsertItem(list, item[, pos])
```

FUNCTION

This function inserts the specified item into the list specified by `list`. `item` can be of any type. If you do not specify the optional argument pos, the item will be appended to the end of the list. If you specify the `pos` argument, the item will be inserted at this position and all succeeding items will be moved one position up. Position counter starts at 0 which is the first element.

Note that this function is rather slow when used with normal Hollywood tables. To accelerate `InsertItem()`, you have to use it with optimized lists created by `CreateList()`. See Section 53.3 [CreateList], page 1100, for details.

INPUTS

`list`	table where to insert the element
`item`	item to insert (can be of any type)
`pos`	optional: where to insert (defaults to -1 which means at the end of the list)

EXAMPLE

```
a = {1, 2, 3, 4, 5, 7, 8, 9, 10}
InsertItem(a, 6, 5)
For k = 1 To ListItems(a) Do Print(a[k - 1] .. " ")
```

Prints "1 2 3 4 5 6 7 8 9 10". The item "6" is inserted at position 5 so that the row is complete.

53.10 IPairs

NAME

IPairs – traverse over all integer keys of a table (V2.0)

SYNOPSIS

```
func, state, val = IPairs(table)
```

FUNCTION

This function can be used in conjunction with the generic `For` statement to traverse over all integer keys of a table. As required by the generic `For` statement, `IPairs()` will return three values: An iterator function, a private state information, and an initial value for the traversal. The iterator function returned by `IPairs()` will stop the traversal when it encounters a key whose value is set to `Nil`.

If you want to traverse over all fields of a table instead of just the integer indices, use the `Pairs()` function instead.

See Section 11.4 [Generic For statement], page 127, for details.

INPUTS

`table`	table to traverse

RESULTS

`func`	iterator function
`state`	private state information
`val`	initial traversal value

EXAMPLE

```
a = {"one", "two", "three"}
For i, v In IPairs(a)
  DebugPrint(i, v)
Next
```

The code above will print "0 one", "1 two" and "2 three".

53.11 IsTableEmpty

NAME

IsTableEmpty – check if table is empty (V6.1)

SYNOPSIS

```
b = IsTableEmpty(t)
```

FUNCTION

This function checks whether the specified table is empty and returns `True` or `False` respectively.

INPUTS

`t` table to check

RESULTS

`b` `True` or `False`

EXAMPLE

```
Print(IsTableEmpty({}))
Print(IsTableEmpty({0}))
```

The first call will print `"1"` (true), the second `"0"` (false).

53.12 ListItems

NAME

ListItems – count items of a list (V2.0)

SYNOPSIS

```
c = ListItems(list)
```

FUNCTION

This function iterates over all items in the list specified by `list` and returns how many items it has. Counting will stop when an element of type `Nil` is found in the list.

Note that this function only counts items at successive integer indices. It starts at index 0 and counts all items at successive integer indices until a `Nil` item is encountered. To count all items of a table, use the `TableItems()` function instead. See Section 53.23 [TableItems], page 1114, for details.

Note that this function is rather slow when used with normal Hollywood tables. To accelerate `ListItems()`, you have to use it with optimized lists created by `CreateList()`. See Section 53.3 [CreateList], page 1100, for details.

INPUTS

`list` table whose items are to be counted

RESULTS

`c` counter

EXAMPLE

```
Print(ListItems({1, 2, 3, 4, 5, 6, 7, 8, 9, 10}))
```

This returns 10.

53.13 NextItem

NAME

NextItem – traverse fields of a table (V2.0)

SYNOPSIS

```
next, item = NextItem(table[, start])
```

FUNCTION

`NextItem()` returns the item that follows after the item `start` in the specified table. If `start` is `Nil`, the first table item is returned. If there is no item after `start`, `Nil` is returned.

This function is mostly used to traverse all fields of the table in argument 1. To do this, you pass the table in argument 1 and leave out the second argument. `NextItem()` then returns an index to the next value in the table and the value at that table index. To traverse all fields, you have to pass the `next` value to `NextItem()` as the second argument and loop over it until the `next` value is `Nil`.

When there are no more items in the table, `Nil` is returned and you can terminate your loop. Be careful when checking variables against `Nil` because 0=`Nil` is actually `True` in Hollywood. Thus, `GetType()` is the only reliable way to find out if a variable is really `Nil`. Simply checking it against `Nil` would also result in `True` if the variable was 0.

Do not expect this function to return the table fields in the order they were assigned. Hollywood often stores them in a different order.

INPUTS

`table`	table to traverse
`start`	optional: where to start the traversal (defaults to `Nil` which means start at the beginning)

RESULTS

`next`	index of the next table item after `start` or `Nil` if there are no more items
`item`	table value of the item next to `start`

EXAMPLE

```
t = {1, 2, 3, 4, 5, "Hello World", {100, 200, 300}, [-1.5] = -1.5,
     b = 66, Function(s) DebugPrint(s) EndFunction}
a, b = NextItem(t)
While GetType(a) <> #NIL
  DebugPrint(b)
  a, b = NextItem(t, a)
Wend
```

The above code traverses a heterogenous table. The output will be the following:

```
2
3
4
5
Hello World
Table: 74cbd42c
```

```
Function: 74cbd3c8
1
-1.5
66
```

You see that the fields are returned in a different order than they were assigned.

53.14 Pack

NAME

Pack – pack a table (V8.0)

SYNOPSIS

```
t = Pack(a[, b, ...])
```

FUNCTION

This function takes all arguments passed to it and stores them in a table, which is then returned. The table will have as many elements as you passed arguments to this function.

To unpack all table elements, use the `Unpack()` function.

INPUTS

`a`	first parameter to pack into table
`b`	second parameter to pack into table
`...`	unlimited number of further parameters to pack

RESULTS

`t`	a table containing all values passed to this function

EXAMPLE

```
t = Pack(1, 2, 3)
Print(t[0], t[1], t[2])
```

This will print `"1 2 3"`.

53.15 Pairs

NAME

Pairs – traverse over all fields of a table (V2.0)

SYNOPSIS

```
func, state, val = Pairs(table)
```

FUNCTION

This function can be used in conjunction with the generic `For` statement to traverse over all fields of a table. As required by the generic `For` statement, `Pairs()` will return three values: An iterator function, a private state information, and an initial value for the traversal. The iterator function will then return the key/value combination of all table fields.

If you want to traverse over the integer indices of a table only, use the `IPairs()` function instead.

See Section 11.4 [Generic For statement], page 127, for details.

INPUTS

`table`	table to traverse

RESULTS

`func`	iterator function
`state`	private state information
`val`	initial traversal value

EXAMPLE

```
a = {"one", "two", "three", x = 5, y = 6}
For i, v In Pairs(a)
  DebugPrint(i, v)
Next
```

The code above will print "0 one", "1 two", "2 three", "x 5", and "y 6".

53.16 RawEqual

NAME

RawEqual – compare two tables without metamethods (V2.0)

SYNOPSIS

```
eq = RawEqual(t1, t2)
```

FUNCTION

This function compares the two tables and returns `True` if they are the same, `False` otherwise. This is basically the same as writing

```
eq = (t1 = t2)
```

The difference is, though, that `RawEqual()` will do the comparison without invoking any metamethod that might be defined in the tables. See Section 9.8 [Metamethods], page 110, for details.

INPUTS

`t1`	table 1 to compare
`t2`	table 2 to compare

RESULTS

`eq`	`True` if tables are equal, `False` otherwise

53.17 RawGet

NAME

RawGet – read value from table without metamethods (V2.0)

SYNOPSIS

```
v = RawGet(t, key)
```

FUNCTION

This function reads the value at index `key` from the specified table and returns it. Basically, this function does the same as the following expression:

```
v = t[key]
```

The difference is that `RawGet()` will never invoke any metamethod and it will not fail if the specified key does not exist. Thus, it is useful for checking if a specific table key exists, or for reading values from tables without invoking any metamethod. See Section 9.8 [Metamethods], page 110, for details.

Please note that string indices are normally in lower case except when using brackets to initialize the table index. Consider the following code:

```
t1 = {TEST = 1}
DebugPrint(RawGet(t1, "TEST"), RawGet(t1, "test")) ; prints Nil/1
t2 = {}
t2.TEST = 1
DebugPrint(RawGet(t2, "TEST"), RawGet(t2, "test")) ; prints Nil/1
t3 = {}
t3["TEST"] = 1
DebugPrint(RawGet(t3, "TEST"), RawGet(t3, "test")) ; prints 1/Nil
```

As you can see, when initializing the `TEST` element using curly braces or the dot operator, the string index `TEST` is automatically converted to lower case. When using brackets to initialize the `TEST` element, however, no conversion is taking place. This also has the consequence that you can't access non-lower case string indices initialized with the bracket syntax using the dot operator because the dot operator always converts the index to lower case.

Starting with Hollywood 6.0 you can also use the convenience function `HaveItem()` to check if a table item exists. See Section 53.8 [HaveItem], page 1103, for details.

INPUTS

`t`	table to query
`key`	key to search for

RESULTS

`v`	value of the specified key or `Nil` if key does not exist in the table

EXAMPLE

```
t = {x = 10, y = 20}
NPrint(RawGet(t, "x"))
NPrint(RawGet(t, "y"))
NPrint(RawGet(t, "z"))
```

The code above will print 10 / 20 / `Nil`.

53.18 RawSet

NAME

RawSet – write value to table without metamethods (V2.0)

SYNOPSIS

```
RawSet(t, key, val)
```

FUNCTION

This function writes the specified value to the specified table at index `key`. The specified value can be of any type (number, string, table, function, etc.). Basically, this function does the same as the following expression:

```
t[key] = v
```

The only difference is that `RawSet()` will never invoke any metamethod so that you have full and immediate access to all table fields. See Section 9.8 [Metamethods], page 110, for details.

INPUTS

`t`	table to modify
`key`	key to write
`val`	value to set key to

53.19 RemoveItem

NAME

RemoveItem – remove item from a list (V2.0)

SYNOPSIS

```
e = RemoveItem(list[, pos])
```

FUNCTION

This function removes an item from the list specified by `list` and returns it. If you omit the optional argument `pos`, the last item of the list will be removed. Otherwise the specified item is removed. Position 0 specifies the first item of the list. After removing the item, the list is reorganized to close the gap.

Note that this function is rather slow when used with normal Hollywood tables. To accelerate `RemoveItem()`, you have to use it with optimized lists created by `CreateList()`. See Section 53.3 [CreateList], page 1100, for details.

INPUTS

`list`	table from where to remove the element
`pos`	optional: item that is to be removed (defaults to -1 which means that the last element should be removed)

RESULTS

`e`	the item that was just removed

EXAMPLE

```
a = {1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10}
e = RemoveItem(a, 7)
For k = 1 To ListItems(a) Do Print(a[k - 1] .. " ")
```

Removes the number 8 because it is twice in the list. The variable e will receive the value 8. After removing the item, the correct row will be printed: "1 2 3 4 5 6 7 8 9 10".

53.20 SetListItems

NAME

SetListItems – convert table to optimized list (V9.0)

SYNOPSIS

```
SetListItems(t, n)
```

FUNCTION

This function can be used to convert an existing table to an optimized list. Optimized lists have the advantage that `InsertItem()`, `RemoveItem()`, `ListItems()` and `GetItem()` are much faster than when used with normal Hollywood tables.

You have to pass the table to convert in the `t` argument and the number of list entries in `n`. Note that the value you pass in `n` must match the number of list entries currently in the table, i.e. it must match the return value of `ListItems()`.

Note that there are some restrictions when using optimized lists. See Section 53.3 [CreateList], page 1100, for details.

INPUTS

`t`	table to convert to optimized list
`n`	number of entries in the list

EXAMPLE

```
t = {}
For Local k = 1 To 10000 Do t[k-1] = k
SetListItems(t, 10000)
Print(ListItems(t))
```

The code above creates a normal Hollywood table, fills it with 10000 items, and then converts it to an optimized list.

53.21 SetMetaTable

NAME

SetMetaTable – assign a table's metatable (V2.0)

SYNOPSIS

```
SetMetaTable(t, mt)
```

FUNCTION

This function assigns the metatable specified in `mt` to the table specified in argument 1. If you pass `Nil` as `mt`, the table's metatable will be removed. See Section 9.8 [Metamethods], page 110, for more information on metatables and metamethods.

INPUTS

`t`	table that shall be modified
`mt`	metatable you would like to assign to `t`

EXAMPLE

See Section 9.8 [Metamethods], page 110.

53.22 Sort

NAME

Sort – sort an array

SYNOPSIS

```
Sort(array[, sortfunc])
```

FUNCTION

This function sorts the array specified by `array`. It supports arrays of type number, type string or an arbitrary data type via a custom callback. This function stops sorting if it finds a `Nil` element or an empty string (`""`) in string arrays. String arrays are sorted alphabetically, number arrays are sorted in ascending order.

Starting with Hollywood 4.5, you can customize the sorting operation by using a custom sort callback. This function has to accept two parameters and it has to return if the first parameter should be inserted before the second one or not. This gives you great flexibility in setting up custom sort operations because you can compare arbitrary values and you can also customize the sorting order.

INPUTS

`array`	array to sort
`sortfunc`	optional: custom function telling Hollywood how to sort (V4.5)

EXAMPLE

```
names = {"Joey", "Dave", "Mark", "Stephen", "Al", "Jefferson"}
Sort(names)
For k = 0 To 5
   NPrint(names[k])
Next
```

The above code defines an array, adds some names to it and then sorts it. The output is "Al, Dave, Jefferson, Joey, Mark, Stephen".

```
nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Sort(nums, Function(a, b) Return(a > b) EndFunction)
For k = 0 To 9
```

```
   NPrint(nums[k])
Next
```

The code above uses a custom sorting function to sort table "nums" in descending order. The result will be: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

53.23 TableItems

NAME

TableItems – count all table items (V6.1)

SYNOPSIS

```
c = TableItems(t)
```

FUNCTION

This function counts all items in the specified table. In contrast to `ListItems()` which only counts items at successive integer indices, `TableItems()` really counts all table elements, including those at string or floating point indices.

INPUTS

`t`	table whose items are to be counted

RESULTS

`c`	number of items in table

EXAMPLE

```
Print(TableItems({x=5, y=6, [6]=1, 3, 4, 5, 6}))
```

This returns 7.

53.24 Unpack

NAME

Unpack – unpack a table (V2.0)

SYNOPSIS

```
a, ... = Unpack(t)
```

FUNCTION

This function unpacks the table specified by `t` and returns all of its elements. `Unpack()` returns as many values as there are elements in the table.

To pack parameters into a table, use the `Pack()` function.

INPUTS

`t`	table to unpack

RESULTS

`a`	first value
`...`	additional return values depending on how much elements there are in the table

EXAMPLE

```
a, b, c = Unpack({1, 2, 3})
```

The above code unpacks the specified table. `a` will get the value 1, `b` will be assigned 2 and `c` will receive the value 3.

```
a = {1, 2, 3, 4, 5, 6}
Print(Unpack(a))
```

This will print "1 2 3 4 5 6".

54 Text library

54.1 Overview

Hollywood's text library contains functions that deal with font management, text measurement, text drawing, as well as text transformation. You can open fonts using the commands `SetFont()`, `OpenFont()`, or `@FONT`. One important thing to know is that Hollywood supports two different text renderers:

1. Inbuilt text renderer: This is the recommended renderer, although it isn't the default renderer due to historical reasons. The inbuilt renderer operates completely independent of the host operating system's text renderer and thus will guarantee that text looks exactly the same on every platform. The pixels drawn by functions such as `Print()` and `TextOut()` will be exactly the same on every platform if you use the inbuilt text renderer. You can access the inbuilt text renderer by setting the `Engine` tag of the `SetFont()`, `OpenFont()`, or `@FONT` commands to `#FONTENGINE_INBUILT`.
2. Native text renderer: This will use the host operating system's text renderer. Due to historical reasons, this is also the default renderer but it isn't recommended to use it because text drawn using this renderer will look slightly different on each platform. If this is no problem for you, you can just go ahead and use it but if you aim to achieve an identical look on every platform, you should use the inbuilt text renderer instead.

The inbuilt text renderer can also open `*.ttf` fonts directly, so you don't even have to install fonts in order to use them with the inbuilt text renderer. You could just use code like this:

```
OpenFont(1, "c:/Windows/Fonts/Arial.ttf", 36, {Engine = #FONTENGINE_
INBUILT})
```

See Section 54.11 [Font specification], page 1126, for details.

See Section 54.42 [Working with fonts], page 1157, for details.

54.2 AddFontPath

NAME

AddFontPath – add additional search path for fonts (V5.0)

SYNOPSIS

`AddFontPath(path$)`

FUNCTION

This function adds the path specified in `path$` to the search paths of Hollywood's inbuilt font engine. By default, the inbuilt font engine only looks for fonts inside a subdirectory "Fonts" in the current directory. On Amiga systems, it also looks in the FONTS: assign. If you want Hollywood to look into other paths as well, you need to add them using `AddFontPath()`.

Please note that the search paths specified here only affect the inbuilt font engine, i.e. they are only used when you specify `#FONTENGINE_INBUILT` in `OpenFont()`, `SetFont()`, or `@FONT`. The search paths specified here are not respected when using `#FONTENGINE_NATIVE`.

INPUTS

`path$` path to add to Hollywood's inbuilt font engine search paths

EXAMPLE

```
AddFontPath("Data/Fonts")
```

Adds the path "Data/Fonts" to the font engine's search paths.

54.3 AddTab

NAME

AddTab – add a tabulator

SYNOPSIS

```
AddTab(pos, ...)
```

FUNCTION

This function adds the tabulator specified by `pos` to the tabulator list of Hollywood. Tabulators can only be used with the `Print()` function. If there is a tabulator character in a string that is passed to `Print()`, then it will jump to the next tabulator position. You can clear the tabulator settings by calling `ResetTabs()`.

New in V2.0: You can pass as many tabulator positions to this command as you like.

INPUTS

`pos` position of the new tabulator (in pixels)

`...` more tabulator positions (V2.0)

EXAMPLE

```
AddTab(100, 200, 300, 400)
SetFontStyle(#UNDERLINED)
NPrint("Last name\tFirst name\tAge\tGender\n")
SetFontStyle(#NORMAL)
NPrint("Doe\tJon\t34\tMale")
NPrint("Smith\tMaggie\t25\tFemale")
NPrint("...\t...\t...\t...")
```

The above code displays a table using tabulators.

54.4 CloseFont

NAME

CloseFont – close an existing font (V4.5)

SYNOPSIS

```
CloseFont(id)
```

FUNCTION

This function frees all memory occupied by the font specified by `id`. To reduce memory consumption, you should close fonts when you do not need them any longer.

INPUTS

`id` identifier of the font to close

54.5 CopyTextObject

NAME

CopyTextObject – clone a text object (V4.0)

SYNOPSIS

```
[id] = CopyTextObject(source, dest)
```

FUNCTION

This function clones the text object specified by `source` and creates a copy of it as text object `dest`. If you specify `Nil` in the `dest` argument, this function will choose an identifier for the cloned text object automatically and return it to you. The new text object is independent from the old text object so you could free up the source text object after it has been cloned.

INPUTS

`source` source text object identifier

`dest` identifier of the text object to be created or `Nil` for auto ID select

RESULTS

`id` optional: identifier of the text object; will only be returned when you pass `Nil` as argument 2 (see above)

EXAMPLE

```
CopyTextObject(1, 10)
FreeTextObject(1)
```

The above code creates a new text object 10 which contains the same graphics data as text object 1. Then it frees text object 1 because it is no longer needed.

54.6 CreateFont

NAME

CreateFont – create font from brush (V10.0)

SYNOPSIS

```
[id] = CreateFont(id, brushid, charmap, width, height, cols[, t])
```

FUNCTION

This function can be used to create a new font from a brush source. This can be useful if you need to work with custom fonts that are distributed as image files instead of common font formats like it was often the case in games from the 1980s and early 1990s and in scene demos. You have to pass the desired identifier for the new font in the `id` argument and the identifier of the brush source in the `brushid` argument. If you pass `Nil` in `id`, `CreateFont()` will automatically choose an identifier and return it. After the font has been created successfully, you can set it as the current font using `UseFont()`.

The `charmap` parameter must be set to a string describing the individual characters in the brush. The character dimensions must be passed in the `width` and `height` parameters. All characters must share the same dimensions. The number of characters per row must be passed in the `cols` argument.

For example, a font which supports the upper-case characters A-Z and whose characters are 32x32 pixels each could be laid out in a brush that has 4 rows containing 8 characters per row, except for the last row which contains only 2 because the English alphabet has just 26 characters so 3 rows with 8 characters plus one last row with 2 characters are sufficient. Thus, you could create such a font from a 256x128 sized brush and pass 32 for `width` and `height`, 8 for `cols` and `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"` in `charmap`.

The optional table argument can be used to set some additional options. The following tags are currently recognized:

`Name:` This allows you to give your font a name. By default, the font's name will be set to `"Font"`.

`RowSpacing:`
If there is some spacing between the different rows of characters in the brush, you can tell `CreateFont()` about it using this tag. Just set this tag to the number of spacing pixels between each row and `CreateFont()` will skip the spacing when it creates font. Defaults to 0 which means no vertical spacing.

`ColSpacing:`
If there is some spacing between the individual characters in the brush, you can tell `CreateFont()` about it using this tag. Just set this tag to the number of spacing pixels between each character and `CreateFont()` will skip the spacing when it creates font. Defaults to 0 which means no horizontal spacing.

`Ascender:`
This tag allows you to set the desired ascender for the font. The ascender of a font is the maximum character extent from the baseline to the top of the line. Hollywood uses the ascender value to determine where to draw the line for the underline text style, for example. This defaults to the character height passed in `height` minus 1.

`CreateFont()` supports palette brushes as well as brushes with mask or alpha channel. If the brush passed in `brushid` is a 1-bit palette brush, you will also be able to change the color of the font using `SetFontColor()` and other functions just like you can do it for normal fonts. If the brush's depth is more than 1-bit, however, the font will be treated as a color font that always uses the same color no matter what the current font color is set to.

Note that `CreateFont()` is quite flexible and could also be used as a tilemapper. Just map each tile to a character and then draw the whole tilemap using a single call to `TextOut()`. This should be much faster than drawing the tiles individually.

INPUTS

`id` identifier for the font or `Nil` for auto id selection

`brushid`	identifier of the source brush
`charmap`	string describing all characters in the brush
`width`	width of each font character
`height`	height of each font character
`cols`	number of characters per row
`t`	optional: table containing further options (see above)

RESULTS

`id`	optional: identifier of the font; will only be returned if you pass `Nil` as argument 1 (see above)

EXAMPLE

```
CreateFont(1, 2, "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!-.:?",30,32,10)
UseFont(1)
NPrint("HELLO WORLD!")
```

The code above constructs a new font from brush 2. There are 41 characters in the source brush and they are laid out as 10 characters per row and 30x32 pixels each. This means that the source brush must be at least 300x160 pixels. After creating the font, it will be selected as the current one and the text "HELLO WORLD!" will be printed.

54.7 CreateTextObject

NAME

CreateTextObject – create a text object

SYNOPSIS

```
[id] = CreateTextObject(id, text$[, table])
```

FUNCTION

This function creates a new text object containing the data specified by `text$` and assigns the specified `id` to it. If you pass `Nil` in `id`, `CreateTextObject()` will automatically choose an identifier and return it. The text is rendered in the current color and with the currently selected font.

The advantage of text objects compared to standard text (output via `Print()` for example) is that you can easily position text objects on the screen, remove them or even scroll them using `MoveTextObject()`.

Starting with Hollywood 2.5, you can use format tags in the string you pass to `CreateTextObject()`. Using these tags you can control the font style and color of your text on-the-fly. Format tags always start and end with a square bracket ('['). In case you just want to print a square bracket, you will have to use two square brackets. If there is only one square bracket, Hollywood will always expect a format tag. Please see the chapter about format tags for more information on this topic.

In Hollywood 5.0 the syntax of this function changed slightly. While the old syntax is still supported for compatibility, new scripts should use the new syntax which accepts a table as argument 4. The table can contain the following elements:

`Align:` Allows you to specify the text's alignment. The following alignments are currently supported:

`#LEFT` Left alignment.

`#RIGHT` Right alignment.

`#CENTER` Center lines.

`#JUSTIFIED`
Lay out text in justified lines. (V7.0)

The default value for `Align` is `#LEFT`.

`WordWrap:`
`CreateTextObject()` can do automatic word-wrapping for you if you specify this additional parameter. You can use this tag to specify a maximum width for your text. `CreateTextObject()` will then use word wrapping to make sure that no text runs beyond this limit. If you do not set this argument or set it to 0 (which is also the default), the text will be as wide as it is required. Starting with Hollywood 9.1, you can also use soft hyphens or zero-width space characters to customize word wrapping but since these are Unicode characters, you need to make sure that you use UTF-8 encoding in that case.

`Encoding:`
This argument can be used to specify the character encoding inside `text$`. This defaults to the default character encoding for the text library as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details.

`Color:` This tag allows you to specify the text color. The color must be provided as an ARGB value. If you do not specify this tag, `CreateTextObject()` will use the color that was set using the `SetFontColor()` command instead.

`Pen:` When palette mode is `#PALETTEMODE_PEN`, this tag can be used to set the pen that should be used for drawing the text. If palette mode is `#PALETTEMODE_PEN` and `Pen` is not specified, the pen set using `SetDrawPen()` will be used instead. (V9.0)

`Linespacing:`
This tag can be used to customize the spacing pixels between lines. It can be set to a positive or a negative value. A negative value moves lines closer together, whereas a positive value increases the spacing between the lines. A value of 0 means no custom line spacing. Defaults to 0. (V10.0)

`Charspacing:`
Allows you to adjust the space between characters. You can set this to a positive or negative value. A positive value will increase the space between characters, a negative value will decrease it. (V10.0)

`NoAdjust:`
When drawing text objects using `DisplayTextObject()` Hollywood will position them in a way that they appear as if they had been drawn using

`TextOut()` which means that they could be offset to the left and top in case parts of some characters are designed to appear in the area of previous characters. This is often the case with characters like "j". If you don't want that, set `NoAdjust` to `True`. In that case, calling `DisplayTextObject()` will never lead to any adjustments in positioning but the text object will strictly be drawn at the specified position. The adjustment offsets applied to a text object by Hollywood in case `NoAdjust` is `False` can be found out by querying the `#ATTRADJUSTX` and `#ATTRADJUSTY` tags. Defaults to `False`. (V10.0)

Note that Hollywood currently only supports standard left-to-right based text aligned on horizontal lines. Right to left and vertical text is currently not supported.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color set via `SetFontColor()` or the `Color` tag above.

INPUTS

`id`	identifier of the new text object or `Nil` for auto id selection
`text$`	text for the text object
`table`	optional: a table containing further options for the text object

RESULTS

`id`	optional: identifier of the text object; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
SetFontColor(#RED)
SetFont("times.font", 18)
CreateTextObject(1, "Hello World!")
DisplayTextObject(1, #CENTER, #CENTER)
```

The above code creates a text object with the font "times" (size 18) and with the color red. The text is "Hello World". After its creation, the text object is displayed in the center of the screen.

54.8 DisplayTextObject

NAME

DisplayTextObject – show a text object

SYNOPSIS

```
DisplayTextObject(id, x, y)
```

FUNCTION

This function displays the text object specified by `id` at the coordinates specified by `x` and `y`.

If layers are enabled this command will add a new layer of the type `#TEXTOBJECT` to the layer stack.

INPUTS

`id` identifier of the text object to play

`x` x position

`y` y position

EXAMPLE

See Section 54.7 [CreateTextObject], page 1121.

54.9 DisplayTextObjectFX

NAME

DisplayTextObjectFX – display a text object with transition effects

SYNOPSIS

```
[handle] = DisplayTextObjectFX(id, x, y[, table])
```

FUNCTION

This function is an extended version of the `DisplayTextObject()` command. It displays the text object specified by `id` at the position specified by `x` and `y` and it uses one of the many Hollywood transition effects to display it. You need also specify the speed for the transition.

If layers are enabled this command will add a new layer of the type `#TEXTOBJECT` to the layer stack.

Starting with Hollywood 4.0 this function uses a new syntax with just a single table as an optional argument. The old syntax is still supported for compatibility reasons. The optional table argument can be used to configure the transition effect. The following options are possible:

`Type:` Specifies the desired effect for the transition. See Section 20.11 [DisplayTransitionFX], page 238, for a list of all supported transition effects. (defaults to `#RANDOMEFFECT`)

`Speed:` Specifies the desired speed for the transition. The higher the value you specify here, the faster the effect will be displayed. (defaults to `#NORMALSPEED`)

`Parameter:`
Some transition effects accept an additional parameter. This can be specified here. (defaults to `#RANDOMPARAMETER`)

`Async:` You can use this field to create an asynchronous draw object for this transition. If you pass `True` here `DisplayTextObjectFX()` will exit immediately, returning a handle to an asynchronous draw object which you can then draw using `AsyncDrawFrame()`. See Section 19.1 [AsyncDrawFrame], page 221, for more information on asynchronous draw objects.

INPUTS

`id` identifier of the text object to display

`x` desired x position for the text object

`y` desired y position for the text object

`table` optional: transition effect configuration

RESULTS

`handle` optional: handle to an asynchronous draw object; will only be returned if `Async` has been set to `True` (see above)

EXAMPLE

```
DisplayTextObjectFX(1, 0, 0, #VLINES, 10)   ; old syntax

OR

DisplayTextObjectFX(1, 0, 0, {Type = #VLINES, Speed = 10})  ; new syntax
```

The above code displays text object 1 at 0:0 with a `#VLINES` transition at speed 10.

54.10 FONT

NAME

FONT – preload a font for later use (V4.5)

SYNOPSIS

```
@FONT id, fontname$, size[, table]
```

FUNCTION

This preprocessor command preloads the font specified by `fontname$` in the desired size (in pixels) and assigns the identifier `id` to it. You can then activate the font from your script by calling `UseFont()`.

The font specified in `fontname$` must adhere to the font specification. See Section 54.11 [Font specification], page 1126, for details.

See Section 54.24 [OpenFont], page 1134, for more information on fonts in Hollywood.

Using `@FONT` is convenient if you want to have all fonts used by your scripts linked to your applet/executable. By default, all fonts specified using `@FONT` are linked to your applet/executable. You can change this behaviour by setting `Link` to `False` in the optional table argument.

The fourth argument is optional. It is a table that can be used to set further options for the loading operation. This table accepts all tags supported by the optional table of the `SetFont()` command as well as the following tags:

`Link:` Set this field to `False` if you do not want to have this font linked to your executable/applet when you compile your script. This field defaults to `True` which means that the font is linked to your to your executable/applet when Hollywood is in compile mode.

If you want to open fonts manually, please use the `OpenFont()` command.

Important note: Please note that most fonts are copyrighted and it is not allowed to link them into your programs without acquiring a licence. So make sure you check the licence of the font you are going to link into your program! If you do not want to pay

for font licences, it is advised to use a free font such as DejaVu or Bitstream Vera or use one of the TrueType fonts that are inbuilt into Hollywood (`#SANS`, `#SERIF`, `#MONOSPACE`, cf. `SetFont()`)

See Section 54.42 [Working with fonts], page 1157, for more information on using fonts in a platform-independent manner.

INPUTS

`id` a value that is used to identify this font later in the code

`fontname$`
the font you want to open

`size` desired font size in pixels

`table` optional argument specifying further options

EXAMPLE

```
@FONT 1, "Arial", 36
```

Opens font Arial in size 36 and makes it available under id 1.

54.11 Font specification

The Hollywood font specification is the notation that needs to be used when opening new fonts using `SetFont()`, `OpenFont()`, or `@FONT`. These three commands require you to pass a string describing the font you would like to open. This string must follow these guidelines:

1. Do not specify a file, specify a font name! e.g.

```
SetFont("dh0:Fonts/Goudyb.font", 23)    ; --> wrong!
SetFont("Goudyb", 23)                   ; --> right!
OpenFont(1, "c:/Windows/Fonts/Arial.ttf", 36)  ; --> wrong!
OpenFont(1, "Arial", 36)                       ; --> right!
```

EXCEPTION: Starting with Hollywood 4.7, there is a new font engine called `#FONTENGINE_INBUILT`. If you are using this engine, you can specify the font file directly, but only for `*.ttf` fonts! So the following code is legal with Hollywood 4.7 and up:

```
OpenFont(1, "c:/Windows/Fonts/Arial.ttf", 36, {Engine = #FONTENGINE_
INBUILT})
```

2. On some systems font names are case sensitive when you use `#FONTENGINE_NATIVE` (for example on macOS). Thus, you should always specify the font name in exactly the same way as it appears in the font. This can avoid potential problems.

```
SetFont("arial", 36)  ; --> wrong!
SetFont("Arial", 36)  ; --> right!
```

3. For TrueType fonts, the font specification consists of two parts: 1) The face name of the font and 2) its style parameters. There must be space between the face name and the style. Also, if there are multiple styles, they must be separated by spaces. E.g.

```
SetFont("Arial", 36)
SetFont("Arial Bold", 36)
```

```
SetFont("Arial Bold Italic", 36)
```

Of course, you could also open "Arial" and then call `SetFontStyle()` with `#BOLD` or `#ITALIC` set, but the advantage of using it directly with `SetFont()` is that this will open the designed bold/italic variant of the TrueType font. `SetFontStyle()` on the other hand, will create bold and italic using an algorithm which does not look as good as specifically designed bold/italic font faces.

4. **Special note for AmigaOS3, MorphOS, and AROS users:** FTManager often uses very awkward names for fonts. For example, if you are trying to install the font "Adobe Caslon Pro Bold Italic" FTManager will install this font as "adobecaslonprobolditalic" by default. You will then be able to open "adobecaslonprobolditalic" on AmigaOS3/MorphOS/AROS with Hollywood but of course it will not work on AmigaOS4 or Windows or macOS because of this awkward font name. Thus, you should edit the font name suggestion made by FTManager in the following way:
   1. Insert spaces between the different components:
      ```
      "adobecaslonprobolditalic" -> "adobe caslon pro bold italic"
      ```
   2. Adapt the spelling to the spelling of the face name (displayed in FTManager):
      ```
      "adobe caslon pro bold italic" -> "Adobe Caslon Pro bold italic"
      ```
   3. Capitalize all style settings:
      ```
      "Adobe Caslon Pro bold italic" -> "Adobe Caslon Pro Bold Italic"
      ```

   If you follow these guidelines, the font will also work on other systems than AmigaOS3, MorphOS, and AROS.

54.12 FreeGlyphCache

NAME

FreeGlyphCache – clear glyph cache (V4.7)

SYNOPSIS

```
FreeGlyphCache(mode[, id])
```

FUNCTION

This command can be used to free the cached glyphs of either a specific font or of all fonts currently in memory. If you want to free the cached glyphs of a specific font, you have to set the `mode` argument to 1 and pass the identifier of the font in the optional `id` argument. If you want to free the glyph cache of all loaded fonts, simply pass 0 in the `mode` argument.

INPUTS

`mode`	set this to 1 to free the glyph cache of the font specified in argument 2 or to 0 to free the glyph cache of all fonts
`id`	optional: identifier of the font whose glyph cache shall be cleared (mode must be set to 1 if this is used)

54.13 FreeTextObject

NAME

FreeTextObject – free a text object

SYNOPSIS

```
FreeTextObject(id)
```

FUNCTION

This function frees the memory of the text object specified by `id`. To reduce memory consumption, you should free text objects when you do not need them any longer.

INPUTS

`id` identifier of the text object

54.14 GetAvailableFonts

NAME

GetAvailableFonts – retrieve list of available fonts (V4.7)

SYNOPSIS

```
t = GetAvailableFonts()
```

FUNCTION

This function scans all fonts installed on the current computer, puts them into a table, and returns the information to you. This is useful to check if a specific font is available without calling `SetFont()` or `OpenFont()`.

The table returned by this function will consist of several subtables. One subtable for each font. The subtables have the following elements initialized:

`Name:` The complete font name (i.e. family name plus style). For example, `"Arial Bold Italic"`.

`Family:` The family name of this font, e.g. `"Arial"`.

`Weight:` The weight of this font. This will be set to one of the following weight constants:

```
#FONTWEIGHT_THIN
#FONTWEIGHT_EXTRALIGHT
#FONTWEIGHT_ULTRALIGHT
#FONTWEIGHT_LIGHT
#FONTWEIGHT_BOOK
#FONTWEIGHT_NORMAL
#FONTWEIGHT_REGULAR
#FONTWEIGHT_MEDIUM
#FONTWEIGHT_SEMIBOLD
#FONTWEIGHT_DEMIBOLD
#FONTWEIGHT_BOLD
#FONTWEIGHT_EXTRABOLD
#FONTWEIGHT_ULTRABOLD
```

```
#FONTWEIGHT_HEAVY
#FONTWEIGHT_BLACK
#FONTWEIGHT_EXTRABLACK
#FONTWEIGHT_ULTRABLACK
```

`Slant:` The slant style of this font. This will be set to one of the following slant constants:

```
#FONTSLANT_ROMAN
#FONTSLANT_ITALIC
#FONTSLANT_OBLIQUE
```

`Bitmap:` `True` if this font is a bitmap font, `False` if it is a vector font. Vector fonts can be freely transformed and antialiased.

`Sizes:` If the font is a bitmap font this will be a table containing a list of available sizes for the font. If the font is a vector font, this table will be empty.

Please note that there is no guarantee that all calls to `OpenFont()` or `SetFont()` will succeed with the fonts returned by this function. It can often happen that `OpenFont()` and `SetFont()` will fail with a specific font although it was returned in the available table by this function. This is because `GetAvailableFonts()` returns the available fonts for all Hollywood font engines. When you call `OpenFont()` or `SetFont()`, however, only one font engine can be specified. So if a call to `OpenFont()` fails although the font was returned by `GetAvailableFonts()`, then this is a sign that you are using the wrong font engine to open this font. Simply switch font engines in that case and it should work correctly.

INPUTS

none

RESULTS

`t` a table containing all available fonts

EXAMPLE

```
t = GetAvailableFonts()
For Local k = 0 To ListItems(t) - 1
  DebugPrint("Family:", t[k].Family, "Weight:", t[k].Weight,
             "Slant:", t[k].Slant, "Bitmap:", t[k].Bitmap)
Next
```

The code above lists all fonts available on this system.

54.15 GetBulletColor

NAME

GetBulletColor – get current bullet color (V9.0)

SYNOPSIS

```
color = GetBulletColor()
```

FUNCTION

This function returns the bullet color set using `SetFontColor()` or `SetBulletColor()`.

INPUTS
none

RESULTS
`color` current bullet color

54.16 GetCharMaps

NAME
GetCharMaps – return character maps supported by font (V9.0)

SYNOPSIS
```
t = GetCharMaps()
```

FUNCTION
This function returns a table that contains all character maps supported by the font that is currently the active one. The following character maps are currently supported by Hollywood:

```
#CHARMAP_DEFAULT
#CHARMAP_MSSYMBOL
#CHARMAP_UNICODE
#CHARMAP_SJIS
#CHARMAP_BIG5
#CHARMAP_WANSUNG
#CHARMAP_JOHAB
#CHARMAP_ADOBESTANDARD
#CHARMAP_ADOBEEXPERT
#CHARMAP_ADOBECUSTOM
#CHARMAP_ADOBELATIN1
#CHARMAP_OLDLATIN2
#CHARMAP_APPLEROMAN
```

Note that character maps are only supported for fonts handled by Hollywood's inbuilt engine, i.e. the `Engine` tag must have been set to `#FONTENGINE_INBUILT` when opening the font. See Section 54.31 [SetFont], page 1139, for details.

INPUTS
none

RESULTS
`t` table containing all character maps supported by the current font

54.17 GetDefaultEncoding

NAME
GetDefaultEncoding – get default character encoding (V7.0)

SYNOPSIS
```
tencoding, sencoding = GetDefaultEncoding()
```

FUNCTION
This function returns the default character encodings for the text library in `tencoding` and for the string library in `sencoding`. See Section 54.30 [SetDefaultEncoding], page 1138, for details.

INPUTS
none

RESULTS
`tencoding`
default character encoding for the text library

`sencoding`
default character encoding for the string library

54.18 GetFontColor

NAME
GetFontColor – get current font color (V7.1)

SYNOPSIS
```
color = GetFontColor()
```

FUNCTION
This function returns the font color set using `SetFontColor()`. See Section 54.32 [SetFontColor], page 1142, for details.

INPUTS
none

RESULTS
`color` current font color

54.19 GetFontStyle

NAME
GetFontStyle – get current font style (V7.1)

SYNOPSIS
```
style[, t] = GetFontStyle()
```

FUNCTION
This function returns the current font style set using `SetFontStyle()`. The return value `style` is set to a combination of the flags `#BOLD`, `#ITALIC`, `#UNDERLINED`, `#ANTIALIAS`, `#SHADOW`, and `#BORDER`. See Section 54.33 [SetFontStyle], page 1143, for details.

If `#SHADOW` is set, `GetFontStyle()` also returns a table as the second return value which contains the following fields:

`ShadowColor:`
The shadow color.

`ShadowSize:`
The distance of the shadow from the main text in pixels.

`ShadowDir:`
The direction of the shadow. This will be one of the directional constants.

If `#BORDER` is set, the return table will contain the following fields:

`BorderColor:`
The color of the border.

`BorderSize:`
The thickness of the border in pixels

See Section 54.33 [SetFontStyle], page 1143, for more information on font styles.

INPUTS
none

RESULTS

`style`	a combination of font style flags
`t`	optional: table containing additional style information (see above)

54.20 GetKerningPair

NAME
GetKerningPair – return kerning setting for two adjacent characters (V5.0)

SYNOPSIS
```
kern = GetKerningPair(a$, b$[, encoding])
```

FUNCTION
This function computes the kerning value that would be applied to the space between the two characters `a$` and `b$` if they were drawn next to each other. Kerning is often used to reduce spaces between two characters. For example, if a `"j"` character is drawn next to an `"i"` character, the `"j"` is usually moved some pixels to the left so that its underhang appears below the `"i"` which makes the text look more smooth. The kerning value returned by this function is specified in pixels. A negative kerning value means a move to the left, while a positive kerning value moves to the right.

The optional argument `encoding` can be used to specify the character encoding inside `a$` and `b$`. This defaults to the default text library character encoding as set by `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details.

Note there must be only one character in `a$` and `b$` for `GetKerningPair()` to work correctly.

INPUTS

`a$`	character one
`b$`	character two
`encoding`	optional: character encoding used by the strings (defaults to the text library encoding specified in the last call to `SetDefaultEncoding()`)

RESULTS

`kern` kerning value for `a$` and `b$`

EXAMPLE

```
SetFont(#SANS, 72)
SetFontStyle(#ANTIALIAS)
kern = GetKerningPair("W", "a")
```

The code above computes the kerning value for characters "W" and "a" using the inbuilt sans-serif font in size 72. It will return -3 which means that the "a" character is moved 3 pixels towards the "W" character.

54.21 Locate

NAME

Locate – set the cursor position

SYNOPSIS

```
Locate(x, y)
```

FUNCTION

This function sets the cursor to `x`,`y`. The cursor position is used by the Print function as the position where the output starts.

Please note: You cannot specify any of Hollywood's special constants for `x` or `y` because there is no reference width or height, therefore things like `#CENTER`, `#BOTTOM`, `#RIGHT` etc. cannot work. If you want to use these special constants, you will have to use the function `TextOut()` to print your text.

INPUTS

`x` desired new position

`y` desired new position

54.22 MoveTextObject

NAME

MoveTextObject – move text object from a to b

SYNOPSIS

```
MoveTextObject(id, xa, ya, xb, yb[, table])
```

FUNCTION

This function moves (scrolls) the text object specified by `id` softly from the location specified by `xa`,`ya` to the location specified by `xb`,`yb`.

Further configuration options are possible using the optional argument table. You can specify the move speed, special effect, and whether or not the move shall be asynchronous. See Section 21.46 [MoveBrush], page 287, for more information on the optional table argument.

INPUTS

`id`	identifier of the text object to use as source
`xa`	source x position
`ya`	source y position
`xb`	destination x position
`yb`	destination y position
`table`	optional: further configuration for the move

EXAMPLE

```
MoveTextObject(1,100,50,0,50,{Speed = 5})
```

Moves the text object 1 from 100:50 to 0:50 with speed 5.

54.23 NPrint

NAME

NPrint – print data and append a linefeed

SYNOPSIS

```
NPrint(var, ...)
```

FUNCTION

Does the same as Print but adds a linefeed at the end.

INPUTS

`var`	data to print
`...`	other arguments (V2.0)

54.24 OpenFont

NAME

OpenFont – open a new font (V4.5)

SYNOPSIS

```
[id] = OpenFont(id, fontname$, size[, table])
```

FUNCTION

This function loads the font specified in `fontname$` and makes it available to your script under the specified id. If you pass `Nil` in `id`, `OpenFont()` will automatically choose an identifier and return it. After the font has been opened successfully, you can set it as the current font using `UseFont()`.

The font specified in `fontname$` must adhere to the font specification. See Section 54.11 [Font specification], page 1126, for details.

The optional table argument can be used to set further options. This is especially useful if you want to use Hollywood's inbuilt font engine which guarantees a pixel-perfect identical

look across different platforms. See Section 54.31 [SetFont], page 1139, for information on what tags can be used in the optional table argument.

Normally, it is more convenient to open fonts using `SetFont()` directly because it saves you the hassle of having to deal with font handles. But in certain circumstances - for instance, if you need to switch between different fonts a lot - it is handy to preload these fonts using `OpenFont()`. They are then available quicker to your script.

This command is also available from the preprocessor: Use `@FONT` to preload fonts! The advantage of using @FONT is that fonts specified in there are automatically linked to your applet/executable when compiling.

See Section 54.42 [Working with fonts], page 1157, for more information on using fonts in a platform-independent manner.

INPUTS

`id` identifier for the font or `Nil` for auto id selection

`fontname$`
font to open

`size` desired font size in pixels

`table` optional: table with further options (V4.7)

RESULTS

`id` optional: identifier of the font; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenFont(1, "Arial", 36)
UseFont(1)
```

Opens Arial in size 36 and makes it the current font.

54.25 Print

NAME

Print – print data to the screen

SYNOPSIS

```
Print(var, ...)
```

FUNCTION

Prints the data specified by `var` to the screen. This function can handle all different data types: You can print strings, numbers, tables, functions. The data is printed at the current cursor position which you can modify by calling `Locate()`.

This function uses word-wrapping, e.g. when the margin is reached and a word cannot be printed in the same line, it will insert a line break automatically. You can manually set the margins by using the `SetMargins()` function. Starting with Hollywood 9.1, you can also use soft hyphens or zero-width space characters to customize word wrapping but since these are Unicode characters, you need to make sure that you use UTF-8 encoding in that case.

This function also respects your tabulator settings. If you print a string which contains a tabulator char ("\t"), print will jump to the next tabulator position. You can define the tabulator settings with the `AddTab()` and `ResetTabs()` commands.

You can also specify escape codes here. See Section 8.3 [String data type], page 98, for details.

If layers are enabled this command will add a new layer of the type `#PRINT` to the layer stack.

Starting with Hollywood 2.0 you can pass as many arguments as you want to this function. If you pass multiple arguments to this function, they will be printed with a space to separate them.

Starting with Hollywood 2.5 you can use format tags in the string you pass to `Print()`. Using these tags you can control the font style and color of your text on-the-fly. Format tags always start and end with a square bracket ('['). In case you just want to print a square bracket, you will have to use two square brackets. If there is only one square bracket Hollywood will always expect a format tag. See Section 54.36 [Format tags], page 1146, for details.

Besides `Print()`, you can also use the functions `NPrint()` and `TextOut()` to draw text to the screen.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the font color set using `SetFontColor()`.

INPUTS

`var`	data to print
`...`	other arguments (V2.0)

EXAMPLE

```
Print("Hello World!")
```

Prints "Hello World!" to the screen at the current cursor position.

54.26 ResetTabs

NAME

ResetTabs – clear tabulator settings

SYNOPSIS

```
ResetTabs()
```

FUNCTION

This function clears all previous tabulator settings and sets the default (a tabulator is converted to 8 spaces). There is no function to remove a single tabulator, so if you want to do this, you will have to call this function and then add all tabulators with `AddTab()` except the one you want to have removed.

INPUTS

none

54.27 RotateTextObject

NAME

RotateTextObject – rotate a text object (V4.0)

SYNOPSIS

```
RotateTextObject(id, angle[, smooth])
```

FUNCTION

This function rotates the text object specified by `id` by the specified angle (in degrees). A positive angle rotates anti-clockwise, a negative angle rotates clockwise. Optionally, you can choose to enable anti-aliased interpolation by passing `True` in the `smooth` argument.

Note that for vector text objects, `RotateTextObject()` will always operate on the untransformed text object. This means that any previous transformations applied to the text object using `RotateTextObject()`, `TransformTextObject()`, or `ScaleTextObject()` will be undone when calling `RotateTextObject()`.

INPUTS

`id`	identifier of the text object to rotate
`angle`	desired rotation angle in degrees
`smooth`	optional: whether or not anti-aliased interpolation shall be used (V9.1)

54.28 ScaleTextObject

NAME

ScaleTextObject – scale a text object (V4.0)

SYNOPSIS

```
ScaleTextObject(id, width, height[, smooth])
```

FUNCTION

This command scales the text object specified by `id` to the specified dimensions. If the text object uses a vector font it will be scaled without a loss of quality. Optionally, you can choose to enable anti-aliased interpolation by passing `True` in the `smooth` argument.

You can pass `#KEEPASPRAT` as either `width` or `height`. Hollywood will calculate the size then automatically by taking the aspect-ratio of the text object into account. The `width` and `height` arguments can also be a string containing a percent specification, e.g. `"50%"`.

Note that for vector text objects, `ScaleTextObject()` will always operate on the untransformed text object. This means that any previous transformations applied to the text object using `ScaleTextObject()`, `TransformTextObject()`, or `RotateTextObject()` will be undone when calling `ScaleTextObject()`.

INPUTS

`id`	identifier of the text object to scale
`width`	desired new width for the text object
`height`	desired new height for the text object

`smooth` optional: whether or not anti-aliased interpolation shall be used (V9.1)

EXAMPLE

```
ScaleTextObject(1, 600, 200)
```

Scales text object 1 to a resolution of 600x200.

54.29 SetBulletColor

NAME

SetBulletColor – set bullet color (V9.0)

SYNOPSIS

```
SetBulletColor(color)
```

FUNCTION

This function sets the color to be used by bullets when using `TextOut()` in list mode. By default, bullets appear in the current font color set using `SetFontColor()`. If you want them to be drawn in a different color, you can use this function to do so. The `color` argument must be either an RGB value or an ARGB value for alpha-blended text.

See Section 54.39 [TextOut], page 1149, for more information on bullet lists.

INPUTS

`color` RGB or ARGB color specification

EXAMPLE

```
SetBulletColor(#GRAY)
```

This code sets the bullet color to some kind of grey.

```
SetBulletColor(ARGB(128, #RED))
```

The above code sets the bullet color to half-red. The background will then shine through the text at a ratio of 50% (128=50% of 255).

54.30 SetDefaultEncoding

NAME

SetDefaultEncoding – set default character encoding (V4.7)

SYNOPSIS

```
SetDefaultEncoding(tencoding[, sencoding])
```

FUNCTION

This function can be used to change the default character encoding for the text and string libraries. Note that for reasons of compatibility Hollywood maintains two different default character encodings: one for the text library and one for the string library. Under normal conditions, however, both default encodings should be set the to the same character encoding.

The default character encoding for the text library is specified in `tencoding` and affects functions such as `Print()`, `NPrint()`, `TextOut()`, and `CreateTextObject()`.

The default character encoding for the string library needs to be specified in the `sencoding` parameter and affects most functions of the string library, i.e. functions such as `ReplaceStr()` and `StrLen()`.

The following character encodings are currently supported by Hollywood:

`#ENCODING_UTF8:`
: Use UTF-8 encoding. This is the default since Hollywood 7.0.

`#ENCODING_ISO8859_1:`
: Use ISO 8859-1 encoding. This was the default prior to Hollywood 7.0. Note that for historical reasons specifying `#ENCODING_ISO8859_1` on AmigaOS and compatibles doesn't really mean ISO 8859-1 but whatever is the system's default character encoding. On most Amiga systems, this is ISO 8859-1 anyway, but Eastern European systems use a different encoding for example.

Starting with Hollywood 7.0, `#ENCODING_ISO8859_1` shouldn't be used any longer. It is still supported for compatibility reasons but it can lead to problems on non-ISO-8859-1 systems. You should always use `#ENCODING_UTF8` starting with Hollywood 7.0.

INPUTS

`tencoding`
: default character encoding for the text library

`sencoding`
: default character encoding for the string library (V7.0)

54.31 SetFont

NAME

SetFont – change the current font

SYNOPSIS

```
SetFont(font$, size[, table])
```

FUNCTION

This function changes the current font to the one specified by `font$` and `size`. The `size` argument specifies the desired font's height in pixels. The current font is used by commands like the `Print()` command but also by `CreateTextObject()`. The font specified in `font$` must adhere to the Hollywood font specification. See Section 54.11 [Font specification], page 1126, for details.

The font style will be reset when calling this command.

Starting with Hollywood 4.7, there is an optional table argument which allows you to configure the following advanced options:

`Engine:`
: This tag specifies which font engine Hollywood should use for this font. This can be either `#FONTENGINE_NATIVE` (uses the native font engine of the host OS) or `#FONTENGINE_INBUILT` (uses the font engine built into Hollywood). If you are using TrueType fonts in your project and want your texts to look exactly the same on every platform, you must make sure that you use

the #FONTENGINE_INBUILT engine because otherwise the text look will be different from platform to platform. Another advantage of the #FONTENGINE_INBUILT engine is that you can directly specify a *.ttf file as font$ without the need of installing the font first on the local system. See Section 54.11 [Font specification], page 1126, for details. For compatibility reasons, this tag defaults to #FONTENGINE_NATIVE. Note that the Engine tag is deprecated since Hollywood 10.0. You should use the Loader tag instead now (see below). Passing native in the Loader tag does the same as setting Engine to #FONTENGINE_NATIVE and passing inbuilt in the Loader tag corresponds to the #FONTENGINE_INBUILT engine. (V4.7)

Cache: Specifies whether or not glyph caching should be employed. Glyph caching can radically increase performance, especially on slower systems like OS3, but of course it needs more memory. Glyph caching is currently only supported by the inbuilt font engine (i.e. #FONTENGINE_INBUILT). To disable glyph caching, set this tag to False. Defaults to True. (V4.7)

UsePoints:
Set this tag to True if you wish to pass a point size instead of a pixel size in the size argument. If you set this tag to True, SetFont() will interpret the value passed in size as a value in points (pt) instead of pixels. Generally, it is not recommended to use this tag because point sizes always depend on the host display's dots-per-inch (DPI), but all your other graphics are typically pixel graphics which are independent of the host system's DPI settings. Thus, when integrating fonts opened using a point height with pixel graphics, those fonts can appear larger or smaller, depending on the host display's DPI settings, and mess up your design. That is why it is generally not recommended to specify the font height in points instead of pixels. Defaults to False. (V7.0)

CharMap: When Engine has been set to #FONTENGINE_INBUILT, the CharMap tag allows you to specify the char map that the font should use. Normally, it's not necessary to set this but some fonts (e.g. Wingdings, Webdings) use custom char maps that can't be consistently mapped to Unicode. In that case, explicitly telling the font engine which char map to use can be useful. CharMap can be set to the following char maps:

```
#CHARMAP_DEFAULT
#CHARMAP_MSSYMBOL
#CHARMAP_UNICODE
#CHARMAP_SJIS
#CHARMAP_BIG5
#CHARMAP_WANSUNG
#CHARMAP_JOHAB
#CHARMAP_ADOBESTANDARD
#CHARMAP_ADOBEEXPERT
#CHARMAP_ADOBECUSTOM
#CHARMAP_ADOBELATIN1
#CHARMAP_OLDLATIN2
```

`#CHARMAP_APPLEROMAN`

The default is `#CHARMAP_DEFAULT`. To find out the char maps supported by a font, use the `GetCharMaps()` command. (V9.0)

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this font. This must be set to a string containing the name(s) of one or more loader(s). Set this to `native` if you want Hollywood to use the native font engine of the host OS for the font. You can also set `Loader` to `inbuilt` to use the font engine built into Hollywood. If you are using TrueType fonts in your project and want your texts to look exactly the same on every platform, you must make sure that you pass `inbuilt` here because otherwise the text look will be different from platform to platform. Another advantage of the inbuilt font loader is that you can directly specify a `*.ttf` file as `font$` without the need of installing the font first on the local system. See Section 54.11 [Font specification], page 1126, for details. Defaults to the loader set using `SetDefaultLoader()`. Keep in mind that if no other default loader has been set using `SetDefaultLoader()`, this will default to `native` for compatibility reasons. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V10.0)

`UserTags:`

This tag can be used to specify additional data that should be passed to font loaders. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

Hollywood also comes with several inbuilt fonts which you can use. You can open these using the following special constants:

`#SANS:` Opens an inbuilt TrueType font without serifs.

`#SERIF:` Opens an inbuilt TrueType font with serifs.

`#MONOSPACE:`

Opens an inbuilt TrueType monospace font (all characters share the same width).

`#BITMAP_DEFAULT:`

Opens the default inbuilt bitmap font. This font is currently only available in size 8, i.e. like the Amiga's default topaz font.

`#TRUETYPE_DEFAULT:`

Opens the default inbuilt TrueType font. This is currently the same as `#SANS`.

Using inbuilt fonts is helpful if you want to make sure your script works on other systems without having to install some fonts first. If you use inbuilt Hollywood fonts only your

script will work immediately out of the box. Note that when you use one of the inbuilt fonts, Hollywood will automatically choose the inbuilt font engine to ensure that the font look is exactly the same on every system.

See Section 54.42 [Working with fonts], page 1157, for more information on using fonts in a platform-independent manner.

INPUTS

`font$` name of the font to load (or one of the special default constants)

`size` desired y size of the font in pixels

`table` optional: table with further options (see above) (V4.7)

EXAMPLE

```
SetFont("times",18)
Print("Hello World")
```

This code sets the font to "times" with size 18 and prints "Hello World".

54.32 SetFontColor

NAME

SetFontColor – change the color of the current font

SYNOPSIS

```
SetFontColor(color)
```

FUNCTION

This function changes the color of the current font to the one specified by `color` which must be an RGB value.

New in Hollywood 2.5: Color can also be an ARGB value for alpha-blended text.

Note that starting with Hollywood 9.0, the color you pass to this function will also be set as the current bullet color. If you would like to use a different color, call the `SetBulletColor()` function.

INPUTS

`color` RGB or ARGB color specification

EXAMPLE

```
SetFontColor(#GRAY)
```

This code sets the font color to some kind of grey.

```
SetFontColor(ARGB(128, #RED))
```

The above code sets the font color to half-red. The background will then shine through the text at a ratio of 50% (128=50% of 255).

54.33 SetFontStyle

NAME

SetFontStyle – change the style of the current font

SYNOPSIS

```
SetFontStyle(style[, t])
```

DEPRECATED SYNTAX

```
SetFontStyle(#SHADOW, color, distance, direction) (V2.5)
SetFontStyle(#BORDER, color, size) (V2.5)
```

FUNCTION

This function changes the current font's style to the one specified by `style` which must be any combination of the following text styles:

`#NORMAL:` Reset the font style to normal. This cannot be combined with any other style flags.

`#BOLD:` Set the font style to bold.

`#ITALIC:` Set the font style to italic.

`#UNDERLINED:`
Set the font style to underlined.

`#ANTIALIAS:`
Set the font style to anti-alias; please note that anti-aliasing is only available for true type fonts (V2.0)

`#SHADOW:` Add a shadow effect to the text. If you set this style, you can pass additional arguments in the optional table argument to control the appearance of the shadow effect. See below for details. (V2.5)

`#BORDER:` Add a border effect to the text. If you set this style, you can pass additional arguments in the optional table argument to control the appearance of the border effect. See below for details. (V2.5)

To combine multiple font styles in a single call simply bit-or them with another, e.g. a call to `SetFontStyle(#BOLD|#ITALIC)` will set the font style to bold and italic. Obviously, the style `#NORMAL` is mutually exclusive and cannot be combined with any other style.

Starting with Hollywood 9.0, `SetFontStyle()` uses a new syntax that accepts an optional table argument that supports the following tags:

`ShadowDir:`
Specifies the direction of the shadow. This must be set to one of Hollywood's directional constants. This tag is only handled when the `#SHADOW` style has been set (see above). (V9.0)

`ShadowColor:`
Specifies the color of the shadow. This must be an ARGB value that can contain a transparency setting. This tag is only handled when the `#SHADOW` style has been set (see above). (V9.0)

`ShadowSize:`
: Specifies the length of the shadow. This tag is only handled when the `#SHADOW` style has been set (see above). (V9.0)

`BorderColor:`
: Specifies the color of the border. This must be an ARGB value that can contain a transparency setting. This tag is only handled when the `#BORDER` style has been set (see above). (V9.0)

`BorderSize:`
: Specifies the size of the border. This tag is only handled when the `#BORDER` style has been set (see above). (V9.0)

Please note that TrueType fonts often have separate font faces for their respective styles. In that case, you should always use these specifically designed font faces because `SetFontStyle()` will create bold and italic styles using a custom algorithm that often does not look as good as hand-crafted bold or italic font faces do. Thus, if you are planning to use Arial in bold style, you should better use `"Arial Bold"` when calling `SetFont()` (or `OpenFont()` / `@FONT`) than using `"Arial"` and calling `SetFontStyle()` with `#BOLD` set afterwards.

INPUTS

`style` special style constant (see list above)

`t` optional: table containing additional arguments (see above) (V9.0)

EXAMPLE

```
SetFontStyle(#BOLD|#ITALIC)
```

The above code sets the font style to bold and italic.

```
SetFontStyle(#SHADOW, {ShadowColor = ARGB(128, $939393),
    ShadowSize = 16, ShadowDir = #SHDWSOUTHEAST})
```

The above code enables a half-transparent grey shadow which will be positioned 16 pixels to the south-east of the main text.

54.34 SetMargins

NAME

SetMargins – define the margins for printed text

SYNOPSIS

```
SetMargins(left, right[, noclip])
```

FUNCTION

This function allows you to define the margins that shall be used for printing text using the `Print()` function. This is very useful if you only want to print text in a specific area of your display. The `left` argument specifies the left end of the margin and the `right` argument is the right end of the margin. Words exceeding those margins will automatically be wrapped to the next line.

The default setting for `left` is 0 and for `right` the default setting is your display width minus 1.

By default, the margins you specify in `left` and `right` will be clipped against the display's boundaries. If you don't want that, you can set the optional `noclip` argument to `True`. If `noclip` is `True`, `left` can also be less than 0 and `right` can be greater than the display width.

INPUTS

`left` left edge of the margin (in pixels)

`right` right edge of the margin (in pixels)

`noclip` optional: set this to `True` if left and right edges of the margin shouldn't be clipped to the display's boundaries (V9.0)

EXAMPLE

```
SetMargins(200, 300)
Print("Hello World. This is my first program using margins.")
```

The above code defines the margins 200 and 300, which means that text output will only be made between pixels 200 and 300. Then it prints some text.

54.35 TransformTextObject

NAME

TransformTextObject – apply affine transformation to text object (V10.0)

SYNOPSIS

```
TransformTextObject(id, sx, rx, ry, sy[, smooth])
```

FUNCTION

This function can be used to apply affine transformation to a text object. You have to pass a 2x2 transformation matrix to this function that will define how each pixel in the text object will be transformed. This function is useful if you want to apply rotation and scaling at the same time. Of course, you could do this with calls to `ScaleTextObject()` and then `RotateTextObject()`, but this would lead to quality losses. If you do the transformation using `TransformTextObject()` instead, everything will be done in a single run.

The 2x2 transformation matrix consists of four floating point factors:

`sx:` Specifies the amount of scaling on the x axis. This must not be zero. If it is negative, the text object is flipped on the y axis.

`rx:` Specifies the amount of rotation on the x axis. This can be 0.

`ry:` Specifies the amount of rotation on the y axis. This can be 0.

`sy:` Specifies the amount of scaling on the y axis. This must not be zero. If it is negative, the text object is flipped on the x axis.

The identity matrix is defined as

```
( 1  0 )
```

```
( 0  1 )
```

If you pass this matrix, then no transformation will be applied because there is no rotation and no scaling defined. I.e. if Hollywood applied this matrix to every pixel in your text object, the result would be just a copy of the text object. But of course, if `TransformTextObject()` detects that you passed an identity matrix, it will return immediately and do nothing.

The optional argument `smooth` can be set to `True` if Hollywood shall use interpolation during the transformation. This yields results that look better but interpolation is quite slow.

Please note: You should always do transformation operations using the original text object. For instance, if you transform text object 1 to 12x8 pixels and then transform it back to 640x480, you will get a messed text object. Therefore you should always keep the original text object and transform only copies of it.

Note that for vector text objects, `TransformTextObject()` will always operate on the untransformed text object. This means that any previous transformations applied to the text object using `TransformTextObject()`, `ScaleTextObject()`, or `RotateTextObject()` will be undone when calling `TransformTextObject()`.

INPUTS

`id`	identifier of the text object to be transformed
`sx`	scale x factor; must never be 0
`rx`	rotate x factor
`ry`	rotate y factor
`sy`	scale y factor; must never be 0
`smooth`	optional: whether or not affine transformation should use interpolation

EXAMPLE

```
angle = Rad(45)    ; convert degrees to radians
TransformTextObject(1, Cos(angle), Sin(angle), -Sin(angle), Cos(angle))
```

The code above rotates text object number 1 by 45 degrees using a 2x2 transformation matrix.

54.36 Text format tags

Since version 2.5 Hollywood is able to do text formatting on-the-fly. The `Print()`, `CreateTextObject()`, and `TextOut()` commands support special format tags that allow you to change the text color and style without calling `SetFontStyle()` or `SetFontColor()`.

The following format tags are currently available:

`[b]:`	Change font style to 'bold'. Use [/b] to cancel 'bold' style.
`[i]:`	Change font style to 'italic'. Use [/i] to cancel 'italic' style.
`[u]:`	Change font style to 'underlined'. Use [/u] to cancel 'underlined' style.

`[shadow=color,size,direction]:`
: Adds a shadow effect to the text. The new shadow will then use the color specified in `color`. It will run for the pixel distance specified in the argument `size` and it will be oriented according to the direction specified in `direction`. Please use any of the 8 directional constants as the `direction` argument. The color can be in RGB or ARGB notation. Shadow transparency is fully supported. Use [/shadow] to cancel 'shadow' style. Note that when palette mode is set to `#PALETTEMODE_PEN` and the text is drawn to a palette target, the `color` argument must not be an RGB color but a palette pen.

`[border=color,size]:`
: Adds a border effect to the text. This border will use the color specified in `color` and the size specified in `size`. The color can be a RGB or ARGB color specification. Border transparency is fully supported. Use [/border] to cancel border style. Note that when palette mode is set to `#PALETTEMODE_PEN` and the text is drawn to a palette target, the `color` argument must not be an RGB color but a palette pen. Before Hollywood 9.0, this tag was known as 'edge'.

`[color=color]:`
: Change font color to `color`. This color can be in RGB or ARGB notation. If you pass an ARGB value, the text will be rendered with blending. Use [/color] to abort rendering in the current color and return to the previously active color.

`[pen=pen]:`
: When palette mode is set to `#PALETTEMODE_PEN` and the text is drawn to a palette target, this tag can be used to change the drawing pen. Use [/pen] to restore the pen that was previously active. (V9.0)

`[bulletcolor=color]:`
: Change bullet color to `color`. Bullets are only used when `TextOut()` is used in list mode. See Section 54.39 [TextOut], page 1149, for details. The color you pass to this tag can be in RGB or ARGB notation. If you pass an ARGB value, the bullet will be rendered with blending. Note that in contrast to all other tags above, this tag must not be closed. It just acts as a directive for the format processor to modify the current bullet color. So you must never close it using [/bulletcolor]. (V9.0)

`[bulletpen=pen]:`
: Change bullet pen to `pen`. Bullets are only used when `TextOut()` is used in list mode. See Section 54.39 [TextOut], page 1149, for details. Note that in contrast to all other tags above, this tag must not be closed. It just acts as a directive for the format processor to modify the current bullet pen. So you must never close it using [/bulletpen]. (V9.0)

Please note that because of these format tags you have to use two square brackets if you want to have a square bracket in your text. If there is only one square bracket Hollywood will always expect a format tag.

Here is an example how you can use these format tags with the commands of the text library:

```
Print("Normal [b]Bold[/b] [i]Italic[/i] [u]Underlined[/u]")
```

As you can see, using format tags is really easy and makes the Hollywood text processor very powerful for advanced text formatting.

54.37 TextExtent

NAME

TextExtent – retrieve detailed information about a text extent (V2.5)

SYNOPSIS

```
extent = TextExtent(string$[, t])
```

DEPRECATED SYNTAX

```
extent = TextExtent(string$[, encoding])
```

FUNCTION

This function returns detailed information about the extent of the specified string with the current font and style settings. Contrary to `TextWidth()` which only returns the cursor advancement `TextExtent()` calculates the exact bounding box for the specified string.

This function returns a table with information in the following fields:

`MinX:` The offset to the left side of the rectangle. This is often negative.

`MinY:` The offset from the baseline to the top of the rectangle. This is always negative.

`MaxX:` The offset to the right side of the rectangle.

`MaxY:` The offset from the baseline to the bottom of the rectangle.

`Width:` This is the same value as returned by `TextWidth()`.

`Height:` The same value as returned by `TextHeight()`.

The values in `MinX`, `MinY`, `MaxX`, and `MaxY` are always relative to the current cursor position. For instance, if `MinX` is -10, this means that `Print()` would start rendering this string -10 pixels from the current cursor position on the x-axis. The value in `Width` specifies where the cursor would end up after the rendering operation. This is often less than `MaxX-1`. For instance in the case of italic text, the last character will usually be much behind the final cursor position.

To calculate the full width of the specified string, simply subtract `MinX` from `MaxX` and add 1, i.e. `full_width=MaxX-MinX+1`.

Starting with Hollywood 10.0, this function accepts an optional table argument that allows you to specify the following additional options:

`Encoding:`
This tag can be used to specify the character encoding used by `string$`. This defaults to the character encoding set as the text library default encoding using `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

`Charspacing:`
Allows you to adjust the space between characters. You can set this to a positive or negative value. A positive value will increase the space between characters, a negative value will decrease it. (V10.0)

INPUTS

`string$` source text

`t` optional: table argument containing further options (see above) (V10.0)

RESULTS

`extent` detailed information about the text's dimensions

54.38 TextHeight

NAME

TextHeight – return the height of a string (V1.5)

SYNOPSIS

```
height = TextHeight(string$)
```

FUNCTION

This function returns the height of the text specified by `string$` if it was rendered on the display. So it takes care of the currently selected font as well as the font style.

INPUTS

`string$` source text

RESULTS

`height` height of the text

EXAMPLE

```
height = TextHeight("Hello World")
pos = (480 - height) / 2
Locate(0, pos)
Print("Hello World")
```

The above code centers the text "Hello World" vertically on a 480 pixel-high display.

54.39 TextOut

NAME

TextOut – draw text to screen

SYNOPSIS

```
TextOut(x, y, text$[, table])
```

FUNCTION

This function outputs the text specified by `text$` at the position specified by coordinates `x` and `y`. This function has the advantage that you can use Hollywood's special constants as the coordinates (e.g. `#CENTER`, `#BOTTOM`...) which is not possible with `Print()` because the `Locate()` function does not handle them.

If layers are enabled, this command will add a new layer of the type `#TEXTOUT` to the layer stack.

Starting with Hollywood 2.5, you can use format tags in the string you pass to `TextOut()`. Using these tags you can control the font style and color of your text on-the-fly. Format tags always start and end with a square bracket ('['). In case you just want to print a square bracket, you will have to use two square brackets. If there is only one square bracket Hollywood will always expect a format tag. See Section 54.36 [Format tags], page 1146, for details.

In Hollywood 4.0 the syntax of this function changed slightly. While the old syntax is still supported for compatibility, new scripts should use the new syntax which accepts a table as argument 4. The table can contain the following elements:

`Align:` Allows you to specify the text's alignment. The following alignments are currently supported:

`#LEFT` Left alignment.

`#RIGHT` Right alignment.

`#CENTER` Center lines.

`#JUSTIFIED`
Lay out text in justified lines. (V7.0)

The default value for `Align` is `#LEFT`.

`WordWrap:`
`TextOut()` can do automatic word-wrapping for you if you specify this additional parameter. You can use this parameter to specify a maximum width for your text. `TextOut()` will then use word wrapping to make sure that no text runs beyond this limit. If you do not set this argument or set it to 0 (which is also the default), the text will be as wide as it is required. Starting with Hollywood 9.1, you can also use soft hyphens or zero-width space characters to customize word wrapping but since these are Unicode characters, you need to make sure that you use UTF-8 encoding in that case.

`Encoding:`
This argument can be used to specify the character encoding inside `text$`. This defaults to the text library default encoding as set by `SetDefaultEncoding()`). See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V4.7)

`Color:` This tag allows you to specify the text color. The color must be provided as an ARGB value. If you do not specify this tag, `TextOut()` will use the color value that was set using the `SetFontColor()` command instead. (V5.0)

`Pen:` When palette mode is `#PALETTEMODE_PEN`, this tag can be used to set the pen that should be used for drawing the text. If palette mode is `#PALETTEMODE_PEN` and `Pen` is not specified, the pen set using `SetDrawPen()` will be used instead. (V9.0)

`Linespacing:`
This tag allows you to adjust the space between lines. You can set this to a positive or negative value. A positive value will increase the space between lines, a negative value will decrease it. (V9.0)

`Charspacing:`
: Allows you to adjust the space between characters. You can set this to a positive or negative value. A positive value will increase the space between characters, a negative value will decrease it. (V10.0)

`Tabs:`
: If you want to use tabs, you need to set this tag to a table containing the desired tab stops. Tab stops must be passed as pixel values relative to the x-position passed to `TextOut()`. If the string you pass to `TextOut()` contains tabs and you don't set this tag, all tabs will be converted to spaces. (V9.0)

`ListMode:`
: Set this tag to `True` to put `TextOut()` into list mode. List mode allows you to create ordered and unordered lists with `TextOut()`. When in list mode, you need to use tabs in the string you pass to `TextOut()` to signal the desired level of indentation. By default, all list items will use the bullet specified in `DefListBullet`. It is also possible to tell `TextOut()` to use custom bullets by setting the `ListBullet` tag. The same is true for the list's indentation, offset and spacing where the default values can be set using `DefListIndent`, `DefListOffset` and `DefListSpacing` and custom values can be set using `ListIndent`, `ListOffset` and `ListSpacing`. Please see below for more details on all these options. Also note that `ListMode` and `Tabs` are mutually exclusive. You cannot use both at the same time. (V9.0)

`DefListBullet:`
: Use this tag to set the default bullet to use when `TextOut()` is in list mode. This can be set to a Unicode character (passed as its numeric codepoint, not as a string!) or one of the following constants:

  `#BULLET_DASH:`
  : Dash bullet. This is the default.

  `#BULLET_CROSS:`
  : Cross bullet.

  `#BULLET_CIRCLE:`
  : Circle bullet.

  `#BULLET_HOLLOWCIRCLE:`
  : Hollow circle bullet. (V9.1)

  `#BULLET_BOX:`
  : Box bullet.

  `#BULLET_CHECKMARK:`
  : Checkmark bullet.

  `#BULLET_ARROW:`
  : Arrow bullet.

  `#BULLET_DIAMOND:`
  : Diamond bullet.

  `#BULLET_NUMERIC:`
  : Numbered list: 1. 2. 3...

`#BULLET_NUMERICSINGLE:`
Numbered list: 1) 2) 3)...

`#BULLET_NUMERICDOUBLE:`
Numbered list: (1) (2) (3)...

`#BULLET_LALPHA:`
Numbered list: a. b. c...

`#BULLET_LALPHASINGLE:`
Numbered list: a) b) c)...

`#BULLET_LALPHADOUBLE:`
Numbered list: (a) (b) (c)...

`#BULLET_UALPHA:`
Numbered list: A. B. C...

`#BULLET_UALPHASINGLE:`
Numbered list: A) B) C)...

`#BULLET_UALPHADOUBLE:`
Numbered list: (A) (B) (C)...

`#BULLET_LROMAN:`
Numbered list: i. ii. iii...

`#BULLET_LROMANSINGLE:`
Numbered list: i) ii) iii)...

`#BULLET_LROMANDOUBLE:`
Numbered list: (i) (ii) (iii)...

`#BULLET_UROMAN:`
Numbered list: I. II. III...

`#BULLET_UROMANSINGLE:`
Numbered list: I) II) III)...

`#BULLET_UROMANDOUBLE:`
Numbered list: (I) (II) (III)...

`#BULLET_NONE:`
No bullet.

If you need to use different types of bullets in your list, you need to use the `ListBullet` tag instead. See below for more information.

Also note that the starting offset of numbered bullet types like `#BULLET_NUMERIC`, `#BULLET_LALPHA`, etc. can be set using the `DefListOffset` and `ListOffset` tags. See below for more information.

By default, bullets will use the color set using `SetFontColor()`. To set a different color, use the `SetBulletColor()` function. See Section 54.29 [SetBulletColor], page 1138, for details. (V9.0)

`ListBullet:`

If you want to use different types of bullets in your list, you have to set this tag to a table that contains the individual bullets your list should use. `TextOut()` will then extract a new bullet type from your table for each new list it starts. Like `DefListBullet`, the individual items in the table you pass to `ListItems` can be either numeric Unicode codepoints or predefined `#BULLET_XXX` constants as described above. If there are more lists than table elements in `ListBullet`, the bullet specified in `DefListBullet` will be used. (V9.0)

`DefListIndent:`

This tag can be used to specify the number of spaces to use for indenting list items. The default indent is 4 which means that by default, list items will be indented using 4 spaces. You can also specify custom levels of indentation for specific lists. This can be done by using the `ListIndent` tag, see below for more details. (V9.0)

`ListIndent:`

If you want to use different levels of indentation in your list, you have to set this tag to a table that contains the individual indentations your list should use. `TextOut()` will then extract a new indentation value from your table for each new list it starts. If there are more lists than table elements in `ListIndent`, the default indentation specified in `DefListIndent` will be used. (V9.0)

`DefListOffset:`

When using a numbered bullet type like `#BULLET_NUMERIC` or `#BULLET_LALPHA`, you can use this tag to specify a starting offset for the numbering. Note that offsets are counted from 0. Thus, specifying an offset of 0 here, will start numbering from 1 for `#BULLET_NUMERIC` and from `"a"` for `#BULLET_ALPHA`. This tag defaults to 0. You can also specify custom offset levels for specific lists. This can be done by using the `ListOffset` tag, see below for more details. (V9.0)

`ListOffset:`

If you want to use different offset levels in your list, you have to set this tag to a table that contains the individual offsets your list should use. `TextOut()` will then extract a new list offset value from your table for each new list it starts. If there are more lists than table elements in `ListOffset`, the default list offset specified in `DefListOffset` will be used. (V9.0)

`DefListSpacing:`

This tag can be used to specify the default spacing to be used between the items in the individual lists. This defaults to 0. You can also specify custom spacing values for specific lists. This can be done by using the `ListSpacing` tag, see below for more details. (V9.1)

`ListSpacing:`

If you want to use different levels of line spacings for your lists, you have to set this tag to a table that contains the individual spacing values your

lists should use. `TextOut()` will then extract a new spacing value from your table for each new list it starts. If there are more lists than table elements in `ListSpacing`, the default spacing value specified in `DefListSpacing` will be used. (V9.1)

`FrameMode:`
When using `TextOut()` in list mode and layers are enabled, the resulting layer of type `#TEXTOUT` will have multiple frames that you can cycle through using `NextFrame()` or all other layer functions that support anim layers. This makes it possible to cycle through the list items one by one or show one item after the other. The first frame will always contain all list items. The content of the other frames will depend on what was specified in the `FrameMode` tag. This tag can be set to the following constants:

`#FRAMEMODE_SINGLE:`
When using this frame mode, only a single list item will be visible per frame, i.e. frame 2 will just contain the first list item, frame 3 will just contain the second list item and so on.

`#FRAMEMODE_FULL:`
When using this frame mode, all previous list items will always be visible as well. This means that frame 3 will contain the first and the second list item, frame 4 will contain the first three list items and so on.

If not specified, `FrameMode` defaults to `#FRAMEMODE_FULL`. Note that you can use the `Frame` tag (see below) to specify which frame should be initially visible. (V9.0)

`Frame:` When using `TextOut()` in list mode and layers are enabled, the resulting layer of type `#TEXTOUT` will have multiple frames that you can cycle through using `NextFrame()` or all other layer functions that support anim layers. This makes it possible to cycle through the list items one by one or show one item after the other. The first frame will always contain all list items. The content of the other frames will depend on what was specified in the `FrameMode` tag (see above). The `Frame` tag can be used to specify the frame that should be initially visible. Frames are counted from 1. (V9.0)

`SimpleList:`
If this is set to `True`, `TextOut()` won't successively extract the list bullet configuration from the `ListBullet` et al. tables but just statically use the item at the specified tab index, i.e. tab position 1 will always use the bullet specified at index 1 in the `ListBullet` table, tab position 2 will use the bullet specified at index 2 in the `ListBullet` table and so on. This will restrict your flexibility but can make things easier if you always want to have the same configuration for each tab position. (V9.1)

Furthermore, the optional table argument can also contain one or more of the standard tags for all drawing commands. See Section 27.17 [Standard draw tags], page 501, for details.

Note that Hollywood currently only supports standard left-to-right based text aligned on horizontal lines. Right to left and vertical text is currently not supported.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color set via `SetFontColor()` or the `Color` tag above.

INPUTS

`x`	x position for the text
`y`	y position for the text
`text$`	string to output
`table`	optional: table containing additional configuration parameters (see above) (V4.0)

EXAMPLE

```
TextOut(#CENTER, #CENTER, "Hello World!")
```

The above code prints "Hello World!" in the center of your display.

```
For Local k = 100 To 600 Step 100 Do
   Line(k, 0, k, 480, #RED)
TextOut(0, 0, "One\tTwo\tThree\tFour\tFive\tSix",
   {Tabs = {100, 200, 300, 400, 500}})
```

This code shows how to use tabs with `TextOut()`.

```
SetFont(#SANS, 18)
TextOut(0, 0, "Pizzas\n"..
    "\tProsciutto\n"..
    "\tFunghi\n"..
    "\tMargarita\n"..
    "Drinks\n"..
    "\tAlcoholic\n"..
    "\t\tBeer\n"..
    "\t\tWine\n"..
    "\tNon-alcoholic\n"..
    "\t\tCoke\n"..
    "\t\tWater",
    {ListMode = True,
    DefListBullet = #BULLET_CIRCLE,
    ListBullet = {#BULLET_DASH}})
```

The code above shows how to create a list with `TextOut()`.

54.40 TextWidth

NAME

TextWidth – return the width of a string

SYNOPSIS

```
width = TextWidth(string$[, t])
```

DEPRECATED SYNTAX

```
width = TextWidth(string$[, encoding])
```

FUNCTION

This function returns the width of the text specified by `string$` if it was rendered on the display. So it takes care of the currently selected font as well as the font style.

Please note: This function returns the cursor advancement of the text. This is often less than the text actually occupies when rendered to the display. If you need detailed information about the real extent of a text, please use the function `TextExtent()` instead.

Starting with Hollywood 10.0, this function accepts an optional table argument that allows you to specify the following additional options:

`Encoding:`
: This tag can be used to specify the character encoding used by `string$`. This defaults to the character encoding set as the text library default encoding using `SetDefaultEncoding()`. See Section 54.30 [SetDefaultEncoding], page 1138, for details. (V10.0)

`Charspacing:`
: Allows you to adjust the space between characters. You can set this to a positive or negative value. A positive value will increase the space between characters, a negative value will decrease it. (V10.0)

INPUTS

`string$` source text

`t` optional: table argument containing further options (see above) (V10.0)

RESULTS

`width` width of the text

EXAMPLE

```
width = TextWidth("Hello World")
pos = (640 - width) / 2
Locate(pos, 0)
Print("Hello World")
```

The above code centers the text "Hello World" horizontally on a 640 pixel-wide display.

54.41 UseFont

NAME

UseFont – change the current font (V4.5)

SYNOPSIS

```
UseFont(id)
```

FUNCTION

This function changes the current font to the font specified by `id`. The font id specified here must have been preloaded by either `OpenFont()` or `@FONT`.

The font style will be reset when calling this command.

INPUTS

`id` identifier of the font to use

EXAMPLE

See Section 54.24 [OpenFont], page 1134.

54.42 Working with fonts

When using Hollywood to compile executables for several platforms, the most common problem is usually the question of what to do with the fonts required by your script. The easiest solution to this problem is to simply link all fonts required by your script into your executable. You can do this by using either the `@FONT` preprocessor command or the `-linkfonts` console argument. However, many fonts are copyrighted and it is not allowed to link them into your executable, so you might want to load fonts manually instead of linking them. How this works depends on the type of the font your script is using. Hollywood supports two font types: Bitmap fonts in AmigaOS format and TrueType fonts. Please read below for information on how to deal with these two font types.

1) Dealing with Amiga bitmap fonts:

Amiga bitmap fonts are natively supported by Hollywood on all platforms. The advantage of Amiga bitmap fonts is that they do not have to be installed first. They can be used immediately. Simply create a `Fonts` subdirectory in the directory of your executable and copy all the Amiga bitmap fonts that your executable needs to this directory. Note that an Amiga bitmap font is not a single file but requires three components:

a. `*.font` descriptor containing information about the font
b. Directory named after the font
c. One or multiple bitmaps containing the raster graphics for the different font sizes

Thus, if you want to use goudyb in size 23 under Windows for example you will require the following files:

```
C:/Program Files/MyProg/MyCoolProgram.exe  ; exe generated by Hollywood
C:/Program Files/MyProg/Fonts/goudyb.font  ; font descriptor
C:/Program Files/MyProg/Fonts/goudyb/23    ; bitmap for size 23
```

On AmigaOS it would look like the following:

```
dh0:Programs/MyProg/MyCoolProgram      ; exe generated by Hollywood
dh0:Programs/MyProg/Fonts/goudyb.font  ; font descriptor
dh0:Programs/MyProg/Fonts/goudyb/23    ; bitmap for size 23
```

On macOS you need to pay attention to the fact that all data files accompanying your program must be put into the `Resources` folder inside the application bundle. So it would look like the following:

```
/Programs/MyProg.app  ; exe generated by Hollywood
```

```
/Programs/MyProg.app/Contents/Resources/Fonts/goudyb.font
/Programs/MyProg.app/Contents/Resources/Fonts/goudyb/23
```

Important note (AmigaOS): Fonts that have an additional `*.otag` file are not bitmap fonts! Fonts that have an accompanying `*.otag` file are usually vector fonts in the TrueType format. TrueType fonts cannot be used by simply copying them to a subdirectory relative to your program. TrueType fonts always have to be installed first! Please see below for more information.

2) Dealing with TrueType fonts:

Working with TrueType fonts is different from working with bitmap fonts in the way that TrueType fonts always have to be installed before you can use them. The only way to use TrueType fonts without installing them is to link them into your executable. However, this is often not possible because of font copyrights. TrueType fonts come as a single file that usually bears the extension `*.ttf`. To install such a `*.ttf` file on your system, you need to do the following:

AmigaOS3/MorphOS/AROS:
: Use the program FTManager. Note that FTManager by default uses a pretty awkward font name which you should change if you plan to compile your script for multiple platforms. See Section 54.11 [Font specification], page 1126, for details.

AmigaOS4:
: Use TypeManager in `SYS:System`.

Windows and macOS:
: Simply double-click the `*.ttf` file and click on Install.

Once you have installed the new font, it is ready for use by Hollywood.

55 Time library

55.1 CompareDates

NAME

CompareDates – compare two date strings (V4.5)

SYNOPSIS

```
result = CompareDates(date1$, date2$[, notime])
```

FUNCTION

This function can be used to compare the time of two date strings and return their relation. Both date strings must be in the default time notation used by Hollywood:

```
dd-mmm-yyyy hh:mm:ss
```

The `mmm` constituent is a string with three characters identifying the month. This can be `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`.

If you set the optional argument `notime` to `True`, only dates are compared. In that case, the two strings you pass to `CompareDates()` must not contain any time specifications.

The return value of `CompareDates()` indicates how the two dates are related. The following return values are possible:

0: `date1$` and `date$` have exactly the same time

1: `date1$` is later in time than `date2$`

2: `date1$` is earlier in time than `date2$`

INPUTS

`date1$` date string in the Hollywood date notation

`date2$` date string in the Hollywood date notation

`notime` optional: `True` to compare dates only (defaults to `False`)

RESULTS

`result` result of comparison

EXAMPLE

```
NPrint(CompareDates("10-Dec-2009 13:34:12", "09-Dec-2009 15:36:21"))
NPrint(CompareDates("12-Dec-2009 23:59:59", "13-Dec-2009 00:00:00"))
NPrint(CompareDates("24-Dec-2009 20:00:00", "24-Dec-2009 20:00:00"))
```

The code above will do three date comparisons. The results will be: 1,2,0

55.2 DateToTimestamp

NAME

DateToTimestamp – convert local date to timestamp (V7.1)

SYNOPSIS

```
s = DateToTimestamp(d$[, isdst])
```

FUNCTION

This function can be used to get the timestamp for the date passed in `d$`. This string must be in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 55.1 [CompareDates], page 1159, for details.

Note that the date that you pass to this function is interpreted as local time whereas the timestamp returned starts from UTC time, i.e. from the Unix epoch which starts on January 1st, 1970, 00:00:00 UTC. This means that passing `01-Jan-1970 00:00:00` will only return 0 if the local timezone is identical to the UTC timezone. On systems east of UTC, passing the `01-Jan-1970 00:00:00` will lead to an error because January 1st, 1970, 00:00:00, east of UTC means December 31st, 1969 UTC which cannot be represented in the Unix epoch.

The optional argument `isdst` specifies whether or not daylight saving time is active at the specified date. Normally, you don't have to specify this argument because Hollywood will automatically query this information from the timezone database. It is only necessary to pass this information in case the specified time is ambiguous, i.e. when switching from daylight saving time back to standard time, a certain period of time (typically an hour) is repeated in the night. In Germany, for example, clocks are set back from 3am to 2am when switching from daylight saving time to standard time. This means that the hour between 2am and 3am happens twice: Once in daylight saving time, once in standard time. The `isdst` argument allows you to specify which hour you are referring to.

To convert a timestamp back into a date, use the `TimestampToDate()` function. See Section 55.20 [TimestampToDate], page 1170, for details.

INPUTS

`d$`	Hollywood date to convert to the timestamp format
`isdst`	optional: whether or not daylight saving time is active at the specified date (defaults to -1 which means that this information should be retrieved from the local timezone database)

RESULTS

`s`	time in seconds that has elapsed since the Unix epoch or -1 if the specified date cannot be represented in Unix time

55.3 DateToUTC

NAME

DateToUTC – convert local date to UTC (V7.1)

SYNOPSIS

```
u$ = DateToUTC(d$[, isdst])
```

FUNCTION

This function can be used to convert the local date passed in `d$` to a UTC date. The `d$` parameter must be in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 55.1 [CompareDates], page 1159, for details.

The optional argument `isdst` specifies whether or not daylight saving time is active at the specified date. Normally, you don't have to specify this argument because Hollywood will

automatically query this information from the timezone database. It is only necessary to pass this information in case the specified time is ambiguous, i.e. when switching from daylight saving time back to standard time, a certain period of time (typically an hour) is repeated in the night. In Germany, for example, clocks are set back from 3am to 2am when switching from daylight saving time to standard time. This means that the hour between 2am and 3am happens twice: Once in daylight saving time, once in standard time. The `isdst` argument allows you to specify which hour you are referring to.

To convert a UTC date back into a local date, use the `UTCToDate()` function. See Section 55.21 [UTCToDate], page 1171, for details.

INPUTS

`d$` local date to convert to UTC date

`isdst` optional: whether or not daylight saving time is active at the specified date (defaults to -1 which means that this information should be retrieved from the local timezone database)

RESULTS

`u$` UTC equivalent of the local date argument

55.4 GetDate

NAME

GetDate – get current date

SYNOPSIS

```
date$ = GetDate([type])
```

FUNCTION

This function can be used to query the current system date and time. Date and time can be returned in various formats depending on the value passed in `type`.

The following formats are currently recognized by `type`:

`#DATELOCALNATIVE:`

This is the default format. If `type` is omitted, `GetDate()` will fall back to this type. `#DATELOCALNATIVE` will return the date in the system's language. For example, on a German system September 4th 2002 will be returned as `"04.09.02"` but on a system in the USA it would be `"09.04.02"`. Note that the time isn't returned at all for this type.

`#DATELOCAL:`

This will return the date in Hollywood's standard date and time format. It looks like the following:

```
dd-mmm-yyyy hh:mm:ss
```

September 4th 2002 at 3.16pm and 23 seconds would look like this in the default Hollywood notation:

```
04-Sep-2002 15:16:23
```

This notation is also used by other Hollywood commands, for example by the following commands: `GetFileAttributes()`, `SetFileAttributes()`, `FileAttributes()`, and `CompareDates()`.

Note that even though `#DATELOCAL` is the most common type for this function, it is not the default due to historic purposes. `#DATELOCALNATIVE` is the default type. (V4.5)

`#DATEUTC:`

If you pass `#DATEUTC` for `type`, `GetDate()` will return the current UTC date and time. The UTC date and time will be passed in Hollywood's default date and time notation (see above). (V7.1)

INPUTS

`type` date and time format that should be used (V4.5)

RESULTS

`date$` current date and time in the desired format

55.5 GetDateNum

NAME

GetDateNum – get date information as a value

SYNOPSIS

```
info = GetDateNum(type)
```

FUNCTION

This function allows you to retrieve date information as a value from Hollywood. The following constants can be specified as `type`:

`#DATEDAY:`

Returns the day of the month (1-31)

`#DATEMONTH:`

Returns the month (1-12)

`#DATETIME:`

Returns the time (00hhmmss)

`#DATEYEAR:`

Returns the year (yyyy)

INPUTS

`type` one of the constants as listed above

RESULTS

`info` day, month, time or year information

55.6 GetTime

NAME
GetTime – get the current time

SYNOPSIS
```
time$ = GetTime([secs])
```

FUNCTION
This function gets the current time at returns it to `time$`. If the optional argument `secs` is `True`, Hollywood will also add the seconds to the time string.

INPUTS

`secs`	optional: set this to `True` if you want to retrieve the seconds also

RESULTS

`time$`	current time as a string

55.7 GetTimer

NAME
GetTimer – get a timer's state

SYNOPSIS
```
time = GetTimer(id)
```

FUNCTION
This function returns the timer's state, which is the time that has passed since the timer was started with `StartTimer()`. The time is returned in milliseconds.

INPUTS

`id`	identifier of the timer to query for its state

RESULTS

`time`	number of milliseconds that have passed since the timer was started (with respect to `PauseTimer()` and `ResumeTimer()`)

EXAMPLE
See Section 55.17 [StartTimer], page 1168.

55.8 GetTimestamp

NAME
GetTimestamp – get timestamp (V7.0)

SYNOPSIS
```
s = GetTimestamp([type])
```

FUNCTION

This function returns a timestamp. The time is returned in seconds as a fractional number, allowing for sufficient precision. The `type` parameter allows you to specify what kind of timestamp you'd like to get. This can be one of the following:

`#TIMESTAMP_START:`
: Return the time in seconds since Hollywood was started. This is the default.

`#TIMESTAMP_UNIX:`
: Return the time that has elapsed since the Unix epoch which started on January 1st, 1970, 00:00:00 UTC. Note that this depends on the system clock so there could be problems if the system clock is changed while your script is running. If you want to be independent of the system clock, use `#TIMESTAMP_RAW` instead (see below).

`#TIMESTAMP_RAW:`
: Return a raw clock value that is independent of the system clock and is monotonically increasing. This can be useful because `#TIMESTAMP_UNIX` depends on the system clock because it returns the number of seconds since January 1st, 1970 so you could be in trouble in case the system clock is changed between two `GetTimestamp()` calls. (V9.1)

`GetTimestamp()` is especially useful in connection with Hollywood's event handler. All event messages will contain a field named `Timestamp` which contains the timestamp the event was generated. If you compare this time stamp against the return value of `GetTimestamp()`, you can filter out very old events, for example. See Section 29.13 [InstallEventHandler], page 553, for details.

To convert a timestamp into a date, you can use the `TimestampToDate()` function. To convert a date into a timestamp, use the `DateToTimestamp()` function.

INPUTS

`type` optional: the kind of timestamp to get; see above for possible types (defaults to `#TIMESTAMP_START`)

RESULTS

`s` timestamp in seconds as a fractional number

55.9 GetTimeZone

NAME

GetTimeZone – get time zone information (V7.1)

SYNOPSIS

```
off, dst = GetTimeZone()
```

FUNCTION

This function can be used to obtain information about the time zone the host system is in. It will return two values: `off` will be set to the number of minutes of this computer's time from UTC and `dst` will be a boolean value that specifies whether or not daylight saving time is currently active in the host system's time zone.

Note that `off` will be negative if the host system is east of UTC and positive if it is west of UTC.

INPUTS

none

RESULTS

`off` offset in minutes from UTC

`dst` `True` if daylight saving time is currently active, `False` otherwise

EXAMPLE

```
Print(GetTimeZone())
```

When run in January on a computer in Germany, this will print `"-60"` and `"0"` because there is no daylight saving time in Germany in January and CET is 60 minutes ahead of UTC in winter.

55.10 GetWeekday

NAME

GetWeekday – get the weekday

SYNOPSIS

```
day$ = GetWeekday()
```

FUNCTION

This function returns the weekday to the string `day$`. Note that the weekday will be returned in the user's native language (depending on his locale settings).

INPUTS

none

RESULTS

`day$` current weekday

55.11 MakeDate

NAME

MakeDate – make Hollywood date from components (V7.1)

SYNOPSIS

```
d$ = MakeDate(t)
```

FUNCTION

This function composes a Hollywood date from a set of individual date components which have to be passed in the table `t`. The date that is returned by this function will be in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 51.15 [CompareStr], page 1032, for details.

You have to pass a table to `MakeDate()` that has the following fields initialized:

`MDay:` Day of the month (1-31).

`Mon:` Month of the year (1-12).

`Year:` Number of the year (e.g. 2018).

`Hour:` Hours since midnight (0-23).

`Min:` Minutes after the hour (0-59).

`Sec:` Seconds after the minute (0-59).

To break down a Hollywood date into its individual components, use the `ParseDate()` function. See Section 55.12 [ParseDate], page 1166, for details.

INPUTS

`t` table describing the date to compose (see above)

RESULTS

`d$` date string in Hollywood's standard date format

55.12 ParseDate

NAME

ParseDate – break down Hollywood date into components (V7.1)

SYNOPSIS

```
t = ParseDate(d$)
```

FUNCTION

This function parses the Hollywood date passed in `d$` and breaks it down into its individual components. Those components are then returned in a table. The date string passed to this function must be in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 51.15 [CompareStr], page 1032, for details.

`ParseDate()` will return a table with the following fields initialized:

`MDay:` Day of the month (1-31).

`Mon:` Month of the year (1-12).

`Year:` Number of the year (e.g. 2018).

`Hour:` Hours since midnight (0-23).

`Min:` Minutes after the hour (0-59).

`Sec:` Seconds after the minute (0-59).

`WDay:` Days since Sunday (0-6).

`YDay:` Days since January 1 (0-365).

To compose a Hollywood date from its individual components, use the `MakeDate()` function. See Section 55.11 [MakeDate], page 1165, for details.

INPUTS

`d$` Hollywood date to decompose

RESULTS

`t` table containing individual date components (see above)

55.13 PauseTimer

NAME

PauseTimer – pause a timer

SYNOPSIS

```
PauseTimer(id)
```

FUNCTION

This function pauses the timer specified by `id`. When a timer is paused, it does not count the time but you can still retrieve the timer's state through `GetTimer()`. The timer can be resumed by calling `ResumeTimer()`.

INPUTS

`id`	identifier of the timer to pause

EXAMPLE

```
none
```

55.14 ResetTimer

NAME

ResetTimer – reset a timer (V4.5)

SYNOPSIS

```
ResetTimer(id[, time])
```

FUNCTION

You can use this function to reset an existing timer to zero or to a specified time. If you want to reset the timer to zero, simply leave out the second argument. Otherwise use the second argument to specify a time in milliseconds for the timer.

Using `ResetTimer()` to clear a timer is generally faster than starting a new using `StartTimer()`, so you should use `ResetTimer()` if you can.

INPUTS

`id`	identifier of the timer that shall be reset
`time`	optional: desired time value for the timer in milliseconds (defaults to 0 which means no time)

EXAMPLE

```
StartTimer(1)
Wait(1000, #MILLISECONDS)
Print(GetTimer(1))
ResetTimer(1, 2000)
Print(GetTimer(1))
```

The code above will start a new timer 1, wait a second and then print the state of the timer which should be 1000 or a few milliseconds more. Then the timer state is set to 2000 milliseconds and printed again. This time it should be 2000 or a few milliseconds more.

55.15 ResumeTimer

NAME
ResumeTimer – resume a paused timer

SYNOPSIS
`ResumeTimer(id)`

FUNCTION
This function resumes the timer specified by `id`. The timer must be in pause mode (set by `PauseTimer()`). If `ResumeTimer()` was successful, the timer resumes counting the time.

INPUTS

`id`	identifier of the timer to resume

EXAMPLE
`none`

55.16 SetTimerElapse

NAME
SetTimerElapse – set timer elapse threshold (V9.0)

SYNOPSIS
`SetTimerElapse(id, elapse[, reset])`

FUNCTION
This function sets the elapse threshold of the timer specified by `id` to the time specified in `elapse`. This time must be specified in milliseconds. You can then call `TimerElapsed()` to find out when the timer has elapsed or use `WaitTimer()` to wait for the timer to elapse.

By default, `SetTimerElapse()` will also reset the timer. If you don't want that, pass `False` in the `reset` argument.

Note that if you pass 0 in the `elapse` argument, elapsing will be disabled for this timer, i.e. `TimerElapsed()` will never return `True` for timers which have an elapse threshold of 0.

INPUTS

`id`	identifier of the timer to modify
`elapse`	elapse threshold in milliseconds or 0 to disable elapsing
`reset`	optional: whether or not the timer should be reset (defaults to `True`)

55.17 StartTimer

NAME
StartTimer – start a new timer

SYNOPSIS

```
[id] = StartTimer(id[, elapse])
```

FUNCTION

This function creates a new timer and assigns the identifier `id` to it. If you pass `Nil` in `id`, `StartTimer()` will automatically choose an identifier and return it. This timer will run until you call `PauseTimer()` or `StopTimer()`. You can retrieve the current state of the timer by calling `GetTimer()`.

Starting with Hollywood 9.0, there is a new optional `elapse` argument. If you set this to a time in milliseconds, `TimerElapsed()` will return `True` as soon as the timer has been running for the specified amount of time. Alternatively, you can also use `WaitTimer()` to wait for a timer to elapse. Finally, the timer's elapse value can also be set or modified using `SetTimerElapse()`. See Section 55.16 [SetTimerElapse], page 1168, for details.

INPUTS

`id`	id for your timer or `Nil` for auto id selection
`elapse`	optional: number of milliseconds until the timer should elapse (defaults to 0 which means that it will never elapse) (V9.0)

RESULTS

`id`	optional: identifier of the timer; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
StartTimer(1)
Wait(200)
t = GetTimer(1)
Print(t)
```

The above code starts a new timer, waits 4 seconds and retrieves the timer state. The timer state is copied to the variable `t` and should have the value of about 4000 milliseconds.

55.18 StopTimer

NAME

StopTimer – stop a timer

SYNOPSIS

```
StopTimer(id)
```

FUNCTION

This function stops the timer specified by `id`. If you stop a timer it will be completely removed from the system, therefore you cannot resume it. If you want to pause a timer, please use `PauseTimer()` instead.

INPUTS

`id`	identifier of the timer that shall be stopped

EXAMPLE
See Section 55.17 [StartTimer], page 1168.

55.19 TimerElapsed

NAME
TimerElapsed – check if timer has elapsed (V9.0)

SYNOPSIS
```
elapsed = TimerElapsed(id[, reset])
```

FUNCTION
This function checks if the timer specified by `id` has elapsed and returns `True` if it has, `False` otherwise. By default, the timer will be reset to 0 when it has elapsed. If you don't want that, pass `False` in the `reset` argument.

The threshold when a timer elapses can be set either when creating a timer using `StartTimer()` or later using `SetTimerElapse()`.

INPUTS

`id`	identifier of the timer to examine
`reset`	optional: `True` if the elapsed timers should be reset to 0, `False` otherwise (defaults to `True`)

RESULTS

`elapse`	`True` if the timer has elapsed, `False` otherwise

EXAMPLE
```
StartTimer(1, 10000)
Repeat
   VWait
Until TimerElapsed(1) = True
```

55.20 TimestampToDate

NAME
TimestampToDate – convert timestamp to date (V7.1)

SYNOPSIS
```
d$ = TimestampToDate(s[, unixtime])
```

FUNCTION
This function can be used to convert a timestamp into a date string in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 51.15 [CompareStr], page 1032, for details. The optional argument `unixtime` specifies whether or not the timestamp is measured from the beginning of the Unix epoch (i.e. January 1st, 1970, 00:00:00 UTC) or from the time when Hollywood was started. By default, timestamps are measured from the time when Hollywood was started.

To convert a date back into a timestamp, use the `DateToTimestamp()` function. See Section 55.2 [DateToTimestamp], page 1159, for details.

INPUTS

`s` timestamp to convert into a date

`unixtime` optional: if set to `True` the timestamp is interpreted as relative to the beginning of the Unix epoch; otherwise it is relative to the time when Hollywood was started (defaults to `False`)

RESULTS

`d$` date in Hollywood's standard date format

55.21 UTCToDate

NAME

UTCToDate – convert UTC date to local date (V7.1)

SYNOPSIS

```
d$ = UTCToDate(u$)
```

FUNCTION

This function can be used to convert the UTC date passed in `u$` to a local date. The `u$` parameter must be in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 51.15 [CompareStr], page 1032, for details.

To convert a local date back into a UTC date, use the `DateToUTC()` function. See Section 55.3 [DateToUTC], page 1160, for details.

INPUTS

`u$` UTC date to convert to local date

RESULTS

`d$` local date equivalent of the UTC date

55.22 ValidateDate

NAME

ValidateDate – check if date is valid (V7.1)

SYNOPSIS

```
b = ValidateDate(d$)
```

FUNCTION

This function can be used to check if the date given in `d$` is valid. `ValidateDate()` makes sure that all individual date and time components are within their valid ranges, e.g. February 29th is only a valid date in leap years. The `d$` parameter must be in Hollywood's standard date format, i.e. `dd-mmm-yyyy hh:mm:ss`. See Section 51.15 [CompareStr], page 1032, for details.

INPUTS

`d$` date to validate

RESULTS

`b` `True` if date is valid, `False` otherwise

55.23 WaitTimer

NAME

WaitTimer – wait until a timer has reached a certain time (V2.0)

SYNOPSIS

```
WaitTimer(id[, time, reset])
t = WaitTimer(table[, reset]) (V9.0)
```

FUNCTION

This function waits until the timer specified by `id` has been running for the time specified in the `time` argument. This time must be specified in milliseconds. If you omit the `time` argument or set it to -1, `WaitTimer()` will wait until the timer has reached its elapse threshold set using `SetTimerElapse()` or using `StartTimer()`.

Before this function returns it will also reset the specified timer so that you can easily use this function in a loop. You can change this behaviour by setting the optional argument `reset` to `False`. In that case, the timer will not be reset.

Starting with Hollywood 9.0, there is an alternative way of using `WaitTimer()`. Instead of the identifier of a single timer, you can also pass a table containing multiple timer identifiers. In that case, `WaitTimer()` will wait until at least one of the timers from the list specified in the `table` argument has elapsed. Once that happens, `WaitTimer()` will return a table to you. That table will be a list containing the identifiers of all timers that have elapsed. If the `reset` argument is `True`, which is also the default, all elapsed timers will be reset by `WaitTimer()`. Note that the table you pass to `WaitTimer()` can also be empty. In that case, `WaitTimer()` will simply wait for a timer to elapse from all timers that are currently running.

`WaitTimer()` can be very useful to throttle the execution of loops so that they don't consume all of the CPU. For instance, if you have a loop that moves a sprite from the left to the right boundary of the display, you should add some kind of throttle because it doesn't make sense to update the screen more often than the monitor refreshes. This is very important. Even if the script runs at perfect speed without `WaitTimer()` you should not forget that there are faster machines than yours. Using `WaitTimer()` in your loops will make sure that your application runs at the same speed on every platform.

See Section 15.3 [Script timing], page 161, for details.

INPUTS

`id` syntax 1: identifier of the timer to query

`time` syntax 1, optional: time in milliseconds that the timer must have (defaults to timer's elapse threshold)

`table` syntax 2: pass a table containing a list of timers here and `WaitTimer()` will return as soon as a timer from the list has elapsed (see above); if you pass an empty table, all running timers will be taken into account

`reset` optional: specifies whether or not the timer shall be reset after `WaitTimer()` returns (defaults to `True` which means that the timer will be reset)

RESULTS

`t` syntax 2: a list of the timers that have elapsed; this will only be returned if you pass a table instead of a timer identifier to `WaitTimer()` (see above for details)

EXAMPLE

```
StartTimer(1)
For k = 0 To 640
   DisplaySprite(1, k, 0)
   WaitTimer(1, 40)
Next
```

The above code scrolls sprite 1 from left to right. After each call to DisplaySprite(), `WaitTimer()` is used to ensure that we wait at least 40 milliseconds until the next DisplaySprite(). Thus, this loop will not be executed more than 25 times a second because 40 * 25 = 1000.

56 Vectorgraphics library

56.1 AddArcToPath

NAME

AddArcToPath – add elliptical arc to path (V5.0)

SYNOPSIS

```
AddArcToPath(id, xc, yc, ra, rb, start, end[, clockwise])
```

FUNCTION

This function adds an elliptical arc to the path specified in the first argument. You have to provide the center point of the arc in the `xc` and `yc` arguments. The arc's radii have to be passed in `ra` and `rb`, and the start and end angles have to be specified in the `start` and `end` arguments respectively. All angles must be specified in degrees. If you want to have a closed ellipse, the start argument needs to be 0 and the end argument needs to be 360. Using the `AddEllipseToPath()` command is of course easier in this case. The optional argument `clockwise` can be used to specify whether or not the elliptical arc shall be drawn in clockwise direction. This tag defaults to `True` which means clockwise drawing. If you set it to `False`, `AddArcToPath()` will connect the angles in anti- clockwise direction.

Note that `AddArcToPath()` doesn't add a center point vertex. If you want the start and end angles of the arc to be connected with the center point, you need to do this manually by calling `MoveTo()` before `AddArcToPath()` and `LineTo()` afterwards.

Also note that `AddArcToPath()` only starts a new subpath in case there is no active subpath. Otherwise it will simply connect its vertices to the currently active subpath. If you don't want this, you'll have to manually open a new subpath before calling `AddArcToPath()`. Furthermore, `AddArcToPath()` also won't close the active subpath when it is finished.

INPUTS

`id`	identifier of path to which the elliptical arc should be added
`xc`	x center point of arc
`yc`	y center point of arc
`ra`	arc radius on the x axis
`rb`	arc radius on the y axis
`start`	start angle in degrees (must be positive)
`end`	end angle in degrees (must be positive)
`clockwise`	optional: whether or not the angles should be connected in clockwise direction (defaults to `True` which means clockwise)

56.2 AddBoxToPath

NAME

AddBoxToPath – add rectangle to path (V5.0)

SYNOPSIS

```
AddBoxToPath(id, x, y, width, height[, table])
```

FUNCTION

This function adds a rectangle to the path specified in `id`. You have to provide the position and size of the rectangle in the other arguments. `AddBoxToPath()` will start a new subpath for the rectangle and close it.

The optional argument `table` allows you to configure the rectangle's style. The following tags are currently recognized:

`RoundLevel:`
: You can specify this tag to create a rectangle with rounded corners. You need to pass a round level in percentage which specifies how round the corners will be (possible values are 0 to 100). Defaults to 0 which means no round corners.

`CornerA, CornerB, CornerC, CornerD:`
: These four tags allow you to fine-tune the corner rounding of the rectangle. You can specify a rounding level (0 to 100) for every corner of the rectangle thus allowing you to create a rectangle where not all corners are rounded, or where the different corners use different rounding levels. These tags will override any setting specified in the `RoundLevel` tag.

INPUTS

`id`	identifier of path to which the rectangle should be added
`x`	x position of the rectangle
`y`	y position of the rectangle
`width`	rectangle width
`height`	rectangle height
`table`	optional: table containing further options (see above)

56.3 AddCircleToPath

NAME

AddCircleToPath – add circle to path (V5.0)

SYNOPSIS

```
AddCircleToPath(id, xc, yc, radius)
```

FUNCTION

This function adds a circle to the path specified in the first argument. You have to provide the center point of the circle in the `xc` and `yc` arguments. The circle's radius has to be passed in the fourth argument.

Note that `AddCircleToPath()` only starts a new subpath in case there is no active subpath. Otherwise it will simply connect its vertices to the currently active subpath. If you don't want this, you'll have to manually open a new subpath before calling `AddCircleToPath()`. Furthermore, `AddCircleToPath()` also won't close the active subpath when it is finished.

INPUTS

`id`	identifier of path to which the circle should be added
`xc`	x center point of circle
`yc`	y center point of circle
`radius`	circle's radius

56.4 AddEllipseToPath

NAME

AddEllipseToPath – add ellipse to path (V5.0)

SYNOPSIS

```
AddEllipseToPath(id, xc, yc, ra, rb)
```

FUNCTION

This function adds an ellipse to the path specified in the first argument. You have to provide the center point of the ellipse in the `xc` and `yc` arguments. The ellipse's radii have to be passed in `ra` and `rb`.

Note that `AddEllipseToPath()` only starts a new subpath in case there is no active subpath. Otherwise it will simply connect its vertices to the currently active subpath. If you don't want this, you'll have to manually open a new subpath before calling `AddEllipseToPath()`. Furthermore, `AddEllipseToPath()` also won't close the active subpath when it is finished.

INPUTS

`id`	identifier of path to which the ellipse should be added
`xc`	x center point of ellipse
`yc`	y center point of ellipse
`ra`	ellipse radius on the x axis
`rb`	ellipse radius on the y axis

56.5 AddTextToPath

NAME

AddTextToPath – add vector text to path (V5.0)

SYNOPSIS

```
AddTextToPath(id, t$[, table])
```

FUNCTION

This function adds the text specified in `t$` to the current path. The text will be added to the path as vector graphics. Because of that, this command will only work when a vector font (e.g. a TrueType font) is currently active. It will not work with bitmap fonts. `AddTextToPath()` will use the active font that was set by the last call to `SetFont()` or `UseFont()`.

Please note that `AddTextToPath()` will add the text above the current y-point. Thus, if the path's current y-point is 240 and you add text to it that is 36 pixels high, the text will be placed at an y-position of 204 (240-36=204) instead of 240 as the current y-point might suggest.

Please note that there are currently some restrictions:

- The font to be used with `AddTextToPath()` must have been opened using `#FONTENGINE_INBUILT`. Fonts that have been opened using `#FONTENGINE_NATIVE` will currently not work. The default inbuilt fonts `#SANS`, `#SERIF`, and `#MONOSPACE` will work just fine with `AddTextToPath()`.
- Any styles set by `SetFontStyle()` will be ignored by `AddTextToPath()`. If you want italic or bold text, you need to open a separate font that has the bold and/or italicized vector graphics already in its face data. Simply calling `SetFontStyle()` does not work with vector text.
- Similarly, all formatting tags that are recognized by `Print()`, `TextOut()`, and `CreateTextObject()` are ignored by `AddTextToPath()`. It is currently not possible to use formatting tags with this command.
- Newline characters ('\n') are also ignored by this function.

The optional argument `table` allows you to specify further options. The following tags are currently recognized:

`Encoding:`
: This argument can be used to specify the character encoding inside `t$`. This defaults to the text library default encoding set by `SetDefaultEncoding()`). See Section 54.30 [SetDefaultEncoding], page 1138, for details.

INPUTS

`id`	identifier of path to which the text should be added
`t$`	text to add to path
`table`	optional: table containing further arguments (see above)

EXAMPLE

```
EnableLayers
SetFillStyle(#FILLCOLOR)
SetFormStyle(#ANTIALIAS)
SetFont("Arial", 100, {Engine = #FONTENGINE_INBUILT})
StartPath(1)
MoveTo(1, 0, 0)
AddTextToPath(1, "Hello World")
DrawPath(1, #CENTER, #CENTER + 100, #BLUE, {AnchorX = 0.5,
    AnchorY = 0.5, Rotate = 45})
```

The code above creates a vector path containing the text "Hello World". The path is then drawn using rotation by 45 degrees.

56.6 AppendPath

NAME

AppendPath – append path to another path (V5.0)

SYNOPSIS

```
AppendPath(id, src)
```

FUNCTION

This function appends the path specified in `src` to the end of the path specified in `id`. All path data is copied so that you can free the `src` path after the append operation.

INPUTS

`id`	identifier of path that shall be modified
`src`	identifier of path that shall be appended

56.7 ClearPath

NAME

ClearPath – remove all vertices from a path (V5.0)

SYNOPSIS

```
ClearPath(id)
```

FUNCTION

This command will clear the specified path completely. All path vertices and all sub-paths will be removed. However, the path itself will not be freed. Thus, you can start adding vertices to it again after this call. If you want to free a path completely, use `FreePath()` instead.

INPUTS

`id`	identifier of path to clear

56.8 ClosePath

NAME

ClosePath – close current sub-path (V5.0)

SYNOPSIS

```
ClosePath(id)
```

FUNCTION

This command will close the current sub-path in the path specified by `id`. The sub-path is closed by adding a line to the starting point of the current sub-path. If you want to add new vertices after calling `ClosePath()`, you have to call `StartSubPath()` and `MoveTo()` to create a new sub-path.

INPUTS

`id`	identifier of path to close

56.9 CopyPath

NAME

CopyPath – clone a path (V5.0)

SYNOPSIS

```
[id] = CopyPath(src, dst)
```

FUNCTION

This command clones the path specified in `src` and creates a new path under the identifier `dst` that is an exact copy of the path specified in argument 1. The `dst` argument can either be an identifier that should be used for the new path or it can be `Nil`, in which case `CopyPath()` will automatically select an identifier and return it to you.

INPUTS

`src`	identifier of path to clone
`dst`	identifier for the new path or `Nil` for auto id selection

RESULTS

`id`	optional: identifier of the new path; this will only be returned when you pass `Nil` as argument 2 (see above)

56.10 CurveTo

NAME

CurveTo – add curve to path (V5.0)

SYNOPSIS

```
CurveTo(id, x1, y1, x2, y2, x3, y3)
```

FUNCTION

This command will add a cubic Bézier spline curve to the path. The curve will run from the current point to the position specified in `x3`,`y3`. The control points of the spline curve have to specified in arguments `x1`,`y1` and `x2`,`y2`. When `CurveTo()` returns, the path's current point will be at `x3`,`y3`.

INPUTS

`id`	identifier of path to add curve to
`x1`	x coordinate of control point `#1`
`y1`	y coordinate of control point `#1`
`x2`	x coordinate of control point `#2`
`y2`	y coordinate of control point `#2`

`x3`	x coordinate of curve destination point
`y3`	y coordinate of curve destination point

56.11 DrawPath

NAME

DrawPath – draw vector path (V5.0)

SYNOPSIS

```
DrawPath(id, x, y[[, color], table])
```

FUNCTION

This draws the vector path specified in `id` to the position specified in `x` and `y` using the color that is passed in argument 4. The vector path will be drawn using the form style specified using `SetFormStyle()` and it will be filled according to the configuration selected using `SetFillStyle()` and `SetFillRule()`. If you are drawing the vector path in outline mode (i.e. fill style is set to `#FILLNONE`), then `DrawPath()` will also take the settings of `SetLineJoin()`, `SetLineCap()`, and `SetDash()` into account. `Color` can either be an RGB value or an ARGB value for alpha-blended drawing.

The optional table argument can be used to specify one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

If layers are enabled, this command will add a new layer of the type `#VECTORPATH` to the layer stack.

Note that `DrawPath()` only allows you to use a single color for per path. If you want to use multi-colored paths, you can use the `PathToBrush()` function to combine multiple paths inside a single vector brush object. See Section 56.26 [PathToBrush], page 1191, for details.

Note that when drawing to a palette-based target and the palette mode is set to `#PALETTEMODE_PEN`, this function will draw using the pen set via `SetDrawPen()` instead of the color passed to the function.

INPUTS

`id`	identifier of path object to draw
`x`	destination x offset
`y`	destination y offset
`color`	optional: RGB or ARGB color (defaults to `#BLACK`) color is optional because it is not required when you draw to a mask or alpha channel
`table`	optional: table containing further arguments; can be any from the ones listed above and from the standard tags

EXAMPLE

```
SetFillStyle(#FILLNONE)
SetFormStyle(#ANTIALIAS)
```

```
x=25.6 y=128.0
x1=102.4 y1=230.4
x2=153.6 y2=25.6
x3=230.4 y3=128.0

StartPath(1)
MoveTo(1, x, y)
CurveTo(1, x1, y1, x2, y2, x3, y3)
SetLineWidth(10)
DrawPath(1, 0, 0, #BLACK)

ClearPath(1)
MoveTo(1, x, y)
LineTo(1, x1, y1)
MoveTo(1, x2, y2)
LineTo(1, x3, y3)
SetLineWidth(6)
DrawPath(1, 0, 0, ARGB(128, #RED))
```

The code above draws a curve and two lines that illustrate the control points of the curve.

```
EnableLayers
SetFillStyle(#FILLCOLOR)
SetFormStyle(#ANTIALIAS)
StartPath(1)
AddBoxToPath(1, 0, 0, 100, 100)
AddBoxToPath(1, 150, 0, 100, 100)
AddBoxToPath(1, 0, 150, 100, 100)
AddBoxToPath(1, 150, 150, 100, 100)
DrawPath(1, #CENTER, #CENTER, #RED, {Border = True, bordersize = 5})
```

The code above draws a vector path that looks a little bit like the flag of Switzerland.

56.12 ForcePathUse

NAME

ForcePathUse – always use path-based drawing (V5.0)

SYNOPSIS

```
ForcePathUse(enable)
```

FUNCTION

This command can be used to redirect all drawing commands of Hollywood's standard graphics primitives library to the new vector-path based draw library. This is especially useful for round shapes like circles, arcs, and ellipses, because the standard drawing library is line-based, which means that round shapes will never look perfectly round because their round shape must be approximated through lines. The vector-path based

drawing library on the other hand can draw perfectly round shapes which will look better than the line-based approach of the standard drawing library.

To enable the path-based drawing for the standard functions, pass `True` to `ForcePathUse()`. Then, the following standard functions will be patched to make use of the new vector-path based drawing: `Arc()`, `Box()`, `Circle()`, `Ellipse()`, and `Polygon()`.

Keep in mind, though, that if you enable path-based drawing for the standard library, you will always get layers of type `#VECTORPATH` if layers are enabled. The standard draw library on the other hand would add layers of type `#ARC`, `#BOX`, `#CIRCLE`, `#ELLIPSE`, and `#POLYGON` respectively. Normally, this does not make much of a difference, but it can be an issue if you try to change the attributes of a layer using `SetLayerStyle()`, because layers of type `#VECTORPATH` do not have the same functionality than the other layer types. See Section 34.48 [SetLayerStyle], page 689, for more information about the supported attributes for the different layer types.

Please also note that in Hollywood versions earlier than 6.0 your script will require a vectorgraphics plugin if you pass `True` here. Starting with Hollywood 6.0 there is an inbuilt vectorgraphics engine as well.

Finally, it should be mentioned that vector-path based drawing is of course slower than the polygon based drawing of the standard drawing library. On modern systems, however, it does not make a great difference.

INPUTS

`enable` flag that specifies whether or not to enable the path-based drawing for standard functions (`True` means enable, `False` means disable)

56.13 FreePath

NAME

FreePath – free path object (V5.0)

SYNOPSIS

```
FreePath(id)
```

FUNCTION

This command will free the path specified in `id`. After calling this command the specified path will not be available any more. If you only want to remove all vertices and sub-paths from a path, you should use `ClearPath()` instead.

INPUTS

`id` identifier of path to free

56.14 GetCurrentPoint

NAME

GetCurrentPoint – get current point of path (V5.0)

SYNOPSIS

```
x, y = GetCurrentPoint(id)
```

FUNCTION

This command returns the current point of the path specified in `id`.

INPUTS

`id`	identifier of path to query

RESULTS

`x`	x position of current point
`y`	y position of current point

56.15 GetDash

NAME

GetDash – get current line dashing style (V7.1)

SYNOPSIS

```
offset[, t] = GetDash()
```

FUNCTION

This function returns the current line dashing style set using `SetDash()`. The first return value will be set to the offset at which the dash pattern starts and the second return value is a table containing the lengths of the individual on/off sections. See Section 56.30 [SetDash], page 1193, for details.

If no dash pattern is currently active, -1 is returned in `offset` and there is no second return value.

INPUTS

none

RESULTS

`offset`	offset at which to start the dash pattern or -1 for no dash pattern
`t`	optional: table containing on/off sections (only if `offset` is not -1)

56.16 GetFillRule

NAME

GetFillRule – get current fill rule for overlapping paths (V7.1)

SYNOPSIS

```
rule = GetFillRule()
```

FUNCTION

This function returns the current fill rule set using `SetFillRule()`. This will be either `#FILLRULEWINDING` or `#FILLRULEEVENODD`. See Section 56.31 [SetFillRule], page 1194, for details.

INPUTS
none

RESULTS

`rule` current fill rule

56.17 GetLineCap

NAME
GetLineCap – get current line cap style (V7.1)

SYNOPSIS
```
style = GetLineCap()
```

FUNCTION
This function returns the current line cap style set using `SetLineCap()`. This will be either `#CAPBUTT`, `#CAPROUND`, or `#CAPSQUARE`. See Section 56.32 [SetLineCap], page 1195, for details.

INPUTS
none

RESULTS

`style` current line cap style

56.18 GetLineJoin

NAME
GetLineJoin – get current line join style (V7.1)

SYNOPSIS
```
style = GetLineJoin()
```

FUNCTION
This function returns the current line join style set using `SetLineJoin()`. This will be either `#JOINMITER`, `#JOINROUND`, or `#JOINBEVEL`. See Section 56.33 [SetLineJoin], page 1195, for details.

INPUTS
none

RESULTS

`style` current line join style

56.19 GetMiterLimit

NAME
GetMiterLimit – get current miter limit (V7.1)

SYNOPSIS
```
limit = GetMiterLimit()
```

FUNCTION
This function returns the current miter limit set using `SetMiterLimit()`. See Section 56.34 [SetMiterLimit], page 1196, for details.

INPUTS
none

RESULTS

`limit`	current miter limit

56.20 GetPathExtents

NAME
GetPathExtents – calculate extents of path (V5.0)

SYNOPSIS
```
x1, y1, x2, y2 = GetPathExtents(id)
```

FUNCTION
This command calculates the extents of the path specified in `id`. The current fill style and form style is taken into account by this function. The extents are returned as a bounding rectangle. Note that all return values specify absolute point positions. To get the width and height of the path from these positions you have to subtract `x1` from `x2` and `y1` from `y2`.

INPUTS

`id`	identifier of path to query

RESULTS

`x1`	start x position
`y1`	start y position
`x2`	end x position
`y2`	end y position

56.21 IsPathEmpty

NAME
IsPathEmpty – check if path is empty (V5.0)

SYNOPSIS
```
ret = IsPathEmpty(id)
```

FUNCTION

This function can be used to check whether or not the specified path object is empty or not. If it is empty, this function returns `True`, otherwise `False`.

INPUTS

`id`	identifier of path to query

RESULTS

`ret`	`True` if path is empty, else `False`

56.22 LineTo

NAME

LineTo – add line to path (V5.0)

SYNOPSIS

```
LineTo(id, x, y)
```

FUNCTION

This command will add a line from the path's current point to the point specified by `x`,`y`. When `LineTo()` returns, the path's current point will be at `x`,`y`.

INPUTS

`id`	identifier of path to add line to
`x`	x coordinate of destination point
`y`	y coordinate of destination point

56.23 MoveTo

NAME

MoveTo – set current point and begin sub-path (V5.0)

SYNOPSIS

```
MoveTo(id, x, y)
```

FUNCTION

This command can be used to begin a new sub-path at the specified point. When `MoveTo()` returns, the path's current point will be at `x`,`y`. For most cases this is the preferable way to start a new sub-path. The command `StartSubPath()` is only recommended for the rare case when you want a sub-path without a current point.

INPUTS

`id`	identifier of path
`x`	x coordinate of destination point
`y`	y coordinate of destination point

56.24 NormalizePath

NAME

NormalizePath – move path to origin (V5.0)

SYNOPSIS

```
NormalizePath(id)
```

FUNCTION

This function can be used to normalize a path. Normalizing means that all blank spaces at the left and top sides of the path are cut off, so that the path is moved to the origin position at the top left corner.

INPUTS

`id` identifier of path to normalize

56.25 PathItems

NAME

PathItems – traverse individual path items (V7.0)

SYNOPSIS

```
f = PathItems(id)
```

FUNCTION

This function can be used together with the generic `For` statement to traverse all items in a path. It returns an iterator function which will return a table that contains information about the next item in the path. Once all path items have been returned, the iterator function will return `Nil` to break the generic `For` statement.

See Section 11.4 [Generic For statement], page 127, for details.

The table returned by `PathItems()` will contain a `Type` field which contains a string that describes the item type. All further table fields depend on the item type as returned in `Type`. The following types are currently supported:

`NewSubPath:`
: This item starts a new sub-path. The current point will be undefined at this time. There are no additional arguments.

`ClosePath:`
: Closes the current path by drawing a line from the current point to the first point in the sub-path. There are no additional arguments.

`MoveTo:` This command begins a new sub-path. The sub-path's current point will be set to the specified position. The `MoveTo` table contains the following three additional arguments:

  `Rel:` This is a boolean value that indicates whether the coordinates are relative or absolute values. If this is `True`, the coordinates have to be interpreted as relative to the current point.

  `X:` The x coordinate of the new position.

`Y:` The y coordinate of the new position.

`LineTo:` This draws a line from the current point to the specified position. Additionally, it will change the current point to the line's end point when it is done. The `LineTo` table contains the following three additional arguments:

`Rel:` This is a boolean value that indicates whether the coordinates are relative or absolute values. If this is `True`, the coordinates have to be interpreted as relative to the current point.

`X:` The x coordinate of the new position.

`Y:` The y coordinate of the new position.

If there is no current point, `LineTo` will behave as if it was `MoveTo`, i.e. it will simply set the current point to the specified vertex.

`CurveTo:` This command draws a Bézier curve that runs from the current point to the position passed in the final two coordinates. The other four coordinates are the control points for the curve. Additionally, it will change the current point to the curve's end point when it is done. The `CurveTo` table contains the following seven additional arguments:

`Rel:` This is a boolean value that indicates whether the coordinates are relative or absolute values. If this is `True`, the coordinates have to be interpreted as relative to the current point.

`X1:` The x coordinate of the first control point.

`Y1:` The y coordinate of the first control point.

`X2:` The x coordinate of the second control point.

`Y2:` The y coordinate of the second control point.

`X3:` The x coordinate of the end point.

`Y3:` The y coordinate of the end point.

If there is no current point, `CurveTo` will use the point passed in (x1,y1) as the current point.

`Arc:` This command will draw an elliptical arc. `Arc` will open a new subpath for the new arc only in case there is no currently active subpath. If there is already a subpath, `Arc` will connect its starting vertex with the current vertex of the subpath. `Arc` will not close the subpath when it has finished adding its vertices. `Arc` will not connect the start and end angles of the arc with its center point automatically. This has to be requested explicitly by issuing separate `MoveTo` and `LineTo` commands before and after `Arc`. The `Arc` table contains the following additional arguments:

`XC:` The x center point of the arc.

`YC:` The y center point of the arc.

`RA:` Arc's radius on the x axis.

`RB:` Arc's radius on the y axis.

`Start:` Start angle in degrees.

`End:` End angle in degrees.

`Clockwise:`
Whether or not the angles should be connected in clockwise direction.

When `Arc` is done, it will set the current point to the position of the end angle.

`Box:` This command will draw a rectangle. `Box` will first open a new subpath, then add the rectangle's vertices to it and close the subpath when it is finished. Optionally, the rectangle can have rounded corners. The `Box` table contains the following additional arguments:

`X:` X position of the rectangle.

`Y:` Y position of the rectangle.

`Width:` Rectangle width.

`Height:` Rectangle height.

`Round:` This is a table which contains four integer values in the range from 0 to 100 specifying the degree of rounding for the four corners of the rectangle. A value of 0 means no rounding, 100 means full rounding.

`Text:` This command will draw vector text relative to the current point. The individual characters are added as closed subpaths. The `Text` table contains the following additional arguments:

`Size:` Desired font size.

`Text:` The string to draw.

When `Text` is done, it will set the current point to where the next character would be displayed.

INPUTS

`id` identifier of path to traverse

RESULTS

`f` iterator function for generic for loop

EXAMPLE

```
For t In PathItems(1) Do DebugPrint(t.type)
```

The code above iterates over all items in path 1 and prints the type of each item to the debug device.

56.26 PathToBrush

NAME

PathToBrush – convert path(s) to vector brush (V7.0)

SYNOPSIS

```
[id] = PathToBrush(id, table)
```

FUNCTION

This function can be used to convert one or more path(s) into a vector brush. `PathToBrush()` will create a new brush with the identifier specified in `id` or if you pass `Nil` in the `id` argument, `PathToBrush()` will automatically choose an identifier for the new brush and return it to you.

Converting paths into vector brushes has the advantage that you can assign different colors to the individual paths combined inside a single vector brush, allowing you to easily manage multi-colored paths inside just a single brush object. Furthermore, the paths combined inside the vector brush can also use different drawing styles.

You have to pass a table in the `table` argument that contains a number of subtables specifying information about the individual paths to be embedded inside the vector brush. The paths are drawn into the vector brush in exactly the same order as they appear in that table.

Note that each path to be embedded inside the vector brush will be normalized first. Thus, by default all paths will be drawn in the top-left corner of the vector brush. You can change this behaviour by specifying the `X` and `Y` arguments in the individual subtables for each path to be added to the vector brush (see below).

The following subtable fields can be specified:

`ID:` This must be set to the identifier of the path object to be embedded inside the vector brush that shall be created. This must always be provided. Note that `PathToBrush()` will make a copy of this path so subsequent modifications of the path won't affect the new vector brush in any way. You may also free this path after adding it to the vector brush.

`X:` The x position where this path should be drawn inside the vector brush. This position must be relative to the left corner of the vector brush. Note that `PathToBrush()` will internally normalize the path before adding it to the vector brush so you will usually have to use this field to position the path correctly inside the vector brush. Defaults to 0.

`Y:` The y position where this path should be drawn inside the vector brush. This position must be relative to the top corner of the vector brush. Note that `PathToBrush()` will internally normalize the path before adding it to the vector brush so you will usually have to use this field to position the path correctly inside the vector brush. Defaults to 0.

`Color:` The path will be drawn in this ARGB color. This color can also contain a transparency setting. Defaults to `#BLACK`.

Note that the form and fill styles to be used by the individual paths are the ones that were active at the time the path was created using `StartPath()`. This is different to the way

form and fill styles work when drawing paths using `DrawPath()`. `DrawPath()` will use the form and fill styles that are active when `DrawPath()` is called whereas `PathToBrush()` will use the form and fill styles that were active when `StartPath()` was called on the individual paths. This allows you to use different form and fill styles for the individual paths to be embedded inside the vector brush.

INPUTS

`id` id for the new brush or `Nil` for auto id selection

`table` table containing paths to combine into the vector brush (see above)

RESULTS

`id` optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
PathToBrush(1, {{ID=1, Color=#RED}, {ID=2, Color=#BLUE, X=100}})
```

The code above combines paths 1 and 2 inside a new vector brush which will use the identifier 1. Path 1 is drawn in red and path 2 is drawn in blue. Additionally, path 2 is drawn at x position 100 whereas path 1 is drawn at 0,0.

56.27 RelCurveTo

NAME

RelCurveTo – add curve to path (V5.0)

SYNOPSIS

```
RelCurveTo(id, dx1, dy1, dx2, dy2, dx3, dy3)
```

FUNCTION

This command does the same as `CurveTo()` except that the coordinates are delta values instead of absolute positions. The delta coordinates are all interpreted as relative offsets to the path's current point.

INPUTS

`id` identifier of path to add curve to

`dx1` delta x coordinate of control point `#1`

`dy1` delta y coordinate of control point `#1`

`dx2` delta x coordinate of control point `#2`

`dy2` delta y coordinate of control point `#2`

`dx3` delta x coordinate of curve destination point

`dy3` delta y coordinate of curve destination point

56.28 RelLineTo

NAME
RelLineTo – add relative line to path (V5.0)

SYNOPSIS
`RelLineTo(id, dx, dy)`

FUNCTION
This command does the same as `LineTo()` except that the coordinates are delta values instead of absolute positions. The delta coordinates are all interpreted as relative offsets to the path's current point.

INPUTS

`id`	identifier of path to add line to
`dx`	delta x coordinate of destination point
`dy`	delta y coordinate of destination point

56.29 RelMoveTo

NAME
RelMoveTo – set relative current point and begin sub-path (V5.0)

SYNOPSIS
`RelMoveTo(id, dx, dy)`

FUNCTION
This command does the same as `MoveTo()` except that the coordinates are delta values instead of absolute positions. The delta coordinates are all interpreted as relative offsets to the path's current point.

INPUTS

`id`	identifier of path
`dx`	delta x coordinate of destination point
`dy`	delta y coordinate of destination point

56.30 SetDash

NAME
SetDash – set line dashing style (V5.0)

SYNOPSIS
`SetDash(offset, on1, off1, ...)`

FUNCTION
This function can be used to define a dash pattern for paths drawn by `DrawPath()`. A dash pattern consists of an unlimited number of on and off line sections that start at the

offset specified in argument 1. Starting with argument 2, you have to pass the length of the line section that shall be visible ("on-section") followed by the length of the line section that shall be invisible ("off-section"), and repeat this pattern as many times as you like. When drawing an outline path, `DrawPath()` will then apply this dash pattern to all lines it is drawing. When the dash pattern does not cover the whole line length, it will be repeated over and over again.

To remove the dash pattern, call this function with the second argument set to -1.

Please note that the line dash style is only used when drawing vector outlines. It is obviously not used when filling vector paths.

Also note that the inbuilt vectorgraphics renderer introduced in Hollywood 6.0 currently does not support line dashing. Use a fully featured vectorgraphics plugin instead if you need line dashing. See Section 56.39 [Vectorgraphics plugin note], page 1198, for details.

INPUTS

`offset`	offset at which to start the dash pattern
`on1`	length of on-section 1
`off1`	length of off-section 1
`...`	optional: define as many on-/off-sections as you like

EXAMPLE

```
SetFillStyle(#FILLNONE)
SetFormStyle(#ANTIALIAS)
SetLineWidth(10)
SetDash(0, 10, 10, 20, 20, 30, 30, 40, 40)
StartPath(1)
MoveTo(1, 0, 0)
LineTo(1, 640, 480)
DrawPath(1, 0, 0, #RED)
```

The code above draws a line using a dash pattern that first has four different on/off sections: The first on/off section is 10 units, the second 20 units, the third 30 units, and the fourth 40 units.

56.31 SetFillRule

NAME

SetFillRule – set fill rule for overlapping paths (V5.0)

SYNOPSIS

```
SetFillRule(rule)
```

FUNCTION

This function can be used to define how `DrawPath()` should fill paths that overlap each other. Currently, the following fill rules are supported:

`#FILLRULEWINDING:`
: Fill all overlapping paths only if they are not winding. This is the default setting.

`#FILLRULEEVENODD:`
Fill overlapping paths if the total number of intersections is odd.

INPUTS

`rule` desired fill rule (see above for possible settings)

56.32 SetLineCap

NAME

SetLineCap – define current line cap style (V5.0)

SYNOPSIS

```
SetLineCap(style)
```

FUNCTION

This function can be used to define how `DrawPath()` should draw the endings of lines that are not connected to another vertices. Currently, the following cap styles are supported:

`#CAPBUTT:`
Do not draw any special line ending. Just stop drawing where the line ends. This is the default mode.

`#CAPROUND:`
Draw round line endings.

`#CAPSQUARE:`
Draw squared line endings.

Please note that the line cap style is only used when drawing vector outlines. It is obviously not used when filling vector paths.

INPUTS

`style` desired line cap style (see above for possible settings)

56.33 SetLineJoin

NAME

SetLineJoin – define current line join style (V5.0)

SYNOPSIS

```
SetLineJoin(style)
```

FUNCTION

This function can be used to define how `DrawPath()` should connect the lines when drawing a vector path. Currently, the following join styles are supported:

`#JOINMITER:`
Use miter join (a sharp angled corner). This is the default join mode. The miter limit can be set using the `SetMiterLimit()` function. See Section 56.34 [SetMiterLimit], page 1196, for details.

`#JOINROUND:`
Join lines by drawing their ends as circles. This gives a thick pen impression.

`#JOINBEVEL:`
Join lines by cutting off the line ends at the half of the line width.

Please note that the line join style is only used when drawing vector outlines. It is obviously not used when filling vector paths.

INPUTS

`style` desired line join style (see above for possible settings)

56.34 SetMiterLimit

NAME

SetMiterLimit – set miter limit (V7.1)

SYNOPSIS

```
SetMiterLimit(limit)
```

FUNCTION

This function sets the miter limit to the value specified by `limit`. This can be a fractional value. The miter limit is used when the join style is set to `#JOINMITER` to determine when to join lines with a bevel and when to join them using a miter. Note that `#JOINMITER` is also the default line join style.

When drawing line ends, the length of the miter is divided by the line width and if the result of this division is greater than the miter limit set using this function, lines are joined using a bevel.

Hollywood's default miter limit is 10.

INPUTS

`limit` desired miter limit

56.35 SetVectorEngine

NAME

SetVectorEngine – choose vectorgraphics renderer (V6.0)

SYNOPSIS

```
SetVectorEngine(engine$)
```

FUNCTION

This command can be used to select the plugin that should be used to draw vector-based shapes. You simply have to pass the name of the plugin to this function. To use the inbuilt vectorgraphics renderer, pass `default`.

Note that it is perfectly allowed to use several vectorgraphics renderers inside a single script. It is even possible to use several vectorgraphics renderers inside a single BGPic. For example, the first layer could use the default vectorgraphics renderer while the second

layer uses a plugin-based vectorgraphics renderer. This is all supported. You can even change the renderer used by single layers by calling `SetLayerStyle()` and setting the `VectorEngine` tag.

INPUTS

`engine$` name of the plugin that should be used to draw vectorgraphics

56.36 StartPath

NAME

StartPath – begin a new path (V5.0)

SYNOPSIS

```
[id] = StartPath(id)
```

FUNCTION

This command can be used to create a new path object that will be made available under the identifier `id`. Alternatively, you can pass `Nil` as `id`. In that case, `StartPath()` will automatically select an identifier and return it to you.

Once the new path is created, you should first define a current point for the path by calling the `MoveTo()` command. After that, you can start adding vertices to the path.

A vector path is a root object for an infinite number of sub-paths. You can think of a vector path as a collection of an infinite number of separate polygons. Each sub-path within a vector path can be regarded as a separate polygon. Keep in mind, though, that when you draw a vector path using `DrawPath()` you can only specify a global color that will be used by all sub-paths within the vector path. Thus, it is not possible to use different colors within a single path object. If you need to use another color, you need to create a new path object first.

INPUTS

`id` identifier for the new path or `Nil` for auto id selection

RESULTS

`id` optional: identifier of the new path; this will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

See Section 56.11 [DrawPath], page 1181.

56.37 StartSubPath

NAME

StartSubPath – begin a new sub-path (V5.0)

SYNOPSIS

```
StartSubPath(id)
```

FUNCTION

This command can be used to begin a new sub-path within the path object specified in `id`. The new sub-path will not get a current point so that most of the time you should better call `MoveTo()` to start a new sub-path. `StartSubPath()` is only preferable for rare cases in which a current point is not desired; for example, when adding a circle/ellipse/arc to a path a current point can be annoying because it would be connected to the circle/ellipse/arc then. For most cases, however, you should use `MoveTo()` instead of `StartSubPath()`.

INPUTS

`id`	identifier of the path to use

56.38 TranslatePath

NAME

TranslatePath – offset a path (V5.0)

SYNOPSIS

```
TranslatePath(id, dx, dy)
```

FUNCTION

This function can be used to translate a path. Translating means that each vertex in the path is shifted by the specified delta offset. A positive delta offset shifts to the right (or bottom) and a negative delta offset shifts to the left (or top). An offset of 0 does not do anything.

INPUTS

`id`	identifier of path to normalize
`dx`	amount of delta shift on the x axis
`dy`	amount of delta shift on the y axis

EXAMPLE

```
TranslatePath(1, -100, 150)
```

The code above shifts path number one 100 pixels to the left and 150 pixels to the bottom.

56.39 Vectorgraphics plugin

Before Hollywood 6.0 all functions of Hollywood's vectorgraphics library could only be used if a separate plugin that implements vector-based drawing was present. Starting with Hollywood 6.0 there is basic support for vector-based drawing even without an external vectorgraphics plugin. However, inbuilt support for vector-based drawing does not support all features offered by the vectorgraphics library and might not look as good as vector-based shapes drawn by a specialized plugin. Still, it should be enough for most purposes.

For the best results and compatibility you should install an external vectorgraphics plugin that uses a dedicated vector render. For example, vectorgraphics plugins could implement

vector-based drawing by using either platform-independent engines like cairo, or OS technologies like Apple's Quartz 2D or Microsoft's GDI+.

The plugin needs to be present in the same directory as the Hollywood executable. In case you want to distribute an executable compiled by Hollywood, you need to put the plugin into the same directory as your compiled executable. On AmigaOS and compatibles, you can also put the plugin inside the directory LIBS:Hollywood. If the "Hollywood" directory in LIBS: does not already exist, please create it yourself. On macOS, you need to put the plugin inside the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources/Plugins` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle, namely in `Hollywood.app/Contents/Resources`.

Once the plugin has been installed, use the `SetVectorEngine()` command to activate it. See Section 56.35 [SetVectorEngine], page 1196, for details.

57 Video library

57.1 Overview

Hollywood's video library provides functions for loading and playing video objects. Video objects are Hollywood objects which contain a video stream that may be bundled with an audio stream. When playing a video object, Hollywood will make sure that video and audio streams are perfectly synchronized with each other.

You can open a video file from disk using the `@VIDEO` preprocessor command or the `OpenVideo()` command. To play a video, use the `PlayVideo()` command.

Hollywood's video library supports two different renderers:

1. Inbuilt video renderer: This is a platform-independent video renderer supported on all platforms. It is the most flexible video renderer and supports advanced features like video layers and plugins. The disadvantage is that decoding is done completely in software which is why large videos (or videos that use 50fps or more) might be stuttering. In that case, you can use the native video renderer instead (see below). The native video renderer is often hardware-accelerated which is why video playback will still be smooth even in very high resolutions or with lots of frames per second. Note that the only video format supported by the inbuilt video renderer is the CDXL video format developed by Commodore in the early 90s. CDXL isn't very useful for today's video requirements, but the big advantage of the inbuilt video renderer is that it can load videos via Hollywood plugins. Installing video plugins can greatly enhance the functionality of the inbuilt video renderer and can enable Hollywood to play lots of different video formats.
2. Native video renderer: This is only supported on Windows, macOS, and iOS. This renderer loads and plays videos through the operating system's video interface. On Windows video playback is done via Media Foundation and DirectShow whereas macOS and iOS use AV Foundation or QuickTime (on older systems). This renderer is not as flexible as the inbuilt video renderer. It doesn't support video layers and it also doesn't support video format loaders made available by Hollywood plugins. But it can be much faster because native video renderers are often hardware-accelerated. The number of video formats that can be played by the native video renderer is also limited. The best format to use with the native video renderer is MPEG4 because this is supported on all platforms except on very old macOS or Windows versions.

By default, Hollywood will first ask the inbuilt video renderer to open the video file. You can change this behaviour by using the `Loader` tag in your call to `@VIDEO` or `OpenVideo()`. See Section 57.17 [VIDEO], page 1212, for details.

57.2 CloseVideo

NAME

CloseVideo – close a video (V5.0)

SYNOPSIS

```
CloseVideo(id)
```

FUNCTION

This function frees any memory occupied by the video specified by `id` and closes the video. If the video is still playing, it will be stopped first. Although Hollywood will automatically free all resources when it exits, you should still call `CloseVideo()` when you are done with a video file because it reduces memory consumption.

INPUTS

`id`	identifier of the video to close

57.3 DisplayVideoFrame

NAME

DisplayVideoFrame – display a single frame of a video (V6.0)

SYNOPSIS

```
DisplayVideoFrame(id, x, y, pos[, table])
```

FUNCTION

This function displays a single frame of a video at the specified coordinates and adds a new layer of type `#VIDEO` to the layer stack. The frame is specified not as an absolute index position but as a timestamp in milliseconds. Thus, to display the very first frame you would have to pass 0 in the `pos` argument.

Please note that this function currently does not work with layers disabled. Layers need to be enabled for this function to work.

`DisplayVideoFrame()` also recognizes an optional table argument which allows you to specify one or more of the standard tags for all drawing commands. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`id`	identifier of the video to use
`x`	destination x coordinate
`y`	destination y coordinate
`pos`	timestamp of the frame to display in milliseconds
`table`	optional: table specifying further options

EXAMPLE

```
DisplayVideoFrame(1, #CENTER, #CENTER, 0)
```

The code above displays the first frame of video 1 in the center of the screen.

57.4 ForceVideoDriver

NAME

ForceVideoDriver – enforce use of specified video driver (V5.1)

SYNOPSIS

```
ForceVideoDriver(driver)
```

FUNCTION

This function can be used to specify the video driver all subsequent calls to `OpenVideo()` should use. Hollywood currently supports the following two video drivers:

`#VIDDRV_HOLLYWOOD:`
: Hollywood's platform independent video renderer. This is the default driver. It supports playback of the CDXL format plus all formats you have a plugin for.

`#VIDDRV_OS:`
: This driver uses the native video system of the OS. This is currently only supported on Windows, macOS, and iOS. On Windows this driver uses the Media Foundation or DirectShow technology while on macOS and iOS it uses AV Foundation or QuickTime (on older systems).

By default, `#VIDDRV_HOLLYWOOD` is given priority over `#VIDDRV_OS` which means that Hollywood will first try to play the video using its inbuilt video renderer. Only if that does not work, will Hollywood switch to the OS native renderer. If you want to change this behaviour, use this function.

Note that this function is obsolete since Hollywood 6.0 because you can now simply use the new `Loader` tag with `OpenVideo()` and `@VIDEO`. The `Loader` equivalent for `#VIDDRV_OS` is `native` and the `Loader` equivalent for `#VIDDRV_HOLLYWOOD` is `inbuilt|plugin`.

INPUTS

`driver` desired video driver to use

57.5 GetVideoFrame

NAME

GetVideoFrame – convert video frame to a brush (V5.0)

SYNOPSIS

```
[id] = GetVideoFrame(brushid, frame, videoid[, unit])
```

FUNCTION

This function can be used to convert a single frame of a video to a brush. The video must have been opened using `OpenVideo()` or the `@VIDEO` preprocessor command. In the first argument, you have to pass an id for the brush you want this function to create (alternatively, you can pass `Nil` for automatic id selection). In the second argument you have to specify which frame of the video should be grabbed, and the third argument specifies the identifier of the video to use as the source.

The optional argument `unit` is used to specify how the value passed in `frame` should be interpreted. If `unit` is set to 0, then the value passed in `frame` is interpreted as an absolute frame index. This is also the default setting. If `unit` is set to 1, then the value passed in `frame` is interpreted as a time stamp in milliseconds and `GetVideoFrame()` will grab the frame at this very timestamp. It is recommended to use unit 1 access to single

frames because this is usually much faster. If you really need to access single frames by their absolute index, please read the word of warning below.

Please note that frame access by absolute index is usually a very expensive operation because, for most video formats, Hollywood needs to traverse all the way through the stream until it reaches the requested frame. Such a traversal requires a lot of time and is thus of limited practical use. However, there is one special case where `GetVideoFrame()` can be used very efficiently and that is the sequential grabbing of frames from a video stream. "Sequential grabbing" means that you read one frame after the other from the video stream, i.e. first frame 1, then frame 2, then frame 3, etc. This can be done very quickly. The only thing that will take lots of time is reading frames in backward direction (i.e. frame 10, frame 9, frame 8, etc.), or making huge leaps between frame reads (i.e. frame 1, then frame 1000, then frame 5000, etc.). This will take a lot of time. Sequential reading will be efficient, however.

To find out the exact number of frames inside a video stream, you can use the `GetAttribute()` command and query the `#ATTRNUMFRAMES` attribute using the `#VIDEO` object type.

INPUTS

`id` identifier for the brush to create or `Nil` for auto id selection

`frame` frame to load; format of this argument depends on the value passed to the unit argument below

`videoid` identifier of the video to use as source

`unit` optional: base to use for the frame argument; this can be either 0 which means that the frame argument specifies an absolute frame index or 1 which means that the frame argument specifies a timestamp in milliseconds (defaults to 0)

RESULTS

`id` optional: identifier of the brush; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
my_frame = GetVideoFrame(Nil, 1, 2)
DisplayBrush(my_frame, #CENTER, #CENTER)
```

The code above extracts frame 1 from video stream 2 and stores it in a new brush. The brush is then displayed at the center of the display.

57.6 IsVideo

NAME

IsVideo – determine if a file is in a supported video format (V5.0)

SYNOPSIS

```
ret = IsVideo(file$[, table])
```

FUNCTION

This function will check if the file specified in `file$` is in a supported video format. If it is, this function will return `True`, otherwise `False`. If this function returns `True`, you can open the video using `OpenVideo()`.

Starting with Hollywood 6.0 this function accepts an optional table argument which allows you to configure further options:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this video. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

See Section 57.8 [OpenVideo], page 1206, for a list of supported video formats.

INPUTS

`file$` file to examine

`table` optional: table configuring further options (V6.0)

RESULTS

`ret` `True` if the video is in a supported format, `False` otherwise

57.7 IsVideoPlaying

NAME

IsVideoPlaying – check if video is currently playing (V5.0)

SYNOPSIS

```
playing = IsVideoPlaying(id)
```

FUNCTION

This function checks if the video specified by `id` is currently playing. If it is, `True` is returned, `False` otherwise.

INPUTS

`id` identifier of video to check

RESULTS

`playing` `True` if video is currently playing; `False` otherwise

57.8 OpenVideo

NAME

OpenVideo – open a video file (V5.0)

SYNOPSIS

```
[id] = OpenVideo(id, filename$[, table])
```

FUNCTION

This function opens the video file specified by `filename$` and assigns the specified `id` to it. If you pass `Nil` in `id`, `OpenVideo()` will automatically choose an identifier and return it. The video file specified in `filename$` will be opened and prepared for playback. Video playback is always done directly from disk which means that `OpenVideo()` will not prebuffer any data at all. It will just initialize all parameters necessary for video playback.

Video formats that are supported on all platforms are CDXL and formats you have a plugin for. Depending on the platform Hollywood is running on, more video formats might be supported. On Windows Hollywood is able to open all video formats for which you have a Media Foundation or DirectShow codec installed. On macOS Hollywood can open all video formats that are supported by AV Foundation (or QuickTime on older Macs).

Starting with Hollywood 6.0 this command accepts an optional table argument which recognizes the following options:

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this video. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. The default loader lets Hollywood first ask all plugins whether they would like to handle the video file, then it will check its inbuilt loaders (currently only CDXL), and finally it will ask the video interface of the host OS to play this video. If you want to customize this order, use this tag. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

This command is also available from the preprocessor: Use `@VIDEO` to preload video files!

INPUTS

`id` identifier for the video or `Nil` for auto id selection

`filename$`
file to load

`table` optional: table that contains further parameters (V6.0)

RESULTS

`id` optional: identifier of the video; will only be returned when you pass `Nil` as argument 1 (see above)

EXAMPLE

```
OpenVideo(1, "intro.avi")
PlayVideo(1)
```

The code above loads & plays "intro.avi".

57.9 PauseVideo

NAME

PauseVideo – pause a playing video (V5.0)

SYNOPSIS

```
PauseVideo(id)
```

FUNCTION

This function pauses the video associated with the identifier `id`. This video must be playing when you call this command. You can resume playback later by using the `ResumeVideo()` command.

Please note that pausing a video will not remove the video from the display. Instead, the currently displayed frame will be frozen until you call `ResumeVideo()`. If you want to remove a video completely from the display, you will always have to call `StopVideo()`.

INPUTS

`id` identifier of the video to pause

57.10 PlayVideo

NAME

PlayVideo – start playback of a video (V5.0)

SYNOPSIS

```
PlayVideo(id[, x, y, table])
```

FUNCTION

This command starts the playback of the video specified by `id`. This video must have been opened using either the `@VIDEO` preprocessor command or the `OpenVideo()` command. The optional arguments `x` and `y` can be used to specify where on the display the video should appear. If you do not specify these arguments, `PlayVideo()` will use

the coordinates specified in the latest call to `SetVideoPosition()`. If you did not call `SetVideoPosition()` on this video at all, the position will default to 0/0 which is the upper left corner of the display.

`PlayVideo()` works in an asynchronous manner. It will return immediately once it has started video playback. After you have started a video, you can control playback by calling functions like `StopVideo()` or `StopLayer()` depending on whether you use normal or layered playback mode.

Hollywood supports two different video playback modes: Normal playback and layered playback. Normal playback is the fastest and most optimized mode as it can utilize hardware acceleration by using video overlays for example. The disadvantage of normal mode is that it has some restrictions (see below for more information). Layered playback mode, on the other hand, is very flexible as your video will be rendered into a Hollywood layer and so you can use all layer functionality on your video object, e.g. you can rotate it, apply transparency and transition effects or special image filters. The disadvantage of layered mode is that it is quite slow because it cannot use hardware acceleration so you will need lots of raw CPU horsepower to get decent framerates here. For most cases, normal playback mode should be sufficient. Layered playback mode is only necessary if you need to do advanced things during video playback.

It is important to note that `PlayVideo()` will always use normal playback mode by default, even if layers are enabled for the current BGPic. To make `PlayVideo()` use layered mode, you will have to request this explicitly by setting the `UseLayer` tag in the optional table argument to `True`.

Normal playback mode comes with the restriction that videos will always appear above everything else. This means that it is impossible for you to draw on top of a video. Instead, all graphics commands will always draw beneath the video area. Even sprites will never appear above the video graphics. Also, the video will stay visible until you call `StopVideo()`. Pausing a video will not remove that video from the display. Instead, the currently displayed frame will be frozen until you call `ResumeVideo()`. If you want to remove a video completely from the display, you will always have to call `StopVideo()`. If you would like to move a video or change its size during normal playback mode, you need to use the commands `SetVideoPosition()` and `SetVideoSize()` for this task.

In layered playback mode you can use all the functions from Hollywood's layers library to control video playback, i.e. you can change the video's size and orientation using `ScaleLayer()` and `RotateLayer()` or redefine the z-order by using `SetLayerZPos()`. You can also show and hide videos using `ShowLayer()` and `HideLayer()` or apply transparency or filters to them using `SetLayerTransparency()` and `SetLayerFilter()`. All functions of Hollywood's layers library can be used with video layers. To stop or pause a video layer, use the `StopLayer()` and `PauseLayer()` functions respectively. To seek to a new position inside the video, use `SeekLayer()`. To change the audio volume of a video layer use `SetLayerVolume()`.

Please note that layered playback is only possible if the video has been opened using Hollywood's inbuilt or plugin-based video handler. Layered playback is not supported when using the video renderer provided by the host OS. You can change the video driver by using the `Loader` tag in `OpenVideo()` or `@VIDEO`.

There are no limits as to how many videos can be played concurrently. Hardware acceleration, however, can often be only used when just a single video is played at a time. When multiple videos are playing at the same time, Hollywood often has to switch back to software rendering, which is slower. Please also note that video playback generally requires a strong CPU. 68k processors are much too slow for this task (except on WinUAE).

Note that when switching BGPics using `DisplayBGPic()`, Hollywood will automatically stop all videos playing in normal mode. Videos playing in layered mode, however, will continue playing even if the BGPic has been changed. Thus, you need to explicitly stop video layers by calling `StopLayer()` before switching BGPics if you want them to stop on this occasion.

Starting with Hollywood 6.0, `PlayVideo()` accepts an optional table argument which can be used to configure the following options:

`UseLayer:`
: If you set this tag to `True`, `PlayVideo()` will use layered playback mode. You need to enable layers before you can use this tag. See Section 34.1 [Layers introduction], page 647, for details. If layered playback mode is used, this command will add a new layer of the type `#VIDEO` to the layer stack. See above for more information on the difference between normal and layered playback mode. Defaults to `False`. (V6.0)

`Channel:`
: Channel to use for playback of this video's audio stream. By default, `PlayVideo()` will automatically choose a vacant channel and will fail if there is no vacant channel. To override this behaviour, you can use this field. When specified, it will always enforce audio playback on the very channel specified here. If the channel is already playing, it will be stopped first. (V6.1)

If layered playback mode is used you can also specify one or more of the standard tags for all drawing commands in the optional table argument. See Section 27.17 [Standard drawing tags], page 501, for more information about the standard tags that nearly all Hollywood drawing commands support.

INPUTS

`id` identifier of the video to play

`x` optional: desired x position for the video (defaults to the position defined using `SetVideoPosition()` or 0)

`y` optional: desired y position for the video (defaults to the position defined using `SetVideoPosition()` or 0)

`table` optional: table configuring further options (V6.0)

EXAMPLE

See Section 57.8 [OpenVideo], page 1206.

57.11 ResumeVideo

NAME

ResumeVideo – resume a paused video (V5.0)

SYNOPSIS

```
ResumeVideo(id)
```

FUNCTION

This function resumes the playback of a paused video that is associated with the identifier `id`. You can pause the playback of a video using the `PauseVideo()` command.

INPUTS

`id`	identifier of the video to be resumed

57.12 SeekVideo

NAME

SeekVideo – seek to a certain position in a video (V5.0)

SYNOPSIS

```
SeekVideo(id, pos)
```

FUNCTION

You can use this function to seek to the specified position in the video specified by `id`. The video does not have to be playing. If the video is playing and you call `SeekVideo()`, it will immediately skip to the specified position. The position is specified in milliseconds. Thus, if you want to skip to the position 3:24, you would have to pass the value 204000 because 3 * 60 * 1000 + 24 * 1000 = 204000.

Please note that video seeking is a complex operation. There are video formats which do not have any position lookup tables so that Hollywood first has to approximate the seeking position and then do some fine- tuning and keyframe seeking so that the final position can always be a bit off from the position you specified in `SeekVideo()`. It can also happen that Hollywood will not seek directly to a keyframe so there might be artefacts from previous frames left on the screen.

INPUTS

`id`	identifier of the video to seek
`pos`	new position for the video (in milliseconds)

57.13 SetVideoPosition

NAME

SetVideoPosition – change output position of a video (V5.0)

SYNOPSIS

```
SetVideoPosition(id, x, y)
```

FUNCTION

This function can be used to change the position of a video. If the video is currently playing, it will be instantly moved to the new position. If it is not playing, the specified position will be memorized until you call `PlayVideo()` the next time.

Please note that this function must not be used for videos that are played back in layered mode. You can change the position of video layers using functions from layers library, e.g. `ShowLayer()`.

INPUTS

`id`	identifier of the video whose position you want to change
`x`	desired x position for the video
`y`	desired y position for the video

EXAMPLE

```
SetVideoPosition(1, #RIGHT, #BOTTOM)
```

The code above moves video to the bottom-right edge of the current display.

57.14 SetVideoSize

NAME

SetVideoSize – change output size of a video (V5.0)

SYNOPSIS

```
SetVideoSize(id, width, height[, smooth])
```

FUNCTION

This function can be used to change the dimensions of a video. If the video is currently playing, it will be instantly scaled to fit the new dimensions. If it is not playing, the specified dimensions will be memorized until you call `PlayVideo()` the next time.

You can pass the special constant `#KEEPASPRAT` as either `width` or `height`. Hollywood will then calculate the size automatically by taking the aspect- ratio of the video into account. Alternatively, `width` and `height` can also be a string containing a percent specification, e.g. `"50%"`.

Starting with Hollywood 5.1 you can pass the optional argument `smooth` which specifies whether or not anti-aliased interpolated scaling should be used. Please note that interpolated scaling is only available for videos played through Hollywood's platform independent video player without any hardware overlay.

Please note that this function must not be used for videos that are played back in layered mode. You can change the size of video layers using functions from layers library, e.g. `ScaleLayer()`.

INPUTS

`id`	identifier of the video whose size you want to change
`width`	desired new width for the video
`height`	desired new height for the video
`smooth`	optional: whether or not to use interpolated scaling; defaults to `False` (V5.1)

EXAMPLE

```
SetVideoSize(1, 640, 480)
```

The code above scales video 1 to a resolution of 640x480 pixels.

```
SetVideoSize(2, "50%", "50%")
```

The code above shrinks video number 2 to half its size.

57.15 SetVideoVolume

NAME

SetVideoVolume – modify volume of a video (V5.0)

SYNOPSIS

```
SetVideoVolume(id, volume)
```

FUNCTION

This function modifies the volume of the video specified by `id`. If the video is currently playing, the volume will be modified on-the-fly which can be used for sound fades etc. The `volume` argument can also be a string containing a percent specification, e.g. `"50%"`.

INPUTS

`id`	identifier of the video
`volume`	new volume for the video (range: 0=mute until 64=full volume or percent specification)

57.16 StopVideo

NAME

StopVideo – stop a currently playing video (V5.0)

SYNOPSIS

```
StopVideo(id)
```

FUNCTION

This function stops the video specified by `id` and removes it from the display. The video must be either in playing or paused state.

INPUTS

`id`	identifier of the video to be stopped

57.17 VIDEO

NAME

VIDEO – preload a video for later use (V5.0)

SYNOPSIS

```
@VIDEO id, filename$[, table]
```

FUNCTION

Use this preprocessor command to preload a video which you want to play later using `PlayVideo()`.

Video formats that are supported on all platforms are CDXL and formats you have a plugin for. Depending on the platform Hollywood is running on, more video formats might be supported. On Windows Hollywood is able to open all video formats for which you have a Media Foundation or DirectShow codec installed. On macOS Hollywood can open all video formats that are supported by AV Foundation (or QuickTime on older Macs).

The third argument is optional. It is a table that can be used to set further options for the opening operation. The following fields of the table can be used:

`Link:` Set this field to `False` if you do not want to have this video linked to your executable/applet when you compile your script. This field defaults to `True` which means that the video is linked to your to your executable/applet when Hollywood is in compile mode.

`Loader:` This tag allows you to specify one or more format loaders that should be asked to load this video. This must be set to a string containing the name(s) of one or more loader(s). Defaults to the loader set using `SetDefaultLoader()`. The default loaders lets Hollywood first ask all plugins whether they would like to handle the video file, then it will check its inbuilt loaders (currently only CDXL), and finally it will ask the video interface of the host OS to play this video. If you want to customize this order, use this tag. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`Adapter:` This tag allows you to specify one or more file adapters that should be asked to open the specified file. This must be set to a string containing the name(s) of one or more adapter(s). Defaults to the adapter set using `SetDefaultAdapter()`. See Section 7.9 [Loaders and adapters], page 92, for details. (V6.0)

`UserTags:`
This tag can be used to specify additional data that should be passed to loaders and adapters. If you use this tag, you must set it to a table of key-value pairs that contain the additional data that should be passed to plugins. See Section 7.10 [User tags], page 95, for details. (V10.0)

If you want to open the video manually, please use the `OpenVideo()` command.

INPUTS

`id` a value that is used to identify this video later in the code

`filename$`
the file you want to have loaded

`table` optional: a table containing further options

EXAMPLE

```
@VIDEO 1, "intro.avi"
```

The code above opens "intro.avi" so that it can be played later using `PlayVideo()`.

58 Windows support library

58.1 CreateShortcut

NAME
CreateShortcut – create a shortcut to a file (V4.7)

SYNOPSIS
`CreateShortcut(src$, dest$, desc$)`

PLATFORMS
Microsoft Windows only

FUNCTION
This function can be used to create a `*.lnk` shortcut to file `src$` in file `dest$`. The shortcut will use the description passed in `desc$`. The source file passed in `src$` can be either an executable or a document file.

INPUTS

`src$`	source file to which the shortcut shall point
`dest$`	shortcut file that shall be created
`desc$`	description string of shortcut

EXAMPLE
```
CreateShortcut("test.exe", "test.lnk", "Test shortcut")
```
The code above creates a link to file "test.exe" as "test.lnk" using the description "Test shortcut".

58.2 GetShortcutPath

NAME
GetShortcutPath – get path from shortcut (V5.2)

SYNOPSIS
`p$, desc$ = GetShortcutPath(f$)`

PLATFORMS
Microsoft Windows only

FUNCTION
This function can be used to get the full path from the `*.lnk` shortcut file specified in `f$`. The path that this shortcut is pointing to will then be returned in the first return value. If the shortcut file contains a description, it will be returned in the second value.

INPUTS

`f$`	shortcut file

RESULTS

`p$`	full path that shortcut points to

`desc$` description of shortcut (if available)

EXAMPLE

```
p$ = GetShortcutPath("test.lnk")
```

The code above returns the full path of the shortcut "test.lnk".

58.3 ReadRegistryKey

NAME

ReadRegistryKey – read a key from the registry (V4.5)

SYNOPSIS

```
value = ReadRegistryKey(base, key$)
```

PLATFORMS

Microsoft Windows only

FUNCTION

This function can be used to read a key from the Windows registry. You have to specify the base tree as well as the key to read. The base tree can be one of the following constants:

```
#HKEY_CLASSES_ROOT
#HKEY_CURRENT_CONFIG
#HKEY_LOCAL_MACHINE
#HKEY_USERS
#HKEY_CURRENT_USER
```

The return value will be a number in case the registry key contains a number. If the registry key contains a string or binary data, you will get a string as a return value. Hollywood strings are capable of holding binary data because they allow `NULL` characters in them.

INPUTS

`base` one of the base tree constants from above

`key$` the registry key to query

RESULTS

`value` value of specified registry key; will be either a number or a string

EXAMPLE

```
program_files$ = ReadRegistryKey(#HKEY_LOCAL_MACHINE,
      "Software/Microsoft/Windows/CurrentVersion/ProgramFilesDir")
```

The code above reads the default location of programs under Windows from the registry. On a German Windows system, this will usually return "C:/Programme".

58.4 WriteRegistryKey

NAME

WriteRegistryKey – write a key to the registry (V4.5)

SYNOPSIS

```
WriteRegistryKey(base, key$, value)
```

PLATFORMS

Microsoft Windows only

FUNCTION

This function can be used to write a key to the Windows registry. You have to specify the base tree, the key, and the value which shall be written in the specified key. If the specified key does not exist, it will be created by this function. The base tree can be one of the following constants:

```
#HKEY_CLASSES_ROOT
#HKEY_CURRENT_CONFIG
#HKEY_LOCAL_MACHINE
#HKEY_USERS
#HKEY_CURRENT_USER
```

The value for the key can be a either number or a string. You can also write binary data to the registry by passing a string. Hollywood strings are capable of holding arbitrary binary data because they allow `NULL` characters in them. Under normal circumstances, however, writing numbers or normal strings to the registry should be sufficient.

INPUTS

`base`	one of the base tree constants from above
`key$`	the registry key to create/modify
`value`	value to set the key to; can be either a number or a string

Appendix A Licenses

A.1 Lua license

Lua 5.0 license

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

A.2 OpenCV license

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistribution's of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution's in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of Intel Corporation may not be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall Intel or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

A.3 ImageMagick license

The authoratitive ImageMagick license can be found at http://www.imagemagick.org/script/license.php and ImageMagick notices at http://www.imagemagick.org/script/notice.php.

Before we get to the text of the license lets just review what the license says in simple terms:

It allows you to:

- freely download and use ImageMagick software, in whole or in part, for personal, company internal, or commercial purposes;

- use ImageMagick software in packages or distributions that you create.

It forbids you to:

- redistribute any piece of ImageMagick-originated software without proper attribution;
- use any marks owned by ImageMagick Studio LLC in any way that might state or imply that ImageMagick Studio LLC endorses your distribution;
- use any marks owned by ImageMagick Studio LLC in any way that might state or imply that you created the ImageMagick software in question.

It requires you to:

- include a copy of the license in any redistribution you may make that includes ImageMagick software;
- provide clear attribution to ImageMagick Studio LLC for any distributions that include ImageMagick software.

It does not require you to:

- include the source of the ImageMagick software itself, or of any modifications you may have made to it, in any redistribution you may assemble that includes it;
- submit changes that you make to the software back to the ImageMagick Studio LLC (though such feedback is encouraged).

A few other clarifications include:

- ImageMagick is freely available without charge;
- you may include ImageMagick on a CD-ROM as long as you comply with the terms of the license;
- you can give modified code away for free or sell it under the terms of the ImageMagick license or distribute the result under a different license, but you need to acknowledge the use of the ImageMagick software;
- the license is compatible with the GPL.

The legally binding and authoritative terms and conditions for use, reproduction, and distribution of ImageMagick follow:

Copyright 1999-2009 ImageMagick Studio LLC, a non-profit organization dedicated to making software imaging solutions freely available.

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 10 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication intentionally sent to the Licensor by its copyright holder or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non- exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non- exclusive, no-charge, royalty-free, irrevocable patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

b. You must cause any modified files to carry prominent notices stating that You changed the files; and

c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License.

A.4 GD Graphics Library license

Portions copyright 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Cold Spring Harbor Laboratory. Funded under Grant P41-RR02188 by the National Institutes of Health.

Portions copyright 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Boutell.Com, Inc.

Portions relating to GD2 format copyright 1999, 2000, 2001, 2002 Philip Warner.

Portions relating to PNG copyright 1999, 2000, 2001, 2002 Greg Roelofs.

Portions relating to gdttf.c copyright 1999, 2000, 2001, 2002 John Ellson (ellson@lucent.com).

Portions relating to gdft.c copyright 2001, 2002 John Ellson (ellson@lucent.com).

Portions copyright 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007 Pierre-Alain Joye (pierre@libgd.org).

Portions relating to JPEG and to color quantization copyright 2000, 2001, 2002, Doug Becker and copyright (C) 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, Thomas G. Lane. This software is based in part on the work of the Independent JPEG Group. See the file README-JPEG.TXT for more information.

Portions relating to WBMP copyright 2000, 2001, 2002 Maurice Szmurlo and Johan Van den Brande.

Permission has been granted to copy, distribute and modify gd in any context without fee, including a commercial application, provided that this notice is present in user-accessible supporting documentation.

This does not affect your ownership of the derived work itself, and the intent is to assure proper credit for the authors of gd, not to interfere with your productive use of gd. If you have questions, ask. "Derived works" includes all programs that utilize the library. Credit must be given in user-accessible documentation.

This software is provided "AS IS." The copyright holders disclaim all warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to this code and accompanying documentation.

Although their code does not appear in gd, the authors wish to thank David Koblas, David Rowley, and Hutchison Avenue Software Corporation for their prior contributions.

A.5 Bitstream Vera fonts license

The fonts have a generous copyright, allowing derivative works (as long as "Bitstream" or "Vera" are not in the names), and full redistribution (so long as they are not *sold* by themselves). They can be be bundled, redistributed and sold with any software.

The fonts are distributed under the following copyright:

Copyright © 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes NULL and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

A.6 Pixman license

The following is the MIT license, agreed upon by most contributors. Copyright holders of new code should use this license statement where possible. They may also add themselves to the list below.

Copyright 1987, 1988, 1989, 1998 The Open Group
Copyright 1987, 1988, 1989 Digital Equipment Corporation
Copyright 1999, 2004, 2008 Keith Packard
Copyright 2000 SuSE, Inc.
Copyright 2000 Keith Packard, member of The XFree86 Project, Inc.
Copyright 2004, 2005, 2007, 2008, 2009, 2010 Red Hat, Inc.
Copyright 2004 Nicholas Miell
Copyright 2005 Lars Knoll & Zack Rusin, Trolltech
Copyright 2005 Trolltech AS

A.7 LuaSocket license

A.8 librs232 license

A.9 UsbSerial license

A.10 SDL license

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

A.11 LGPL license

GNU LESSER GENERAL PUBLIC LICENSE Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W