Hollywood SDK 7.0

The Cross-Platform Multimedia Application Layer

Andreas Falkenhahn

Table of Contents

1 General information

1.1 Introduction

Welcome to the Hollywood Software Development Kit (SDK). This package contains all the necessary information and files to write your own Hollywood plugins. Plugins can greatly enhance Hollywood's functionality. They 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.

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 Mac OS X, 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. On AmigaOS and compatible systems, plugins in a compiled program's directory have a higher priority than those in `LIBS:Hollywood`. So if a plugin is present in both locations, Hollywood will load the one from the program's directory. When compiling application bundles for Mac OS X, plugins need to be put in the "Resources" directory of the application bundle, i.e. in `MyProject.app/Contents/Resources`.

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.

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. If you want to compile your own plugins, make sure that you compile in thumb mode.

1.2 Terms and conditions

The software is provided "as-is" and the author can not be made responsible of any possible harm done by it. You are using this software absolutely at your own risk. No warranties are implied or given by the author.

This software must not be distributed without the written permission of the author.

It is generally not allowed to release any kind of wrapper programs that make Hollywood commands available to other programming languages or the end-user. It is also generally not allowed to release any sort of mediator programs that would enable the user to access Hollywood commands through a mediating software.

No changes may be made to the SDK without the permission of the author.

This software is based in part on the Lua programming language by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo. See Section A.1 [Lua license], page 347, for details.

This software is based in part on the work of the Independent JPEG Group.

This software is based in part on the libpng link library by the PNG Development Group and the zlib link library by Jean-loup Gailly and Mark Adler.

This software is based in part on 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 347, for details.

This software is based in part on ImageMagick by ImageMagick Studio LLC. See Section A.3 [ImageMagick license], page 347, for details.

This software is based in part on the GD Graphics Library by Thomas Boutell. See Section A.4 [GD Graphics Library license], page 351, for details.

This software uses the pixman library. See Section A.6 [Pixman license], page 352, 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 351, 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.7 [LGPL license], page 353, for details.

Amiga is a registered trademark of Amiga, Inc. All other trademarks belong to their respective owners.

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.

1.3 Requirements

This SDK is targetted at C programmers only. Using C for writing Hollywood plugins is recommended because C compilers are available for a multitude of platforms and thus it is usually not too difficult to make your plugin available for more than one platform.

You need a C compiler and some knowledge about how to write shared libraries in C and how to compile them using your C compiler.

It's recommended to use the following compilers:

`Windows:` Microsoft Visual C

`Linux, Android, Mac OS X, AmigaOS 4, AROS, MorphOS:`
gcc

`AmigaOS 3:`
vbcc

`WarpOS:` vbcc

1.4 Examples

The Hollywood SDK comes with several example plugins and makefiles for lots of different platforms. You can study the source code of these examples to learn more about the practical aspects of writing a Hollywood plugin. The following example plugins are currently included with the SDK:

`AIFF:` This is a plugin of type `HWPLUG_CAPS_SOUND` which adds a loader for the AIFF audio file format popular on the Mac OS platforms to Hollywood. The AIFF file format's layout is very easy to understand so this plugin is a good starting point for learners who want to write a sound plugin. See Section 19.1 [Sound plugins], page 155, for details.

`Library:` This is a plugin of type `HWPLUG_CAPS_LIBRARY`. It adds some primitive functions and constants to Hollywood and is a good starting point if you want to extend Hollywood's command set using your own functions. It can also be used to learn about the way the Lua VM works. See Section 15.1 [Library plugins], page 129, for details.

`PCX:` This is a plugin of type `HWPLUG_CAPS_IMAGE` which adds a loader for the PCX image file format to Hollywood. Have a look at this source code if you intend to

write an image loader plugin for Hollywood. See Section 13.1 [Image plugins], page 117, for details.

As an exercise, you might want to use the AIFF or PCX plugin source codes as a starting point and extend them to include a saver for these formats as well. See Section 14.1 [Image saver plugins], page 125, for details. See Section 18.1 [Sample saver plugins], page 151, for details.

More plugin source codes are available at the official Hollywood portal at `http://www.hollywood-mal.com`. You can find them in the download section next to the binary distributions of the plugins.

1.5 History

Please consult the Hollywood documentation for a detailed list of changes in the course of Hollywood's evolution.

1.6 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 Conceptual overview

2.1 Getting started

The best way to learn to write Hollywood plugins is a learning-by-doing based approach. Just take a look at the source code of one of the example plugins that come with this SDK and use this documentation to try to understand what is going on there. After you have understood the structure and operation modes of some basic plugins, you can start to extend them by adding your own code, using this documentation as a reference manual.

This SDK expects that you are familiar with the C language and know how to use a C compiler to create libraries. If you aren't an experienced Amiga programmer, it is advised that you start out on Windows or Linux first, because it's much easier to write Hollywood plugins on these systems than on AmigaOS-based systems. Once you have developed a stable plugin on Windows or Linux and you are confident with Hollywood's plugin API, you may tackle an AmigaOS build as well, although there are several pitfalls that you have to avoid. See Section 3.1 [AmigaOS peculiarities], page 25, for details.

If you need help, don't hesitate to ask on the official Hollywood forums which are online at `http://forums.hollywood-mal.com/` There you will find a friendly community from all around the world which can surely help you to solve your programming problems.

2.2 Plugin types and Hollywood APIs

This documentation can be divided into two major parts: The first part describes all the different plugin types supported by Hollywood and their interfaces. There is detailed information on all the functions you will have to implement for the individual plugin types as well as on the way Hollywood interacts with them. As of Hollywood 7.0, the following plugin types are supported:

- Convert script plugins: These plugins automatically convert a custom file format into a Hollywood script. See Section 8.1 [Convert script plugins], page 53, for details.
- Library plugins: These plugins can add new commands and constants to Hollywood's script language. See Section 15.1 [Library plugins], page 129, for details.
- Image plugins: These plugins can provide loaders for additional image formats. See Section 13.1 [Image plugins], page 117, for details.
- Animation plugins: These plugins can provide loaders for additional animation formats. See Section 4.1 [Animation plugins], page 31, for details.
- Sound plugins: These plugins can provide loaders for additional sound and music formats. See Section 19.1 [Sound plugins], page 155, for details.
- Vectorgraphics plugins: These plugins can provide code for drawing vector-based graphics. See Section 21.1 [Vectorgraphics plugins], page 163, for details.
- Video plugins: These plugins can provide loaders for additional video formats. See Section 22.1 [Video plugins], page 175, for details.
- Image saver plugins: These plugins can provide savers for additional image formats. See Section 14.1 [Image saver plugins], page 125, for details.
- Animation saver plugins: These plugins can provide savers for additional animation formats. See Section 5.1 [Animation saver plugins], page 37, for details.

- Sample saver plugins: These plugins can provide savers for additional sample formats. See Section 18.1 [Sample saver plugins], page 151, for details.
- Require hook plugins: These plugins can hook into Hollywood's `@REQUIRE` preprocessor command to do their initialization only when explicitly asked to. See Section 17.1 [Require hook plugins], page 149, for details.
- Display adapter plugins: These plugins can completely replace Hollywood's internal display driver with a custom one. This is a very powerful plugin type and can be used to achieve quite astonishing things. See Section 10.1 [Display adapter plugins], page 59, for details.
- Timer adapter plugins: These plugins can completely replace Hollywood's internal timer handler with a custom one. See Section 20.1 [Timer adapter plugins], page 161, for details.
- Requester adapter plugins: These plugins can completely replace Hollywood's internal requester handler with a custom one. See Section 16.1 [Requester adapter plugins], page 137, for details.
- File adapter plugins: These plugins can hook into Hollywood's file handler and intercept any file they like. This can be used to make Hollywood magically able to handle compressed files or entirely new file formats. See Section 12.1 [File adapter plugins], page 105, for details.
- Directory adapter plugins: These plugins can hook into Hollywood's directory handler and intercept access on any directory they like. This can be used to allow Hollywood to deal with custom directory types, e.g. a plugin that treats a ZIP file as a directory could be written using the directory adapter plugin interface. See Section 9.1 [Directory adapter plugins], page 55, for details.
- Audio adapter plugins: These plugins can completely replace Hollywood's internal audio driver with a custom one. See Section 6.1 [Audio adapter plugins], page 43, for details.
- Extension plugins: This is a special plugin type that does not offer any functionality on its own. Its only purpose is to extend other plugin types. See Section 11.1 [Extension plugins], page 101, for details.

The second major part of this manual is the documentation of the functions that are publically exposed by Hollywood to every plugin. When Hollywood calls the `InitPlugin()` function of a plugin, it passes a pointer to a `hwPluginAPI` data structure which contains pointers to all the API functions that Hollywood has made publically available. Currently, the `hwPluginAPI` structure looks like this:

```
typedef struct _hwPluginAPI
{
    int hwVersion;
    int hwRevision;
    hwCRTBase *CRTBase;
    hwSysBase *SysBase;
    hwDOSBase *DOSBase;
    hwGfxBase *GfxBase;
    hwAudioBase *AudioBase;
```

```
    hwRequesterBase *RequesterBase;
    hwFontBase *FontBase;
    hwFT2Base *FT2Base;
    hwLuaBase *LuaBase;
    hwMiscBase *MiscBase;

    /****** V5.3 vectors start here *****/
    hwZBase *ZBase;
    hwJPEGBase *JPEGBase;

    /****** V6.0 vectors start here *****/
    hwPluginLibBase *PluginBase;
    hwUtilityBase *UtilityBase;

    /****** V7.0 vectors start here *****/
    hwUnicodeBase *UnicodeBase;
} hwPluginAPI;
```

All the individual structure members point to libraries, i.e. collections of functions that your plugin can use to interact with Hollywood. The way you have to call these functions is described in the second major part of this documentation. See Section 28.1 [GfxBase functions], page 223, to find out about all functions supported by the GfxBase library, for example.

2.3 What is a Hollywood plugin?

A Hollywood plugin is a shared library that is dynamically loaded by Hollywood at run time. Every Hollywood plugin has to export a number of function symbols that Hollywood can call when necessary. The actual file format of a plugin is platform-dependent. Here is an overview of the file formats used by Hollywood plugins on the individual platforms:

`Windows:` Hollywood plugins have to be compiled as standard Windows DLLs. Hollywood loads plugins using `LoadLibrary()`.

`Mac OS X:` Hollywood plugins have to be compiled as dynamic libraries (dylib). Hollywood loads plugins using `dlopen()`.

`Linux and Android:`
Hollywood plugins have to be compiled as shared objects. Hollywood loads plugins using `dlopen()`.

`AmigaOS and compatibles:`
Hollywood plugins have to be compiled as executables that can be loaded via `LoadSeg()`. As AmigaOS doesn't supported named symbol export, some glue code is necessary to allow access to named symbols. See Section 3.1 [AmigaOS glue code], page 25, for details. Another speciality on AmigaOS is that you cannot use certain functions from the standard ANSI C runtime library. See Section 3.2 [AmigaOS C runtime limitations], page 26, for details.

Please note that although Hollywood uses common file formats like DLLs on Windows, dylibs on Mac OS X, and shared objects on Linux/Android, the file extension of a Hollywood plugin always has to be `*.hwp`. Otherwise Hollywood won't be able to detect plugins.

2.4 Basic plugin design

Every Hollywood plugin needs to export the functions `InitPlugin()` and `ClosePlugin()`. As their names imply, these two functions initialize and close a plugin. By calling `InitPlugin()` Hollywood asks your plugin to identify itself, i.e. it needs to report certain information back to Hollywood, e.g. its feature set, author, version, minimum required Hollywood version and so on. Depending on this information, Hollywood will then import other function pointers from the plugin, depending on the feature set the plugin has reported to Hollywood.

To describe its feature set, the plugin sets the `hwPluginBase.CapsMask` field to a combination of capability bits taken from the `HWPLUG_CAPS_XXX` definitions in `hollywood/plugin.h`. Hollywood then knows what this plugin can do and will import the needed function pointers. For example, if `hwPluginBase.CapsMask` is set to `HWPLUG_CAPS_SOUND|HWPLUG_CAPS_SAVESAMPLE`, Hollywood will know that this plugin can load sound files and save sample files. Hollywood will therefore import the following additional function pointers from the plugin:

```
OpenStream()
CloseStream()
SeekStream()
StreamSamples()
GetFormatName()
RegisterSampleSaver()
SaveSample()
```

If one of the function pointers listed above cannot be resolved, Hollywood won't load this plugin.

See Section 7.3 [InitPlugin()], page 47, for details.

See Section 7.2 [ClosePlugin()], page 47, for details.

2.5 Auto and manual init plugins

All available plugins will be loaded automatically by Hollywood upon startup. However, not all plugins will be initialized automatically. Some plugin types will only be initialized if the user explicitly calls `@REQUIRE` on them. The reason for this is simple: Some plugin types can completely replace core components of Hollywood like the inbuilt display or audio driver. It would not make any sense to activate these plugins automatically upon startup because it could happen then that several plugins try to replace the same core component and it would also make it impossible to revert to Hollywood's inbuilt drivers. That's the reason why certain plugin types are only initialized if explicitly demanded by the user by calling `@REQUIRE` on them. Other plugin types, however, like loaders and savers for additional file formats are normally initialized automatically so that they are immediately available to all Hollywood applications right after their installation. Starting with Hollywood 6.0 it is possible to prevent automatic initialization of image, animation, sound, and video loader plugins using extension flags.

If you want to write a plugin that requires manual initialization, you need to set the `HWPLUG_CAPS_REQUIRE` capability flag in your `InitPlugin()` implementation. Hollywood will then call your plugin whenever the user runs the `@REQUIRE` preprocessor command on your plugin. You will then be able to perform all necessary initialization in your implementation of the `RequirePlugin()` call.

Here's a brief overview which plugin types are automatically initialized and which plugin types have to be manually initialized from your `RequirePlugin()` function:

`HWPLUG_CAPS_CONVERT:`
: Initialized automatically when plugin is loaded.

`HWPLUG_CAPS_LIBRARY:`
: Initialized automatically when plugin is loaded.

`HWPLUG_CAPS_IMAGE:`
: Initialized automatically when plugin is loaded. Starting with Hollywood 6.0 automatic loading can be disabled by setting the `HWEXT_IMAGE_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_ANIM:`
: Initialized automatically when plugin is loaded. Starting with Hollywood 6.0 automatic loading can be disabled by setting the `HWEXT_ANIM_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_SOUND:`
: Initialized automatically when plugin is loaded. Starting with Hollywood 6.0 automatic loading can be disabled by setting the `HWEXT_SOUND_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_VECTOR:`
: Initialized automatically when plugin is loaded. Starting with Hollywood 6.0, however, the user will have to call `SetVectorEngine()` manually to make Hollywood's vectorgraphics library use the plugin.

`HWPLUG_CAPS_VIDEO:`
: Initialized automatically when plugin is loaded. Starting with Hollywood 6.0 automatic loading can be disabled by setting the `HWEXT_VIDEO_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_SAVEIMAGE:`
: Initialized automatically when plugin is loaded.

`HWPLUG_CAPS_SAVEANIM:`
: Initialized automatically when plugin is loaded.

`HWPLUG_CAPS_SAVESAMPLE:`
: Initialized automatically when plugin is loaded.

`HWPLUG_CAPS_REQUIRE:`
: Initialized automatically when plugin is loaded.

`HWPLUG_CAPS_DISPLAYADAPTER`
: Use `hw_SetDisplayAdapter()` to initialize this plugin type. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

`HWPLUG_CAPS_TIMERADAPTER`
: Use `hw_SetTimerAdapter()` to initialize this plugin type. See Section 34.41 [hw_SetTimerAdapter], page 326, for details.

`HWPLUG_CAPS_REQUESTERADAPTER`
: Use `hw_SetRequesterAdapter()` to initialize this plugin type. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

`HWPLUG_CAPS_FILEADAPTER`
: Use `hw_AddLoaderAdapter()` to initialize this plugin type. See Section 34.2 [hw_AddLoaderAdapter], page 277, for details.

`HWPLUG_CAPS_DIRADAPTER`
: Use `hw_AddLoaderAdapter()` to initialize this plugin type. See Section 34.2 [hw_AddLoaderAdapter], page 277, for details.

`HWPLUG_CAPS_AUDIOADAPTER`
: Use `hw_SetAudioAdapter()` to initialize this plugin type. See Section 23.3 [hw_SetAudioAdapter], page 188, for details.

`HWPLUG_CAPS_EXTENSION:`
: This is a special plugin type that extends other plugin types. It does not offer any functionality on its own.

You will normally call the functions listed above in your `RequirePlugin()` implementation. See Section 17.2 [RequirePlugin], page 149, for details.

Some plugin types only support the initialization of a single plugin, e.g. it is not possible to have multiple display adapter plugins running. Only a single display adapter can be active at a time. Thus, only the first call to `hw_SetDisplayAdapter()` will succeed. All other attempts to install another display adapter will fail once a plugin has installed a custom display adapter.

2.6 Compiling plugins

All source files that include `hollywood/plugin.h` need to be compiled with certain preprocessor commands defined, depending on the platform. Here is a list of preprocessor commands that may need to be defined, depending on your platform:

`HW_AMIGAOS3`
: Needs to be defined for AmigaOS 3 builds.

`HW_AMIGAOS4`
: Needs to be defined for AmigaOS 4 builds.

`HW_ANDROID`
: Needs to be defined for Android builds.

`HW_AROS` Needs to be defined for AROS builds.

`HW_LINUX` Needs to be defined for Linux builds.

`HW_LITTLE_ENDIAN`
: Needs to be defined for little endian builds.

`HW_MACOS` Needs to be defined for Mac OS X builds.

`HW_MORPHOS`
Needs to be defined for MorphOS builds.

`HW_WARPOS`
Needs to be defined for WarpOS builds.

`HW_WIN32` Needs to be defined for Windows builds.

Also make sure to use the `HW_EXPORT` macro on all function declarations that you export as shared library functions to Hollywood, i.e.

```
HW_EXPORT int InitPlugin(hwPluginBase *self, hwPluginAPI *cl, STRPTR p)
{
    ...
}
```

Finally, don't forget to target your plugin for the right architecture. The 32-bit version of Hollywood can only load plugins compiled for a 32-bit architecture whereas the 64-bit version can only load plugins compiled for 64-bit. Modern compilers often default to 64-bit binaries nowadays so keep in mind that these binaries can't be loaded by the 32-bit versions of Hollywood. To make it easier for you to distinguish between 32-bit and 64-bit builds, the Hollywood SDK will automatically define the `HW_64BIT` preprocessor constant when it has detected that you are building for a 64-bit target.

On AmigaOS you also need to make sure that you do not link the compiler's startup code against the plugin as this can cause conflicts. You also must not use any library auto-open features provided by the compiler. You need to manually open all Amiga libraries that your plugin requires. See Section 3.2 [AmigaOS C runtime limitations], page 26, for details.

2.7 Error codes

Many functions require you to return a Hollywood error code to indicate success or failure. All standard error codes are defined in `hollywood/errors.h` but you can also register custom error codes using `hw_RegisterError()`. To indicate success, you have to return 0 which is equivalent to the constant `ERR_NONE`.

A special error code is `ERR_USERABORT`. If you return this error code, Hollywood will immediately shut down and quit without showing any error message.

2.8 Data types

The Hollywood SDK defines a few additional data types that are often used by its functions. They are all defined in `<hollywood/types.h>`. Here is a brief overview:

`APTR:` The arbitrary pointer. This is used to declare pointers to memory buffers that contain data of various sizes. The C equivalent to the `APTR` is the `void` pointer.

`STRPTR:` The string pointer. This data type is used to refer to null-terminated strings. See Section 2.11 [Unicode support], page 14, for details.

`UBYTE:` This data type is used for an unsigned byte (8-bit).

`UWORD:` This data type is used for an unsigned word (16-bit).

`ULONG:` This data type is used for an unsigned long-word (32-bit).

`DOSINT64:`
This is a platform-dependent data type for IO functions that deal with 64-bit integers. On platforms that support 64-bit file IO, this is set to a signed 64-bit quantity whereas on platforms that do not support 64-bit file IO, e.g. AmigaOS 3.x, this is set to a signed 32-bit quantity, limiting large file support to 2 gigabytes.

`IPTR:` An unsigned integer that is large enough to hold a pointer. This is 4 bytes on 32-bit systems and 8 bytes on 64-bit systems.

2.9 Version compatibility

It is possible to write plugins which dynamically make use of features of newer Hollywood versions. However, you need to be very careful when trying to access structure members or functions that are not available in all Hollywood versions. For example, let's consider the `struct LoadSoundCtrl` structure which looks like this:

```
struct LoadSoundCtrl
{
    ULONG Samples;
    int Channels;
    int Bits;
    int Frequency;
    ULONG Flags;
    int SubSong;        // V5.3
    int NumSubSongs;    // V5.3
    STRPTR Adapter;     // V6.0
};
```

You can see that the `SubSong` and `NumSubSongs` members are not available in Hollywood versions before 5.3 and `Adapter` is not available before Hollywood 6.0. This means that if your plugin is targetted at Hollywood 5.0 and up, you must only access these fields if you've verified that the user is running at least Hollywood 5.3 or 6.0 respectively. Otherwise you are going to access uninitialized memory which can easily lead to a crash. Hollywood will pass its version and revision numbers to your `InitPlugin()` function. It's recommended to store this information somewhere in your plugin so that your `OpenStream()` function can verify that the user is running Hollywood 5.3 or 6.0 before accessing the new structure members. Such a check could look like this:

```
ctrl->Frequency = 48000;
ctrl->Channels = 2;
ctrl->Bits = 16;
ctrl->Samples = numsamples;
ctrl->Flags = HWSNDFLAGS_SIGNEDINT;

// only return NumSubSongs if Hollywood >= 5.3
if(hwver > 5 || (hwver == 5 && hwrev >= 3)) {
    ctrl->NumSubSongs = numsubsongs;
}
```

Furthermore, be extra careful when using `memset()` to zero-out structure memory. For example, to zero-out `struct LoadSoundCtrl` you could do the following inside your `OpenStream()` implementation:

```
memset(ctrl, 0, sizeof(struct LoadSoundCtrl));
```

This, however, will cause memory access faults when the plugin is loaded by Hollywood versions older than version 6.0 because in that case the structure pointer passed to your `OpenStream()` function will point to a memory block that is smaller than the one used by Hollywood 6.0. If you just do a `memset()` without verifying version numbers first, you will write to unallocated memory which is likely to crash Hollywood. Thus, always make sure to check version numbers before reading from or writing to structure memory passed to your plugin by Hollywood.

If this is too much hassle for you, you can also declare that your plugin requires at least Hollywood 6.0 in your `InitPlugin()` implementation. If you do that, then you won't have to do all these checks of course because all Hollywood versions earlier than 6.0 will simply refuse to load your plugin.

You also have to be careful when calling Hollywood API functions from the different library bases. On its man page, every library base API function is followed by a version number in brackets which indicates the Hollywood version in which this API was first introduced. Before calling Hollywood API functions you must make sure that the user is running a Hollywood version that has this API using one of the methods described above.

2.10 Tag lists

People familiar with AmigaOS programming will already know about the concept of tag lists which can be used to pass additional arguments to functions or receive additional return values. The Hollywood SDK makes extensive use of this concept so that the capabilities of the individual functions can be easily extended in the future simply by offering new tags.

A tag list is an array of a number of `struct hwTagList` elements that is terminated by a last element that has its `Tag` member set to 0. Many Hollywood SDK API functions accept such a tag list as their last argument in order to be extensible. `struct hwTagList` looks like this:

```
struct hwTagList
{
    ULONG Tag;
    union {
        ULONG iData;
        void *pData;
    } Data;
};
```

As you can see, each tag item is accompanied by a data item that can either be an integer value or a pointer. Tag items can either be used to pass additional parameters to a function or they can also be used to receive additional return values from a function by setting the `pData` pointer to a variable that is to receive an additional return value. The data that needs to be passed in the `iData` or `pData` elements depends on the tag of course. This is all documented alongside the respective API functions.

Here is an example of how a tag list is used with the `hw_SetAudioAdapter()` API:

```
struct hwTagList[3];

t[0].Tag = HWSAATAG_BUFFERSIZE;
t[0].Data.iData = 2048;
t[1].Tag = HWSAATAG_CHANNELS;
t[1].Data.iData = 8;
t[2].Tag = 0;

error = hw_SetAudioAdapter(self, HWSAAFLAGS_PERMANENT, t);
```

In this case, a tag list is used to pass some additional parameters to `hw_SetAudioAdapter()`.

2.11 Unicode support

Starting with version 7.0, Unicode is now fully supported by Hollywood. By default, Hollywood always runs in Unicode mode now which means that all strings that your plugin gets from Hollywood or that your plugin passes to Hollywood must be in the UTF-8 encoding. This even applies to standard C functions in CRTBase like `fopen()`, `stat()`, `rename()`, `remove()`, etc. Normally, those functions from the C runtime library aren't UTF-8 compatible on all systems (most notably they aren't UTF-8 compatible on Windows and AmigaOS), but when calling those standard C runtime functions through Hollywood's CRTBase they will expect UTF-8 strings.

Note that scripts may explicitly request running in compatibility mode which puts Hollywood back into ISO 8859-1 mode which was used before Hollywood 7.0. In that case, all strings that your plugin gets from Hollywood or passes to Hollywood must be in ISO 8859-1 encoding. Supporting this compatibility mode can greatly increase the complexity of your plugin which is why you should think about whether your plugin really needs to support this mode or if you just make Hollywood running in Unicode mode a requirement for your plugin.

If you do decide to support compatibility mode in your plugin, the recommended way to detect the compatibility mode is to install a callback for the `HWCB_ENCODINGCHANGE` type in your `InitPlugin()` implementation using `hw_RegisterCallback()`. This callback will then be called if a script explicitly requests running in compatibility mode.

Note that if you want your plugin to be compatible with Hollywood versions older than 7.0, you have to support the ISO 8859-1 compatibility mode as well of course, because before Hollywood 7.0 all strings were in ISO 8859-1 encoding anyway.

Here's how you could deal with both cases in your `InitPlugin()` implementation. We first check the Hollywood version. If it is older than 7.0, we run in the ISO 8859-1 encoding. If it is at least 7.0, we install an encoding change callback to find out whether Hollywood is running in compatibility mode or UTF-8 mode. Here is the code:

```
// we store the encoding used by Hollywood here
static int useencoding;

// this callback will be run if the encoding changes; note that this
// will never be called more than once; the encoding can only change
// during Hollywood's program startup, not on the fly!
```

```
static SAVEDS void encodingchange(int encoding, APTR userdata)
{
    useencoding = encoding;
}

HW_EXPORT int InitPlugin(hwPluginBase *self, hwPluginAPI *cl, STRPTR
    path)
{
    // on Hollywood 7 or better, UTF-8 is the default encoding whereas
    // older version use ISO 8859-1 instead
    if(self->hw_Version > 6) {
        useencoding = HWOS_ENCODING_UTF8;
    } else {
        useencoding = HWOS_ENCODING_ISO8859_1;
    }

    // do initialization here
    ....

    // if we are on Hollywood 7 or better, let us know if the script
    // requests compatibility mode
    if(hwcl->hwVersion > 6) {
        hwcl->SysBase->hw_RegisterCallback(HWCB_ENCODINGCHANGE, (APTR)
            encodingchange, NULL);
    }

    return TRUE;
}
```

To convert between different encodings there is a convenience function named `hw_ConvertString()`. See Section 34.8 [hw_ConvertString], page 281, for details.

2.12 File IO information

As Hollywood can deal with virtual files as well as with files linked into other files like applets or executables there are some things to attend to when writing plugins that deal with files. If you want your plugin to support all these Hollywood-specific extensions you must make sure that you only use IO functions provided by Hollywood in the `DOSBase` pointer to deal with files. If you use functions like `fopen()` from the ANSI C library instead, your plugin will only work with normal files that are physically existent on a system drive.

For example, when writing plugins that provide loaders for additional file formats like images, sounds, or videos it can often happen that the filename that is passed to your plugin is a specially formatted specification that Hollywood uses to load files that have been linked to applets or executables. If you do not use Hollywood's IO functions to open this file, your plugin won't be able to load files that have been linked to applets or executables. This can be quite annoying for the end-user because the ability to link data files into applets and executables is a key functionality of Hollywood and thus your plugin should strive to be

compatible with it. If you use `fopen()` instead, it will just fail whenever your function is passed a specially formatted specification to open one of Hollywood's virtual files.

If your plugin has to use IO functions from the C runtime for some particular reason and you are unable to use the functions from `DOSBase` instead, you can translate virtual file specifications into physical files using the `hw_TranslateFileName()` or `hw_TranslateFileNameExt()` APIs. See Section 25.32 [hw_TranslateFileName], page 215, for details.

2.13 File attributes

The following attributes are supported by Hollywood for file system objects, i.e. files and directories:

`HWOS_FILEATTR_READ:`
: Read access is granted (user scope). This is unsupported on Win32.

`HWOS_FILEATTR_WRITE:`
: Write access is granted (user scope). This is unsupported on Win32.

`HWOS_FILEATTR_DELETE:`
: Delete access is granted. This is only supported on AmigaOS and compatibles.

`HWOS_FILEATTR_EXECUTE:`
: Execute access is granted (user scope). This is unsupported on Win32.

`HWOS_FILEATTR_PURE:`
: File can be made resident. This is only supported on AmigaOS and compatibles.

`HWOS_FILEATTR_ARCHIVE:`
: Archive bit. This is supported on AmigaOS and Win32.

`HWOS_FILEATTR_SCRIPT:`
: File is an AmigaDOS script. This is only supported on AmigaOS and compatibles.

`HWOS_FILEATTR_HIDDEN:`
: File is hidden. This is supported on AmigaOS and Win32.

`HWOS_FILEATTR_SYSTEM:`
: User for system files on Win32.

`HWOS_FILEATTR_READG:`
: Read access is granted (group scope). This is unsupported on Win32 and AmigaOS.

`HWOS_FILEATTR_WRITEG:`
: Write access is granted (group scope). This is unsupported on Win32 and AmigaOS.

`HWOS_FILEATTR_EXECUTEG:`
: Execute access is granted (group scope). This is unsupported on Win32 and AmigaOS.

`HWOS_FILEATTR_READO:`
: Read access is granted (others scope). This is unsupported on Win32 and AmigaOS.

`HWOS_FILEATTR_WRITEO:`
: Write access is granted (others scope). This is unsupported on Win32 and AmigaOS.

`HWOS_FILEATTR_EXECUTEO:`
: Execute access is granted (others scope). This is unsupported on Win32 and AmigaOS.

`HWOS_FILEATTR_READONLY:`
: Only read-access is allowed for this file. Supported on Win32 only.

You'll have to deal with these attributes if you write a file or directory adapter.

See Section 12.12 [Stat], page 114, for details.

See Section 9.3 [NextDirEntry], page 55, for details.

2.14 Bitmap information

Hollywood supports two different kinds of bitmaps: Software bitmaps and hardware bitmaps. Software bitmaps, often also called device-independent bitmaps (DIBs), are bitmaps that are usually allocated in CPU memory. Hollywood will often need to read from and write to these bitmaps. That's why it's advised that they are stored in memory that the CPU can access efficiently. The downside of software bitmaps is that it is quite slow to draw them to the screen and that it's not possible to apply hardware-accelerated transformations like scaling, rotating, blending, etc. to them. This is only possible with hardware bitmaps.

Hardware bitmaps, on the other hand, are usually stored in GPU memory. They are often also called video or device-dependent bitmaps (DDBs). Hardware bitmaps are optimized for efficient blitting to the display and for hardware-accelerated transformations. Hollywood will never modify the pixels of hardware bitmaps using the CPU because this would be too slow. Instead, hardware bitmaps are uploaded to GPU memory once and then only the GPU is used to access hardware bitmaps. Only the Amiga versions of Hollywood have inbuilt support for hardware bitmaps. On all other systems hardware bitmap support is not available in Hollywood but can be provided by third party plugins by installing a display adapter using `hw_SetDisplayAdapter()` and setting the `HWSDAFLAGS_VIDEOBITMAPADAPTER` flag.

Software bitmaps always store the color and transparency channels in separate bitmaps. This is because Hollywood is still compatible with 15-bit and 16-bit screen modes which do not have enough room for an 8-bit alpha channel that carries transparency information. Thus, software bitmaps, even if they use 32-bits per pixel, will never contain alpha channel information in their most significant bits. This is always stored in a separate bitmap. Plugins which want to override Hollywood's inbuilt software bitmap handler by setting the `HWSDAFLAGS_BITMAPADAPTER` flag need to adhere to this design as well and allocate color and transparency channels separately. See Section 10.4 [AllocBitMap], page 62, for details.

Hardware bitmaps, on the other hand, can store color and transparency channels in any way they like because Hollywood will never access the pixels of hardware bitmaps directly. The way a plugin allocates hardware bitmaps is completely up to the plugin. A limitation

of hardware bitmaps is that they can only be drawn when Hollywood is in hardware double-buffer mode. Thus, if you want to write a plugin which offers support for hardware bitmaps, you also have to set the `HWSDAFLAGS_DOUBLEBUFFERADAPTER` flag along with `HWSDAFLAGS_VIDEOBITMAPADAPTER` or your hardware bitmap support won't be of much use.

If you want to replace Hollywood's inbuilt bitmap handler with your custom versions, you have to write a display adapter plugin. See Section 10.1 [Display adapter plugins], page 59, for details.

2.15 Pixel formats

Hollywood software bitmaps can currently use the following pixel formats:

`HWOS_PIXFMT_RGB15:`
: Two bytes per pixel. Colors are stored as 0rrrrrgg gggbbbbb.

`HWOS_PIXFMT_BGR15:`
: Two bytes per pixel. Colors are stored as 0bbbbbgg gggrrrrr.

`HWOS_PIXFMT_RGB15PC:`
: Two bytes per pixel. Colors are stored as gggbbbbb 0rrrrrgg.

`HWOS_PIXFMT_BGR15PC:`
: Two bytes per pixel. Colors are stored as gggrrrrr 0bbbbbgg.

`HWOS_PIXFMT_RGB16:`
: Two bytes per pixel. Colors are stored as rrrrrggg gggbbbbb.

`HWOS_PIXFMT_BGR16:`
: Two bytes per pixel. Colors are stored as bbbbbggg gggrrrrr.

`HWOS_PIXFMT_RGB16PC:`
: Two bytes per pixel. Colors are stored as gggbbbbb rrrrrggg.

`HWOS_PIXFMT_BGR16PC:`
: Two bytes per pixel. Colors are stored as gggrrrrr bbbbbggg.

`HWOS_PIXFMT_RGB24:`
: Three bytes per pixel. Colors are stored as rrrrrrrr gggggggg bbbbbbbb.

`HWOS_PIXFMT_BGR24:`
: Three bytes per pixel. Colors are stored as bbbbbbbb gggggggg rrrrrrrr.

`HWOS_PIXFMT_ARGB32:`
: Four bytes per pixel. Colors are stored as aaaaaaaa rrrrrrrr gggggggg bbbbbbbb.

`HWOS_PIXFMT_BGRA32:`
: Four bytes per pixel. Colors are stored as bbbbbbbb gggggggg rrrrrrrr aaaaaaaa.

`HWOS_PIXFMT_RGBA32:`
: Four bytes per pixel. Colors are stored as rrrrrrrr gggggggg bbbbbbbb aaaaaaaa.

`HWOS_PIXFMT_ABGR32:`
: Four bytes per pixel. Colors are stored as aaaaaaaa bbbbbbbb gggggggg rrrrrrrr.

`HWOS_PIXFMT_ALPHA8:`
: One byte per pixel. Stored as aaaaaaaa.

`HWOS_PIXFMT_MONO1:`
: One bit per pixel (visible and invisible).

Please note that although many pixel formats support the storage of alpha channel information next to the color channel information, Hollywood's software bitmaps always store alpha channel information in separate bitmaps for compatibility with 15-bit and 16-bit screen modes. See Section 2.14 [Bitmap information], page 17, for details.

Keep in mind that for 16-bit and 32-bit pixel formats the actual byte storage order is dependent on the endianness of the host system. The constants defined above always specify the byte order when reading words or longwords from memory. Thus, in case a little endian system is used, the actual byte order in memory will be inverted for all 16-bit and 32-bit pixel formats, i.e. if you access a bitmap that uses `HWOS_PIXFMT_ARGB32` as its pixel format on a little endian system, the bytes will actually be stored in BGRA order in memory so that you get an ARGB pixel whenever you read a longword from the pixel buffer. Conversely, bitmaps that use `HWOS_PIXFMT_BGRA32` will store bytes as ARGB on little endian systems so that you get BGRA pixels when reading a longword from the pixel buffer.

2.16 Differences between Hollywood and Lua

If you plan to write plugins that extend Hollywood's script language by installing new commands and constants, you will have to deal with the Lua VM which is at the heart of Hollywood. Hollywood uses Lua 5.0.2 as its virtual machine but with major modifications. Here is a non-exhaustive list of the differences between Lua 5.0.2 and Hollywood:

1. At a first glance, Hollywood in contrast to Lua does not seem to distinguish between lower and upper case characters for keywords, preprocessor commands, variable, function, and constant names. You can mix upper and lower case characters any way you please. Internally, however, Hollywood still does the distinction between upper and lower case in true Lua fashion. The reason why you don't notice this, is because Hollywood's parser converts everything to lower case when it parses your script so all the differences are levelled at parsing time already and you don't have to care about upper and lower case characters when writing your script. However, if you write a plugin and you push elements into the stack or pop them from the stack, you need to be very careful that you use lower case strings only when describing these elements. Otherwise the user won't be able to access the elements that you have pushed or you won't be able to access the elements the user has pushed because internally Hollywood still distinguishes between upper and lower case characters. This must be kept in mind when writing plugins that push/pop stack elements. Always use lower case characters for everything and your plugin will fit in just fine.
2. The handling of `Nil` is different between Hollywood and Lua. Comparing 0 against `Nil` will be `True` in Hollywood, but `False` in Lua. This change has been made to allow you to work with uninitialiazed variables. If you pass an uninitialized, i.e. a `Nil` variable to a function or you use an uninitialized variable in an equation, Hollywood will just treat this uninitialized variable as if its value was 0. Lua, on the other hand, will fail if you try do arithmetics with `Nil` variables or pass a `Nil` variable to a function which expects a numerical value. Hollywood will just assume a numerical value of 0 for all uninitialized variables. The only exception from this rule is with table elements. Hollywood will fail

if you try to index table elements that are `Nil`. It will not automatically assume 0 for them. That is why you have to explicitly initialize all table elements you want to use. Variables, on the other hand, don't have to be initialized explicitly. You can just use them and if they are still `Nil`, Hollywood will assume they are 0.

3. Hollywood does not support the boolean object type. In Hollywood, the values `True` and `False` are simply special constants that will be mapped to the numerical values 1 and 0 respectively. There is no special object type for boolean values. This means that comparing 0 against `False` will be `True` in Hollywood, whereas in Lua it would be `False` because you would be comparing two different object types. Internally, Lua's boolean API is still supported by the VM and your plugin could use the respective functions from LuaBase but this is not recommended since Hollywood itself will never use Lua's boolean object type. It will always just use numbers. Not to mention that it is impossible to pass a real Lua boolean value to one of your plugin's functions because the parser will map all the `True` and `False` keywords to plain numbers.
4. The syntax is different. Whereas Lua uses the `end` keyword to close all kinds of different scopes, Hollywood has scope-dependent closing keywords like `Wend`, `Next`, `EndIf` and so on to make script files better readable.
5. The operators are different. For example, Lua uses `~=` for the `not equal` operator whereas Hollywood uses `<>`. Hollywood also supports much more operators than Lua does. For example, Hollywood comes with a variety of bitwise operators that Lua is missing entirely.
6. Lua uses 1-based tables and arrays whereas in Hollywood they are 0-based as in almost every other programming language. Though 1-based arrays might make more sense from a strictly logical point of view, 0-based arrays are the de facto standard in the programming world.
7. Hollywood's preprocessor has support for preprocessor commands, e.g. `@BRUSH` or `@INCLUDE`. Preprocessor commands are prefixed by the at character (@).
8. Hollywood supports constants. Constants are always prefixed by the hash tag (#).
9. Hollywood supports additional program flow statements like `Repeat/Until` and `Switch/EndSwitch`.
10. Hollywood still supports labels and `Goto()` and `Gosub()`, although this is considered obsolete and is only included for compatibility with Hollywood 1.x.
11. Hollywood has a `continue` statement.
12. Hollywood introduces a new data type named `lua_ID` and the functions `luaL_checkid()` and `luaL_checknewid()` to deal with its object identifiers. See Section 2.17 [Object identifiers], page 21, for details.
13. There is a difference in the error handling when writing C functions that are callable from Lua. If an error occurs in a C function with Lua 5.0.2, your C function has to call the `luaL_error()` function which will directly jump to Lua's error handler. In Hollywood, however, you have to return an error code from your C function to indicate that an error has occurred. If that is not possible for some reason, you may also call `lua_throwerror()` to jump directly into Hollywood's error handler but the recommended way is returning an error code. The reason for this design is that working with error codes is preferable to doing a `longjmp()` because it gives your code a chance to free resources before it error-exits. Note that Lua functions like `luaL_checklstring()` and

`luaL_checknumber()` will still jump into the error handler directly, so be prepared to deal with this.

See Section 15.1 [Library plugins], page 129, for details.

See Section 30.1 [LuaBase], page 263, for details.

2.17 Object identifiers

All Hollywood objects like brushes, videos, samples, etc. have an object identifier that is used to refer to the object when calling a function that deals with this object type. Hollywood objects can use two different kinds of identifiers: They can either use a numerical identifier or an automatically chosen identifier that uses the `LUA_TLIGHTUSERDATA` object type internally. The user can request an automatically chosen identifier by passing `Nil` as the desired identifier when creating the object. In that case, Hollywood will automatically choose an identifier for the object and return it. This is usually done by using the raw memory pointer to the newly allocated object as an identifier because this guarantees its uniqueness.

Internally, Hollywood object identifiers are managed using the `lua_ID` structure which looks like this:

```
typedef struct _lua_ID
{
    int num;
    void *ptr;
} lua_ID;
```

Every Hollywood object has such a `lua_ID` associated with it. The two structure members are initialized like this:

`num:` If the object uses a numerical identifier, this identifier is stored in `num` and the `ptr` member is set to `NULL`. If the `ptr` member is not `NULL`, Hollywood will ignore whatever is in `num` and the object will automatically use the value in `ptr` as its identifier.

`ptr:` If the object has been created using automatic ID selection, this member contains the object's unique identifier of type `LUA_TLIGHTUSERDATA`. This is typically set to the raw memory pointer of the newly allocated object. If `ptr` is `NULL`, Hollywood will automatically use the numerical identifier specified in `num`.

Whenever you call an API function that expects an object identifier you need to pass a pointer to a `lua_ID` to it. For example, let's assume you want to use `hw_LockBrush()` to access the raw pixel data of brush 1. In that case, you'd have to call `hw_LockBrush()` like this:

```
lua_ID id = {1, NULL};
struct hwos_LockBrushStruct lb;
APTR handle;

handle = hw_LockBrush(&id, NULL, &lb);
if(handle) hw_UnLockBrush(handle);
```

If you are writing a plugin that has the `HWPLUG_CAPS_LIBRARY` capability flag set and you want to offer a function that accepts either a numerical or an automatically chosen object

identifier, you can use the `luaL_checkid()` function for that. `luaL_checkid()` will check whether the value at the specified stack position is a number or a light userdata value and then it will initialize the `lua_ID` structure accordingly. The function from above would look like this then:

```
static SAVEDS int plug_LockBrush(lua_State *L)
{
    lua_ID id;
    struct hwos_LockBrushStruct lb;
    APTR handle;

    luaL_checkid(L, 1, &id);

    handle = hw_LockBrush(&id, NULL, &lb);
    if(handle) hw_UnLockBrush(handle);
}
```

By using `luaL_checkid()` your function can be made to accept numerical as well as light user data identifiers without much hassle.

You can also register your own Hollywood object types by calling the `hw_RegisterUserObject()` function. See Section 34.35 [hw_RegisterUserObject], page 317, for details.

2.18 Designer compatibility

Hollywood Designer, the WYSIWYG editor for Hollywood scripts, also supports Hollywood plugins. However, it supports only a fraction of Hollywood's plugin API. Thus, you need to be very careful when calling plugin API functions because they might not be available when Hollywood Designer has opened your plugin.

The first version of Hollywood Designer that supports Hollywood plugins is version 4.0. Hollywood Designer supports the following plugin types:

`HWPLUG_CAPS_IMAGE`
: Plugin provides a loader for additional image formats. See Section 13.1 [Image plugins], page 117, for details.

`HWPLUG_CAPS_ANIM`
: Plugin provides a loader for additional animation formats. See Section 4.1 [Animation plugins], page 31, for details.

`HWPLUG_CAPS_VIDEO`
: Plugin provides a loader for additional video formats. See Section 22.1 [Video plugins], page 175, for details.

`HWPLUG_CAPS_SAVEIMAGE`
: Plugin provides a saver for additional image formats. See Section 14.1 [Image saver plugins], page 125, for details.

All the other plugin types are currently unsupported. Thus, you don't have to be careful about which plugin API functions you call if your plugin is of a type that Designer doesn't support anyway. Be careful about the `InitPlugin()` and `ClosePlugin()` functions, though.

These will be called for all plugins so you must be very careful about the plugin API functions you call from these functions.

Let's suppose you are going to write a plugin that adds an image loader and a file adapter. In that case you would set `hwPluginBase.CapsMask` to `HWPLUG_CAPS_IMAGE|HWPLUG_CAPS_FILEADAPTER`. File adapters usually call `hw_ConfigureLoaderAdapter()` from their `InitPlugin()` implementation. `hw_ConfigureLoaderAdapter()`, however, is not supported by Designer because Designer doesn't support file adapters. That's why you have to check if Hollywood or Hollywood Designer has opened your plugin before you call `hw_ConfigureLoaderAdapter()`. You can check for Hollywood Designer by looking at the first function of `LuaBase` and see if it is `NULL`. If it is, you can be sure that you are being called by Hollywood Designer. Your implementation of `InitPlugin()` could then look like this:

```
HW_EXPORT int InitPlugin(hwPluginBase *self, hwPluginAPI *cl, STRPTR p)
{
    int isdesigner = (cl && cl->LuaBase->lua_gettop == NULL);

    self->CapsMask = HWPLUG_CAPS_IMAGE|HWPLUG_CAPS_FILEADAPTER;
    self->hwVersion = 1;
    self->hwRevision = 0;
    ...

    if(cl) {
        if(!isdesigner) cl->SysBase->hw_ConfigureLoaderAdapter(...);
    }

    return TRUE;
}
```

The code above will only call `hw_ConfigureLoaderAdapter()` if `InitPlugin()` has been called by Hollywood. If you don't do this check, Hollywood Designer will crash on every startup because your plugin tries to jump to a function that is `NULL`. So make sure your `InitPlugin()` implementation is compatible with Hollywood Designer.

To find out which plugin APIs are supported by Hollywood Designer, look at the "Designer compatibility" section that is part of every function's documentation.

To find out which version of Hollywood Designer is calling you, you must use the `HWMCP_GETDESIGNERVERSION` tag with the `hw_MasterControl()` function. It is not sufficient to look at the `hwVersion` and `hwRevision` members of the `hwPluginBase` that is passed to your `InitPlugin()` implementation because these just contain the version of the Hollywood plugin API provided by the Designer version that is calling your plugin. For example, Designer 4.0 will identify itself as Hollywood 5.0 whereas Designer 4.5 will identify itself as Hollywood 6.0. To get the real Designer version numbers, use `HWMCP_GETDESIGNERVERSION`. See Section 34.24 [hw_MasterControl], page 291, for details.

2.19 Multithreading

Generally, all of Hollywood's API functions are not thread-safe and the plugin functions need also not be thread-safe except where stated. There are a few thread-safe API calls

and there are also a few plugin functions that must be thread-safe. In that case, this requirement is explicitly mentioned in this documentation. If nothing is mentioned in an API call's documentation, then the call is not thread-safe or need not be thread-safe in case it is a plugin function to be written by you.

2.20 Legacy plugins

Plugin support was introduced with Hollywood 1.5. However, Hollywood's plugin interface was completely redesigned for Hollywood 5.0 and isn't compatible to previous versions any more. Thus, all the documentation provided here only applies to Hollywood 5.0 and up. If you need to target older Hollywood versions, too, please contact me for developer information for older Hollywood versions.

3 AmigaOS peculiarities

3.1 Glue code

As AmigaOS doesn't support the export of named symbols from library files, Hollywood plugins need to provide some glue code on AmigaOS and compatible systems so that Hollywood can locate symbols by name instead of by jumptable.

After calling `LoadSeg()` on your plugin, Hollywood will look for two magic cookies that must be embedded in the `MagicCookie[]` array in the following structure:

```
typedef struct _hwAmigaEntry
{
    ULONG MagicCookie[2];
    int Platform;
    void *(*GetProcAddress)(STRPTR name);
} hwAmigaEntry;
```

Thus, your plugin needs to declare a global constant based on this structure so that Hollywood can identify the file as a Hollywood plugin. An example declaration could be like this:

```
const hwAmigaEntry entry = {
    {HWPLUG_COOKIE1, HWPLUG_COOKIE2},
    HWARCH_OS3,
    GetProcAddress,
};
```

Make sure that the compiler doesn't optimize this declaration away just because it isn't referenced anywhere. Otherwise Hollywood won't be able to load your plugin. As you can see `MagicCookie[]` needs to be set to `HWPLUG_COOKIE1` and `HWPLUG_COOKIE2` which are both defined in `hollywood/plugin.h`. Note that different cookies are used on little endian systems so make sure to define the preprocessor constant `HW_LITTLE_ENDIAN` if you target little endian systems.

It is very important to set `hwAmigaEntry.Platform` to the correct architecture constant. In the above example we set it to `HWARCH_OS3` indicating that this is a plugin for AmigaOS 3.

You can also see that the declaration above references a function named `GetProcAddress()`. You need to implement this function in your glue code as well. Hollywood calls this function whenever it needs to resolve a function pointer from a symbol name. Thus, your implementation of `GetProcAddress()` needs to look at the string argument it has received and then return the appropriate function pointer. This can be implemented using a lookup table like this:

```
static const struct
{
    STRPTR name;
    void *func;
} funcs[] =
{
```

```
    {"InitPlugin", (void *) InitPlugin},
    {"ClosePlugin", (void *) ClosePlugin},
    // ...more function pointers here, depending on plugin type
    {NULL, NULL}
};

HW_EXPORT void *GetProcAddress(STRPTR name)
{
    int k;

    for(k = 0; funcs[k].name; k++) {
        if(!strcmp(name, funcs[k].name)) return funcs[k].func;
    }

    return NULL;
}
```

Another speciality on AmigaOS is that you cannot use certain functions from the standard ANSI C runtime library. See Section 3.2 [AmigaOS C runtime limitations], page 26, for details.

3.2 C runtime limitations

As Hollywood plugins on AmigaOS are loaded using `LoadSeg()` they do not contain the C compiler's runtime library startup code and there is no standardized way of executing the constructor and destructor functions of your C compiler's runtime library from a Hollywood plugin. This means that you won't be able to use any functions from the ANSI C runtime that require the compiler constructor or destructor code. Instead, you either have to call into AmigaOS API functions directly or you can use the C runtime functions that Hollywood makes available to your plugin in `CRTBase` that is passed to your `InitPlugin()` function.

Here is a list of functions that typically require the constructor and destructor code of your C compiler and therefore cannot be used from Hollywood plugins. Depending on your compiler, there may be more functions which cannot be used:

`Memory allocation functions`
: Functions like malloc(), calloc(), realloc(), free(), strdup()...

`File IO functions`
: All functions that deal with file handles like fopen(), fclose(), fread(), fwrite(), fgetc()...

`Standard IO functions`
: Functions like printf(), sprintf(), scanf(), sscanf()...

`Date and time functions`
: Functions like time(), localtime(), mktime(), gettimeofday()...

`Locale dependent functions`
: Functions like toupper(), tolower(), isgraph()...

Please note that it is usually not possible to write Hollywood plugins in C++ on AmigaOS because C++ most of the time also needs custom compiler constructor and destructor code.

You also must not use any library auto-open features provided by the compiler. You need to manually open and close all Amiga libraries that your plugins requires.

Very experienced users might be able to work around all these limitations by finding a way to run the compiler's constructor and destructor code manually from the `InitPlugin()` and `ClosePlugin()` functions but this requires quite some effort and is different from compiler to compiler.

3.3 clib2 versus newlib

Developers targetting the AmigaOS 4 platform can choose between two different C runtime libraries: clib2 and newlib. clib2 is linked statically into the binary whereas newlib is available as a shared library. Since Hollywood uses clib2 it is recommended that you use clib2 for your plugins as well. This can be achieved by compiling your sources with the following compiler setting:

```
gcc -mcrt=clib2 ...
```

Please note that clib2 is not thread-safe by default. Thus, if you need a thread-safe C runtime you need to link against `clib2-ts` instead, i.e.

```
gcc -mcrt=clib2-ts ...
```

If you want to use newlib for your plugins, you need to be very careful not to mix clib2 and newlib structures and handles. For example, all handles (e.g. an stdio `FILE` handle) allocated by Hollywood will be clib2 handles since Hollywood uses clib2. Thus, you must not pass them to a newlib stdio function since the internal representation of these handles might differ. Also, you must be careful when using structures that are shared between the C runtime provided by Hollywood and your plugin. Hollywood's C runtime will use the structures as defined in the clib2 headers whereas your plugin will use the format defined the newlib headers.

Finally, you also have to open `newlib.library` on your own if you want to use newlib in your plugin.

3.4 __saveds keyword

If you compile plugins for the Motorola 680x0 processors or for WarpOS you have to make sure that all your functions that will be called by Hollywood are declared using the `__saveds` keyword or your compiler's equivalent of this keyword (some compilers use a function called `geta4()` or a `__geta4` keyword instead). This is to make sure that the compiler generates code that loads the near data pointer in register `a4` on each function entry so that your function is able to access its data. If you don't use `__saveds` in your functions that can be called from Hollywood, then the index register will still point to to the data section within Hollywood and not within your plugin which will lead to all sorts of trouble.

Note that you need not use `__saveds` for all your functions but only the ones that Hollywood will directly call into. This includes the functions your plugin exports, the Lua functions offered by your plugin as well as callback functions within your plugin whose pointers you pass to Hollywood functions.

When declaring your plugin functions, the `HW_EXPORT` macro will automatically set `__saveds` for you, e.g.

```
HW_EXPORT int InitPlugin(hwPluginBase *self, hwPluginAPI *cl, STRPTR p)
```

```
{
    ...
}
```

However, you will also need to use `__saveds` when defining the Lua functions for your `HWPLUG_CAPS_LIBRARY` plugin, e.g.

```
static SAVEDS int MyDiv(lua_State *L)
{
    double a = luaL_checknumber(L, 1);
    double b = luaL_checknumber(L, 2);

    // catch division by zero CPU exception and handle
    // it cleanly
    if(b == 0) return ERR_ZERODIVISION;

    lua_pushnumber(L, a / b);

    // push 1 to indicate one return value
    return 1;
}

static const struct hwCmdStruct plug_commands[] = {
    {"MyDiv", MyDiv},
    ...
    {NULL, NULL}
};
```

Here we use the macro `SAVEDS` which will only insert the `__saveds` keyword when building for 680x0 or WarpOS-based systems.

Finally, don't forget to set `__saveds` when writing callback functions that you pass to a Hollywood API call. These must also be declared with the `__saveds` keyword because, obviously, Hollywood calls into them, e.g.

```
static SAVEDS int dispatcher(APTR h, int op, APTR data, APTR udata)
{
    ...
}

handle = hw_AttachDisplaySatellite(&id, dispatcher, data, tags);
```

Make sure that you don't forget the `__saveds` keyword for all these functions! Trying to debug a crash that is caused by a missing `__saveds` declaration can be a really frustrating experience because very strange things will start to happen if the data index register hasn't been set up correctly.

3.5 Building for 68881 and 68882

Some extra care needs to be taken when compiling plugins for the 68881 and 68882 FPUs. Hollywood's plugin interface was designed to allow plugins compiled for 68881/2 to be used with the non-FPU version of Hollywood as well. This means that return values from

plugins compiled for 68881/2 must never be stored in FPU registers like `fp0`. Instead, return values must always be returned in CPU registers. If a 64-bit value is to be returned from a plugin that has been compiled for 68881/2, it must be returned in registers `d0` and `d1`, not in `fp0`. Even the FPU version of Hollywood will expect floating point return values in CPU registers. Hollywood will never look for them in FPU registers! This design makes it possible to use FPU-compiled plugins with the non-FPU version of Hollywood and also vice versa.

Most compilers, however, will use FPU registers for floating point return values by default. If you're using vbcc, you can change this behaviour by compiling your source codes using the `-no-fp-return` command line argument. If this is specified, vbcc will always use CPU registers for return values. If you compile your sources with `-no-fp-return` enabled, however, you'll soon run into another problem, namely that all the ANSI C runtime library functions which return floating point values will return them in `fp0`. This will cause conflicts because the compiler expects them in `d0` and `d1` now. To solve this problem, you will have to link your project against runtime libraries which also have been compiled using `-no-fp-return`. At the time of this writing, the standard vbcc distribution does not come with these special libraries, but they're available on request from Frank Wille, who is the maintainer of the Amiga vbcc distribution.

3.6 Building for WarpOS

If you build plugins for WarpOS, the `hwPluginAPI` pointer that is passed to your `InitPlugin()` will contain function pointers to PPC code. However, many of these function pointers will immediately do a context switch and run on the 68k context. You might already be aware of the fact that context switches are quite expensive so you should do your best to minimize the number of context switches your plugin has to perform. To achieve this, you need to know which of the functions in the `hwPluginAPI` base pointer run PPC-native and which of them need a context switch. So here's an overview on the functions that don't need a context switch. All other functions not listed here will immediately perform a context switch.

`CRTBase:` The following functions are available in PPC native versions: `malloc()`, `calloc()`, `realloc()`, `free()`, `strdup()`, `qsort()`, `stricmp()`, `strnicmp()`, `toupper()`, `tolower()`, `strtolower()`, `strtoupper()`, `gettimeofday()`, `time()`, `lrint()`, `strtol()`, `strtoul()`, `strtod()`, `vsscanf()`, `vsnprintf()`, `atol()`.

`SysBase:` The following functions are available in PPC native versions: `hw_GetSysTime()`, `hw_SubTime()`, `hw_AddTime()`, `hw_CmpTime()`, `hw_Delay()`, `hw_MasterControl()` (but this function is only PPC-native when querying the `HWMCP_GETPOWERPCBASE` tag to make Hollywood return its `PowerPCBase` pointer to you), `hw_TrackedAlloc()`, `hw_TrackedFree()`, `hw_AllocSemaphore()`, `hw_FreeSemaphore()`, `hw_LockSemaphore()`, `hw_UnLockSemaphore()`.

`DOSBase:` All functions here are implemented in 68k code and will immediately do a context switch.

`GfxBase:` All functions here are implemented in 68k code and will immediately do a context switch.

`AudioBase:`
: All functions here are implemented in 68k code and will immediately do a context switch.

`RequesterBase:`
: All functions here are implemented in 68k code and will immediately do a context switch.

`FontBase:`
: All functions here are implemented in 68k code and will immediately do a context switch.

`FT2Base:`
: All functions here are PPC-native and don't need a context switch. Note that the PPC-native `FT2Base` for WarpOS is not available before Hollywood 5.3. You need to make sure that your plugin was opened by Hollywood 5.3 at least before trying to make any calls in `FT2Base`.

`LuaBase:`
: All functions here are implemented in 68k code and will immediately do a context switch.

`ZBase:`
: All functions here are PPC-native and don't need a context switch.

`JPEGBase:`
: All functions here are PPC-native and don't need a context switch.

`PluginBase:`
: All functions here are implemented in 68k code and will immediately do a context switch.

`UtilityBase:`
: All functions here are PPC-native and don't need a context switch.

`UnicodeBase:`
: All functions here are PPC-native and don't need a context switch.

Also, be careful about structure alignment when developing for WarpOS. The WarpOS version of Hollywood doesn't use the traditional Amiga structure alignment of 2 bytes but is optimized for performance. This means, for example, that 64-bit double values will always be aligned on an 8 byte boundary.

4 Anim plugins

4.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_ANIM` set will be called whenever Hollywood has to load an animation. The plugin can check then whether the animation file is in a format that the plugin recognizes and if it is, it can open the animation and return the raw pixel data of the individual frames to Hollywood. This makes it possible to load custom animation formats with Hollywood.

By default, anim plugins are automatically activated when Hollywood loads them. Starting with Hollywood 6.0 this behaviour can be changed by setting the `HWEXT_ANIM_NOAUTOINIT` extension bit. If this bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you will have to manually call `hw_AddLoaderAdapter()` to activate your plugin. For example, you could call `hw_AddLoaderAdapter()` from your `RequirePlugin()` implementation. In that case, the anim plugin would only be activated if the user called `@REQUIRE` on it. If you do not call `hw_AddLoaderAdapter()` on a plugin that has auto-initialization disabled, it will only be available if the user addresses it directly through the `Loader` tag. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

4.2 CloseAnim

NAME

CloseAnim – close animation handle (V5.0)

SYNOPSIS

```
void CloseAnim(APTR handle);
```

FUNCTION

This function must close the specified animation handle that has been allocated by your plugin's `OpenAnim()` function. Hollywood will call `CloseAnim()` when it is done with your animation.

INPUTS

`handle` handle returned by `OpenAnim()`

4.3 FreeFrame

NAME

FreeFrame – free frame pixel data (V5.0)

SYNOPSIS

```
void FreeFrame(ULONG *raw);
```

FUNCTION

This function must free the raw pixel data returned by `LoadFrame()`.

RESULTS

`raw` frame buffer allocated by `LoadFrame()`

4.4 GetFrameDelay

NAME

GetFrameDelay – get frame presentation time stamp (V5.0)

SYNOPSIS

```
int delay = GetFrameDelay(APTR handle, int frame);
```

FUNCTION

This function must return the presentation time stamp of the specified frame. Frame indices are counted from 0 to number of frames minus 1. Presentation time stamp means the duration that this frame should be shown before skipping to the next frame. This value has to be specified in milliseconds. If the animation format doesn't support frame-based time stamps you can also return a global frame delay value here or even 0. Returning 0 for every frame will display the animation as fast as possible.

INPUTS

`handle`	handle returned by `OpenAnim()`
`frame`	index of frame to query (starts from 0)

RESULTS

`delay`	delay in milliseconds for the specified frame

4.5 LoadFrame

NAME

LoadFrame – load raw pixel data of a single frame (V5.0)

SYNOPSIS

```
ULONG *raw = LoadFrame(APTR handle, int frame, struct LoadAnimCtrl *ctrl);
```

FUNCTION

This function must load the specified frame and return its raw pixel data encoded as an array of 32-bit ARGB values. Frame indices are counted from 0 to number of frames minus 1. The returned pixel array must use the exact dimensions returned by `OpenAnim()`, i.e. it must contain exactly `width * height * 4` bytes. A line width different from the animation width is currently not supported.

This function also has to provide certain information about the frame it has just loaded. This information has to be written to the `struct LoadAnimCtrl` that is passed in the third parameter. This structure looks like this:

```
struct LoadAnimCtrl
{
    int Width;              // [unused]
    int Height;             // [unused]
    int LineWidth;          // [unused]
    int NumFrames;          // [unused]
    int AlphaChannel;       // [in]
    int ForceAlphaChannel;  // [out]
```

```
        STRPTR Adapter;          // [unused] -- V6.0
        ULONG Flags;             // [in]     -- V6.0
    };
```

In contrast to `OpenAnim()`, `LoadFrame()` only uses some members from the `struct LoadAnimCtrl` structure pointer. The following members are used by `LoadFrame()`:

`AlphaChannel:`
This will be set to `True` whenever the "LoadAlpha" tag has been set to `True`.

`ForceAlphaChannel:`
If you set this to `True`, Hollywood will automatically create an alpha channel for this frame. For example, if the user calls `OpenAnim()` on your animation but does not set the "LoadAlpha" tag to `True`, the frame will still get an alpha channel if you set "ForceAlphaChannel" to `True`. Note that this functionality was broken before Hollywood 6.0.

`Flags:` This member can be set to a combination of the following flags:

`HWANMFLAGS_TRANSPARENCY:`
This flag will be set whenever the user sets the "LoadTransparency" tag to `True`. You may then choose to write the frame's transparency information to its alpha channel and set the `ForceAlphaChannel` member to `True`. See above for more information. (V6.0)

Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions

Hollywood will call `FreeFrame()` to free the pixel array returned by this function.

INPUTS

`handle` handle returned by `OpenAnim()`

`frame` index of frame to load (starts from 0)

`ctrl` pointer to a `struct LoadAnimCtrl`

RESULTS

`raw` an array of raw 32-bit ARGB pixels

4.6 OpenAnim

NAME

OpenAnim – open an animation file (V5.0)

SYNOPSIS

```
APTR handle = OpenAnim(STRPTR filename, struct LoadAnimCtrl *ctrl);
```

FUNCTION

This function has to open the specified filename, check if it is in an animation format that the plugin wants to handle, and, if it is, return a handle to the animation back to Hollywood. Otherwise it has to return `NULL`. The handle returned by `OpenAnim()` is an

opaque datatype that only your plugin knows about. Hollywood will simply pass this handle back to your `LoadFrame()` function when it wants to have the raw pixel data of a single frame.

This function also has to provide certain information about the animation it has just loaded. This information has to be written to the `struct LoadAnimCtrl` that is passed in the second parameter. This structure looks like this:

```
struct LoadAnimCtrl
{
    int Width;                // [out]
    int Height;               // [out]
    int LineWidth;            // [out]
    int NumFrames;            // [out]
    int AlphaChannel;         // [out]
    int ForceAlphaChannel;    // [out]
    STRPTR Adapter;           // [in]  -- V6.0
    ULONG Flags;              // [in]  -- V6.0
};
```

The following information has to be written to the `struct LoadAnimCtrl` pointer by `OpenAnim()`:

`NumFrames:`
: Must be set to the number of frames in this animation.

`Width:`
: Must be set to the animation width in pixels. Note that Hollywood only supports animations which use the same width for every frame.

`Height:`
: Must be set to the animation height in pixels. Note that Hollywood only supports animations which use the same height for every frame.

`AlphaChannel:`
: Must be set to `True` or `False`, depending on whether or not this animation uses frames that have an alpha channel.

`ForceAlphaChannel:`
: If this is set to `True`, Hollywood will automatically create an alpha channel for all objects that load this animation. For example, if the user calls `OpenAnim()` on your animation but does not set the "LoadAlpha" tag to `True`, the animation will still get an alpha channel if you set "ForceAlphaChannel" to `True`. Note that this functionality was broken before Hollywood 6.0.

`Adapter:`
: Starting with Hollywood 6.0 users can specify the file adapter that should be used to open certain files. If this member is non-`NULL`, Hollywood wants your plugin to use the file adapter specified in `Adapter` to open the animation. This means that you have to use `hw_FOpenExt()` instead of `hw_FOpen()` to open the animation. Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions. See Section 25.16 [hw_FOpenExt], page 202, for details. (V6.0)

`Flags:`
: The following flags may be set by Hollywood:

`HWANMFLAGS_TRANSPARENCY:`
: This flag will be set whenever the user sets the "LoadTransparency" tag to `True`. You may then choose to write a frame's transparency information to its alpha channel and set the `ForceAlphaChannel` member to `True`. See above for more information. (V6.0)

Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions

Please note that you should not use ANSI C functions like `fopen()` to open the file that is passed to this function because the filename that is passed to this function can also be a specially formatted filename specification that Hollywood uses to load files that have been linked to applets or executables. In order to be able to load these files correctly, you have to use special IO functions provided by Hollywood. See Section 2.12 [File IO information], page 15, for details.

INPUTS

`filename` filename to open

`ctrl` pointer to a `struct LoadAnimCtrl` for storing information about the animation

RESULTS

`handle` a handle that identifies this animation or `NULL` if plugin doesn't want to handle this animation

5 Anim saver plugins

5.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_SAVEANIM` set can register one or more additional output animation formats. The user will then be able to save animations in the output formats supported by the plugin. Plugins have to register new output animation formats by passing the name of a constant that should be used to access the new format. For example, a plugin might choose to register a new output animation format under the constant `#ANMFMT_CUSTOMFORMAT`. Whenever the user calls a command like `SaveAnim()` now and passes `#ANMFMT_CUSTOMFORMAT` as the animation format, Hollywood will ask the plugin to save the animation.

5.2 BeginAnimStream

NAME

BeginAnimStream – create a new animation stream (V5.0)

SYNOPSIS

```
APTR handle = BeginAnimStream(STRPTR filename, int width, int height,
                  int format, int quality, int fps);
```

FUNCTION

This function must create a new animation stream in the specified filename. After Hollywood has called this function, it will then call `WriteAnimFrame()` to add a number of frames to your animation stream. Once all frames have been added, Hollywood will call `FinishAnimStream()` on the stream handle.

The 'format' parameter specifies the pixel format of the source frame data that will be passed by `WriteAnimFrame()` later. This can be one of the following constants:

`HWSAVEANMFMT_ARGB:`
: Data is delivered as a 32-bit array consisting of ARGB pixels.

`HWSAVEANMFMT_CHUNKY:`
: Data is delivered as 8-bit indices into a color look-up table.

You will only have to handle those formats that you have explicitly declared as supported when Hollywood called your `RegisterAnimSaver()` function.

The 'quality' parameter contains a value between 0 and 100 indicating the desired quality for the output file. Animation formats that use lossy compression can use this member to determine compression settings for the animation. Animation formats that don't use any compression or offer lossless compression can ignore this parameter.

The 'fps' parameter contains the desired playback rate for the animation in frames per second. This is not supported by all animation formats so you can ignore it if you want. Some animation formats also support a frame-based delay value that Hollywood will pass to you in its `WriteAnimFrame()` calls.

Please note that in case your plugin supports multiple output animation formats, you'll have to wait until the first call to `WriteAnimFrame()` on that stream until you can tell

which format the user has chosen for the stream. This inconvenience is due to a design flaw in Hollywood: Support for multiple output animation formats wasn't available before Hollywood 5.3 but the API was designed for Hollywood 5.0. So there's just no room for another parameter in the `BeginAnimStream()` prototype declaration because it doesn't accept a tag list or any other parameter that could by dynamically extended as Hollywood functionality increases. Thus, you'll have to wait until `WriteAnimFrame()` which gives you the information in the `FormatID` structure member.

This function has to return a handle to the stream if the animation has been successfully created or `NULL` if there was an error.

INPUTS

`filename` desired location for the animation file on disk

`width` animation width in pixels

`height` animation height in pixels

`format` format for the animation (see above)

`quality` quality for the animation (see above)

`fps` frames per second for the animation

RESULTS

`handle` animation handle or `NULL` in case of an error

5.3 FinishAnimStream

NAME

FinishAnimStream – finish animation stream (V5.0)

SYNOPSIS

```
int ok = FinishAnimStream(APTR stream);
```

FUNCTION

This function must finish all writes to the specified animation stream and then close its file handle so that the file can be used.

This function has to return `True` if the stream has been successfully finished or `False` in case of an error.

INPUTS

`stream` output anim stream handle created by `BeginAnimStream()`

RESULTS

`ok` `True` or `False` indicating success or failure

5.4 RegisterAnimSaver

NAME

RegisterAnimSaver – register a new animation saver (V5.0)

SYNOPSIS

```
void RegisterAnimSaver(struct SaveAnimReg *reg)
```

FUNCTION

Hollywood will call this function to get information about the animation saver your plugin wants to register. In addition, `RegisterAnimSaver()` has to tell Hollywood whether it wants to register another animation saver. Hollywood will pass a pointer to a `struct SaveAnimReg` to this function. This structure looks like this:

```
struct SaveFormatReg
{
    ULONG CapsMask;     [out]
    ULONG FormatID;     [in/out]
    STRPTR FormatName;  [out]
};

struct SaveAnimReg
{
    struct SaveFormatReg hdr;
};
```

Your implementation has to do the following with the individual structure members:

`CapsMask:`

This must be set to a combination of flags that tell Hollywood about the capabilities of the animation saver that is to be registered. The following flags are currently supported:

`HWSAVEANMCAPS_ARGB:`

Your animation saver supports source animation data that is delivered as a 32-bit ARGB pixel array.

`HWSAVEANMCAPS_CHUNKY:`

Your animation saver supports source animation data that is delivered as 8-bit chunky pixels that are index values for a palette look-up table.

`HWSAVEANMCAPS_ALPHA:`

Your animation saver supports alpha channel saving. This is only supported if you also set `HWSAVEANMCAPS_ARGB`.

`HWSAVEANMCAPS_MORE:`

If you set this flag, Hollywood will call `RegisterAnimSaver()` again so that you can register another saver. If you don't want to register another saver, don't set this flag. (V5.3)

Note that `HWSAVEANMCAPS_ARGB` and `HWSAVEANMCAPS_CHUNKY` are not mutually exclusive. You can set them both if the target animation format supports both true colour and palette-based pixel data storage.

`FormatID:`

This member must be set to a unique 32-bit value that should be assigned to the constant that is registered for accessing this animation saver from Hollywood scripts. Values smaller than 32768 are reserved for internal Hollywood use. You may use values larger than 32768 for your saver but if you want to publish your plugin, you need to contact Airsoft Softwair to obtain a unique value that is still vacant. This won't cost you anything; it's just needed to make sure that plugin animation savers don't use conflicting identifiers. Also, once you have published your animation saver plugin, the `FormatID` you have specified must not be changed or you will break compatibility with applets or executables that have been compiled with previous versions. If you are registering more than one animation saver using `HWSAVEANMCAPS_MORE`, you can look at the `FormatID` member to tell how many times Hollywood has already called `RegisterAnimSaver()` because `FormatID` will contain the identifier of the last animation saver you registered. If `FormatID` is 0, then this is the first call to `RegisterAnimSaver()`. Note that it is not recommended to keep your own counter because Hollywood might call `RegisterAnimSaver()` multiple times, i.e. it might first loop over `RegisterAnimSaver()` to determine how many animation savers there are in total and then it might loop over `RegisterAnimSaver()` again to actually register their names.

`FormatName:`

This must be set to a string that should form the second half of the constant that Hollywood registers for your animation saver. This string you specify here must follow the naming restrictions for Hollywood constants, i.e. only alphabetical characters, numbers and very few special characters like the underscore character are allowed. The `#ANMFMT_` prefix must not be included in the string you pass. Hollywood will add this automatically, i.e. if you pass the string "TESTFORMAT" here, Hollywood will make your animation saver available under the constant `#ANMFMT_TESTFORMAT`.

INPUTS

`reg` pointer to a `struct SaveAnimReg` to be filled out by your implementation

5.5 WriteAnimFrame

NAME

WriteAnimFrame – write single animation frame to disk (V5.0)

SYNOPSIS

```
int ok = WriteAnimFrame(APTR stream, struct SaveAnimCtrl *ctrl);
```

FUNCTION

This function must save the frame described in the second parameter to the animation stream that is passed in the first parameter. This stream pointer is a handle that has been created by the `BeginAnimStream()` function. The second parameter is a pointer to a `struct SaveAnimCtrl`. This structure looks like this:

```
struct SaveAnimCtrl
```

```
{
    APTR Data;        // [in]
    int *Palette;     // [in]
    int Modulo;       // [in]
    int Colors;       // [in]
    int TransIndex;   // [in]
    int Delay;        // [in]
    ULONG Flags;      // [in]
    ULONG FormatID;   // [in] -- V5.3
};
```

Hollywood passes the following information to your `WriteAnimFrame()` function:

`Data:` The pixel data to save to the frame. The actual format of this data depends on the `Format` member.

`Modulo:` Number of bytes used by a single row of pixel data. This may be more than needed to store for the width that has been passed to `BeginAnimStream()` since there may be some padding.

`Colors:` This contains the number of colors in the color look-up table passed in the `Palette` member. This member is only used if `HWSAVEANMFMT_CHUNKY` has been passed to `BeginAnimStream()`.

`Palette:` Contains the look-up table that you need to convert the chunky pixel values to RGB color values. This table consists of as many 32-bit ARGB values as has been set in the `Colors` member. Note that `Palette` is only used if `HWSAVEANMFMT_CHUNKY` has been passed to `BeginAnimStream()`.

`TransIndex:`
If this animation stream has been created using `HWSAVEANMFMT_CHUNKY` this member specifies the index of the color that should appear transparent in the animation. The value specified here is only valid if the `HWSAVEANMFLAGS_TRANSINDEX` flag has been set (see below).

`Delay:` The delay for this frame in milliseconds or 0 if there should be no delay. Not all animation formats support frame-based delaying.

`Flags:` Contains a combination of flags specifying further options:

`HWSAVEANMFLAGS_ALPHA:`
Pixel data contains alpha channel transparency values.

`HWSAVEANMFLAGS_TRANSINDEX:`
The `TransIndex` member contains the index of a palette entry that should be made transparent in the output animation.

`FormatID:`
This member contains the identifier of the animation format the frame should be saved in. You only need to look at this member if your plugin supports more than one output animation format. But be careful, you are only allowed to look at this member if the user is running at least Hollywood 5.3. Otherwise, you must not access this member because older versions of Hollywood don't support it. (V5.3)

This function has to return `True` if the frame has been successfully saved or `False` in case of an error.

INPUTS

`stream`	output anim stream handle created by `BeginAnimStream()`
`ctrl`	pointer to a `struct SaveAnimCtrl` containing the frame to be saved

RESULTS

`ok`	`True` or `False` indicating success or failure

6 Audio adapter plugins

6.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_AUDIOADAPTER` set can replace Hollywood's inbuilt audio driver with a customized version. This is a very powerful feature and allows you to reroute Hollywood's complete audio output through an entirely different backend, thus making it possible to adapt Hollywood to completely different audio environments.

Please note that audio adapters are not automatically initialized when Hollywood loads the plugin. Instead, you have to manually call `hw_SetAudioAdapter()` in your `RequirePlugin()` function to activate the audio adapter. The audio adapter will then only be activated if the user calls `@REQUIRE` on your plugin. Otherwise, Hollywood will use its default audio driver. See Section 2.5 [Auto and manual plugin initialization], page 8, for details.

See Section 23.3 [hw_SetAudioAdapter], page 188, for information on how to install your audio adapter.

This plugin type is supported since Hollywood 6.0.

6.2 AllocAudioChannel

NAME

AllocAudioChannel – allocate new audio channel (V6.0)

SYNOPSIS

```
APTR chandle = AllocAudioChannel(APTR handle, int fmt, int freq, int vol,
                   int (*feedproc)(APTR handle, APTR chandle, APTR buf,
                   int count, APTR userdata), ULONG flags, APTR userdata,
                   struct hwTagList *tags);
```

FUNCTION

This function must allocate a new audio channel on the audio device passed in the first parameter. Hollywood will inform you about the PCM sample format that should be played on this channel in parameters 2 to 4. The `fmt` parameter can be one of the following formats:

`HWSMPFMT_U8M:`
: Unsigned 8-bit mono PCM data is fed to this channel.

`HWSMPFMT_U8S:`
: Unsigned 8-bit stereo PCM data is fed to this channel.

`HWSMPFMT_S8M:`
: Signed 8-bit mono PCM data is fed to this channel.

`HWSMPFMT_S8S:`
: Signed 8-bit stereo PCM data is fed to this channel.

`HWSMPFMT_S16M:`
: Signed 16-bit mono PCM data is fed to this channel.

`HWSMPFMT_S16S:`
Signed 16-bit stereo PCM data is fed to this channel.

Parameter 3 contains the number of PCM frames that will be played on this channel per second. Common values are 44100 or 48000 here. Parameter 4 contains the desired volume for this channel. This can range from 0 (mute) to 64 (full volume).

Hollywood will also pass a pointer to an audio feed procedure to this function. Whenever you need more PCM data to play on this audio channel, call this feed procedure. The prototype of this procedure looks like this:

```
int feedproc(APTR hdl, APTR ch, APTR buf, int count, APTR userdata);
```

Here is how you have to call this feed procedure:

`hdl:` This parameter must be set to the handle of the audio device opened via `OpenAudio()`.

`ch:` This parameter must be set to the handle of the audio channel allocated via `AllocAudioChannel()`.

`buf:` You have to pass a pointer to a memory buffer that should receive the new PCM data here.

`count:` You have to pass the number of PCM frames you want to receive here. Please note that this parameter must be specified in PCM frames, not in bytes. So if you set this to 1024 and your PCM samples are formatted as 16-bit wide stereo frames, the buffer you pass would have to be at least 4096 bytes in size. You should request PCM frames from Hollywood only in small portions. For a playback rate of 44100 frames per second, a request of 2048 PCM frames per `feedproc()` call is a reasonable size.

`userdata:`
You always have to set this parameter to the user data pointer that Hollywood has passed to your `AllocAudioChannel()` function here.

`feedproc()` returns the number of PCM frames successfully copied. Once again, be careful that this value is in PCM frames, not in bytes (see above). If this value is less than you requested, the channel has finished playing and Hollywood will soon call `FreeAudioChannel()` on it.

The feed procedure that Hollywood passes to you is thread-safe so you can call this from worker threads or audio interrupts as well.

INPUTS

`handle` audio device allocated by `OpenAudio()`

`fmt` format of PCM data passed to this channel (see above)

`freq` number of PCM frames per second to be played on this channel

`vol` initial channel volume (0 to 64)

`feedproc` pointer to a function that needs to be called to request more PCM data from Hollywood

`flags` reserved for future use (currently 0)

`userdata` userdata that needs to be passed to `feedproc()`

`tags` reserved for future use (currently `NULL`)

RESULTS

`chandle` handle to this audio channel or `NULL` on error

6.3 CloseAudio

NAME

CloseAudio – close audio device (V6.0)

SYNOPSIS

```
void CloseAudio(APTR handle);
```

FUNCTION

This function must close the specified audio device handle that has been opened by `OpenAudio()`.

INPUTS

`handle` audio device allocated by `OpenAudio()`

6.4 FreeAudioChannel

NAME

FreeAudioChannel – free audio channel (V6.0)

SYNOPSIS

```
void FreeAudioChannel(APTR handle, APTR chandle);
```

FUNCTION

This function must free the specified channel on the audio device.

INPUTS

`handle` audio device allocated by `OpenAudio()`

`chandle` audio channel allocated by `AllocAudioChannel()`

6.5 OpenAudio

NAME

OpenAudio – open audio device (V6.0)

SYNOPSIS

```
APTR handle = OpenAudio(ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function must open your plugin's audio device and return a handle to it. The `flags` and `tags` parameters are currently unused and reserved for future use.

INPUTS

`flags` reserved for future use (currently 0)

`tags` reserved for future use (currently `NULL`)

RESULTS

`handle` handle to audio device or `NULL` on error

6.6 SetChannelAttributes

NAME

SetChannelAttributes – change audio channel attributes (V6.0)

SYNOPSIS

```
int error = SetChannelAttributes(APTR handle, APTR chandle,
                struct hwTagList *tags);
```

FUNCTION

This function must change attributes of the specified audio channel. Attributes are passed as a taglist. The following attributes may currently be changed:

`HWSCATAG_VOLUME:`
Change the channel's volume to the volume specified in the `iData` member of this tag item. Volumes are specified in the range of 0 (mute) to 64 (full volume).

`HWSCATAG_PANNING:`
Change the channel's panning value to the value specified in the `iData` member of this tag item. Panning values range from 0 (left speaker only) to 255 (right speaker only). The default panning is 128 which means centered audio output.

`HWSCATAG_PITCH:`
Change the channel's pitch to the value specified in the `iData` member of this tag item. The pitch value specifies the number of PCM frames that are to be played per second.

INPUTS

`handle` audio device allocated by `OpenAudio()`

`chandle` audio channel allocated by `AllocAudioChannel()`

`tags` taglist containing attributes to modify (see above)

RESULTS

`error` error code or 0 for success

7 Base plugin functions

7.1 Overview

The functions `InitPlugin()` and `ClosePlugin()` must be implemented by every plugin. Hollywood will load the plugin depending on the information returned to it by `InitPlugin()`.

7.2 ClosePlugin

NAME
ClosePlugin – close plugin (V5.0)

SYNOPSIS
```
void ClosePlugin(void);
```

FUNCTION
This function must free all resources allocated by the plugin. `ClosePlugin()` will be the final call Hollywood makes to your plugin. After that the plugin will be expunged from memory.

Important: Be very careful about the plugin API functions that you call here because `ClosePlugin()` can also be called by Hollywood versions that are older than the one you requested in your `InitPlugin()` implementation. See Section 7.3 [InitPlugin], page 47, for a detailed description of this issue.

INPUTS
none

7.3 InitPlugin

NAME
InitPlugin – init plugin (V5.0)

SYNOPSIS
```
int success = InitPlugin(hwPluginBase *self, hwPluginAPI *cl, STRPTR path);
```

FUNCTION
This function must initialize your plugin and report information about it back to Hollywood. Your `InitPlugin()` implementation must fill out all fields of the `hwPluginBase` structure that is passed to it:

```
typedef struct _hwPluginBase
{
    ULONG CapsMask;       // [out]
    int Version;          // [out]
    int Revision;         // [out]
    int hwVersion;        // [in/out]
    int hwRevision;       // [in/out]
    STRPTR Name;          // [out]
```

```
    STRPTR ModuleName;    // [out]
    STRPTR Author;        // [out]
    STRPTR Description;   // [out]
    STRPTR Copyright;     // [out]
    STRPTR URL;           // [out]
    STRPTR Date;          // [out]
    STRPTR Settings;      // [out]
    STRPTR HelpFile;      // [out]
} hwPluginBase;
```

Here is an explanation about the function of the different structure members:

`CapsMask:`
: This is a bitmask describing the capabilities of your plugin, i.e. which features your plugin provides. Hollywood uses this bitmask to determine which function pointers it has to import from your plugin. This member must be set to one or more of the following capabilities:

  `HWPLUG_CAPS_CONVERT`
  : Plugin can convert custom file types into Hollywood scripts. See Section 8.1 [Convert script plugins], page 53, for details.

  `HWPLUG_CAPS_LIBRARY`
  : Plugin adds new commands and constants. See Section 15.1 [Library plugins], page 129, for details.

  `HWPLUG_CAPS_IMAGE`
  : Plugin provides a loader for additional image formats. See Section 13.1 [Image plugins], page 117, for details.

  `HWPLUG_CAPS_ANIM`
  : Plugin provides a loader for additional animation formats. See Section 4.1 [Animation plugins], page 31, for details.

  `HWPLUG_CAPS_SOUND`
  : Plugin provides a loader for additional sound formats. See Section 19.1 [Sound plugins], page 155, for details.

  `HWPLUG_CAPS_VECTOR`
  : Plugin provides an implementation to draw vector graphics. See Section 21.1 [Vectorgraphics plugins], page 163, for details.

  `HWPLUG_CAPS_VIDEO`
  : Plugin provides a loader for additional video formats. See Section 22.1 [Video plugins], page 175, for details.

  `HWPLUG_CAPS_SAVEIMAGE`
  : Plugin provides a saver for additional image formats. See Section 14.1 [Image saver plugins], page 125, for details.

  `HWPLUG_CAPS_SAVEANIM`
  : Plugin provides a saver for additional animation formats. See Section 5.1 [Animation saver plugins], page 37, for details.

`HWPLUG_CAPS_SAVESAMPLE`
: Plugin provides a saver for additional sound formats. See Section 18.1 [Sample saver plugins], page 151, for details.

`HWPLUG_CAPS_REQUIRE`
: Plugin wants to be called when the user does a `@REQUIRE` on it. See Section 17.1 [Require hook plugins], page 149, for details. (V6.0)

`HWPLUG_CAPS_DISPLAYADAPTER`
: Plugin replaces Hollywood's inbuilt display handler. See Section 10.1 [Display adapter plugins], page 59, for details. (V6.0)

`HWPLUG_CAPS_TIMERADAPTER`
: Plugin replaces Hollywood's inbuilt timer handler. See Section 20.1 [Timer adapter plugins], page 161, for details. (V6.0)

`HWPLUG_CAPS_REQUESTERADAPTER`
: Plugin replaces Hollywood's inbuilt requester handler. See Section 16.1 [Requester adapter plugins], page 137, for details. (V6.0)

`HWPLUG_CAPS_FILEADAPTER`
: Plugin provides a loader for additional file formats. See Section 12.1 [File adapter plugins], page 105, for details. (V6.0)

`HWPLUG_CAPS_DIRADAPTER`
: Plugin provides a loader for additional directory formats. See Section 9.1 [Directory adapter plugins], page 55, for details. (V6.0)

`HWPLUG_CAPS_AUDIOADAPTER`
: Plugin replaces Hollywood's inbuilt audio driver. See Section 6.1 [Audio adapter plugins], page 43, for details. (V6.0)

`HWPLUG_CAPS_EXTENSION:`
: This is a special plugin type that does not offer any functionality on its own. Its only purpose is to extend other plugin types. See Section 11.1 [Extension plugins], page 101, for details. (V6.0)

You have to implement all functions for every capability bit you set in `CapsMask` otherwise Hollywood will fail to load your plugin.

`Version:` Set this to the current version of your plugin.

`Revision:`
: Set this to the current revision of your plugin.

`hwVersion:`
: This contains the Hollywood version that has just opened your plugin. You should store this value somewhere because you might need it later to check whether a certain feature is available in this Hollywood version or not. After

that, set this member to the minimum Hollywood version required by your plugin.

`hwRevision:`
This contains the Hollywood revision that has just opened your plugin. You should store this value somewhere because you might need it later to check whether a certain feature is available in this Hollywood revision or not. After that, set this to the minimum Hollywood revision required by your plugin.

`Name:` Set this to a string describing the name of your plugin. This can contain spaces and need not be unique. Non-ASCII characters must be encoded as UTF-8.

`ModuleName:`
Set this to the module name of your plugin. The module name of your plugin must be identical to its file name minus the `*.hwp` extension. If file and module names do not match, Hollywood will refuse to load your plugin. This may only contain ASCII characters which are allowed in file names.

`Author:` Set this to the name(s) of the plugin author(s). Non-ASCII characters must be encoded as UTF-8.

`Description:`
Set this to a string describing the plugin's functionality. Non-ASCII characters must be encoded as UTF-8.

`Copyright:`
Set this to a string containing relevant copyright information. Non-ASCII characters must be encoded as UTF-8.

`URL:` Set this to a string containing a link to the plugin's website. This may be `NULL`.

`Date:` Set this to the build date of the plugin. This may be `NULL`. If set, it should use the format `"dd.mm.yy"`.

`Settings:`
This can be set to the full path of an external program that can be used to configure settings for your plugin. It is advised to store this program relative to your plugin's path. You can find out the full path of your plugin by looking at the third argument that is passed to `InitPlugin()`. This will tell you where the user has installed your plugin. If you set this member, the user will be able to launch the external program from the Hollywood GUI. If your plugin doesn't feature such a program, set this member to `NULL`.

`HelpFile:`
This can be set to the full path of a help file that acts as a user manual for the plugin. It is advised to store this help file relative to your plugin's path. You can find out the full path of your plugin by looking at the third argument that is passed to `InitPlugin()`. This will tell you where the user has installed your plugin. If you set this member, the user will be able to open this help file from the Hollywood GUI. If your plugin doesn't come with a help file, set this member to `NULL`.

Note that all string pointers you use to initialize the `hwPluginBase` structure must stay valid until `ClosePlugin()` is called.

The second parameter that is passed to `InitPlugin()` is a pointer to a `hwPluginAPI` vector which is the gateway to all plugin API functions provided by Hollywood. Plugin API functions are grouped into several different library bases like `GfxBase` and `SysBase`. Please note that this parameter can be `NULL`. If this is the case, your plugin should not initialize itself but just fill out the `hwPluginBase` structure and return `True`. If Hollywood passes `NULL` in `hwPluginAPI` it only wants to collect information about your plugin without actually loading it.

Important: You have to be very careful about the plugin API functions you call from your InitPlugin() implementation. This is because an older Hollywood version might have called your `InitPlugin()` function and if you try to call newer plugin API functions that are unavailable in the Hollywood version that has just called `InitPlugin()`, your plugin will crash terribly. You always have to check the `hwVersion` and `hwRevision` members of the `hwPluginBase` structure that is passed to your `InitPlugin()` function before you call any of the plugin APIs. These checks only have to be done in `InitPlugin()` and `ClosePlugin()` since they can be called by any Hollywood version. All the other functions of your plugin will only be called if the host Hollywood version matches the one you request in your `InitPlugin()` implementation using the `hwVersion` and `hwRevision` members. `InitPlugin()` and `ClosePlugin()`, however, may be called by any arbitrary Hollywood version and the only assumption you can make is that it will be at least Hollywood 5.0 which is calling you because 5.0 is the version that introduced the new plugin system explained here. **Please take this advice very seriously because a plugin which does not cleanly work with older Hollywood versions will also crash all executables compiled by these previous Hollywood versions because they will usually also scan and load all available plugins and if there is a plugin which isn't compatible with older versions, projects compiled with older Hollywood versions will suddenly crash badly.**

Additionally, `InitPlugin()` can also be called by Hollywood Designer. In that case you have to be careful as well, because Hollywood Designer supports only a subset of the official Hollywood plugin API so many function pointers inside `hwPluginBase` will be `NULL` and you cannot call them. That is why your `InitPlugin()` implementation also has to check whether it was called by Hollywood or by Hollywood Designer and act accordingly then. See Section 2.18 [Designer compatibility], page 22, for details.

The third parameter contains the full path to the plugin's location on the user's hard drive. This may be useful information if you need to load files from the plugin's directory or store preferences files in the plugin's directory, etc.

`InitPlugin()` has to return either `True` or `False` to signal success or failure. If it returns `False`, `ClosePlugin()` won't ever be called on this plugin.

INPUTS

`self`	pointer to structure that your plugin has to fill out
`cl`	pointer to Hollywood's plugin API functions or `NULL` (see above)
`path`	full path to the plugin's shared library file

RESULTS

`success` `True` or `False` indicating whether initialization was successful

8 Convert script plugins

8.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_CONVERT` set are called when Hollywood loads the file the user has passed to the program. The plugin can then examine the file and if it is in a format that the plugin recognizes, it can convert the file into a Hollywood script and return this script to Hollywood. Hollywood will then run the script it has received from the plugin instead of the file it was originally passed.

This makes it possible to enable Hollywood to open custom file formats. The Malibu plugin, which makes Hollywood able to open Scala project files, uses `HWPLUG_CAPS_CONVERT` for this job for example.

8.2 FreeScript

NAME

FreeScript – free converted script (V5.0)

SYNOPSIS

```
void FreeScript(STRPTR buf);
```

FUNCTION

This function must free the script returned by `GetScript()`. Note that this is called before Hollywood actually runs your script so if your `GetScript()` implementation has created some temporary files that are required by the script, you must not free them in `FreeScript()` but in `ClosePlugin()` which is called when Hollywood shuts down.

INPUTS

`buf` script buffer allocated by `GetScript()`

8.3 GetScript

NAME

GetScript – convert custom file to Hollywood script (V5.0)

SYNOPSIS

```
STRPTR buf = GetScript(STRPTR file);
```

FUNCTION

This function must examine the file that is passed to this function and if it is in a format that the plugin can handle, it must convert the file to a Hollywood script and return this script as a null-terminated string. Hollywood will then skip loading this file and it will run the script that it has received from `GetScript()` instead.

If `GetScript()` does not want to handle the file it is passed, it must return `NULL`.

For compatibility reasons, the returned script is expected to be in ISO 8859-1 encoding. You can change this by including the magic phrase `::utf8` in the very first 5 bytes that your `GetScript()` implementation returns. In that case, Hollywood will assume UTF-8 encoding for your script.

INPUTS

`file` path to a file that the user wants Hollywood to run

RESULTS

`buf` null-terminated string containing a Hollywood script or `NULL`

9 Directory adapter plugins

9.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_DIRADAPTER` set can hook into Hollywood's directory handler. Whenever Hollywood has to scan a directory, it will first ask all the plugins that have hooked themselves into Hollywood's directory handler if one of them wants to scan it instead.

Please note that directory adapters are not automatically initialized when Hollywood loads the plugin. Instead, you have to manually call `hw_AddLoaderAdapter()` in your `RequirePlugin()` function to activate the directory adapter. The directory adapter will then only be activated if the user calls `@REQUIRE` on your plugin. See Section 2.5 [Auto and manual plugin initialization], page 8, for details. If you do not call `hw_AddLoaderAdapter()` on your directory adapter plugin, it will only be available if the user addresses it directly through the `Adapter` tag.

See Section 34.2 [hw_AddLoaderAdapter], page 277, for information on how to add your directory adapter.

This plugin type is supported since Hollywood 6.0.

9.2 CloseDir

NAME

CloseDir – close a directory handle (V6.0)

SYNOPSIS

```
void CloseDir(APTR handle);
```

FUNCTION

This function must close the specified directory handle allocated by `OpenDir()`.

INPUTS

`handle` directory handle returned by `OpenDir()`

9.3 NextDirEntry

NAME

NextDirEntry – return next directory object (V6.0)

SYNOPSIS

```
int more = NextDirEntry(APTR handle, struct hwos_StatStruct *st,
               struct hwTagList *tags);
```

FUNCTION

This function has to return the next file system object from the directory handle passed in parameter 1. File system objects can be returned in a random order. They do not need to be alphabetically sorted. `NextDirEntry()` needs to return `True` if it has obtained a file system object from the directory or `False` if all entries have been retrieved. If it returns

`True`, it has to write information about the file system object retrieved to the `struct hwos_StatStruct` pointer passed in parameter 2. `struct hwos_StatStruct` looks like this:

```
struct hwos_StatStruct
{
    int Type;                                   // [out]
    DOSINT64 Size;                              // [out]
    ULONG Flags;                                // [out]
    struct hwos_DateStruct Time;                // [out]
    struct hwos_DateStruct LastAccessTime;      // [out]
    struct hwos_DateStruct CreationTime;        // [out]
    STRPTR FullPath;                            // [out]
    STRPTR Comment;                             // [out]
    int LinkMode;                               // [out]
    STRPTR Container;                           // [out]
};
```

Your `NextDirEntry()` implementation needs to write the following information to the individual structure members:

`Type:` This must be set to one of the following types:

`HWSTATTYPE_FILE:`
: The file system object is a file.

`HWSTATTYPE_DIRECTORY:`
: The file system object is a directory.

`Size:` Size of object in bytes if it is a file, 0 for directories. Note that this can also be set to -1 in case the file size isn't know, for example because the file is being streamed from a network source.

`Flags:` Combination of flags describing the file system object attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

`Time:` Time stamp indicating when this file system object was last modified. This information is optional. Do not touch this member if you don't have this time information.

`LastAccessTime:`
: Time stamp indicating when this file system object was last accessed. This information is optional. Do not touch this member if you don't have this time information.

`CreationTime:`
: Time stamp indicating when this file system object was created. This information is optional. Do not touch this member if you don't have this time information.

`FullPath:`
: Name of the file system object. This must be provided. The string pointer you use here must stay valid until the next call to `NextDirEntry()`. Please

note that in contrast to its name, `FullPath` must not be set to a fully qualified path but just to the name of the file system object without any path components.

`Comment:` Comment stored for this object in the file system. Set this to `NULL` if you do not have this information or the file system doesn't support storage of comments. The string pointer you pass here must stay valid until the next call to `NextDirEntry()`.

`LinkMode:`
Currently unused. Set to 0.

`Container:`
Currently unused. Set to `NULL`.

INPUTS

`handle` directory handle returned by `OpenDir()`

`st` pointer to a `struct hwos_StatStruct` for storing information about the file system object

`tags` reserved for future use (currently `NULL`)

RESULTS

`ok` `True` if file system object has been retrieved, `False` if there are no more objects

9.4 OpenDir

NAME

OpenDir – open a directory (V6.0)

SYNOPSIS

```
APTR handle = OpenDir(STRPTR name, int mode, struct hwTagList *tags);
```

FUNCTION

This function is called for every directory that Hollywood opens. Your `OpenDir()` implementation has to check whether your plugin wants to handle this directory or not. If your plugin wants to handle this directory, your `OpenDir()` implementation needs to open it and return a handle back to Hollywood. Otherwise `OpenDir()` must return `NULL`. The handle returned by this function is an opaque data type only your plugin knows about. Hollywood will pass this handle to you whenever it wants to get the next object from this directory.

The `mode` and `tags` parameters are currently unused and reserved for future use.

INPUTS

`name` directory to open

`mode` reserved for future use (currently 0)

`tags` reserved for future use (currently `NULL`)

RESULTS

`handle` handle to refer to this directory later or `NULL` if your plugin doesn't want to handle this directory

10 Display adapter plugins

10.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_DISPLAYADAPTER` set can replace Hollywood's inbuilt display handler with a custom display handler. This is a very powerful feature and allows you to reroute Hollywood's complete graphics output and event handler through an entirely different toolkit or display driver, thus making it possible to use Hollywood scripts in completely new environments.

Optionally, display adapters can also choose to override Hollywood's inbuilt bitmap handler with custom implementations and it is also possible to offer support for custom hardware bitmaps and double buffers from display adapters. This allows you to take full advantage of device dependent bitmaps for optimized drawing.

If you're just starting out with display adapters, it is advised that you begin with a pretty basic display adapter first and then you may choose to add support for advanced features later. Functions like `BltBitMap()` can get quite complex if your display adapter supports all the advanced functionality that Hollywood makes available, so it might be a better idea to first implement a barebones display adapter that doesn't support video bitmaps or hardware double buffers but just reroutes all of Hollywood's graphics through a custom device. You can then use this implementation as a basis to add support for more advanced features.

Also, be sure to benchmark raw performance of scripts with your display adapter and compare them to the performance of Hollywood's inbuilt display handler to see if your adapter needs optimizing. See Section 10.28 [HandleEvents], page 83, for details.

Please note that display adapters are not automatically initialized when Hollywood loads the plugin. Instead, you have to manually call `hw_SetDisplayAdapter()` in your `RequirePlugin()` function to activate the display adapter. The display adapter will then only be activated if the user calls `@REQUIRE` on your plugin. Otherwise, Hollywood will use its default display handler. See Section 2.5 [Auto and manual plugin initialization], page 8, for details.

You don't have to implement all functions offered by the display adapter API. Many functions are optional and only have to be implemented if you explicitly request their use in your call to `hw_SetDisplayAdapter()`. However, it is mandatory that all functions defined by the display adapter API are declared so that Hollywood can import their symbols when it loads the plugin. Functions that are optional and that you don't enable via `hw_SetDisplayAdapter()` can just be dummies then. Here is an overview of all display adapter APIs that are optional:

`Sleep()` Only used if activated by setting `HWSDAFLAGS_SLEEP`.

`VWait()` Only used if activated by setting `HWSDAFLAGS_VWAIT`.

`GetMonitorInfo()`
Only used if activated by setting `HWSDAFLAGS_MONITORINFO`.

`FreeMonitorInfo()`
Only used if activated by setting `HWSDAFLAGS_MONITORINFO`.

`GrabScreenPixels()`
: Only used if activated by setting `HWSDAFLAGS_GRABSCREEN`.

`FreeGrabScreenPixels()`
: Only used if activated by setting `HWSDAFLAGS_GRABSCREEN`.

`BeginDoubleBuffer()`
: Only used if activated by setting `HWSDAFLAGS_DOUBLEBUFFER`.

`EndDoubleBuffer()`
: Only used if activated by setting `HWSDAFLAGS_DOUBLEBUFFER`.

`Flip()` Only used if activated by setting `HWSDAFLAGS_DOUBLEBUFFER`.

`Cls()` Only used if activated by setting `HWSDAFLAGS_DOUBLEBUFFER`.

`AllocBitMap()`
: Only used if activated by setting `HWSDAFLAGS_BITMAPADAPTER`.

`FreeBitMap()`
: Only used if activated by setting `HWSDAFLAGS_BITMAPADAPTER`.

`LockBitMap()`
: Only used if activated by setting `HWSDAFLAGS_BITMAPADAPTER`.

`UnLockBitMap()`
: Only used if activated by setting `HWSDAFLAGS_BITMAPADAPTER`.

`GetBitMapAttr()`
: Only used if activated by setting `HWSDAFLAGS_BITMAPADAPTER`.

`AllocVideoBitMap()`
: Only used if activated by setting `HWSDAFLAGS_VIDEOBITMAPADAPTER`.

`FreeVideoBitMap()`
: Only used if activated by setting `HWSDAFLAGS_VIDEOBITMAPADAPTER`.

`ReadVideoPixels()`
: Only used if activated by setting `HWSDAFLAGS_VIDEOBITMAPADAPTER`.

`FreeVideoPixels()`
: Only used if activated by setting `HWSDAFLAGS_VIDEOBITMAPADAPTER`.

`DoVideoBitMapMethod()`
: Only used if activated by setting `HWSDAFLAGS_VIDEOBITMAPADAPTER`.

`AdapterMainLoop()`
: Only used if you set the `HWEXT_DISPLAYADAPTER_MAINLOOP` extension flag. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

See Section 28.29 [hw_SetDisplayAdapter], page 254, for information on how to install your display adapter.

This plugin type is supported since Hollywood 6.0.

10.2 ActivateDisplay

NAME

ActivateDisplay – activate a display (V6.0)

SYNOPSIS

```
void ActivateDisplay(APTR handle, ULONG flags);
```

FUNCTION

This function must assign the focus to the specified display. Optionally, the following flags may be set:

`HWACTDISPFLAGS_TOFRONT:`
: If this flag is set, the display should also be brought to the front.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`flags` additional options (see above)

10.3 AdapterMainLoop

NAME

AdapterMainLoop – enter display adapter's main loop (V6.1, optional)

SYNOPSIS

```
int error = AdapterMainLoop(lua_State *L, int (*f)(APTR data), APTR data,
                  ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function is optional and must only be implemented if the `HWEXT_DISPLAYADAPTER_MAINLOOP` extension bit has been set. See Section 11.1 [Extension plugins], page 101, for details. In that case, it must start the display adapter's main loop and after that run the function that is passed in the second parameter forwarding the third parameter `data` to it.

Normally, Hollywood doesn't explicitly ask display adapters to start their main loop since not all toolkits are main loop-based. Instead, by default, Hollywood repeatedly asks display adapters to wait for events by calling the `WaitEvents()` function and to handle events by calling the `HandleEvents()` function. Some toolkits, however, don't fit into this model very well because their API expects you to call a function which runs the main loop which in turn does all the event waiting and handling. If the toolkit your display adapter uses employs such a design, you can set the `HWEXT_DISPLAYADAPTER_MAINLOOP` extension bit to make Hollywood ask your plugin to start its main loop and call the function which hands back control to Hollywood after the main loop has been started.

Note that even if `HWEXT_DISPLAYADAPTER_MAINLOOP` has been set, Hollywood will still call your display adapter's `WaitEvents()` and `HandleEvents()` functions but you don't have to do anything in `WaitEvents()` then. It can just be an empty function. However, in `HandleEvents()` you still have to call into the master server and run timer callbacks. See Section 10.28 [HandleEvents], page 83, for details.

Since this function starts the main loop of your display adapter, it is not expected to return until Hollywood should shut down, i.e. until the user has closed all displays managed by your display adapter.

INPUTS

`L`	pointer to the `lua_State`
`f`	function you have to call right after starting your main loop
`data`	data you have to pass to the function passed in the second parameter
`flags`	reserved for future use (currently 0)
`tags`	reserved for future use (currently `NULL`)

RESULTS

`error`	error code or 0 for success

10.4 AllocBitMap

NAME

AllocBitMap – allocate a new software bitmap (V6.0, optional)

SYNOPSIS

```
APTR handle = AllocBitMap(int type, int width, int height, ULONG flags,
                  struct hwTagList *tags);
```

FUNCTION

This function has to allocate a software bitmap of the specified type in the requested dimensions. The `type` parameter can be one of the following constants:

`HWBMTYPE_RGB:`
: Hollywood wants you to allocate a color bitmap.

`HWBMTYPE_ALPHA:`
: Hollywood wants you to allocate an 8-bit alpha channel bitmap. Alpha channel bitmaps contain transparency information ranging from 0 (pixel is transparent) to 255 (pixel is fully opaque) for every pixel. Hollywood's software bitmaps always store alpha channel information for color bitmaps in separate bitmaps to be compatible with 15-bit and 16-bit screenmodes which don't have enough room to store an 8-bit alpha channel in a bitmap.

`HWBMTYPE_MASK:`
: Hollywood wants you to allocate a monochrome 1-bit mask bitmap. Mask bitmaps are used to store information about pixel transparency settings. In contrast to alpha channel bitmaps only two different states are supported: Visible pixels (1) or invisible pixels (0).

The `flags` parameter can be a combination of the following flags:

`HWABMFLAGS_CLEAR:`
: If this flag is set, Hollywood wants you to clear the bitmap after allocating. Clearing a bitmap means that all bits should be set to 0.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWABMTAG_FRIENDBITMAP:`
: If `type` is `HWBMTYPE_MASK`, the `pData` element of this tag item can contain a pointer to a color bitmap that Hollywood wants to use this mask for. The mask can then be allocated in a way that allows efficient blitting.

`HWABMTAG_DATA:`
: If this tag is given, your `AllocBitMap()` implementation must initialize the bitmap it has just allocated with the bits provided in the `pData` member of this tag. If this tag is set, Hollywood will also always pass the `HWABMTAG_DATABYTESPERROW` to inform your implementation about the byte length of a single row of the pixel array that has been passed to this function in the `pData` member. `HWABMTAG_DATA` is only used for types `HWBMTYPE_MASK` or `HWBMTYPE_ALPHA`.

`HWABMTAG_DATABYTESPERROW:`
: If `HWABMTAG_DATA` is set, this tag contains the byte width of a single row in its `iData` member. To find out the total size of the array passed to you in `HWABMTAG_DATA`, multiply this value with the height of the bitmap.

See Section 2.14 [Bitmap information], page 17, for more information on Hollywood bitmaps.

`AllocBitMap()` is an optional API and must only be implemented if `HWSDAFLAGS_BITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`type`	desired bitmap type (see above)
`width`	width of the bitmap in pixels
`height`	height of the bitmap in pixels
`flags`	allocation flags (see above)
`tags`	taglist for additional options (see above)

RESULTS

`handle`	handle to the bitmap or `NULL` in case of an error

10.5 AllocVideoBitMap

NAME

AllocVideoBitMap – allocate a video bitmap (V6.0, optional)

SYNOPSIS

```
APTR handle = AllocVideoBitMap(int width, int height, ULONG flags,
                  struct hwTagList *tags);
```

FUNCTION

This function must allocate a new video bitmap in the requested dimensions. Video bitmaps, also called hardware or device-dependent bitmaps (DDBs), are usually stored in GPU memory and can thus be drawn and transformed with hardware acceleration. Hollywood can only draw video bitmaps to the custom hardware double buffer implemented by your plugin. Thus, whenever you write a plugin that supports video bitmaps you will also have to implement a custom hardware double buffer and set the `HWSDAFLAGS_DOUBLEBUFFERADAPTER` flag accordingly. See Section 2.14 [Bitmap information], page 17, for details.

Hollywood will pass a taglist which contains further parameters for the operation to `AllocVideoBitMap()`. Your implementation must be able to deal with the following tags:

`HWAVBMTAG_DATA:`
: The `pData` member of this tag item will be set to an object that should be used to initialize the video bitmap's pixels. This tag will always be provided because there is no way to set the video bitmap's pixel data at a later stage. That is why the pixel data for the video bitmap is already provided by Hollywood at allocation time. If the `HWAVBMFLAGS_BITMAPDATA` flag is set, then `pData` will contain a handle to another video bitmap. This means that the new video bitmap should copy all pixels from this video bitmap handle. It is guaranteed that the video bitmap handle provided here matches the size of the new video bitmap that is to be allocated. If `HWAVBMFLAGS_BITMAPDATA` is not set, then `pData` contains a pointer to a 32-bit ARGB pixel array that contains the raw pixels that should be used to initialize the new video bitmap. This 32-bit ARGB array will always be of the size `width * height * 4`. No row padding is applied to this buffer.

`HWAVBMTAG_SRCWIDTH:`
: If this tag is set and the `HWAVBMTAG_MATRIX2D` tag is not passed, then the width value passed in parameter 1 is to be interpreted as a scaled value while the width passed in this tag's `iData` member contains the width of the source pixel data that is passed in `HWAVBMTAG_DATA`. This means that your implementation has to scale the source pixel data provided in `HWAVBMTAG_DATA` to the dimensions passed in parameters 1 and 2. It also has to take the scale mode into account that is passed in `HWAVBMTAG_SCALEMODE`. Hollywood uses this tag to offer hardware-accelerated scaling of video bitmaps. Please note that `HWAVBMTAG_SRCWIDTH` will only ever be passed to `AllocVideoBitMap()` if your plugin has explicitly declared that it supports video bitmap scaling by setting the `HWVBMCAPS_SCALE` flag in `HWSDATAG_VIDEOBITMAPCAPS`. If `HWAVMTAG_SRCWIDTH` and `HWAVBMTAG_MATRIX2D` are both passed, then you have to apply a transformation matrix to the source pixel data. See the documentation of `HWAVBMTAG_MATRIX2D` below for more information.

`HWAVBMTAG_SRCHEIGHT:`
: This is the height counterpart for `HWAVBMTAG_SRCWIDTH`. See above for a description on what this tag is used for.

`HWAVBMTAG_SCALEMODE:`
: If `HWAVBM_SRCWIDTH` and `HWAVBM_SRCHEIGHT` or `HWAVBM_MATRIX2D` are set, this tag will also be provided to tell you about the scale mode that Hollywood wants you to use. Currently, only 0 and 1 are supported here. A value of 0 in `iData` means that you should do hard scaling without any interpolation whereas a value of 1 means that Hollywood wants you to do use anti-alias interpolation. See above in the description of `HWAVBM_SRCWIDTH` for more information.

`HWAVBMTAG_MATRIX2D:`
: If this tag is set, then Hollywood wants you to apply a transformation to the source pixel data provided in `HWAVBMTAG_DATA` and store the transformation's resulting pixels in the new video bitmap that is to be allocated. Hollywood has already calculated the dimensions for the new video bitmap and passed them to this function in parameters 1 and 2. To get to know about the dimensions of the source pixel data, you have to examine the tags `HWAVBMTAG_SRCWIDTH` and `HWAVBM_SRCHEIGHT` which are always passed if `HWAVBMTAG_MATRIX2D` is set. The `pData` member of this tag item will be set to a pointer to a `struct hwMatrix2D` that contains all parameters for the transformation. Note that your implementation also has to take the scale mode that is passed in `HWAVBMTAG_SCALEMODE` into account. Hollywood uses this tag to offer hardware-accelerated transformation of video bitmaps. Please note that `HWAVBMTAG_MATRIX2D` will only ever be passed to `AllocVideoBitMap()` if your plugin has explicitly declared that it supports video bitmap transformation by setting the `HWVBMCAPS_TRANSFORM` flag in `HWSDATAG_VIDEOBITMAPCAPS`.

`HWAVBMTAG_DISPLAY:`
: If you've set the `HWSDAFLAGS_TIEDVIDEOBITMAP` flag when installing your display adapter using `hw_SetDisplayAdapter()` then this tag will always be provided and its `pData` member will be set to a handle to the display that this video bitmap should be allocated for. This will always be a handle returned by your `OpenDisplay()` implementation.

In addition to the taglist items described above, Hollywood can also set the following flags and your plugin has to be prepared to handle them:

`HWAVBMFLAGS_BLEND:`
: If this flag is set, then the data provided in `HWAVBMTAG_DATA` contains alpha channel transparency information and your plugin is expected to do the alpha blending with this data whenever it draws this bitmap. Note that the alpha channel data is always non-premultiplied.

`HWAVBMFLAGS_BITMAPDATA:`
: If this flag is set, then the data provided in `HWAVBMTAG_DATA` is a handle to another video bitmap which Hollywood wants you to use as the pixel data source. If this flag isn't set, then `HWAVBMTAG_DATA` contains a pointer to a 32-bit ARGB pixel array. See the documentation of `HWAVBMTAG_DATA` above for more information.

`AllocVideoBitMap()` is an optional API and must only be implemented if `HWSDAFLAGS_VIDEOBITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`width`	desired bitmap width in pixels
`height`	desired bitmap height in pixels
`flags`	allocation flags (see above)
`tags`	taglist containing additional parameters (see above)

RESULTS

`handle`	handle to the bitmap or `NULL` in case of an error

10.6 BeginDoubleBuffer

NAME

BeginDoubleBuffer – start hardware double buffer mode for display (V6.0, optional)

SYNOPSIS

```
int error = BeginDoubleBuffer(APTR handle, struct hwTagList *tags);
```

FUNCTION

This function must put the specified display into hardware double buffering mode. If Hollywood is in hardware double buffering mode, all calls to functions that draw something to the display like `BltBitMap()` or `RectFill()` must draw to the back buffer instead. This back buffer must then be drawn to the display whenever Hollywood calls the `Flip()` function of your plugin.

Please note that this function will only be called if the Hollywood script explicitly requests a hardware double buffer, i.e. the user has to set the hardware parameter to `True` when he calls Hollywood's `BeginDoubleBuffer()` function. If he doesn't do that, Hollywood will use its own software double buffering method and your plugin's `BeginDoubleBuffer()` function will never be called at all.

`BeginDoubleBuffer()` is an optional API and must only be implemented if `HWSDAFLAGS_DOUBLEBUFFERADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle`	display handle returned by `OpenDisplay()`
`tags`	reserved for future use (currently `NULL`)

RESULTS

`error`	error code or 0 for success

10.7 BltBitMap

NAME

BltBitMap – copy pixels from source bitmap to destination (V6.0)

SYNOPSIS

```
void BltBitMap(APTR bmap, APTR handle, struct hwBltBitMapCtrl *ctrl,
        ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function must copy pixels from a source bitmap to a destination which can be a display or another bitmap. The source bitmap can be a software bitmap allocated by Hollywood or your plugin or it can be a custom video bitmap allocated by your plugin's `AllocVideoBitMap()` function. `BltBitMap()` can be used in many different contexts so you need to pay close attention to implement it correctly. If you only need a barebones implementation that doesn't support offscreen rendering and video bitmaps, the implementation is really simple and straight-forward, though. See towards the end of the `BltBitMap()` documentation for information on what the most basic implementation of `BltBitMap()` has to look like.

In parameter 3, Hollywood will pass a `struct hwBltBitMapCtrl` which looks like this:

```
struct hwBltBitMapCtrl
{
    int SrcX;          // [in]
    int SrcY;          // [in]
    int DstX;          // [in]
    int DstY;          // [in]
    int Width;         // [in]
    int Height;        // [in]
    int ScaleWidth;    // [in]
    int ScaleHeight;   // [in]
    ULONG ScaleMode;   // [in]
    APTR Mask;         // [in]
    APTR Alpha;        // [in]
};
```

Here's a description of the individual structure members:

`SrcX:` Contains the x position in the source bitmap that marks the start offset for the copy operation. This is relative to the upper-left corner of the source bitmap.

`SrcY:` Contains the y position in the source bitmap that marks the start offset for the copy operation. This is relative to the upper-left corner of the source bitmap.

`DstX:` Contains the destination x position relative to the upper-left corner.

`DstY:` Contains the destination y position relative to the upper-left corner.

`Width:` Contains the number of columns to copy.

`Height:` Contains the number of rows to copy.

`ScaleWidth:`
: If you've passed the `HWSDAFLAGS_CUSTOMSCALING` flag in your call to `hw_SetDisplayAdapter()` then this will contain the desired scale width when autoscaling mode is active. If you haven't passed `HWSDAFLAGS_CUSTOMSCALING`, Hollywood will take care of scaling automatically and this member will always be zero.

`ScaleHeight:`
: If you've passed the `HWSDAFLAGS_CUSTOMSCALING` flag in your call to `hw_SetDisplayAdapter()` then this will contain the desired scale height when autoscaling mode is active. If you haven't passed `HWSDAFLAGS_CUSTOMSCALING`, Hollywood will take care of scaling automatically and this member will always be zero.

`ScaleMode:`
: If you've passed the `HWSDAFLAGS_CUSTOMSCALING` flag in your call to `hw_SetDisplayAdapter()` then this will contain the desired scale mode when autoscaling mode is active. This can be either 0 for brute force scaling or 1 for interpolated scaling using pixel antialiasing.

`Mask:`
: Contains a pointer to mask bitmap or `NULL` if there is no mask. This member is only used when drawing software bitmaps to video bitmaps or hardware double buffers.

`Alpha:`
: Contains a pointer to an alpha channel bitmap or `NULL` if there is no alpha channel. This member is only used when drawing software bitmaps to video bitmaps or hardware double buffers.

The way `BltBitMap()` should operate is defined by the `flags` parameter. The following flags are currently recognized:

`HWBBFLAGS_SRCVIDEOBITMAP:`
: The source bitmap passed in parameter 1 is a video bitmap allocated by `AllocVideoBitMap()`. If this flag isn't set, then the source bitmap will be a software bitmap that has either been allocated by Hollywood or by your plugin (if you've set the `HWSDAFLAGS_BITMAPADAPTER` flag). Note that video bitmaps can only be drawn to a hardware double buffer allocated by your plugin or to another offscreen video bitmap. Thus, if `HWBBFLAGS_SRCVIDEOBITMAP` is set you can be certain that Hollywood is either currently in hardware double buffer mode, i.e. it has previously called your plugin's `BeginDoubleBuffer()` function, or that the `HWBBFLAGS_DESTVIDEOBITMAP` flag is set.

`HWBBFLAGS_DESTVIDEOBITMAP:`
: The destination handle passed in parameter 2 is a video bitmap allocated by `AllocVideoBitMap()`. This can only happen if you've set the `HWVBMCAPS_OFFSCREENCOLOR` or `HWVBMCAPS_OFFSCREENALPHA` capabilities in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmaps. Otherwise, `HWBBFLAGS_DESTVIDEOBITMAP` will never be set.

`HWBBFLAGS_DESTALPHAONLY:`
: This is only set in connection with `HWBBFLAGS_DESTVIDEOBITMAP`. If Hollywood wants you to draw to the alpha channel of your video bitmap allocated by `AllocVideoBitMap()`, it will indicate this by setting `HWBBFLAGS_DESTALPHAONLY`. If `HWBBFLAGS_DESTVIDEOBITMAP` is set and `HWBBFLAGS_DESTALPHAONLY` isn't, you have to draw to the color channels of the video bitmap instead. Note that `HWBBFLAGS_DESTALPHAONLY` will only ever be set if you've set the `HWVBMCAPS_OFFSCREENALPHA` capability flag in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmap alphachannels.

`HWBBFLAGS_DESTBITMAP:`
: The destination handle passed in parameter 2 is a software bitmap allocated by Hollywood or your plugin's `AllocBitMap()` function if you've set the `HWSDAFLAGS_BITMAPADAPTER` in your call to `hw_SetDisplayAdapter()`. Note that `HWBBFLAGS_DESTBITMAP` will only ever be set if you've passed either `HWBMAHOOK_BLTBITMAP`, `HWBMAHOOK_BLTMASKBITMAP` or `HWBMAHOOK_BLTALPHAHOOK` in `HWSDAFLAGS_BITMAPHOOK`. Otherwise, Hollywood will do the rendering to the software bitmap on its own and you don't have to care. `HWBBFLAGS_DESTBITMAP` will only ever be set if you've explicitly requested that you want to do offscreen drawing to software bitmaps on your own by setting the appropriate bitmap hook flags.

`HWBBFLAGS_DONOTBLEND:`
: This flag indicates that `BltBitMap()` should not do any alpha blending. Instead, it should just do a raw copy of the color and alpha channel pixel data without any blending. This may only ever be set if the source or destination bitmap is a video bitmap allocated by your plugin, i.e. `HWBBFLAGS_DONOTBLEND` may only be set if `HWBBFLAGS_DESTVIDEOBITMAP` or `HWBBFLAGS_SRCVIDEOBITMAP` is defined, too. It will never be set when dealing with software bitmaps.

`HWBBFLAGS_IGNOREBKBUFFER:`
: If autoscaling is active and `HWSDAFLAGS_CUSTOMSCALING` hasn't been set in your call to `hw_SetDisplayAdapter()` this flag may be set to indicate that the source bitmap doesn't match the dimensions of your back buffer. Thus, if this flag is set you must not draw into the back buffer first. Instead, you must draw directly to your display. If `HWBBFLAGS_IGNOREBKBUFFER` is set, the destination handle will always be a display handle allocated by `OpenDisplay()`. Since `HWBBFLAGS_IGNOREBKBUFFER` is only used when Hollywood is in autoscale mode, it will always refresh the whole display, i.e. `SrcX`, `SrcY`, `DstX`, and `DstY` will all be 0 and `Width` and `Height` will correspond to the display's dimensions.

This may all sound quite complicated but in fact, the complexity of the `BltBitMap()` function can be greatly reduced if your plugin doesn't support offscreen drawing and video bitmaps. In that case, all your `BltBitMap()` implementation has to do is to grab the pixels from the source bitmap and draw them to the destination display. You won't have to support the `Mask` and `Alpha` members of the `struct hwBltBitMapCtrl` in that

case either, because they will only ever be set if `BltBitMap()` is used to draw a software bitmap to a video bitmap or a hardware double buffer.

To get the raw pixels of Hollywood bitmaps, you can use the `hw_LockBitMap()` function. If you've installed your own bitmap and video bitmap adapters, you can also directly interpret the handle pointers that are passed to this function because you've allocated them. It's not necessary to use `hw_LockBitMap()` on bitmaps that have been allocated by your plugin.

A typical implementation of `BltBitMap()` should do the following whenever it has to draw directly to the display: It should first draw the pixels into a back buffer (unless `HWBBFLAGS_IGNOREBKBUFFER` is set or a hardware double buffer is used) and then it should draw the pixels from this back buffer to the display.

This function doesn't have to do any clipping. Hollywood will perform clipping itself before calling `BltBitMap()`.

If your plugin supports hardware double buffering and Hollywood has put your display into hardware double buffering mode by calling your plugin's `BeginDoubleBuffer()` function, this function must not draw anything to the display but only to the back buffer. Hollywood will call your plugin's `Flip()` function when it wants you to draw the back buffer to the display.

You might want to use the `hw_RawBltBitMap()` function in your implementation to copy the pixels of bitmaps stored as raw pixel buffers. See Section 28.23 [hw_RawBltBitMap], page 248, for details.

INPUTS

`bmap`	source bitmap
`handle`	destination display or bitmap (depends on the flags that are set, see above)
`ctrl`	pointer to a `struct hwBltBitMapCtrl` containing parameters for the blit
`flags`	flags specifying how to copy the pixels (see above)
`tags`	additional options (currently always `NULL`)

10.8 ChangeBufferSize

NAME

ChangeBufferSize – change size of back buffer (V6.0)

SYNOPSIS

```
int error = ChangeBufferSize(APTR handle, int width, int height,
                ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function will usually be called by Hollywood when the display's size has changed. It tells you about the new size of Hollywood's internal back buffer and if your drawing mechanism is back buffer based as well, you need to adapt the size of your back buffer when Hollywood calls this function.

Please note that the back buffer size is not necessarily the same as the window's physical dimensions. If autoscaling is active, back buffer size and the window's physical dimensions can be different.

If you don't use a back buffer you won't have to do anything here. This is often the case for plugins which are designed to draw in hardware double buffer mode. In that case, an additional back buffer is not necessary and would only slow things down.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`width` new back buffer width

`height` new back buffer height

`flags` additional flags (currently 0)

`tags` taglist for additional options (currently always `NULL`)

RESULTS

`error` error code or 0 for success

10.9 CloseDisplay

NAME

CloseDisplay – close a display (V6.0)

SYNOPSIS

```
int error = CloseDisplay(APTR handle);
```

FUNCTION

This function must close the specified display that has been opened by your plugin's `OpenDisplay()` function. Hollywood will call `CloseDisplay()` when it is done with your display. After calling this function, there will be no more accesses to the display handle and you can safely free all resources associated with it.

INPUTS

`handle` display handle returned by `OpenDisplay()`

RESULTS

`error` error code or 0 for success

10.10 Cls

NAME

Cls – clear back buffer in hardware double buffer mode (V6.0, optional)

SYNOPSIS

```
int error = Cls(APTR handle, ULONG color, struct hwTagList *tags);
```

FUNCTION

This must clear the back buffer of the hardware double buffer in the specified display with the color specified in the second parameter. The color is specified as a 24-bit RGB value.

`Cls()` is an optional API and must only be implemented if `HWSDAFLAGS_DOUBLEBUFFERADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle`	display handle returned by `OpenDisplay()`
`color`	color to use for clearing
`tags`	reserved for future use (currently `NULL`)

RESULTS

`error`	error code or 0 for success

10.11 CreatePointer

NAME

CreatePointer – create a custom mouse pointer (V6.0)

SYNOPSIS

```
APTR handle = CreatePointer(ULONG *rgb, int hx, int hy, int *width,
                  int *height, struct hwTagList *tags);
```

FUNCTION

This function must create a mouse pointer that uses the custom imagery passed to this function. Hollywood passes the desired imagery for the custom mouse pointer as a 32-bit ARGB pixel array in the first parameter. Please note that transparency settings for the mouse pointer image are always provided as 8-bit alpha channel values in the 8 most significant bits. If your hardware doesn't support alpha channeled mouse pointers, you have to convert this information first.

Additionally, Hollywood passes the desired hotspot for the mouse pointer to `CreatePointer()`. A mouse pointer's hotspot is a single pixel within the pointer imagery that is used to determine the mouse pointer's pixel position by the operating system. The hotspot is described as an offset in pixels that is relative to the upper-left corner of the mouse pointer image.

The desired pixel dimensions for the new mouse pointer are passed as pointer values because your implementation of `CreatePointer()` is allowed to modify them in case there are some hardware restrictions for mouse pointer dimensions on your system. You have to return the actual pixel dimensions of the new mouse pointer in parameters 4 and 5.

The last parameter is a pointer to a tag list. This is reserved for future use and is currently always `NULL`.

INPUTS

`rgb`	pointer to a 32-bit ARGB pixel array containing the pointer image

`hx` x offset of the pointer hotspot in pixels

`hy` y offset of the pointer hotspot in pixels

`width` pointer to an integer containing the pointer width in pixels; you have to write the real pointer width in pixels to this pointer on exit

`height` pointer to an integer containing the pointer height in pixels; you have to write the real pointer height in pixels to this pointer on exit

`tags` tag list containing additional parameters (see above)

RESULTS

`handle` handle to the custom pointer or `NULL` if `CreatePointer()` failed

10.12 DetermineBorderSizes

NAME

DetermineBorderSizes – obtain information about window border sizes (V6.0)

SYNOPSIS

```
void DetermineBorderSizes(ULONG flags, int *left, int *right,
         int *top, int *bottom);
```

FUNCTION

Hollywood will call this function before `OpenDisplay()` to determine the border sizes of the display your plugin is about to open. Hollywood needs this information prior to opening the display in order to align the display on its host screen or center it. You have to write the border size values in pixels to the integer pointers passed to this function.

Additionally, Hollywood passes some flags to this function that tell you something about the display's style. `Flags` can be a combination of the following bits:

`HWDISPFLAGS_BORDERLESS:`
Display should be opened without any border decoration.

`HWDISPFLAGS_SIZEABLE:`
Display should be resizeable.

`HWDISPFLAGS_FIXED:`
Display dragging should be disabled.

`HWDISPFLAGS_NOHIDE:`
Display should not have a widget for minimizing.

`HWDISPFLAGS_NOCLOSE:`
Display should not have a close widget.

INPUTS

`flags` flags describing the display facilities

`left` integer pointer to receive width of the left window border

`right` integer pointer to receive width of the right window border

`top` integer pointer to receive height of the top window border

`bottom` integer pointer to receive height of the bottom window border

10.13 DoVideoBitMapMethod

NAME

DoVideoBitMapMethod – perform custom action on video bitmap (V6.0, optional)

SYNOPSIS

```
int error = DoVideoBitMapMethod(APTR handle, int method, APTR data);
```

FUNCTION

This function is used to perform custom actions on a video bitmap. It accepts a `method` and a `data` parameter. The contents of the `data` parameter depend on the specified method.

The following methods are currently recognized:

`HWVBMMTHD_SETBLEND:`
: Hollywood will call this method to toggle the blend flag of the video bitmap. If the blend flag is set, this video bitmap has to be drawn with alpha blending enabled. If the blend flag isn't set, the video bitmap should be drawn without any alpha blending. If `HWVBMMTHD_SETBLEND` is passed, the `data` parameter will just be an `int` that is set to `True` or `False` to enable or disable the blend flag.

`DoVideoBitMapMethod()` is an optional API and must only be implemented if `HWSDAFLAGS_VIDEOBITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` handle to a bitmap allocated by `AllocVideoBitMap()`.

`method` method to execute (see above)

`data` method specific data

RESULTS

`error` error code or 0 for success

10.14 EndDoubleBuffer

NAME

EndDoubleBuffer – stop hardware double buffer mode (V6.0, optional)

SYNOPSIS

```
int error = EndDoubleBuffer(APTR handle, struct hwTagList *tags);
```

FUNCTION

This function must stop hardware double buffer mode for the specified display.

`EndDoubleBuffer()` is an optional API and must only be implemented if `HWSDAFLAGS_DOUBLEBUFFERADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`tags` reserved for future use (currently `NULL`)

RESULTS

`error` error code or 0 for success

10.15 Flip

NAME

Flip – flip front and back buffers (V6.0, optional)

SYNOPSIS

```
int error = Flip(APTR handle, struct hwTagList *tags);
```

FUNCTION

When in hardware double buffer mode, this function must bring the display's back buffer into view and install the former front buffer as the new back buffer. See Section 10.6 [BeginDoubleBuffer], page 66, for details.

The following tags may be passed in the second parameter:

`HWFLIPTAG_VSYNC:`
: If the `iData` element of this tag item is set to `True`, Hollywood wants you to synchronize buffer flips with the vertical blank interrupt. If this is `False`, Hollywood doesn't want you to do any synchronization and your `Flip()` implementation should exit immediately after flipping front and back buffers.

`Flip()` is an optional API and must only be implemented if `HWSDAFLAGS_DOUBLEBUFFERADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`tags` taglist for additional options (see above)

RESULTS

`error` error code or 0 for success

10.16 ForceEventLoopIteration

NAME

ForceEventLoopIteration – wake up WaitEvents() (V6.0)

SYNOPSIS

```
void ForceEventLoopIteration(struct hwTagList *tags);
```

FUNCTION

This function must wake up your `WaitEvents()` implementation and make it return control to Hollywood. Hollywood will call `ForceEventLoopIteration()` from worker threads whenever it needs control back from your plugin.

The tag list parameter is reserved for future use. It's currently always `NULL`.

This function must be implemented in a thread-safe way.

INPUTS

`t`	tag list containing additional parameters (currently always `NULL`)

10.17 FreeBitMap

NAME

FreeBitMap – free a software bitmap (V6.0, optional)

SYNOPSIS

```
void FreeBitMap(APTR handle);
```

FUNCTION

This function must free bitmaps allocated by `AllocBitMap()`.

`FreeBitMap()` is an optional API and must only be implemented if `HWSDAFLAGS_BITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle`	bitmap handle returned by `AllocBitMap()`

10.18 FreeGrabScreenPixels

NAME

FreeGrabScreenPixels – free grabbed screen pixels (V6.0, optional)

SYNOPSIS

```
void FreeGrabScreenPixels(ULONG *pixels);
```

FUNCTION

This function must free the pixel array allocated by `GrabScreenPixels()`.

`FreeGrabScreenPixels()` is an optional API and must only be implemented if `HWSDAFLAGS_GRABSCREEN` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`pixels`	pixel array allocated by `GrabScreenPixels()`

10.19 FreeMonitorInfo

NAME

FreeMonitorInfo – free data allocated by GetMonitorInfo() (V6.0, optional)

SYNOPSIS

```
void FreeMonitorInfo(int what, APTR data);
```

FUNCTION

This function must free the data allocated by `GetMonitorInfo()`. You have to specify the type of the data in `what`. See Section 10.24 [GetMonitorInfo], page 79, for a list of supported data types.

`FreeMonitorInfo()` is an optional API and must only be implemented if `HWSDAFLAGS_MONITORINFO` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`what`	which data to free
`data`	data allocated by `GetMonitorInfo()`

10.20 FreePointer

NAME

FreePointer – free a custom mouse pointer (V6.0)

SYNOPSIS

```
void FreePointer(APTR handle);
```

FUNCTION

This function has to free a mouse pointer handle allocated by `CreatePointer()`. Hollywood will make sure that this function is only called if the custom mouse pointer is no longer attached to any display.

INPUTS

`handle`	pointer handle returned by `CreatePointer()`

10.21 FreeVideoBitMap

NAME

FreeVideoBitMap – free video bitmap (V6.0, optional)

SYNOPSIS

```
void FreeVideoBitMap(APTR handle);
```

FUNCTION

This function must free the specified video bitmap that has been allocated using `AllocVideoBitMap()`. See Section 10.5 [AllocVideoBitMap], page 63, for details.

`FreeVideoBitMap()` is an optional API and must only be implemented if `HWSDAFLAGS_VIDEOBITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` handle to a bitmap allocated by `AllocVideoBitMap()`.

10.22 FreeVideoPixels

NAME

FreeVideoPixels – free pixel array allocated by ReadVideoPixels() (V6.0, optional)

SYNOPSIS

```
void FreeVideoPixels(APTR pixels);
```

FUNCTION

This function must free the pixel array allocated by `ReadVideoPixels()`. See Section 10.33 [ReadVideoPixels], page 91, for details.

`FreeVideoPixels()` is an optional API and must only be implemented if `HWSDAFLAGS_VIDEOBITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`pixels` pixel array allocated by `ReadVideoPixels()`

10.23 GetBitMapAttr

NAME

GetBitMapAttr – query software bitmap attribute (V6.0, optional)

SYNOPSIS

```
int val = GetBitMapAttr(APTR handle, int attr, struct hwTagList *tags);
```

FUNCTION

This function must return the requested information about the specified bitmap. The `attr` parameter will tell you which information you have to return. The following attributes are currently recognized:

`HWBMATTR_WIDTH:`
: Return the bitmap's width in pixels.

`HWBMATTR_HEIGHT:`
: Return the bitmap's height in pixels.

`HWBMATTR_BYTESPERROW:`
: Return the bitmap's bytes per row.

`GetBitMapAttr()` is an optional API and must only be implemented if `HWSDAFLAGS_BITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` bitmap handle allocated by `AllocBitMap()`

`attr` which information should be queried

`tags` reserved for future use (currently `NULL`)

RESULTS

`val` value of the requested attribute

10.24 GetMonitorInfo

NAME

GetMonitorInfo – get information about monitor hardware (V6.0, optional)

SYNOPSIS

```
int error = GetMonitorInfo(int what, int monitor, APTR *data,
                struct hwTagList *tags);
```

FUNCTION

This function must query all available monitors and return information about their configuration and capabilities. Hollywood will use the `what` parameter to tell your implementation which information about the monitor hardware the program wants to have. `what` can be one of the following values:

`HWGMITYPE_MONITORS:`

Hollywood wants to have information about all monitors available to the system and their positions on the extended desktop relative to the primary monitor. If `what` has been set to `HWGMITYPE_MONITORS`, your `GetMonitorInfo()` implementation must allocate a list of all available monitors as an array of `struct hwMonitorInfo` elements. The list must be terminated by a last `struct hwMonitorInfo` element with all structure members set to zero. Here is what `struct hwMonitorInfo` looks like:

```
struct hwMonitorInfo
{
    int X;        // [out]
    int Y;        // [out]
    int Width;    // [out]
    int Height;   // [out]
};
```

Hollywood expects the following information in the individual structure members:

`X:` Must be set to the x position of this monitor on the extended desktop.

`Y:` Must be set to the y position of this monitor on the extended desktop.

`Width:` This monitor's width on the extended desktop.

`Height:` This monitor's height on the extended desktop.

All values must be specified in pixels. The pointer to the list of `struct hwMonitorInfo` elements must be written to the `data` pointer that is passed to `GetMonitorInfo()` as parameter 3. Hollywood will call `FreeMonitorInfo()` to give you a chance to free the list you've allocated.

The `monitor` parameter is ignored if `what` is `HWGMITYPE_MONITORS`.

`HWGMITYPE_VIDEOMODES:`
Hollywood wants to know about all video modes that a specific monitor supports. The number of the monitor to examine is passed in the `monitor` parameter. Monitors are counted from 0 which describes the primary monitor. Your `GetMonitorInfo()` implementation must allocate a list of all available video modes for this monitor as an array of `struct hwVideoModeInfo` elements. The list must be terminated by a last `struct hwVideoModeInfo` element with all structure members set to zero. Here is what `struct hwVideoModeInfo` looks like:

```
struct hwVideoModeInfo
{
    int Width;    // [out]
    int Height;   // [out]
    int Depth;    // [out]
};
```

Hollywood expects the following information in the individual structure members:

`Width:` Width of video mode in pixels.

`Height:` Height of video mode in pixels.

`Depth:` Depth of video mode.

The pointer to the list of `struct hwVideoModeInfo` elements must be written to the `data` pointer that is passed to `GetMonitorInfo()` as parameter 3. Hollywood will call `FreeMonitorInfo()` to give you a chance to free the list you've allocated.

`GetMonitorInfo()` is an optional API and must only be implemented if `HWSDAFLAGS_MONITORINFO` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`what`	which information to query (see above)
`monitor`	monitor index to query
`data`	pointer to store this function's return data
`tags`	taglist containing additional parameters (currently always `NULL`)

RESULTS

`error`	error code or 0 for success

10.25 GetMousePos

NAME

GetMousePos – query mouse pointer position (V6.0)

SYNOPSIS

```
void GetMousePos(APTR handle, int *mx, int *my);
```

FUNCTION

This function must query the mouse pointer position for the specified window and write it to the pointers passed in parameters 2 and 3. The mouse position you return in this function must be relative to the upper-left corner of your display's client area.

INPUTS

`handle`	display handle returned by `OpenDisplay()`
`mx`	pointer to write the mouse pointer's x position to
`my`	pointer to write the mouse pointer's y position to

10.26 GetQualifiers

NAME

GetQualifiers – query current qualifier key state (V6.0)

SYNOPSIS

```
ULONG state = GetQualifiers(APTR handle);
```

FUNCTION

This function must query the state of all keyboard qualifiers for the specified display and return them to Hollywood. The following keyboard qualifiers are currently defined:

`HWKEY_QUAL_MASK:`
: This is an internal control bit and must always be set.

`HWKEY_QUAL_LSHIFT:`
: Left shift key is down.

`HWKEY_QUAL_RSHIFT:`
: Right shift key is down.

`HWKEY_QUAL_LALT:`
: Left alt key is down.

`HWKEY_QUAL_RALT:`
: Right alt key is down.

`HWKEY_QUAL_LCOMMAND:`
: Left command key is down.

`HWKEY_QUAL_RCOMMAND:`
: Right command key is down.

`HWKEY_QUAL_LCONTROL:`
: Left control key is down.

`HWKEY_QUAL_RCONTROL:`
Right control key is down.

INPUTS

`handle` display handle returned by `OpenDisplay()`

RESULTS

`state` bitmask containing all qualifier keys that are currently down; do not forget to always set `HWKEY_QUAL_MASK`

10.27 GrabScreenPixels

NAME

GrabScreenPixels – grab pixels of display's host screen (V6.0, optional)

SYNOPSIS

```
ULONG *rgb = GrabScreenPixels(APTR handle, int x, int y, int width,
                 int height, ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function must grab the pixels of the screen that is currently hosting the specified display and return them to Hollywood as a 32-bit RGB pixel array. The pixel array that is returned by this function must be exactly `width * height * 4` bytes in size. No padding bytes are allowed.

Hollywood will call your `FreeGrabScreenPixels()` function to free the pixel array allocated by this function.

`GrabScreenPixels()` is an optional API and must only be implemented if `HWSDAFLAGS_GRABSCREEN` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`x` x offset marking the start position of the grab operation

`y` y offset marking the start position of the grab operation

`width` grab width in pixels

`height` grab height in pixels

`flags` reserved for future use (currently 0)

`tags` reserved for future use (currently `NULL`)

RESULTS

`rgb` 32bit RGB array containing the grabbed pixels

10.28 HandleEvents

NAME

HandleEvents – handle display events (V6.0)

SYNOPSIS

```
int error = HandleEvents(lua_State *L, ULONG flags, struct hwTagList *t);
```

FUNCTION

This function must handle all display events that have come in. Hollywood will call this function many times per second so that your application stays responsive. As this is called so very often, you may want to implement a throttle here so that you only poll for events 50 times per second or so. Otherwise, `HandleEvents()` has the potential to slow down your script's execution significantly if polling for events is too expensive and you do not implement a throttle. Be sure to benchmark raw performance of scripts with your display adapter and compare them to the performance of Hollywood's inbuilt display handler to see if your adapter needs optimizing.

The following flags may be passed in the second parameter:

`HWHEFLAGS_LINEHOOK:`
: This flag is set if `HandleEvents()` has been called from the Lua line hook.

`HWHEFLAGS_MODAL:`
: If this flag is set, then `HandleEvents()` has been called from a modal loop that Hollywood is currently running. A modal loop is a temporary event loop set up by functions that block the script execution until certain events happen, e.g. `WaitLeftMouse()` or `InKeyStr()`.

`HWHEFLAGS_CHECKEVENT:`
: This flag is set if `HandleEvents()` has been called as a result of the script calling Hollywood's `CheckEvent()` command.

`HWHEFLAGS_WAITEVENT:`
: This flag is set if `HandleEvents()` has been called because `WaitEvents()` has triggered.

Your `HandleEvents()` function should always call `hw_MasterServer()` with the `HWMSFLAGS_DRAWVIDEOS` flag set so that Hollywood can update any videos that are currently playing. If you don't do that, video playback won't work correctly.

The third parameter is a tag list which is currently always `NULL`. This might change in the future, though.

`HandleEvents()` must return an error code or 0 for success. A special return value is `ERR_USERABORT`. If your `HandleEvents()` returns `ERR_USERABORT`, Hollywood will quit. Thus, it is suggested that you return `ERR_USERABORT` if the user has clicked your display's close widget, pressed the escape key, etc. Note that you should not return `ERR_USERABORT` in case the `HWDISPSATAG_USERCLOSE` attribute has been set to `True` using `SetDisplayAttributes()`. See Section 10.35 [SetDisplayAttributes], page 93, for details.

Please note that this function must handle events on all displays that have currently been opened by `OpenDisplay()`. Additionally, it could also happen that no display is

open at all and your `HandleEvents()` function is called. Be prepared to deal with these cases.

All standard window events like sizing a window, moving a window, key presses and mouse events, etc. should be forwarded to Hollywood using `hw_PostEventEx()` so that Hollywood is able to notify any handlers that might listen to these events.

INPUTS

`L`	pointer to the `lua_State`
`flags`	combination of flags (see above)
`t`	tag list containing additional parameters (see above)

RESULTS

`error`	error code or 0 for success; returning `ERR_USERABORT` tells Hollywood to quit

10.29 Line

NAME

Line – draw a line (V6.0)

SYNOPSIS

```
void Line(APTR handle, int x1, int y1, int x2, int y2, ULONG color,
          ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function must draw a line between the two points passed to this function. The destination handle can be either a display or a bitmap. You have to look at `flags` parameter to find out how to interpret the handle that Hollywood has passed to your function. The following flags are currently recognized:

`HWLIFLAGS_DESTVIDEOBITMAP:`
: The handle passed is a video bitmap allocated by `AllocVideoBitMap()`. This can only happen if you've set the `HWVBMCAPS_OFFSCREENCOLOR` or `HWVBMCAPS_OFFSCREENALPHA` capabilities in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmaps. Otherwise, `HWLIFLAGS_DESTVIDEOBITMAP` will never be set.

`HWLIFLAGS_DESTALPHAONLY:`
: This is only set in connection with `HWLIFLAGS_DESTVIDEOBITMAP`. If Hollywood wants you to draw to the alpha channel of your video bitmap allocated by `AllocVideoBitMap()`, it will indicate this by setting `HWLIFLAGS_DESTALPHAONLY`. If `HWLIFLAGS_DESTVIDEOBITMAP` is set and `HWLIFLAGS_DESTALPHAONLY` isn't, you have to draw to the color channels of the video bitmap instead. Note that `HWLIFLAGS_DESTALPHAONLY` will only ever be set if you've set the `HWVBMCAPS_OFFSCREENALPHA` capability flag in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmap alphachannels. In that case, the `color` parameter will contain just an 8-bit alpha transparency value ranging from 0 to 255.

`HWLIFLAGS_DESTBITMAP`:
: The handle passed is a software bitmap allocated by Hollywood or your plugin's `AllocBitMap()` function if you've set the `HWSDAFLAGS_BITMAPADAPTER` in your call to `hw_SetDisplayAdapter()`. Note that `HWLIFLAGS_DESTBITMAP` will only ever be set if you've passed `HWBMAHOOK_WRITEPIXEL` in `HWSDAFLAGS_BITMAPHOOK`. Otherwise, Hollywood will do the rendering to the software bitmap on its own and you don't have to care. `HWLIFLAGS_DESTBITMAP` will only ever be set if you've explicitly requested that you want to do offscreen drawing to software bitmaps on your own by setting the appropriate bitmap hook flags.

If you've set the `HWSDAFLAGS_ALPHADRAW` flag when calling `hw_SetDisplayAdapter()` to initialize your plugin, the color value passed in parameter 6 will contain an alpha value in its 8 most significant bits and your implementation is expected to draw to the destination with alpha blending enabled. If you haven't set `HWSDAFLAGS_ALPHADRAW`, the color will be just a 24-bit RGB value. If the `HWLIFLAGS_DESTALPHAONLY` flag is set, the `color` parameter will contain just an 8-bit alpha transparency value ranging from 0 to 255.

This function doesn't have to do any clipping. Hollywood will perform clipping itself before calling `Line()`.

If your display adapter doesn't support video bitmaps or hooks into Hollywood's bitmap handler, `Line()` only has to be able to draw to the display which should be quite simple and straight-forward to implement.

If your plugin supports hardware double buffering and Hollywood has put your display into hardware double buffering mode by calling your plugin's `BeginDoubleBuffer()` function, this function must not draw anything to the display but only to the back buffer. Hollywood will call your plugin's `Flip()` function when it wants you to draw the back buffer to the display.

You might want to use the `hw_RawLine()` function in your implementation to draw lines to bitmaps stored as raw pixel buffers. See Section 28.24 [hw_RawLine], page 250, for details.

INPUTS

`handle`	destination display or bitmap (depends on the flags that are set, see above)
`x1`	start x offset for the line
`y1`	start y offset for the line
`x2`	end x offset for the line
`y2`	end y offset for the line
`color`	desired line color
`flags`	flags specifying further parameters
`tags`	additional options (currently always `NULL`)

10.30 LockBitMap

NAME

LockBitMap – lock a software bitmap (V6.0, optional)

SYNOPSIS

```
APTR lock = LockBitMap(APTR handle, ULONG flags, struct
                hwos_LockBitMapStruct *bmlock, struct hwTagList *tags);
```

FUNCTION

This function must lock a bitmap to allow access to the bitmap's underlying pixel data. Hollywood will pass a pointer to a `struct hwos_LockBitMapStruct` to this function. Your implementation must then fill this structure with all necessary information. `struct hwos_LockBitMapStruct` looks like this:

```
struct hwos_LockBitMapStruct
{
    APTR Data;          // [out]
    int Modulo;         // [out]
    int PixelFormat;    // [out]
    int BytesPerPixel;  // [out]
    int Width;          // [out]
    int Height;         // [out]
};
```

Your `LockBitMap()` implementation has to write the following information to the individual structure members:

`Data:` Must be set to a pointer to the bitmap's actual pixel data. The pointer must be valid until Hollywood calls `UnLockBitMap()` on the handle that is returned by this function.

`Modulo:` Must be set to the bitmap's modulo width, i.e. the length of a single row of pixels. For bitmaps of type `HWBMTYPE_RGB` this value must be specified in pixels. For the other bitmap types, this value must be specified in bytes.

`PixelFormat:`
This must be set to the pixel format used by the raw pixel array that you've set in `Data`. See Section 2.15 [Pixel format information], page 18, for details.

`BytesPerPixel:`
This must be set to how many bytes are needed for a single pixel.

`Width:` Must be set to the bitmap's width.

`Height:` Must be set to the bitmap's height.

Hollywood will also pass a combination of flags to this function. The following flags are currently supported:

`HWLBMFLAGS_READONLY:`
If this flag is set, Hollywood will only need read access to the bitmap. It won't write to the pixel array you return in the `Data` member of the `struct hwos_LockBitMapStruct`.

`LockBitMap()` has to return a lock handle for this bitmap. This is an opaque datatype only known by your plugin. It's only used to unlock the bitmap again when Hollywood is finished with it. Hollywood will call `UnLockBitMap()` then, passing the handle which was returned by `LockBitMap()`. If this function fails to lock the bitmap, return `NULL`.

This function will usually be called very often by Hollywood so your implementation should be efficient and should not have to copy the pixels to a new memory block first. Instead, it should be designed in a way that allows immediate access to the pixel data.

`LockBitMap()` is an optional API and must only be implemented if `HWSDAFLAGS_BITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle`	bitmap handle allocated by `AllocBitMap()`
`flags`	flags for the lock operation (see above)
`bmlock`	pointer to a `struct hwos_LockBitMapStruct` to be filled out by this function
`tags`	reserved for future use (currently `NULL`)

RESULTS

`lock`	bitmap lock or `NULL` in case there was an error

10.31 MovePointer

NAME

MovePointer – move the mouse pointer (V6.0)

SYNOPSIS

```
void MovePointer(APTR handle, int x, int y);
```

FUNCTION

This function must move the mouse pointer to the specified position. The position that is passed to this function is relative to the display's client area top-left corner.

INPUTS

`handle`	display handle returned by `OpenDisplay()`
`x`	desired new x position for mouse pointer
`y`	desired new y position for mouse pointer

10.32 OpenDisplay

NAME

OpenDisplay – open a display (V6.0)

SYNOPSIS

```
APTR handle = OpenDisplay(STRPTR title, int x, int y, int width,
                  int height, ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function has to open a new display that is described by the parameters passed to this function. Hollywood will pass the position of the display on screen as well as its dimensions. All values are in pixels. Additionally, a flags bitfield and a tag list is passed to this function.

Your implementation has to return a handle that is used to refer to this display later or `NULL` in case an error has occurred.

The `flags` parameter can be a combination of the following individual flags:

`HWDISPFLAGS_BORDERLESS:`
: Window should be opened without any border decoration.

`HWDISPFLAGS_SIZEABLE:`
: Window should be resizeable. Note that this flag refers to user resizability only. Even if this flag isn't set, Hollywoood might still ask your window to resize by calling the `SizeMoveDisplay()` function.

`HWDISPFLAGS_FIXED:`
: Window dragging should be disabled.

`HWDISPFLAGS_NOHIDE:`
: Window should not have a widget for minimizing.

`HWDISPFLAGS_NOCLOSE:`
: Window should not have a close widget.

`HWDISPFLAGS_AUTOSCALE:`
: This flag is set if the user has enabled auto scaling for this window. If that is the case, the window's physical dimensions won't necessarily match the dimensions of its back buffer.

`HWDISPFLAGS_LAYERSCALE:`
: This flag is set if the user has enabled layer scaling for this window. Layer scaling uses a different technique than auto scaling which means that the back buffer size will match the window's physical size in case layer scaling is active.

`HWDISPFLAGS_LAYERS:`
: This flag is set if layers are enabled for this window.

`HWDISPFLAGS_DOUBLEBUFFER:`
: This flag is set if Hollywood will use double buffer drawing for this window.

`HWDISPFLAGS_HARDWAREDB:`
: This flag is set if Hollywood is using a hardware double buffer provided by the plugin for this window.

`HWDISPFLAGS_FULLSCREEN:`
: If this flag is set, the display should be opened in full screen mode. You may choose to ignore the `x` and `y` parameters in that case.

`HWDISPFLAGS_DISABLEBLANKER:`
: If this flag is set, the screen blanker should be disabled while this display is opened.

Additionally, Hollywood will pass a tag list to `OpenDisplay()`. This tag list can contain the following tags:

`HWDISPTAG_BUFFERWIDTH:`
: This tag is always provided and contains the pixel width of Hollywood's back buffer for this display. Please note that this is not necessarily the same as the window's physical dimensions. If autoscaling is active, back buffer size and the window's physical dimensions can be different. It is suggested that you allocate a pixel buffer of this size and first draw everything into this back buffer. After that you refresh the display using the graphics you have drawn into the back buffer. For optimized drawing you should implement a custom double buffer using `HWPLUG_CAPS_DOUBLEBUFFER`. Custom double buffers don't have to draw into the back buffer first because double buffer frames are usually updated multiple times per second so there's no need to cache an additional back buffer for refreshing the window when parts of it have to be redrawn.

`HWDISPTAG_BUFFERHEIGHT:`
: This tag is always provided and contains the pixel height of Hollywood's back buffer for this display. See above for details.

`HWDISPTAG_OPTIMIZEDREFRESH:`
: This tag is always provided and it is set to a pointer to a `ULONG`. Your plugin can write `True` to this pointer if it wants Hollywood to enable optimized refresh for this display. Optimized refresh should be enabled on systems where it's more efficient to refresh the complete display instead of several smaller parts. By default, Hollywood prefers to refresh just the parts of the display that actually need refreshing. On some backends, however, this is quite slow if several smaller parts have to be updated. It is often faster to refresh the complete display on these systems. This is especially true for backends that are double buffer based because refreshing a single tile on double buffer based backends means doing a complete buffer flip which doesn't look so nice and is quite slow. That's why it is wise to enable this option on some systems. By default, optimized refresh is disabled.

`HWDISPTAG_SINGLEREFRESHFX:`
: This tag is always provided and it is set to a pointer to a `ULONG`. Your plugin can write `True` to this pointer if it wants Hollywood to enable single refresh transition effect drawing for this display. This option is related to the `HWDISPTAG_OPTIMIZEDREFRESH` tag (see above). If you set the pointer that you are passed here to `True`, Hollywood will draw one transition effect frame in exactly one video frame. If it is set to `False` (which is also the default), Hollywood might emit more than one video frame for a single transition effect frame. This is especially so for transition effect frames that consist of many small parts that need to be updated. If `HWDISPTAG_SINGLEREFRESHFX` is not set to `True`, Hollywood will prefer to draw many small parts instead of the complete frame all at once. On some systems, however, drawing many small bits totally kills the performance. In that case you need to set

this tag to `True`. See above in `HWDISPTAG_OPTIMIZEDREFRESH` for additional information.

`HWDISPTAG_LUASTATE:`
This tag is always provided and contains a pointer to the `lua_State`.

`HWDISPTAG_MONITOR:`
This tag is always provided and specifies the monitor that this display should be opened on. Monitors are counted from 0 which specifies the primary monitor.

`HWDISPTAG_SCREENWIDTH:`
If `HWDISPFLAGS_FULLSCREEN` has been set, this tag contains the desired pixel width for the full screen mode. This is not necessarily the same as the display width since the user might explicitly choose a full screen resolution that is larger than the display, e.g. an 800x600 screen mode for a 640x480 display.

`HWDISPTAG_SCREENHEIGHT:`
Contains the desired pixel screen height if `HWDISPFLAGS_FULLSCREEN` has been set. See above for details.

`HWDISPTAG_SCREENDEPTH:`
Contains the desired screen depth if `HWDISPFLAGS_FULLSCREEN` has been set. On most systems you may choose to ignore this since most modern systems all operate in 32-bit true colour mode.

`HWDISPTAG_SCALEWIDTH:`
If `HWDISPFLAGS_AUTOSCALE` or `HWDISPFLAGS_LAYERSCALE` is active, this tag contains the current scale width.

`HWDISPTAG_SCALEHEIGHT:`
If `HWDISPFLAGS_AUTOSCALE` or `HWDISPFLAGS_LAYERSCALE` is active, this tag contains the current scale height.

`HWDISPTAG_SCALEMODE:`
If `HWDISPFLAGS_AUTOSCALE` or `HWDISPFLAGS_LAYERSCALE` is active, this tag contains the current scale mode. Currently, only 0 and 1 are defined here. 0 means hard scaling with no interpolation whereas 1 means anti-alias interpolated scaling.

Please note that Hollywood supports multiple displays so `OpenDisplay()` can be called several times. Hollywood also supports multiple full screen displays if they are to appear on different monitors. Be prepared to deal with these cases.

INPUTS

`title`	caption string for the display's window border
`x`	desired x position of this display in pixels
`y`	desired y position of this display in pixels
`width`	desired width for this display in pixels
`height`	desired height for this display in pixels

`flags` flags describing additional options (see above)

`tags` tag list describing additional options (see above)

RESULTS

`handle` a handle to the newly allocated display or `NULL` in case of an error

10.33 ReadVideoPixels

NAME

ReadVideoPixels – get raw pixels from video bitmap (V6.0, optional)

SYNOPSIS

```
APTR rgb = ReadVideoPixels(APTR handle, struct hwTagList *tags);
```

FUNCTION

This function must return the raw pixels of the specified video bitmap. This is often very slow because it has to read from GPU memory. That's why Hollywood calls this function only under very special circumstances. This function has to return a pixel buffer that is exactly of the size `width * height * bytes_per_pixel`. No line padding may be involved.

Additionally, your implementation has to handle the following taglist:

`HWRVPTAG_BLEND:`
: The `pData` member of this tag item will be set to a pointer to an `int`. Your implementation has to set this pointer to either `True` or `False`, depending on whether the returned video pixels contain alpha channel transparency information or not.

`HWRVPTAG_PIXELFORMAT:`
: The `pData` member of this tag item will be set to a pointer to an `int`. Your implementation has to set this pointer to the pixel format the returned data is in. See Section 2.15 [Pixel formats], page 18, for details.

When Hollywood is done with the data returned by this function, it will call `FreeVideoPixels()` on it.

`ReadVideoPixels()` is an optional API and must only be implemented if `HWSDAFLAGS_VIDEOBITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle` handle to a bitmap allocated by `AllocVideoBitMap()`.

`tags` taglist containing additional parameters (see above)

RESULTS

`rgb` array containing raw pixel data or `NULL` in case of an error

10.34 RectFill

NAME

RectFill – fill rectangular pixel area with color (V6.0)

SYNOPSIS

```
void RectFill(APTR handle, int x, int y, int width, int height,
         ULONG color, ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function must fill the specified rectangular area with the color passed in parameter 6. The destination handle can be either a display or a bitmap. You have to look at `flags` parameter to find out how to interpret the handle that Hollywood has passed to your function. The following flags are currently recognized:

`HWRFFLAGS_DESTVIDEOBITMAP:`
: The handle passed is a video bitmap allocated by `AllocVideoBitMap()`. This can only happen if you've set the `HWVBMCAPS_OFFSCREENCOLOR` or `HWVBMCAPS_OFFSCREENALPHA` capabilities in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmaps. Otherwise, `HWRFFLAGS_DESTVIDEOBITMAP` will never be set.

`HWRFFLAGS_DESTALPHAONLY:`
: This is only set in connection with `HWRFFLAGS_DESTVIDEOBITMAP`. If Hollywood wants you to draw to the alpha channel of your video bitmap allocated by `AllocVideoBitMap()`, it will indicate this by setting `HWRFFLAGS_DESTALPHAONLY`. If `HWRFFLAGS_DESTVIDEOBITMAP` is set and `HWRFFLAGS_DESTALPHAONLY` isn't, you have to draw to the color channels of the video bitmap instead. Note that `HWRFFLAGS_DESTALPHAONLY` will only ever be set if you've set the `HWVBMCAPS_OFFSCREENALPHA` capability flag in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmap alphachannels. In that case, the `color` parameter will contain just an 8-bit alpha transparency value ranging from 0 to 255.

`HWRFFLAGS_DESTBITMAP:`
: The handle passed is a software bitmap allocated by Hollywood or your plugin's `AllocBitMap()` function if you've set the `HWSDAFLAGS_BITMAPADAPTER` in your call to `hw_SetDisplayAdapter()`. Note that `HWRFFLAGS_DESTBITMAP` will only ever be set if you've passed `HWBMAHOOK_RECTFILL` in `HWSDAFLAGS_BITMAPHOOK`. Otherwise, Hollywood will do the rendering to the software bitmap on its own and you don't have to care. `HWRFFLAGS_DESTBITMAP` will only ever be set if you've explicitly requested that you want to do offscreen drawing to software bitmaps on your own by setting the appropriate bitmap hook flags.

If you've set the `HWSDAFLAGS_ALPHADRAW` flag when calling `hw_SetDisplayAdapter()` to initialize your plugin, the color value passed in parameter 6 will contain an alpha value in its 8 most significant bits and your implementation is expected to draw to the destination with alpha blending enabled. If you haven't set `HWSDAFLAGS_ALPHADRAW`, the color will be just a 24-bit RGB value. If the `HWRFFLAGS_DESTALPHAONLY` flag is set, the

`color` parameter will contain just an 8-bit alpha transparency value ranging from 0 to 255.

This function doesn't have to do any clipping. Hollywood will perform clipping itself before calling `RectFill()`.

If your display adapter doesn't support video bitmaps or hooks into Hollywood's bitmap handler, `RectFill()` only has to be able to draw to the display which should be quite simple and straight-forward to implement.

If your plugin supports hardware double buffering and Hollywood has put your display into hardware double buffering mode by calling your plugin's `BeginDoubleBuffer()` function, this function must not draw anything to the display but only to the back buffer. Hollywood will call your plugin's `Flip()` function when it wants you to draw the back buffer to the display.

You might want to use the `hw_RawRectFill()` function in your implementation to draw rectangles to bitmaps stored as raw pixel buffers. See Section 28.25 [hw_RawRectFill], page 251, for details.

INPUTS

`handle`	destination display or bitmap (depends on the flags that are set, see above)
`x`	x offset of the rectangle to fill
`y`	y offset of the rectangle to fill
`width`	width in pixels of the area to fill
`height`	height in pixels of the area to fill
`color`	filling color
`flags`	flags specifying further parameters
`tags`	additional options (currently always `NULL`)

10.35 SetDisplayAttributes

NAME

SetDisplayAttributes – modify display attributes (V6.0)

SYNOPSIS

```
int error = SetDisplayAttributes(APTR handle, struct hwTagList *tags);
```

FUNCTION

This function is used by Hollywood to change certain attributes of your display. Hollywood will pass a tag list to this function. The following tags are currently defined:

`HWDISPSATAG_USERCLOSE:`

If `iData` is set to `True` here, then you must not shut down Hollywood when the user clicks on your display's close widget, i.e. your `HandleEvents()` implementation should not return `ERR_USERABORT` in that case. Instead, you should just post an `HWEVT_CLOSEDISPLAY` event and leave everything else to Hollywood. This attribute is often set to `True` in case the user

listens to `CloseWindow` events using `InstallEventHandler()` to perform some custom action when the window's close widget is pressed.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`t` tag list containing additional attributes (see above)

RESULTS

`error` error code or 0 for success

10.36 SetDisplayTitle

NAME

SetDisplayTitle – change display title (V6.0)

SYNOPSIS

```
void SetDisplayTitle(APTR handle, STRPTR title);
```

FUNCTION

This function must change the display's window caption to the specified string.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`title` new caption for the display

10.37 SetPointer

NAME

SetPointer – change mouse pointer (V6.0)

SYNOPSIS

```
void SetPointer(APTR handle, int type, APTR data);
```

FUNCTION

This function must change the display's mouse pointer to the one that is requested by Hollywood. The second parameter describes the requested mouse pointer type. This can be one of the following types:

`HWPOINTER_SYSTEM:`
Mouse pointer should be changed to the system's default mouse pointer.

`HWPOINTER_BUSY:`
Mouse pointer should be changed to a system mouse pointer that indicates that the application is currently busy.

`HWPOINTER_CUSTOM:`
Mouse pointer should be changed to a custom mouse pointer allocated by `CreatePointer()`. If `type` is set to `HWPOINTER_CUSTOM`, a handle to the custom mouse pointer is passed in the `data` parameter. Otherwise the `data` parameter is `NULL`.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`type` desired mouse pointer type (see above for supported types)

`data` if type is `HWPOINTER_CUSTOM`, this is set to a handle returned by `CreatePointer()`, otherwise it is `NULL`

10.38 ShowHideDisplay

NAME

ShowHideDisplay – show or hide the display (V6.0)

SYNOPSIS

```
int error = ShowHideDisplay(APTR handle, int show, struct hwTagList *t);
```

FUNCTION

This function must show or hide the display depending on the state passed as parameter 2.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`show` `True` to show the display, `False` to hide it

`t` taglist for additional options (currently always `NULL`)

RESULTS

`error` error code or 0 for success

10.39 ShowHidePointer

NAME

ShowHidePointer – show or hide the mouse pointer (V6.0)

SYNOPSIS

```
void ShowHidePointer(APTR handle, int show);
```

FUNCTION

This function must show or hide the mouse pointer for the specified display.

INPUTS

`handle` display handle returned by `OpenDisplay()`

`show` `True` to show the pointer, `False` to hide it

10.40 SizeMoveDisplay

NAME

SizeMoveDisplay – change display position and/or size (V6.0)

SYNOPSIS

```
int error = SizeMoveDisplay(APTR handle, int x, int y, int width,
                int height, ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function must change the display position to the specified new position and the display size must be changed to the specified new dimensions. All values have to be passed in pixels.

After a display size change Hollywood will usually also call `ChangeBufferSize()` so that your plugin can adapt the size of its back buffer. The only time in which `ChangeBufferSize()` is not called after `SizeMoveDisplay()` is with enabled autoscaling because in that case the size of the back buffer does not change with the output display's size.

INPUTS

`handle`	display handle returned by `OpenDisplay()`
`x`	new x position for this display
`y`	new y position for this display
`width`	new width for this display
`height`	new height for this display
`flags`	additional flags (currently 0)
`t`	taglist for additional options (currently always `NULL`)

RESULTS

`error`	error code or 0 for success

10.41 Sleep

NAME

Sleep – sleep for a certain amount of time (V6.0, optional)

SYNOPSIS

```
int error = Sleep(lua_State *L, int ms);
```

FUNCTION

This function must put the application to sleep for the requested amount of milliseconds. It is important that this is done in a way that lets the application stay responsive, i.e. you must make sure that you keep handling window events and you also must call into `hw_MasterServer()` frequently to keep videos playing. If the user closes the display while sleeping, you should return `ERR_USERABORT` so that Hollywood can shutdown.

`Sleep()` is an optional API and must only be implemented if `HWSDAFLAGS_SLEEP` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

Note that when compiling for 64-bit Windows, this function must be called `_Sleep` instead. Otherwise there will be clashes with the function of the same name from `kernel32`.lib. On 32-bit Windows this isn't a problem because 32-bit Windows uses decorated function exports.

INPUTS

`L`	pointer to the `lua_State`
`ms`	number of milliseconds to sleep

RESULTS

`error`	error code or 0 for success

10.42 UnLockBitMap

NAME

UnLockBitMap – unlock a software bitmap (V6.0, optional)

SYNOPSIS

```
void UnLockBitMap(APTR handle);
```

FUNCTION

This must unlock the software bitmap specified by `handle`. Note that the `handle` parameter is not a bitmap but the lock handle as returned by `LockBitMap()`.

`UnLockBitMap()` is an optional API and must only be implemented if `HWSDAFLAGS_BITMAPADAPTER` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle`	lock handle returned by `LockBitMap()`

10.43 VWait

NAME

VWait – wait for next vertical blank (V6.0, optional)

SYNOPSIS

```
void VWait(APTR handle, struct hwTagList *tags);
```

FUNCTION

This function must wait for the next vertical blank on the specified display. The following tags can currently be passed in the tag list:

`HWVWAITTAG_DOUBLEBUFFER:`
: The `iData` value of this tag item is set to `True` if Hollywood is currently in double-buffer mode, otherwise it is set to `False`.

`VWait()` is an optional API and must only be implemented if `HWSDAFLAGS_VWAIT` has been passed to `hw_SetDisplayAdapter()`. See Section 28.29 [hw_SetDisplayAdapter], page 254, for details.

INPUTS

`handle`	display handle returned by `OpenDisplay()`
`t`	taglist for additional options (see above)

10.44 WaitEvents

NAME

WaitEvents – wait until events come in (V6.0)

SYNOPSIS

```
int error = WaitEvents(lua_State *L, ULONG flags, struct hwTagList *t);
```

FUNCTION

This function must halt the program's execution until an event comes in. The event must then be handled by your `HandleEvents()` implementation. `WaitEvents()` has to return an error code or 0 for success.

The following flags may be passed in the second parameter:

`HWWEFLAGS_MODAL:`
: If this flag is set, then `WaitEvents()` has been called from a modal loop that Hollywood is currently running. A modal loop is a temporary event loop set up by functions that block the script execution until certain events happen, e.g. `WaitLeftMouse()` or `InKeyStr()`. If `HWWEFLAGS_MODAL` is not set, then you can be sure that Hollywood is currently running the script's main loop. This means that your `WaitEvents()` implementation has been called as a result of the script calling Hollywood's `WaitEvent()` function.

The third parameter is a tag list that can contain the following tags:

`HWWETAG_AMIGASIGNALS:`
: This tag is always passed on Amiga systems and your `WaitEvents()` implementation must take it into account. The `pData` item of this tag is set to a `ULONG` pointer that contains a combination of signal bits that your `WaitEvents()` implementation has to take into account. Thus, on Amiga systems all `WaitEvents()` implementations must be based on `exec.library/Wait()`. You may choose to wait on additional signals but you also must take the signals that you get from Hollywood into account. Furthermore, you must write the signals that have broken your `WaitEvents()` implementation back to the `ULONG` pointer before you return from this function. This is because Hollywood needs to know which signals have triggered so that it can take appropriate action.

Please note that this function must wait for events on all displays that have currently been opened by `OpenDisplay()`. Additionally, it could also happen that no display is

open at all and your `WaitEvents()` function is called. Be prepared to deal with these cases.

Also note that your `WaitEvents()` implementation must be capable of being woken up by `ForceEventLoopIteration()` when this is called from worker threads. See Section 10.16 [ForceEventLoopIteration], page 75, for details.

INPUTS

`L`	pointer to the `lua_State`
`flags`	combination of flags (see above)
`t`	tag list containing additional parameters (see above)

RESULTS

`error`	error code or 0 for success

10.45 WritePixel

NAME

WritePixel – draw a single pixel (V6.0)

SYNOPSIS

```
void WritePixel(APTR handle, int x, int y, ULONG color, ULONG flags,
         struct hwTagList *tags);
```

FUNCTION

This function must draw a single pixel with the color passed in parameter 4. The destination handle can be either a display or a bitmap. You have to look at `flags` parameter to find out how to interpret the handle that Hollywood has passed to your function. The following flags are currently recognized:

`HWWPFLAGS_DESTVIDEOBITMAP:`
: The handle passed is a video bitmap allocated by `AllocVideoBitMap()`. This can only happen if you've set the `HWVBMCAPS_OFFSCREENCOLOR` or `HWVBMCAPS_OFFSCREENALPHA` capabilities in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmaps. Otherwise, `HWWPFLAGS_DESTVIDEOBITMAP` will never be set.

`HWWPFLAGS_DESTALPHAONLY:`
: This is only set in connection with `HWWPFLAGS_DESTVIDEOBITMAP`. If Hollywood wants you to draw to the alpha channel of your video bitmap allocated by `AllocVideoBitMap()`, it will indicate this by setting `HWWPFLAGS_DESTALPHAONLY`. If `HWWPFLAGS_DESTVIDEOBITMAP` is set and `HWWPFLAGS_DESTALPHAONLY` isn't, you have to draw to the color channels of the video bitmap instead. Note that `HWWPFLAGS_DESTALPHAONLY` will only ever be set if you've set the `HWVBMCAPS_OFFSCREENALPHA` capability flag in `HWSDATAG_VIDEOBITMAPCAPS` to enable offscreen rendering to video bitmap alphachannels. In that case, the `color` parameter will contain just an 8-bit alpha transparency value ranging from 0 to 255.

`HWWPFLAGS_DESTBITMAP`:
: The handle passed is a software bitmap allocated by Hollywood or your plugin's `AllocBitMap()` function if you've set the `HWSDAFLAGS_BITMAPADAPTER` in your call to `hw_SetDisplayAdapter()`. Note that `HWWPFLAGS_DESTBITMAP` will only ever be set if you've passed `HWBMAHOOK_WRITEPIXEL` in `HWSDAFLAGS_BITMAPHOOK`. Otherwise, Hollywood will do the rendering to the software bitmap on its own and you don't have to care. `HWWPFLAGS_DESTBITMAP` will only ever be set if you've explicitly requested that you want to do offscreen drawing to software bitmaps on your own by setting the appropriate bitmap hook flags.

If you've set the `HWSDAFLAGS_ALPHADRAW` flag when calling `hw_SetDisplayAdapter()` to initialize your plugin, the color value passed in parameter 4 will contain an alpha value in its 8 most significant bits and your implementation is expected to draw to the destination with alpha blending enabled. If you haven't set `HWSDAFLAGS_ALPHADRAW`, the color will be just a 24-bit RGB value. If the `HWWPFLAGS_DESTALPHAONLY` flag is set, the `color` parameter will contain just an 8-bit alpha transparency value ranging from 0 to 255.

This function doesn't have to do any clipping. Hollywood will perform clipping itself before calling `WritePixel()`.

If your display adapter doesn't support video bitmaps or hooks into Hollywood's bitmap handler, `WritePixel()` only has to be able to draw to the display which should be quite simple and straight-forward to implement.

If your plugin supports hardware double buffering and Hollywood has put your display into hardware double buffering mode by calling your plugin's `BeginDoubleBuffer()` function, this function must not draw anything to the display but only to the back buffer. Hollywood will call your plugin's `Flip()` function when it wants you to draw the back buffer to the display.

You might want to use the `hw_RawWritePixel()` function in your implementation to plot pixels to bitmaps stored as raw pixel buffers. See Section 28.26 [hw_RawWritePixel], page 252, for details.

INPUTS

`handle`	destination display or bitmap (depends on the flags that are set, see above)
`x`	x offset for the pixel
`y`	y offset for the pixel
`color`	pixel color
`flags`	flags specifying further parameters
`tags`	additional options (currently always `NULL`)

11 Extension plugins

11.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_EXTENSION` set are a special plugin type that doesn't offer any functionality on its own but just extends an existing plugin type. Certain plugin types support a number of extensions in newer Hollywood versions and the extension plugin type can be used to tell Hollywood which extensions your plugin actually supports. There is only one function, `GetExtensions()`, which Hollywood will call to get information about the extensions supported by your plugin.

This plugin type is supported since Hollywood 6.0.

11.2 GetExtensions

NAME

GetExtensions – query supported extensions for plugin type (V6.0)

SYNOPSIS

```
ULONG exts = GetExtensions(ULONG capbit, struct hwTagList *tags);
```

FUNCTION

Hollywood will call this function to see which extensions are supported by your plugin for a certain plugin type. Hollywood will pass the capability bit of the plugin type whose supported extensions it wants to know to `GetExtensions()`. Note that this is a not a combination of capability flags but only a single capability bit will ever be set in each call to `GetExtensions()` Hollywood makes, i.e. Hollywood will call `GetExtensions()` for each plugin type whose supported extensions it wants to get individually.

The following plugin types currently support extensions:

`HWPLUG_CAPS_LIBRARY:`

Library plugins currently support the following extensions:

`HWEXT_LIBRARY_MULTIPLE:`

If this extension bit is set, your library plugin wants to install multiple libraries and has implemented the `GetLibraryCount()` and `SetCurrentLibrary()` functions to handle this. See Section 15.1 [Library plugins], page 129, for details. (V6.0)

`HWEXT_LIBRARY_NOAUTOINIT:`

If this extension bit is set, the commands and constants of your library won't be added automatically to Hollywood's set of commands and constants. Instead, they'll only be added when the script performs a `@REQUIRE` on your plugin. See Section 15.1 [Library plugins], page 129, for details. (V6.1)

`HWEXT_LIBRARY_HELPSTRINGS:`

If this extension bit is set, your library plugin wants to provide help strings and nodes for the individual plugin commands and needs to implement the `GetHelpStrings()` function to handle

this. See Section 15.1 [Library plugins], page 129, for details. (V7.0)

`HWPLUG_CAPS_IMAGE:`
Image plugins currently support the following extensions:

`HWEXT_IMAGE_NOAUTOINIT:`
If this extension bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you'll have to do this manually by calling the `hw_AddLoaderAdapter()` function. See Section 13.1 [Image plugins], page 117, for details. (V6.0)

`HWPLUG_CAPS_ANIM:`
Anim plugins currently support the following extensions:

`HWEXT_ANIM_NOAUTOINIT:`
If this extension bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you'll have to do this manually by calling the `hw_AddLoaderAdapter()` function. See Section 4.1 [Anim plugins], page 31, for details. (V6.0)

`HWPLUG_CAPS_SOUND:`
Sound plugins currently support the following extensions:

`HWEXT_SOUND_NOAUTOINIT:`
If this extension bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you'll have to do this manually by calling the `hw_AddLoaderAdapter()` function. See Section 19.1 [Sound plugins], page 155, for details. (V6.0)

`HWPLUG_CAPS_VIDEO:`
Video plugins currently support the following extensions:

`HWEXT_VIDEO_NOAUTOINIT:`
If this extension bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you'll have to do this manually by calling the `hw_AddLoaderAdapter()` function. See Section 22.1 [Video plugins], page 175, for details. (V6.0)

`HWPLUG_CAPS_VECTOR:`
Vectorgraphics plugins currently support the following extensions:

`HWEXT_VECTOR_EXACTFIT:`
If this extension bit is set, `GetPathExtents()` must take the transformation matrix it is passed into account when computing the path's extents. If `HWEXT_VECTOR_EXACTFIT` is not set, Hollywood will compute the extents of the transformed path but this is not recommended since it is your plugin that knows best about the real extents. See Section 21.1 [Vectorgraphics plugins], page 163, for details. (V6.0)

`HWPLUG_CAPS_DISPLAYADAPTER:`
Display adapter plugins currently support the following extensions:

`HWEXT_DISPLAYADAPTER_MAINLOOP:`
: If this extension bit is set, the display adapter is marked as main loop-based which means that Hollywood will explicitly ask the display adapter to run its main loop instead of just calling the `HandleEvents()` and `WaitEvents()` functions from time to time. See Section 10.3 [AdapterMainLoop], page 61, for details. (V6.1)

INPUTS

`capbit` single capability bit of the plugin type whose extensions should be queried

`tags` reserved for future use (currently `NULL`)

RESULTS

`exts` combination of extension bits for the specified plugin type (see above)

12 File adapter plugins

12.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_FILEADAPTER` set can hook into Hollywood's file handler. Whenever Hollywood has to open a file, it will first ask all the plugins that have hooked themselves into Hollywood's file handler if one of them wants to open it instead. If one of the plugins chooses to handle this file type, all file IO will be done through the file adapter functions implemented in the plugin. Otherwise, Hollywood will do all file IO through its default file handler.

Please note that file adapters are not automatically initialized when Hollywood loads the plugin. Instead, you have to manually call `hw_AddLoaderAdapter()` in your `RequirePlugin()` function to activate the file adapter. The file adapter will then only be activated if the user calls `@REQUIRE` on your plugin. See Section 2.5 [Auto and manual plugin initialization], page 8, for details. If you do not call `hw_AddLoaderAdapter()` on your file adapter plugin, it will only be available if the user addresses it directly through the `Adapter` tag.

See Section 34.2 [hw_AddLoaderAdapter], page 277, for information on how to add your file adapter.

All functions of this plugin type have to be implemented in a thread-safe manner.

This plugin type is supported since Hollywood 6.0.

12.2 FClose

NAME

FClose – close a file handle (V6.0)

SYNOPSIS

```
int ok = FClose(APTR handle);
```

FUNCTION

This function must close the specified file handle, finishing all pending writes. `FClose()` must return `True` on success, `False` otherwise.

This function must be implemented in a thread-safe manner.

INPUTS

`handle`	file handle returned by `FOpen()`

RESULTS

`ok`	`True` to indicate success, `False` on failure

12.3 FEof

NAME

FEof – check if end-of-file marker has been reached (V6.0)

SYNOPSIS

```
int ok = FEof(APTR handle);
```

FUNCTION

This function must return `True` if the end-of-file marker has been reached for the specified file handle.

This function must be implemented in a thread-safe manner.

INPUTS

`handle` file handle returned by `FOpen()`

RESULTS

`ok` `True` or `False`

12.4 FFlush

NAME

FFlush – flush all pending writes to file (V6.0)

SYNOPSIS

```
int ok = FFlush(APTR handle);
```

FUNCTION

This function must flush any pending buffered write operations to the specified file handle and return `True` on success, `False` otherwise.

If your file adapter doesn't support writing to files, this function can be a dummy stub.

This function must be implemented in a thread-safe manner.

INPUTS

`handle` file handle returned by `FOpen()`

RESULTS

`ok` `True` to indicate success, `False` on failure

12.5 FGetC

NAME

FGetC – read a single character from a file (V6.0)

SYNOPSIS

```
int c = FGetC(APTR handle);
```

FUNCTION

This function must read a single character from the specified file handle and return it. In case the end-of-file marker has been reached or an error has occurred, -1 must be returned.

This function must be implemented in a thread-safe manner.

INPUTS

`handle` file handle returned by `FOpen()`

RESULTS

`c` character read or -1 on error or EOF

12.6 FOpen

NAME

FOpen – open a file (V6.0)

SYNOPSIS

```
APTR handle = FOpen(STRPTR name, int mode, struct hwTagList *tags);
```

FUNCTION

This function is called for every file that Hollywood opens. Your `FOpen()` implementation has to check whether your plugin wants to handle this file or not. If your plugin wants to handle this file, your `FOpen()` implementation needs to open it and return a handle to Hollywood. Otherwise `FOpen()` must return `NULL`. The handle returned by this function is an opaque data type only your plugin knows about. Hollywood will pass this handle to you whenever it wants to do IO on this file.

The second parameter specifies whether Hollywood wants to open this file for reading and/or writing. It can be one of the following values:

`HWFOPENMODE_READ_NEW:`
: File should be opened for reading. `FOpen()` must fail if file doesn't exist.

`HWFOPENMODE_WRITE:`
: File should be opened for writing. If it doesn't exist, `FOpen()` has to create it first.

`HWFOPENMODE_READWRITE:`
: File should be opened for reading and writing.

Additionally, Hollywood will pass a tag list to your implementation in parameter 3. This tag list can contain the following items:

`HWFOPENTAG_FLAGS:`
: This tag allows you to report certain flags about the file back to Hollywood. The `pData` member of this tag will be set to a pointer to a `ULONG`. You may then set one or more of the following flags in this `ULONG` to inform Hollywood about the properties of this file. The following flags are currently recognized:

  `HWFOPENFLAGS_STREAMING:`
  : Setting this flag tells Hollywood that the file is being streamed from a network source. If this flag is set, Hollywood will try to avoid operations that are inefficient on streaming sources like excessive seeking operations.

  `HWFOPENFLAGS_NOSEEK:`
  : Setting this flag tells Hollywood that the file cannot be seeked. Note that if you set this flag, you will still have to implement

the FSeek() function but it only needs to support rewinding (i.e. reverting the read/write cursor to the beginning of the file) and querying the current file cursor position. Note that if you set HWFOPENFLAGS_NOSEEK several file format handlers which depend on the seek functionality might stop working. Plugins may choose to work-around this problem by setting the HWFOPENMODE_EMULATESEEK flag when calling hw_FOpen(). See Section 25.15 [hw_FOpen], page 200, for details.

HWFOPENTAG_CHUNKFILE:
: If you've set the HWCLAFAFLAGS_CHUNKLOADER flag to indicate that your file adapter supports opening of virtual files that do not exist physically but only as parts of other files, you can use this tag to find out the path to the real file that contains the virtual file. If Hollywood passes the HWFOPENTAG_CHUNKFILE tag to your FOpen() implementation, the pData member will be set to a string containing the path to the real file Hollywood wants you to open, but keep in mind that Hollywood wants you to look at a part of this file only. This part is described by the HWFOPENTAG_CHUNKOFFSET and HWFOPENTAG_CHUNKLENGTH tags which are always passed alongside HWFOPENTAG_CHUNKFILE. HWFOPENTAG_CHUNKOFFSET specifies the offset where the virtual file inside the file passed in HWFOPENTAG_CHUNKFILE starts and HWFOPENTAG_CHUNKLENGTH specifies its length. Your file adapter implementation must remap all accesses to the virtual file to the physical file specified in HWFOPENTAG_CHUNKFILE then, i.e. if the user calls FSeek() to seek to the beginning of the file, your implementation of FSeek() must actually seek to the position specified in HWFOPENTAG_CHUNKOFFSET and so on. You only need to implement support for HWFOPENTAG_CHUNKFILE if you set LinkMode to HWSTATLKMODE_CONTAINER in FStat(). Otherwise, it's not necessary to implement HWFOPENTAG_CHUNKFILE. See Section 12.10 [FStat], page 111, for details.

HWFOPENTAG_CHUNKMEMORY:
: This is similar to HWFOPENTAG_CHUNKFILE except that the pData member of this tag doesn't point to a string containing a file path but to a memory block containing the data of the virtual file. You can look at the HWFOPENTAG_CHUNKLENGTH to find out the size of the memory block. HWFOPENTAG_CHUNKOFFSET is not used for this tag. See above for more information.

HWFOPENTAG_CHUNKOFFSET:
: If HWFOPENTAG_CHUNKFILE is set, the iData member of this tag will be set to the starting offset of the virtual file inside the physical file specified in HWFOPENTAG_CHUNKFILE.

HWFOPENTAG_CHUNKLENGTH:
: If either HWFOPENTAG_CHUNKFILE or HWFOPENTAG_CHUNKMEMORY is set, the iData member of this tag will be set to the length of the virtual file in bytes.

This function must be implemented in a thread-safe manner.

INPUTS

`name` file to open

`mode` desired access mode (see above)

`tags` tag list with additional options (see above)

RESULTS

`handle` handle to refer to this file later or `NULL` if your plugin doesn't want to handle this file

12.7 FPutC

NAME

FPutC – write single character to file (V6.0)

SYNOPSIS

```
int ok = FPutC(APTR handle, int ch);
```

FUNCTION

This function must write the specified character to the specified file handle. It must return `True` on success or `False` on failure.

If your file adapter doesn't support writing to files, this function can be a dummy stub.

This function must be implemented in a thread-safe manner.

INPUTS

`handle` file handle returned by `FOpen()`

`ch` character to write to file (0-255)

RESULTS

`ok` `True` to indicate success, `False` on failure

12.8 FRead

NAME

FRead – read file data into memory buffer (V6.0)

SYNOPSIS

```
int read = FRead(APTR handle, APTR buf, int size);
```

FUNCTION

This function has to read the specified number of bytes into the memory buffer specified in parameter 2. It has to return the number of bytes actually read.

This function must be implemented in a thread-safe manner.

INPUTS

`handle` file handle returned by `FOpen()`

`buf` pointer to memory buffer to receive the data read

`size` number of bytes to read from file handle

RESULTS

`read` number of bytes actually read

12.9 FSeek

NAME

FSeek – seek file to new position (V6.0)

SYNOPSIS

```
DOSINT64 oldpos = FSeek(APTR handle, DOSINT64 pos, int mode);
```

FUNCTION

This function has to seek the file handle's read/write cursor to the specified position. Additionally, it has to return the position of the read/write cursor before the seek operation. The specified position is relative to the seek mode passed in parameter 3. This can be one of the following modes:

`HWFSEEKMODE_CURRENT:`
New seek position is relative to the current position.

`HWFSEEKMODE_BEGINNING:`
New seek position is relative to the beginning of the file.

`HWFSEEKMODE_END:`
New seek position is relative to the end of the file.

Note that `FSeek()` is often called with a 0 zero position and `HWFSEEKMODE_CURRENT` to query the position of the read/write cursor.

If there was an error, `FSeek()` has to return -1.

If you set the `HWFOPENFLAGS_NOSEEK` flag in your `FOpen()` implementation, your `FSeek()` implementation only has to support rewinding the file and querying the position of the cursor. In terms of code, this means that `FSeek()` only has to support the following two operations if `HWFOPENFLAGS_NOSEEK` is set:

```
FSeek(fh, 0, HWFSEEKMODE_BEGINNING);     // rewind
pos = FSeek(fh, 0, HFSEEKMODE_CURRENT);  // query cursor position
```

For any other operation, it has to return -1, i.e. an error has occurred.

This function must be implemented in a thread-safe manner.

INPUTS

`handle` file handle returned by `FOpen()`

`pos` destination seek position

`mode` seek mode (see above)

RESULTS

`oldpos` previous position of file cursor or -1 on error

12.10 FStat

NAME

FStat – obtain information about open file (V6.0)

SYNOPSIS

```
int ok = FStat(APTR handle, ULONG flags, struct hwos_StatStruct *st,
                    struct hwTagList *tags);
```

FUNCTION

This function has to do the same as `Stat()` but instead of a string describing a path to a file system object it has to be able to obtain information about a file from its handle allocated by `FOpen()`. `FStat()` needs to write the information about the file to the structure pointer passed in parameter 3. `struct hwos_StatStruct` looks like this:

```
struct hwos_StatStruct
{
    int Type;                                   // [out]
    DOSINT64 Size;                              // [out]
    ULONG Flags;                                // [out]
    struct hwos_DateStruct Time;                // [out]
    struct hwos_DateStruct LastAccessTime;      // [out]
    struct hwos_DateStruct CreationTime;        // [out]
    STRPTR FullPath;                            // [out]
    STRPTR Comment;                             // [out]
    int LinkMode;                               // [out]
    STRPTR Container;                           // [out]
};
```

Your `FStat()` implementation needs to write the following information to the individual structure members:

`Type:` This must always be set to `HWSTATTYPE_FILE`.

`Size:` This must be set to the size of the file in bytes or -1 if the size is not known, maybe because the file is being streamed from a network source.

`Flags:` Combination of flags describing the file system object attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

`Time:` Time stamp indicating when this file system object was last modified. This information is optional. Do not touch this member if you don't have this time information.

`LastAccessTime:`
Time stamp indicating when this file system object was last accessed. This information is optional. Do not touch this member if you don't have this time information.

`CreationTime:`
Time stamp indicating when this file system object was created. This information is optional. Do not touch this member if you don't have this time information.

`FullPath:`

Fully qualified path to the file. This must be provided. If the `HWSTATFLAGS_ALLOCSTRINGS` flag is not set, you can set this to a static string buffer which must stay valid until the next call to `FStat()`. If `HWSTATFLAGS_ALLOCSTRINGS` has been set, you need to allocate a string buffer using `hw_TrackedAlloc()`.

`Comment:` Comment stored for this file in the file system. Set this to `NULL` if you do not have this information or the file system doesn't support storage of comments. If the `HWSTATFLAGS_ALLOCSTRINGS` flag is not set, you can set this to a static string buffer which must stay valid until the next call to Stat(). If `HWSTATFLAGS_ALLOCSTRINGS` has been set, you need to allocate a string buffer using `hw_TrackedAlloc()`.

`LinkMode:`

This member has to be set to the link mode to use when Hollywood needs to link this file into an applet or executable. This can be one of the following pre-defined link modes:

`HWSTATLKMODE_NORMAL:`

Normal link mode. This means that all data is simply read from the file and is written to the applet or executable. Consequently, your file adapter is no longer necessary when running the compiled applet or executable since the data has already been converted to its raw form. Thus, if you use `HWSTATLKMODE_NORMAL`, the compiled applet or executable won't require your file adapter plugin any more. Also, your file adapter won't be called at all when the user runs the compiled applet or executable because Hollywood has already obtained the raw data from the file adapter during linking stage.

`HWSTATLKMODE_NONE:`

This file should never be linked to applets or executables. If you use this link mode, Hollywood will not link the file and just keep the original reference that was specified in the Hollywood script, whatever it may be. This can be useful when writing a file adapter that streams data from a network source like an HTTP server. It wouldn't make sense then for the Hollywood linker to always download the whole file and link it to your applet or executable. Instead, just the URL specification should be linked so that the data is streamed from this URL when the user runs the compiled applet or executable. In that case `HWSTATLKMODE_NONE` is the right choice since it skips linking for this file altogether.

`HWSTATLKMODE_CONTAINER:`

This link mode allows you to specify a container file that should be linked instead of the current file. Imagine you are writing a file adapter that can load a compressed file format like gzip. If you used `HWSTATLKMODE_NORMAL` now, Hollywood

would always link the uncompressed data to the applet or executable. However, you might want to make Hollywood link the compressed data instead. This can be achieved by setting the link mode to `HWSTATLKMODE_CONTAINER` and then setting the `Container` member of this structure to the file that contains the compressed data. When setting `LinkMode` to `HWSTATLKMODE_CONTAINER`, Hollywood will always link the file specified in `Container` instead of the current file. Please note that if you use `HWSTATLKMODE_CONTAINER`, your implementation of `FOpen()` has to support the `HWFOPENTAG_CHUNKXXX` tags and you have to set the `HWCLAFAFLAGS_CHUNKLOADER` flag using `hw_ConfigureLoaderAdapter()`. see{FOpen, FOpen}

`Container:`

If you set `LinkMode` to `HWSTATLKMODE_CONTAINER`, you need to set this member to a path to a file that should be linked instead of the current file when Hollywood is in linking mode. This can be used for fine-tuned control over Hollywood's linker. See above for more information. The string buffer you use to pass a container file to Hollywood must stay valid until the next call to `FStat()`. Note that `HWSTATFLAGS_ALLOCSTRINGS` (see below) doesn't affect `Container`. It must always use a static string buffer. If link mode isn't set to `HWSTATLKMODE_CONTAINER`, set this member to `NULL`.

The following flags are supported by `FStat()`:

`HWSTATFLAGS_ALLOCSTRINGS:`

If this flag is set, `FStat()` must not use static string buffers for the `FullPath` and `Comment` structure members but allocate new private string buffers for them. Hollywood will then call `hw_TrackedFree()` on these buffers once it is done with them. This flag is often set when `FStat()` is used in a multithreaded setup.

`FStat()` has to return `True` on success or `False` on failure.

This function must be implemented in a thread-safe manner if the `HWSTATFLAGS_ALLOCSTRINGS` flag is set.

INPUTS

`handle`	file handle returned by `FOpen()`
`flags`	additional flags (see above)
`st`	pointer to a `struct hwos_StatStruct` for storing information about the file
`tags`	reserved for future use (currently `NULL`)

RESULTS

`ok`	`True` to indicate success, `False` on failure

12.11 FWrite

NAME

FWrite – write data to file handle (V6.0)

SYNOPSIS

```
int written = FWrite(APTR handle, APTR buf, int size);
```

FUNCTION

This function has to write the specified number of bytes from the memory buffer specified in parameter 2 to the file handle passed in parameter 1. It has to return the number of bytes actually written.

If your file adapter doesn't support writing to files, this function can be a dummy stub.

This function must be implemented in a thread-safe manner.

INPUTS

`handle`	file handle returned by `FOpen()`
`buf`	source memory buffer
`size`	number of bytes to write to file handle

RESULTS

`written`	number of bytes actually written

12.12 Stat

NAME

Stat – examine a file system object (V6.0)

SYNOPSIS

```
int ok = Stat(STRPTR name, ULONG flags, struct hwos_StatStruct *st,
                  struct hwTagList *tags);
```

FUNCTION

This function has to examine the file system object specified in parameter 1 and write information about it to the structure pointer passed in parameter 3. `struct hwos_StatStruct` looks like this:

```
struct hwos_StatStruct
{
    int Type;                                // [out]
    DOSINT64 Size;                           // [out]
    ULONG Flags;                             // [out]
    struct hwos_DateStruct Time;             // [out]
    struct hwos_DateStruct LastAccessTime;   // [out]
    struct hwos_DateStruct CreationTime;     // [out]
    STRPTR FullPath;                         // [out]
    STRPTR Comment;                          // [out]
    int LinkMode;                            // [out]
```

```
        STRPTR Container;                       // [out]
    };
```

Your `Stat()` implementation needs to write the following information to the individual structure members:

`Type:` This must be set to one of the following types:

`HWSTATTYPE_FILE:`
: The file system object examined is a file.

`HWSTATTYPE_DIRECTORY:`
: The file system object examined is a directory.

`Size:` Size of object in bytes if it is a file, 0 for directories. Note that this can also be set to -1 in case the file size isn't know, for example because the file is being streamed from a network source.

`Flags:` Combination of flags describing the file system object attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

`Time:` Time stamp indicating when this file system object was last modified. This information is optional. Do not touch this member if you don't have this time information.

`LastAccessTime:`
: Time stamp indicating when this file system object was last accessed. This information is optional. Do not touch this member if you don't have this time information.

`CreationTime:`
: Time stamp indicating when this file system object was created. This information is optional. Do not touch this member if you don't have this time information.

`FullPath:`
: Fully qualified path to the file system object. This must be provided. If the `HWSTATFLAGS_ALLOCSTRINGS` flag is not set, you can set this to a static string buffer which must stay valid until the next call to `Stat()`. If `HWSTATFLAGS_ALLOCSTRINGS` has been set, you need to allocate a string buffer using `hw_TrackedAlloc()`.

`Comment:` Comment stored for this object in the file system. Set this to `NULL` if you do not have this information or the file system doesn't support storage of comments. If the `HWSTATFLAGS_ALLOCSTRINGS` flag is not set, you can set this to a static string buffer which must stay valid until the next call to `Stat()`. If `HWSTATFLAGS_ALLOCSTRINGS` has been set, you need to allocate a string buffer using `hw_TrackedAlloc()`.

`LinkMode:`
: Currently unused. Set to 0.

`Container:`
: Currently unused. Set to `NULL`.

The following flags are supported by `Stat()`:

`HWSTATFLAGS_ALLOCSTRINGS:`
: If this flag is set, `Stat()` must not use static string buffers for the `FullPath` and `Comment` structure members but allocate new private string buffers for them. Hollywood will then call `hw_TrackedFree()` on these buffers once it is done with them. This flag is often set when `Stat()` is used in a multithreaded setup.

`Stat()` has to return `True` on success or `False` on failure.

`Stat()` is often used by Hollywood to find out whether a certain file system object is a file or a directory. It is also used to resolve relative file name specifications into absolute, fully-qualified paths. So make sure your implementation provides this information in the `FullPath` structure member above.

This function must be implemented in a thread-safe manner if the `HWSTATFLAGS_ALLOCSTRINGS` flag is set.

INPUTS

`name`	name of file system object to examine
`flags`	additional flags (see above)
`st`	pointer to a `struct hwos_StatStruct` for storing information about the file system object
`tags`	reserved for future use (currently `NULL`)

RESULTS

`ok`	`True` to indicate success, `False` on failure

13 Image plugins

13.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_IMAGE` set will be called whenever Hollywood has to load an image. The plugin can check then whether the image is in a format that the plugin recognizes and if it is, it can open the image and return the raw pixel data to Hollywood. This makes it possible to load custom image formats with Hollywood.

Image plugins can support two different image types: Raster and vector images. If your plugin supports vector images, Hollywood will always call your plugin whenever it needs to transform the image. Your plugin can then do the lossless vector image transformation on its own and return the new pixel data to Hollywood. For raster images, image transformation is always done by Hollywood and your plugin doesn't have to do anything.

By default, image plugins are automatically activated when Hollywood loads them. Starting with Hollywood 6.0 this behaviour can be changed by setting the `HWEXT_IMAGE_NOAUTOINIT` extension bit. If this bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you will have to manually call `hw_AddLoaderAdapter()` to activate your plugin. For example, you could call `hw_AddLoaderAdapter()` from your `RequirePlugin()` implementation. In that case, the image plugin would only be activated if the user called `@REQUIRE` on it. If you do not call `hw_AddLoaderAdapter()` on a plugin that has auto-initialization disabled, it will only be available if the user addresses it directly through the `Loader` tag. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

The SDK distribution comes with an example image plugin which contains a loader for the PCX image format. Feel free to study this example code to learn how image plugins are written in practice.

13.2 FreeImage

NAME

FreeImage – free image handle (V5.0)

SYNOPSIS

```
void FreeImage(APTR handle);
```

FUNCTION

This function must free the specified image handle that has been allocated by your plugin's `LoadImage()` function. Hollywood will call `FreeImage()` when it is done with your image.

INPUTS

`handle` image handle returned by `LoadImage()`

13.3 GetImage

NAME

GetImage – get raw pixel image data (V5.0)

SYNOPSIS

```
ULONG *raw = GetImage(APTR handle, struct LoadImageCtrl *ctrl);
```

FUNCTION

This function must return the image's raw pixel data encoded as an array of 32-bit ARGB values. If the image type is `HWIMAGETYPE_VECTOR`, `GetImage()` also needs to take possible transformations that have been applied via `TransformImage()` into account.

Furthermore, `GetImage()` has to provide some additional information in the `struct LoadImageCtrl` pointer that is passed as the second parameter. See Section 13.5 [LoadImage], page 119, for details on this structure. The following information has to be provided by `GetImage()`:

`Width:` Must be set to the image width in pixels. If the image type is `HWIMAGETYPE_VECTOR` and `TransformImage()` has been called prior to `GetImage()`, this value must exactly match the width that has been passed to the last call of `TransformImage()`.

`Height:` Must be set to the image height in pixels. If the image type is `HWIMAGETYPE_VECTOR` and `TransformImage()` has been called prior to `GetImage()`, this value must exactly match the height that has been passed to the last call of `TransformImage()`.

`LineWidth:`
Must be set to the image modulo width in pixels. This is often the same as the image width.

`AlphaChannel:`
Must be set to `True` or `False`, depending on whether or not this image has an alpha channel.

The pointer that is returned by `GetImage()` must stay valid at least until the next call to `GetImage()` or `FreeImage()` on this handle.

INPUTS

`handle` image handle as returned by `LoadImage()`

`ctrl` pointer to a `struct LoadImageCtrl` for storing information about the image

RESULTS

`raw` an array of raw 32-bit ARGB pixels

13.4 IsImage

NAME

IsImage – check if a file is in a supported image format (V5.0)

SYNOPSIS

```
int ok = IsImage(STRPTR filename, struct LoadImageCtrl *ctrl);
```

FUNCTION

This function has to check whether the specified file is in an image format that the plugin wants to handle. If it is, the plugin has to return `True` and provide information about

the image's size and whether or not it has an alpha channel. This is done by setting the following members of the `struct LoadImageCtrl` pointer that is passed to `IsImage()` in the second argument:

`Width:` Must be set to the image width in pixels.

`Height:` Must be set to the image height in pixels.

`AlphaChannel:`
Must be set to `True` or `False`, depending on whether or not this image has an alpha channel.

`Flags:` Your implementation may set the following flags:

`HWIMGFLAGS_TRANSPARENCY:`
Set this flag to tell Hollywood that the image has a monochrome transparency channel (e.g. a transparent pen in a palette-based image). (V6.0)

The following members of `struct LoadImageCtrl` are set by Hollywood before it calls your implementation of `IsImage()`:

`Adapter:` Starting with Hollywood 6.0 users can specify the file adapter that should be used to open certain files. If this member is non-`NULL`, Hollywood wants your plugin to use the file adapter specified in `Adapter` to open the file. This means that you have to use `hw_FOpenExt()` instead of `hw_FOpen()` to open the file. Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions. See Section 25.16 [hw_FOpenExt], page 202, for details. (V6.0)

You must not touch any other members of the `struct LoadImageCtrl` pointer that is passed to this function. See Section 13.5 [LoadImage], page 119, for details on this structure.

INPUTS

`filename` filename to examine

`ctrl` pointer to a `struct LoadImageCtrl` for storing information about the image

RESULTS

`ok` `True` if the plugin wants to handle this file, `False` otherwise

13.5 LoadImage

NAME

LoadImage – load image into memory (V5.0)

SYNOPSIS

```
APTR handle = LoadImage(STRPTR filename, struct LoadImageCtrl *ctrl);
```

FUNCTION

This function has to open the specified filename, check if it is in a format that the plugin wants to handle, and, if it is, return a handle to the image back to Hollywood. Otherwise

it has to return NULL. The handle returned by LoadImage() is an opaque datatype that only your plugin knows about. Hollywood will simply pass this handle back to your GetImage() function when it wants to have the raw pixel data.

This function also has to provide certain information about the image it has just loaded. This information has to be written to the struct LoadImageCtrl that is passed in the second parameter. This structure looks like this:

```
struct LoadImageCtrl
{
    int Width;              // [out]
    int Height;             // [out]
    int LineWidth;          // [out]
    int AlphaChannel;       // [out]
    int ForceAlphaChannel; // [out]
    int Type;               // [out]
    ULONG Flags;            // [in/out] -- V5.3
    int ScaleWidth;         // [in]     -- V5.3
    int ScaleHeight;        // [in]     -- V5.3
    int BaseWidth;          // [out]    -- V5.3
    int BaseHeight;         // [out]    -- V5.3
    ULONG ScaleMode;        // [in]     -- V5.3
    STRPTR Adapter;         // [in]     -- V6.0
};
```

The following information has to be provided by LoadImage():

Type: This must be set to either HWIMAGETYPE_RASTER or HWIMAGETYPE_VECTOR. If you set this to HWIMAGETYPE_VECTOR, Hollywood will call your TransformImage() function whenever it needs to transform the image. This allows you to do lossless transformation of the vector image. For images of type HWIMAGETYPE_RASTER, TransformImage() is never called. Instead, Hollywood does all transformations itself.

Width: Must be set to the image width in pixels.

Height: Must be set to the image height in pixels.

LineWidth:
Must be set to the image modulo width in pixels. This is often the same as the image width.

AlphaChannel:
Must be set to True or False, depending on whether or not this image has an alpha channel.

ForceAlphaChannel:
If this is set to True, Hollywood will automatically create an alpha channel for all objects that load this image. For example, if the user calls LoadBrush() on your image but does not set the "LoadAlpha" tag to True, the brush will still get an alpha channel if you set "ForceAlphaChannel" to True.

`Flags:` The following flags may be set by Hollywood:

`HWIMGFLAGS_TRANSPARENCY:`
This flag will be set whenever the user sets the "LoadTransparency" tag to `True`. You may then choose to write the image's transparency information to its alpha channel and set the `ForceAlphaChannel` member to `True`. See above for more information. (V6.0)

The following flags may be set by your implementation:

`HWIMGFLAGS_DIDSCALE:`
If your plugin has scale-loaded this image, you have to set the `HWIMGFLAGS_DIDSCALE` flag here so that Hollywood knows that your plugin has loaded and scaled the image. See below for more information on scaled loading of images. (V5.3)

`Adapter:` Starting with Hollywood 6.0 users can specify the file adapter that should be used to open certain files. If this member is non-`NULL`, Hollywood wants your plugin to use the file adapter specified in `Adapter` to open the image. This means that you have to use `hw_FOpenExt()` instead of `hw_FOpen()` to open the image. Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions. See Section 25.16 [hw_FOpenExt], page 202, for details. (V6.0)

Starting with Hollywood 5.3 `LoadImage()` also supports scaled loading of images. This is optional functionality and need not be supported by `LoadImage()`. If you want to support it, you have to look at the members `ScaleWidth` and `ScaleHeight` of the `struct LoadImageCtrl` pointer that is passed to `LoadImage()`. **Warning! Make sure that you access these members only if you have checked that your plugin has been opened by version 5.3 or higher of Hollywood. Otherwise, these members won't be there and trying to access them will read from bad memory locations and give you back random values**. So if you want to implement support for scaled loading of images, first check for Hollywood 5.3 and then take a look at the following members of the `struct LoadImageCtrl`:

`ScaleWidth:`
If Hollywood wants your plugin to scale the image while loading, this member will be set to either a positive or negative integer. A positive integer value specifies the desired width in pixels for this image while a negative integer value is to be interpreted as a percentage value specifying the desired scaling factor relative to the original image width, i.e. `"-75"` means that the image should be scaled to 75% of its original width. If `ScaleWidth` is 0, Hollywood doesn't want to have any scaling. (V5.3)

`ScaleHeight:`
This works in the same way as described above for `ScaleWidth` except that it deals with the image height. (V5.3)

`ScaleMode:`
Contains the ID of a scale mode. Currently, this can be either 0 for hard scaling or 1 for interpolated scaling using anti-aliasing. (V5.3)

`BaseWidth:`
: If your plugin supports scaled loading, you need to set this member to the original width of the image. This is important because otherwise Hollywood won't know the original size of the image as the `Width` member needs to be set to the scaled width if you do scaling. (V5.3)

`BaseHeight:`
: Same as `BaseWidth` but for the image height. (V5.3)

Please note that you should not use ANSI C functions like `fopen()` to open the file that is passed to this function because the filename that is passed to this function can also be a specially formatted filename specification that Hollywood uses to load files that have been linked to applets or executables. In order to be able to load these files correctly, you have to use special IO functions provided by Hollywood. See Section 2.12 [File IO information], page 15, for details.

INPUTS

`filename` filename to open

`ctrl` pointer to a `struct LoadImageCtrl` for storing information about the image

RESULTS

`handle` a handle that identifies this image or `NULL` if plugin doesn't want to handle this image

13.6 TransformImage

NAME

TransformImage – transform a vector image (V5.0)

SYNOPSIS

```
int ok = TransformImage(APTR handle, struct hwMatrix2D *m, int width,
                int height);
```

FUNCTION

This function must transform the specified vector image according to the 2D transformation matrix passed in parameter 2. It must also clip the resulting image to the specified width and height in pixels. After calling `TransformImage()`, Hollywood will then call your plugin's `GetImage()` function again to obtain the raw pixel data of the newly transformed image. It is very important that the dimensions and the pixel array returned by the next call to `GetImage()` match the dimensions passed to `TransformImage()` in parameters 3 and 4 exactly.

`TransformImage()` is only ever called for images of type `HWIMAGETYPE_VECTOR`. If your `LoadImage()` function sets the image type to `HWIMAGETYPE_RASTER`, `TransformImage()` won't be called at all and Hollywood will do all image transformations on its own.

If the transformation was successful, `TransformImage()` must return `True`. Otherwise it has to return `False`.

INPUTS

`handle` image handle as returned by `LoadImage()`

`m`	2D matrix describing the desired transformation
`width`	clipping width for resulting image
`height`	clipping height for resulting image

RESULTS

`ok`	`True` or `False` indicating success or failure

14 Image saver plugins

14.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_SAVEIMAGE` set can register one or more additional output image formats. The user will then be able to save images in the output formats supported by the plugin. Plugins have to register new output image formats by passing the name of a constant that should be used to access the new format. For example, a plugin might choose to register a new output image format under the constant `#IMGFMT_CUSTOMFORMAT`. Whenever the user calls `SaveBrush()` or `SaveSnapshot()` now and passes `#IMGFMT_CUSTOMFORMAT` as the image format, Hollywood will ask the plugin to save the image.

14.2 RegisterImageSaver

NAME

RegisterImageSaver – register a new image saver (V5.0)

SYNOPSIS

```
void RegisterImageSaver(struct SaveImageReg *reg)
```

FUNCTION

Hollywood will call this function to get information about the image saver your plugin wants to register. In addition, `RegisterImageSaver()` has to tell Hollywood whether it wants to register another image saver. Hollywood will pass a pointer to a `struct SaveImageReg` to this function. This structure looks like this:

```
struct SaveFormatReg
{
    ULONG CapsMask;     [out]
    ULONG FormatID;     [in/out]
    STRPTR FormatName;  [out]
};

struct SaveImageReg
{
    struct SaveFormatReg hdr;
};
```

Your implementation has to do the following with the individual structure members:

`CapsMask:`

This must be set to a combination of flags that tell Hollywood about the capabilities of the image saver that is to be registered. The following flags are currently supported:

`HWSAVEIMGCAPS_ARGB:`

Your image saver supports source image data that is delivered as a 32-bit ARGB pixel array.

HWSAVEIMGCAPS_CHUNKY:
: Your image saver supports source image data that is delivered as 8-bit chunky pixels that are index values for a palette look-up table.

HWSAVEIMGCAPS_ALPHA:
: Your image saver supports alpha channel saving. This is only supported if you also set HWSAVEIMGCAPS_ARGB.

HWSAVEIMGCAPS_MORE:
: If you set this flag, Hollywood will call RegisterImageSaver() again so that you can register another saver. If you don't want to register another saver, don't set this flag. (V5.3)

Note that HWSAVEIMGCAPS_ARGB and HWSAVEIMGCAPS_CHUNKY are not mutually exclusive. You can set them both if the target image format supports both true colour and palette-based pixel data storage.

FormatID:
: This member must be set to a unique 32-bit value that should be assigned to the constant that is registered for accessing this image saver from Hollywood scripts. Values smaller than 32768 are reserved for internal Hollywood use. You may use values larger than 32768 for your saver but if you want to publish your plugin, you need to contact Airsoft Softwair to obtain a unique value that is still vacant. This won't cost you anything; it's just needed to make sure that plugin image savers don't use conflicting identifiers. Also, once you have published your image saver plugin, the FormatID you have specified must not be changed or you will break compatibility with applets or executables that have been compiled with previous versions. If you are registering more than one image saver using HWSAVEIMGCAPS_MORE, you can look at the FormatID member to tell how many times Hollywood has already called RegisterImageSaver() because FormatID will contain the identifier of the last image saver you registered. If FormatID is 0, then this is the first call to RegisterImageSaver(). Note that it is not recommended to keep your own counter because Hollywood might call RegisterImageSaver() multiple times, i.e. it might first loop over RegisterImageSaver() to determine how many image savers there are in total and then it might loop over RegisterImageSaver() again to actually register their names.

FormatName:
: This must be set to a string that should form the second half of the constant that Hollywood registers for your image saver. This string you specify here must follow the naming restrictions for Hollywood constants, i.e. only alphabetical characters, numbers and very few special characters like the underscore character are allowed. The #IMGFMT_ prefix must not be included in the string you pass. Hollywood will add this automatically, i.e. if you pass the string "TESTFORMAT" here, Hollywood will make your image saver available under the constant #IMGFMT_TESTFORMAT.

INPUTS

reg — pointer to a struct SaveImageReg to be filled out by your implementation

14.3 SaveImage

NAME

SaveImage – save image to disk (V5.0)

SYNOPSIS

```
int ok = SaveImage(STRPTR filename, struct SaveImageCtrl *ctrl);
```

FUNCTION

This function must save the image provided by the pointer in the second parameter to the filename specified in the first parameter. Hollywood passes a pointer to a `struct SaveImageCtrl` to this function. This structure looks like this:

```
struct SaveImageCtrl
{
    APTR Data;        // [in]
    int *Palette;     // [in]
    int Width;        // [in]
    int Height;       // [in]
    int Modulo;       // [in]
    int Format;       // [in]
    int Quality;      // [in]
    int Colors;       // [in]
    int TransIndex;   // [in]
    ULONG Flags;      // [in]
    ULONG FormatID;   // [in] -- V5.3
};
```

In this structure Hollywood passes the following information to your `SaveImage()` function:

`Data:` The pixel data to save to the file. The actual format of this data depends on the `Format` member.

`Width:` Width of the image in pixels.

`Height:` Height of the image in pixels.

`Modulo:` Number of bytes used by a single row of pixel data. This may be larger than the specified width because there may be some padding involved.

`Format:` This specifies the pixel format of the source data passed in `Data`. May be one of the following constants:

`HWSAVEIMGFMT_ARGB:`
Data is a 32-bit array consisting of ARGB pixels.

`HWSAVEIMGFMT_CHUNKY:`
Data contains 8-bit indices into a color look-up table. This color look-up table is passed in `Palette` below.

You will only have to handle those formats here that you have explicitly declared as supported when Hollywood called your `RegisterImageSaver()` function.

`Quality:` This contains a value between 0 and 100 indicating the desired quality for the output file. Image formats that use lossy compression can use this member to determine compression settings for the image. Image formats that don't use any compression or offer lossless compression can ignore this member.

`Colors:` This contains the number of colors in the color look-up table passed in the `Palette` member. This member is only used if `Format` is `HWSAVEIMGFMT_CHUNKY`.

`Palette:` Contains the look-up table that you need to convert the chunky pixel values to RGB color values. This table consists of as many 32-bit ARGB values as has been set in the `Colors` member. Note that `Palette` is only used if `Format` is `HWSAVEIMGFMT_CHUNKY`.

`TransIndex:`
If `Format` is `HWSAVEIMGFMT_CHUNKY` this member specifies the index of the color that should appear transparent in the image. The value specified here is only valid if the `HWSAVEIMGFLAGS_TRANSINDEX` flag has been set (see below).

`Flags:` Contains a combination of flags specifying further options:

`HWSAVEIMGFLAGS_ALPHA:`
Pixel data contains alpha channel transparency values.

`HWSAVEIMGFLAGS_TRANSINDEX:`
The `TransIndex` member contains the index of a palette entry that should be made transparent in the output image.

`FormatID:`
This member contains the identifier of the image format the file should be saved in. You only need to look at this member if your plugin supports more than one output image format. But be careful, you are only allowed to look at this member if the user is running at least Hollywood 5.3. Otherwise, you must not access this member because older versions of Hollywood don't support it. (V5.3)

This function has to return `True` if the image has been successfully saved or `False` in case of an error.

INPUTS

`filename` path to a destination file

`ctrl` pointer to a `struct SaveImageCtrl` containing the image to be saved

RESULTS

`ok` `True` or `False` indicating success or failure

15 Library plugins

15.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_LIBRARY` set can add new commands and constants to Hollywood's set of inbuilt commands and constants. If you want to write such a plugin, you need to familiarize yourself with the Lua which Hollywood uses as a VM. You can access Hollywood's Lua VM through the `LuaBase` pointer that is passed inside the `hwPluginAPI` table which your `InitPlugin()` function receives.

Here is a brief explanation of how Lua calls C functions: Your function will receive just a single parameter - a pointer to the `lua_State`. All parameters that the script passes to your function will be pushed into the stack. If you want to return values to Lua, you have to push them into the stack as well and return the number of values you pushed. Alternatively, you can also return an error code. Standard error codes are defined in `hollywood/errors.h`. You can also register custom error codes using `hw_RegisterError()`.

Here is how a custom function that simply divides the first parameter by the second:

```
static SAVEDS int MyDiv(lua_State *L)
{
    double a = luaL_checknumber(L, 1);
    double b = luaL_checknumber(L, 2);

    // catch division by zero CPU exception and handle
    // it cleanly
    if(b == 0) return ERR_ZERODIVISION;

    lua_pushnumber(L, a / b);

    // push 1 to indicate one return value
    return 1;
}
```

This is just a primitive example. Check the Lua manual for more information on how to implement Lua functions in C. Please note that Hollywood uses Lua 5.0.2 so make sure you consult the correct manual. See Section 30.1 [LuaBase functions], page 263, for details.

The SDK distribution also comes with an example library plugin which adds several functions and constants to Hollywood. Feel free to study this example code to learn how library plugins are written in practice.

Starting with Hollywood 6.0 library plugins support the `HWEXT_LIBRARY_MULTIPLE` extension. If this extension is set, a library can install multiple libraries instead of just a single one. If you set the `HWEXT_LIBRARY_MULTIPLE` extension bit, you need to implement the `GetLibraryCount()` and `SetCurrentLibrary()` functions. Hollywood will then call `GetLibraryCount()` to find out how many libraries your plugin wants to install. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

Starting with Hollywood 6.1 library plugins can set the `HWEXT_LIBRARY_NOAUTOINIT` extension flag. If this is set, the library's commands and constants won't be added to Hollywood's

set of commands and constants automatically when the plugin is loaded. Instead, the library's commands and constants will only be added if the user performs a `@REQUIRE` on the plugin. This is useful for library plugins which add lots of commands and constants. Adding them to every single Hollywood script is unnecessary overhead, so it is a good idea to use `HWEXT_LIBRARY_NOAUTOINIT` for those libraries so that their commands and constants are only installed when a script explicitly requests them. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

Starting with Hollywood 7.0 library plugins support the `HWEXT_LIBRARY_HELPSTRINGS` extension bit. If this is set, your plugin needs to implement a function named `GetHelpStrings()` which needs to return a table containing a help string and a node in the plugin's help file for each command. See Section 15.6 [GetHelpStrings], page 132, for details.

15.2 FreeLibrary

NAME

FreeLibrary – free library initialization (V5.0)

SYNOPSIS

```
void FreeLibrary(lua_State *L);
```

FUNCTION

This function must free any initialization done by `InitLibrary()`. Any cleanup that requires a valid `lua_State` must be done here as the `lua_State` is no longer there once `ClosePlugin()` is called.

Note that when compiling for 64-bit Windows, this function must be called `_FreeLibrary` instead. Otherwise there will be clashes with the function of the same name from `kernel32.lib`. On 32-bit Windows this isn't a problem because 32-bit Windows uses decorated function exports.

INPUTS

`L` pointer to the `lua_State`

15.3 GetBaseTable

NAME

GetBaseTable – get name of base table for plugin functions (V5.0)

SYNOPSIS

```
STRPTR name = GetBaseTable(void);
```

FUNCTION

This function must return the name of a table that should host all the functions this plugin makes available. Functions made available by plugins should always be subsumed under a table so that there won't be any conflicts with existing or future Hollywood APIs and the user will be able to clearly distinguish plugin APIs from inbuilt APIs.

A good base table name is the module name that has been specified in `InitPlugin()`. For example, if your plugin is called `"cooladdon.hwp"`, then you might want to use

"cooladdon" as a base table name. All functions added by `GetCommands()` will then be stored in a table named "cooladdon".

INPUTS

none

RESULTS

`name` name of a table that should host all functions

15.4 GetCommands

NAME

GetCommands – get list of plugin commands (V5.0)

SYNOPSIS

```
struct hwCmdStruct *list = GetCommands(void);
```

FUNCTION

This function must return a pointer to an array of `struct hwCmdStruct` items that contains a number of commands this plugin wants to make available. The array must be terminated by two `NULL` elements. `struct hwCmdStruct` looks like this:

```
struct hwCmdStruct
{
    STRPTR Name;
    int (*Func)(lua_State *L);
};
```

You have to specify a name and a function pointer for every command that you want to add. Functions must be implemented as Lua C functions. Please check the manual of Lua 5.0.2 to find out how Lua C functions are implemented. See Section 15.1 [Library plugins], page 129, for some more details on how to implement Lua functions. The array that is to be returned by this function may look like this then:

```
struct hwCmdStruct plugin_commands[] = {
    {"TestFunc", hw_TestFunc},
    {"TestFunc2", hw_TestFunc2},
    ...
    {NULL, NULL}
};
```

All functions that you specify here will then be pushed into the Lua table whose name has been specified using `GetBaseTable()`. The user can then call your functions like this:

```
testplugin.TestFunc()
```

INPUTS

none

RESULTS

`list` an array of `struct hwCmdStruct` elements

15.5 GetConstants

NAME

GetConstants – get list of plugin constants (V5.0)

SYNOPSIS

```
struct hwCstStruct *list = GetConstants(void);
```

FUNCTION

This function must return a pointer to an array of `struct hwCstStruct` items that contains a number of constants this plugin wants to make available. The array must be terminated by three `NULL` elements. `struct hwCstStruct` looks like this:

```
struct hwCstStruct
{
    STRPTR Name;
    STRPTR StrVal;
    double Val;
};
```

Constants can be either a string or a number. If the `StrVal` member is set to `NULL`, the constant will use the numerical value specified in `Val`. Please note that the name you pass in `Name` must not contain the hash prefix. The array that is to be returned by this function may look like this then:

```
struct hwCstStruct plugin_constants[] = {
    {"NUMBERTEST", NULL, 1234},
    {"STRINGTEST", "Hello World", 0},
    ...
    {NULL, NULL, 0}
};
```

With a table like this the user would then be able to access the two new constants `#NUMBERTEST` and `#STRINGTEST` if your plugin is installed.

Please note that your constant names should be chosen in a way that they do not conflict with inbuilt constants. It is good practice to prefix the constant names with the name of your plugin so that they can be clearly distinguished from inbuilt constants and there is no risk of conflict with existing constants.

If your plugin doesn't define any constants, you may also return `NULL`.

INPUTS

none

RESULTS

`list` an array of `struct hwCstStruct` elements or `NULL`

15.6 GetHelpStrings

NAME

GetHelpStrings – get help strings for plugin commands (V7.0)

SYNOPSIS

```
struct hwHelpStruct *list = GetHelpStrings(void);
```

FUNCTION

This function is optional and must only be implemented if the `HWEXT_LIBRARY_HELPSTRINGS` extension bit has been set. See Section 11.1 [Extension plugins], page 101, for details. In that case, `GetHelpStrings()` needs to return a table containing a help string and a node in the plugin's help file for each command. This is done by returning a pointer to an array of `struct hwHelpStruct` items. `struct hwHelpStruct` looks like this:

```
struct hwHelpStruct
{
    STRPTR HelpText;
    STRPTR Node;
};
```

Your implementation of `GetHelpStrings()` has to return an array of `struct hwHelpStruct` which has exactly the same number of items as returned by your `GetCommands()` implementation in exactly the same order. Both `HelpText` and `Node` can also be `NULL` if you do not want to provide a help text or node for a certain command. If `Node` is `NULL`, Hollywood will assume the name of the command (minus any dots) as the default node name. Note that in contrast to `GetCommands()`, `NULL` doesn't act as a list terminator for `GetHelpStrings()`. The number of items in the list returned by `GetHelpStrings()` is solely determined by the number of items in the list returned by `GetCommands()`. See Section 15.4 [GetCommands], page 131, for details.

On Windows systems, `Node` usually describes the node of a CHM file. When pressing F1 in the Hollywood IDE on Windows while the cursor is currently over a plugin command, the IDE will automatically jump to this node in the CHM file. On AmigaOS, the node specifies a node inside an AmigaGuide file.

The `HelpText` item should follow the formatting rules of Hollywood's inbuilt commands. Here are some examples of what `HelpText` strings should look like:

```
id=OpenDirectory(id,dir$[,table]) -- open a directory
r=IsDirectory(f$) -- check for file or directory
MakeDirectory(dir$) -- make a new directory
```

Your `HelpText` should first specify the calling convention of your command (similar to what appears in the SYNOPSIS sections of the Hollywood manual) and then a short description of what the command does. The description should be separated from the calling convention by using two minus characters.

INPUTS

none

RESULTS

`list` an array of `struct hwHelpStruct` elements

15.7 GetLibraryCount

NAME

GetLibraryCount – get number of libraries to install (V6.0, optional)

SYNOPSIS

```
int c = GetLibraryCount(void);
```

FUNCTION

This function is optional and must only be implemented if the `HWEXT_LIBRARY_MULTIPLE` extension bit has been set. See Section 11.1 [Extension plugins], page 101, for details. In that case, `GetLibraryCount()` has to return the total number of libraries your plugin wants to install. Hollywood will call your `GetBaseTable()`, `GetCommands()` and `GetConstants()` functions as many times as this function requests so that you can add more than one library into the system. Before calling these functions, Hollywood will first invoke your `SetCurrentLibrary()` function to tell you whose library's commands, constants or base table you should return. See Section 15.9 [SetCurrentLibrary], page 135, for details.

The value to be returned by `GetLibraryCount()` is 1-based so a return value of 1 means there is just one library to install. In that case, of course, it wouldn't be necessary to use the `HWEXT_LIBRARY_MULTIPLE` extension at all.

INPUTS

none

RESULTS

`c`	the total number of libraries to be installed by this plugin

15.8 InitLibrary

NAME

InitLibrary – perform library initialization (V5.0)

SYNOPSIS

```
int error = InitLibrary(lua_State *L);
```

FUNCTION

This function can be used to perform additional library initialization. Certain plugins might need to do some additional things once the `lua_State` has been setup correctly. For example, a plugin might want to setup some metatables or push additional helper tables into the stack. It's impossible to do such things in `InitPlugin()` since the `lua_State` isn't ready yet at `InitPlugin()` call time. That's why this function is available. Many plugins, however, probably don't need to do anything here at all.

`InitLibrary()` has to return an error code or 0 for success.

INPUTS

`L`	pointer to the `lua_State`

RESULTS

`error`	error code or 0 for success

15.9 SetCurrentLibrary

NAME

SetCurrentLibrary – set current library number (V6.0, optional)

SYNOPSIS

```
void SetCurrentLibrary(int n);
```

FUNCTION

This function is optional and must only be implemented if the `HWEXT_LIBRARY_MULTIPLE` extension bit has been set. See Section 11.1 [Extension plugins], page 101, for details. In that case, Hollywood will call `SetCurrentLibrary()` to tell your plugin whose library's commands, constants or base table you should return when Hollywood makes the next call to your `GetBaseTable()`, `GetCommands()` or `GetConstants()` functions. The library numbers passed by Hollywood are 0-based, so a value of 0 means the first library. Hollywood will call this function as many times as you've specified in your `GetLibraryCount()` implementation. See Section 15.7 [GetLibraryCount], page 134, for details.

INPUTS

`n`	number of the current library (zero-based)

16 Requester adapter plugins

16.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_REQUESTERADAPTER` set can replace Hollywood's inbuilt requester handler with their customized version. If you're writing a display adapter using an alternative toolkit, it might be good idea to also use the requester facilities of this toolkit for consistency reasons and an overall more polished appearance.

Please note that requester adapters are not automatically initialized when Hollywood loads the plugin. Instead, you have to manually call `hw_SetRequesterAdapter()` in your `RequirePlugin()` function to activate the requester adapter. The requester adapter will then only be activated if the user calls `@REQUIRE` on your plugin. Otherwise, Hollywood will use its default requester handler. See Section 2.5 [Auto and manual plugin initialization], page 8, for details.

You don't have to implement all functions offered by the requester adapter API. Many functions are optional and only have to be implemented if you explicitly request their use in your call to `hw_SetRequesterAdapter()`. However, it is mandatory that all functions defined by the display adapter API are declared so that Hollywood can import their symbols when it loads the plugin. Functions that are optional and that you don't enable via `hw_SetRequesterAdapter()` can just be dummies then. Here is an overview of all requester adapter APIs that are optional:

`SystemRequest()`
: Only used if activated by setting `HWSRAFLAGS_SYSTEMREQUEST`.

`FileRequest()`
: Only used if activated by setting `HWSRAFLAGS_FILEREQUEST`.

`PathRequest()`
: Only used if activated by setting `HWSRAFLAGS_PATHREQUEST`.

`StringRequest()`
: Only used if activated by setting `HWSRAFLAGS_STRINGREQUEST`.

`ListRequest()`
: Only used if activated by setting `HWSRAFLAGS_LISTREQUEST`.

`FontRequest()`
: Only used if activated by setting `HWSRAFLAGS_FONTREQUEST`.

`ColorRequest()`
: Only used if activated by setting `HWSRAFLAGS_COLORREQUEST`.

See Section 33.5 [hw_SetRequesterAdapter], page 275, for information on how to install your requester adapter.

This plugin type is supported since Hollywood 6.0.

16.2 ColorRequest

NAME

ColorRequest – open a color requester (V6.0, optional)

SYNOPSIS

```
int error = ColorRequest(APTR handle, STRPTR title, ULONG flags,
                int *result, struct hwTagList *tags);
```

FUNCTION

This function must open a color requester, i.e. a dialog box that prompts the user to select a color. The color must be returned to Hollywood by writing an RGB value to the `int` pointer that has been passed as parameter 4. If the user has cancelled the requester, `ColorRequest()` has to write -1 to the `result` pointer.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWCOLORREQTAG_FROMSCRIPT:`
: The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWCOLORREQTAG_DEFCOLOR:`
: If this tag is present, Hollywood wants you to preselect the color that has been passed in the `iData` member of this tag item in the color requester.

Note that Hollywood won't call `FreeRequest()` for this requester type because `ColorRequest()` shouldn't have to allocate any resources.

`ColorRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_COLORREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle` display handle or `NULL` if no display is open

`title` title string for the requester's window

`flags` reserved for future use (currently 0)

`result` `int` pointer for storing the user's selection

`tags` taglist for additional options (see above)

RESULTS

`error` error code or 0 for success

16.3 FileRequest

NAME

FileRequest – open a file requester (V6.0, optional)

SYNOPSIS

```
int error = FileRequest(APTR handle, STRPTR title, ULONG flags,
                STRPTR *result, struct hwTagList *tags);
```

FUNCTION

This function must open a file requester (also known as an open dialog box or file chooser dialog) that prompts the user to select a file for opening or saving. The function must then return a fully qualified path to this file to Hollywood by setting the fourth parameter to a string pointer that your function has allocated. Hollywood will then call `FreeRequest()` on this string when it is done with it. If the user cancels the file requester, you have to write `NULL` to the `result` string pointer.

The `flags` and `tags` parameters are used to control the appearance of the file requester. The following flags are currently defined:

`HWFILEREQFLAGS_MULTISELECT:`

If this flag is set, your requester has to allow the selection of multiple files. In multi-select mode the return string pointer that you write to the `result` parameter has to be a list of fully qualified paths to files. The individual filenames are separated from one another by a single `NULL` terminator byte whereas the complete list is terminated by two `NULL` terminator bytes to signal the list end to Hollywood. This flag cannot be combined with `HWFILEREQFLAGS_SAVEMODE`.

`HWFILEREQFLAGS_SAVEMODE:`

If this flag is set, Hollywood wants your requester to open in save mode, i.e. the user should select a file for saving. In contrast to open file mode, the user might select a file that doesn't exist when in save mode and the requester should also make the user confirm that the file can be overwritten in case he selects an existing file. This flag cannot be combined with `HWFILEREQFLAGS_MULTISELECT`.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWFILEREQTAG_FROMSCRIPT:`

The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWFILEREQTAG_EXTENSIONS:`

If this tag is set, the requester should only show files that use the specified file extension. The `pData` member of this tag item is set to a string that contains a list of file extensions that should be shown. The individual extensions do not contain a dot and are separated by a vertical bar character (`|`), for example `"jpg|jpeg|png|bmp|gif|lbm|ilbm"`.

`HWFILEREQTAG_DEFDRAWER:`
: If this tag is set, Hollywood wants your requester to show the files of this directory when it opens. The directory is passed as a string in the `pData` member of this tag item.

`HWFILEREQTAG_DEFFILE:`
: If this tag is set, Hollywood wants your requester to preselect this file when it opens. The file is passed as a string in the `pData` member of this tag item.

`FileRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_FILEREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle`	display handle or `NULL` if no display is open
`title`	title string for the requester's window
`flags`	flags controlling the requester's appearance (see above)
`result`	`STRPTR` pointer for storing the user's selection
`tags`	taglist for additional options (see above)

RESULTS

`error`	error code or 0 for success

16.4 FontRequest

NAME

FontRequest – open a font requester (V6.0, optional)

SYNOPSIS

```
int error = FontRequest(APTR handle, STRPTR title, ULONG flags, STRPTR *
                result, ULONG *style, struct hwTagList *tags);
```

FUNCTION

This function must open a font requester, i.e. a dialog box that prompts the user to select a font face, size, color, and style. The function must then return a string to Hollywood that contains information about the user's selection. This string must be formatted like this:

```
facename|size|color (e.g. "Arial|72|16711680")
```

The color is specified as an RGB value but in decimal notation. Your `FontRequest()` implementation must set parameter 4 to a string pointer that adheres to the format as described above. The string must be allocated by your function. Hollywood will call `FreeRequest()` on this string when it is done with it so that you can free it. If the user cancels the font requester, you have to write `NULL` to the `result` string pointer.

Additionally, your `FontRequest()` implementation must set the `style` pointer to a combination of the following flags which indicate the style the user has selected for this font:

```
#define HWFONTREQWEIGHT_THIN       0x00000001
```

```
#define HWFONTREQWEIGHT_EXTRALIGHT 0x00000002
#define HWFONTREQWEIGHT_LIGHT      0x00000004
#define HWFONTREQWEIGHT_BOOK       0x00000008
#define HWFONTREQWEIGHT_NORMAL     0x00000010
#define HWFONTREQWEIGHT_MEDIUM     0x00000020
#define HWFONTREQWEIGHT_SEMIBOLD   0x00000040
#define HWFONTREQWEIGHT_BOLD       0x00000080
#define HWFONTREQWEIGHT_EXTRABOLD  0x00000100
#define HWFONTREQWEIGHT_BLACK      0x00000200
#define HWFONTREQWEIGHT_EXTRABLACK 0x00000400
#define HWFONTREQSLANT_ROMAN       0x00000800
#define HWFONTREQSLANT_ITALIC      0x00001000
#define HWFONTREQSLANT_OBLIQUE     0x00002000
#define HWFONTREQSTYLE_UNDERLINED  0x00004000
#define HWFONTREQSTYLE_STRIKEOUT   0x00008000
#define HWFONTREQSTYLE_BOLD        0x00010000
#define HWFONTREQSTYLE_ITALIC      0x00020000
```

The `HWFONTREQWEIGHT_XXX` flags control the font's weight (i.e. how bold the individual characters should appear) and the `HWFONTREQSLANT_XXX` flags contain the font's slant (i.e. how italic the individual characters should appear). You may only set one flag from the `HWFONTREQWEIGHT_XXX` and `HWFONTREQSLANT_XXX` groups.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWFONTREQTAG_FROMSCRIPT:`
: The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWFONTREQTAG_DEFFONT:`
: If this tag is in the list, Hollywood wants you to initialize the requester's initial font to the one specified in this tag. The `pData` member of this tag item is set to a string pointer containing the name of the font.

`HWFONTREQTAG_DEFSIZE:`
: If this tag is in the list, Hollywood wants you to preselect the font size specified in the `iData` member of this tag item when first opening the requester.

`FontRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_FONTREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle` display handle or `NULL` if no display is open

`title` title string for the requester's window

`flags` reserved for future use (currently 0)

`result` `STRPTR` pointer for storing the user's selection (see above)

`style` `ULONG` pointer for storing the font's style (see above)

`tags` taglist for additional options (see above)

RESULTS

`error` error code or 0 for success

16.5 FreeRequest

NAME

FreeRequest – free requester specific data (V6.0)

SYNOPSIS

```
void FreeRequest(int type, STRPTR data);
```

FUNCTION

This function must free the data that has been allocated by one of the requester functions of your plugin. The first parameter tells `FreeRequest()` which function has allocated the data in the second parameter. The following types are currently recognized:

`HWREQTYPE_FILE:`
Data has been allocated by `FileRequest()`.

`HWREQTYPE_PATH:`
Data has been allocated by `PathRequest()`.

`HWREQTYPE_STRING:`
Data has been allocated by `StringRequest()`.

`HWREQTYPE_FONT:`
Data has been allocated by `FontRequest()`.

INPUTS

`type` type of data to free

`data` the actual data item allocated by one of the requester functions

16.6 ListRequest

NAME

ListRequest – open a list requester (V6.0, optional)

SYNOPSIS

```
int error = ListRequest(APTR handle, STRPTR title, STRPTR body, STRPTR
                choices, ULONG flags, int *result, struct hwTagList *tags);
```

FUNCTION

This function must open a list requester that prompts the user to select one item from a list of choices. The list of choices is passed in the fourth parameter as a list of an

unlimited number of strings which are separated by `NULL` characters. The end of this list is marked by two `NULL` characters.

Your implementation has to write the index of the list item chosen by the user to the `result` parameter which is a pointer to an `int`. List items are counted from 0 which marks the first item. If the user has cancelled the requester, your `ListRequest()` implementation has to write -1 to the `result` pointer.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWLISTREQTAG_FROMSCRIPT:`
: The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWLISTREQTAG_ACTIVE:`
: If this tag is provided, Hollywood wants your list requester to preselect the list item at the index specified in this tag item's `iData` member. Item indices are counted from 0 so if `iData` is 0, you'd have to preselect the first list item.

Note that Hollywood won't call `FreeRequest()` for this requester type because `ListRequest()` shouldn't have to allocate any resources.

`ListRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_LISTREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle`	display handle or `NULL` if no display is open
`title`	title string for the requester's window
`body`	message string for the requester's body
`choices`	list of strings separated by `NULL` characters
`flags`	for future use (currently 0)
`result`	`int` pointer for storing the user's selection
`tags`	taglist for additional options (see above)

RESULTS

`error`	error code or 0 for success

16.7 PathRequest

NAME

PathRequest – open a path requester (V6.0, optional)

SYNOPSIS

```
int error = PathRequest(APTR handle, STRPTR title, ULONG flags,
                STRPTR *result, struct hwTagList *tags);
```

FUNCTION

This function must open a path requester (also known as a browse folder dialog box) that prompts the user to select a path. The function must then return this path to Hollywood by setting the fourth parameter to a string pointer that your function has allocated. Hollywood will then call `FreeRequest()` on this string when it is done with it. If the user cancels the path requester, you have to write `NULL` to the `result` string pointer.

The `flags` and `tags` parameters are used to control the appearance of the path requester. The following flags are currently defined:

`HWPATHREQFLAGS_SAVEMODE:`
: If this flag is set, Hollywood wants your requester to open in save mode, i.e. the user should select a path where files can be saved.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWPATHREQTAG_FROMSCRIPT:`
: The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWPATHREQTAG_DEFDRAWER:`
: If this tag is set, Hollywood wants your requester to show this directory when it initially opens. The directory is passed as a string in the `pData` member of this tag item.

`PathRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_PATHREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle`	display handle or `NULL` if no display is open
`title`	title string for the requester's window
`flags`	flags controlling the requester's appearance (see above)
`result`	`STRPTR` pointer for storing the user's selection
`tags`	taglist for additional options (see above)

RESULTS

`error`	error code or 0 for success

16.8 StringRequest

NAME

StringRequest – open a string requester (V6.0, optional)

SYNOPSIS

```
int error = StringRequest(APTR handle, STRPTR title, STRPTR body,
                ULONG flags, STRPTR *result, struct hwTagList *tags);
```

FUNCTION

This function must open a string requester, i.e. a dialog box that prompts the user to enter a string. The function must then return this string to Hollywood by setting parameter 5 to a string pointer that your function has allocated. Hollywood will then call `FreeRequest()` on this string when it is done with it. If the user cancels the string requester, you have to write `NULL` to the `result` string pointer.

The `flags` and `tags` parameters are used to control further parameters of the string requester. The following flags are currently defined:

`HWSTRINGREQTYPE_ALPHANUMERICAL:`
: If this flag is set, only alphabetical and numerical characters should be accepted by the requester.

`HWSTRINGREQTYPE_ALPHABETICAL:`
: If this flag is set, only alphabetical characters should be accepted by the requester.

`HWSTRINGREQTYPE_NUMERICAL:`
: If this flag is set, only numerical characters should be accepted by the requester.

`HWSTRINGREQTYPE_HEXANUMERICAL:`
: If this flag is set, only hexadecimal numerical characters should be accepted by the requester, i.e. the letters A to F and the number 0 to 9.

`HWSTRINGREQFLAGS_PASSWORD:`
: If this flag is set, the requester should open in password mode, i.e. it should not show the letters that are being entered.

Please note that all the `HWSTRINGREQTYPE_XXX` flags are mutually exclusive. Only one from this group will be set.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWSTRINGREQTAG_FROMSCRIPT:`
: The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWSTRINGREQTAG_DEFTEXT:`
: If this tag is in the list, Hollywood wants you to initialize the requester's string widget with the text provided in the string pointer in this tag item's `pData` member.

`HWSTRINGREQTAG_MAXCHARS:`
: If this tag is set, your string requester should limit the text that can be entered to the number of characters provided in the `iData` member of this tag.

`StringRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_STRINGREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle`	display handle or `NULL` if no display is open
`title`	title string for the requester's window
`body`	message string for the requester's body
`flags`	flags controlling the requester's appearance (see above)
`result`	`STRPTR` pointer for storing the user's selection
`tags`	taglist for additional options (see above)

RESULTS

`error`	error code or 0 for success

16.9 SystemRequest

NAME

SystemRequest – open a system requester (V6.0, optional)

SYNOPSIS

```
int error = SystemRequest(APTR handle, STRPTR title, STRPTR body,
                ULONG flags, int *result, struct hwTagList *tags);
```

FUNCTION

This function must open a system requester (also known as a message box) that presents the string passed in the `body` parameter to the user. The user then has to acknowledge the requester by pressing a button. The `flags` parameter specifies which button(s) should be shown and it also tells you whether or not there should be an icon the requester. The following flags are currently defined:

`HWSYSREQTYPE_OK:`
: Requester should contain an "OK" button.

`HWSYSREQTYPE_OKCANCEL:`
: Requester should contain "OK" and "Cancel" buttons.

`HWSYSREQTYPE_YESNO:`
: Requester should contain "Yes" and "No" buttons.

`HWSYSREQTYPE_YESNOCANCEL:`
: Requester should contain "Yes", "No", and "Cancel" buttons.

`HWSYSREQTYPE_CUSTOM:`
: Requester should contain custom buttons. If this flag is set, Hollywood will pass a string in `HWSYSREQTAG_CHOICES` which contains the names for the custom buttons.

`HWSYSREQICON_NONE:`
: There should be no icon in the requester.

`HWSYSREQICON_INFORMATION:`
: There should be an information icon in the requester.

`HWSYSREQICON_ERROR:`
: There should be an error icon in the requester.

`HWSYSREQICON_WARNING:`
: There should be a warning icon in the requester.

`HWSYSREQICON_QUESTION:`
: There should be a question icon in the requester.

Please note that all `HWSYSREQTYPE_XXX` and all `HWSYSREQICON_XXX` flags are mutually exclusive. There will only be one flag from each group set.

Hollywood also passes a taglist to this function. Your implementation has to handle the following tags:

`HWSYSREQTAG_FROMSCRIPT:`
: The `iData` member of this tag item is set to `True` if Hollywood has called you while the script is running. This might be important to know because requesters should not block window refresh so you might want to setup a temporary modal event loop if this tag has been set to `True` to enable your display to stay responsive.

`HWSYSREQTAG_CHOICES:`
: If the `HWSYSREQTYPE_CUSTOM` flag has been set, the `pData` member of this tag item contains a pointer to a string that contains the name(s) of one or more buttons. If there is more than one button, the individual button names will be separated by the vertical bar character (`|`). If this tag is provided, your implementation must setup a custom requester that contains the buttons specified here.

Your `SystemRequest()` implementation has to write the id of the button that has been pressed to the `int` pointer passed as the fifth parameter. The right-most button always has the id 0. If there is only one button, it will also have the id 0. The ids of the other buttons are counted from left to right starting at 1. This arrangement has been chosen so that in case there are two buttons like `"OK|Cancel"` or `"Yes|No"`, the affirmative button's id will correspond to `True` whereas the negative response button's id will correspond to `False`.

Note that Hollywood won't call `FreeRequest()` for this requester type because `SystemRequest()` shouldn't have to allocate any resources.

`SystemRequest()` is an optional API and must only be implemented if `HWSRAFLAGS_SYSTEMREQUEST` has been passed to `hw_SetRequesterAdapter()`. See Section 33.5 [hw_SetRequesterAdapter], page 275, for details.

INPUTS

`handle`	display handle or `NULL` if no display is open
`title`	title string for the requester's window
`body`	message string for the requester's body
`flags`	flags controlling the requester's appearance (see above)
`result`	`int` pointer for storing the user's selection
`tags`	taglist for additional options (see above)

RESULTS

`error`	error code or 0 for success

17 Require hook plugins

17.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_REQUIRE` set can be used to hook your plugin into the `@REQUIRE` preprocessor command, i.e. your plugin will be called whenever the user runs `@REQUIRE` on your plugin. Hooking into the `@REQUIRE` preprocessor command is mandatory for certain plugin types that are not activated automatically when Hollywood loads the plugin. File or display adapter plugins, for example, are only initialized once your plugin explicitly calls `hw_AddLoaderAdapter()` or `hw_SetDisplayAdapter()` on them. This is usually done in the `RequirePlugin()` function of your plugin.

See Section 2.5 [Auto and manual plugin initialization], page 8, for details.

This plugin type is supported since Hollywood 6.0.

17.2 RequirePlugin

NAME

RequirePlugin – require a plugin (V6.0)

SYNOPSIS

```
int error = RequirePlugin(lua_State *L, int version, int revision,
                ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function will be called when the user runs the `@REQUIRE` preprocessor command on the plugin. It is especially useful to initialize plugins that are not automatically set up when Hollywood starts. You will typically initialize these plugin types from your `RequirePlugin()` function then. See Section 2.5 [Auto and manual plugin initialization], page 8, for details.

`RequirePlugin()` can also be used to perform some custom initialization or configuration according to the user's specific needs as it is possible to pass parameters from the `@REQUIRE` preprocessor statement to your `RequirePlugin()` implementation through the `HWRPTAG_USERTAGS` tag item (see below).

Here is a list of tags supported by this function:

`HWRPTAG_USERTAGS:`

Hollywood will set this tag to a `struct hwRequireUserTagList` containing a list of all parameters the user has passed to the `@REQUIRE` preprocessor command. The `struct hwRequireUserTagList` looks like this:

```
struct hwRequireUserTagList
{
    struct hwRequireUserTagList *Succ;
    STRPTR Name;
    STRPTR Str;
    double Val;
};
```

Here is a description each member variable:

`Succ:` Contains a pointer to the next node in the list or `NULL` for the tail node.

`Name:` Name of the parameter.

`Str:` If the user has set the parameter to a string value, it is passed here. If this is `NULL`, then you have to examine the `Val` member below.

`Val:` If the user has set the parameter to a numerical value, it is passed in this structure member.

`HWRPTAG_PLUGINFLAGS:`

This tag is used to return plugin flags back to Hollywood. Hollywood will pass a pointer to a `ULONG` variable here which you can write to if your plugin wants to request certain features. The following flags are currently recognized:

`HWPLUGINFLAGS_HIDEDISPLAYS:`

If you set this flag, Hollywood will hide all displays upon startup. Even the ones that have been explicitly declared as open will be hidden if this tag is set.

INPUTS

`L` pointer to the `lua_State`

`version` plugin version requested in `@REQUIRE` call

`revision` plugin revision requested in `@REQUIRE` call

`flags` currently unused, will be 0

`tags` pointer to a taglist or `NULL` (see above)

RESULTS

`error` error code or 0 for success

18 Sample saver plugins

18.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_SAVESAMPLE` set can register one or more additional output sample formats. The user will then be able to save samples in the output formats supported by the plugin. Plugins have to register new output sample formats by passing the name of a constant that should be used to access the new format. For example, a plugin might choose to register a new output sample format under the constant `#SMPFMT_CUSTOMFORMAT`. Whenever the user calls `SaveSample()` now and passes `#SMPFMT_CUSTOMFORMAT` as the sample format, Hollywood will ask the plugin to save the sample.

18.2 RegisterSampleSaver

NAME

RegisterSampleSaver – register a new sample saver (V5.0)

SYNOPSIS

```
void RegisterSampleSaver(struct SaveSampleReg *reg)
```

FUNCTION

Hollywood will call this function to get information about the sample saver your plugin wants to register. In addition, `RegisterSampleSaver()` has to tell Hollywood whether it wants to register another sample saver. Hollywood will pass a pointer to a `struct SaveSampleReg` to this function. This structure looks like this:

```
struct SaveFormatReg
{
    ULONG CapsMask;     [out]
    ULONG FormatID;     [in/out]
    STRPTR FormatName;  [out]
};

struct SaveSampleReg
{
    struct SaveFormatReg hdr;
};
```

Your implementation has to do the following with the individual structure members:

`CapsMask:`

This must be set to a combination of flags that tell Hollywood about the capabilities of the sample saver that is to be registered. The following flags are currently supported:

`HWSAVESMPCAPS_MORE:`

If you set this flag, Hollywood will call `RegisterSampleSaver()` again so that you can register another saver. If you don't want to register another saver, don't set this flag. (V5.3)

`FormatID:`

This member must be set to a unique 32-bit value that should be assigned to the constant that is registered for accessing this sample saver from Hollywood scripts. Values smaller than 32768 are reserved for internal Hollywood use. You may use values larger than 32768 for your saver but if you want to publish your plugin, you need to contact Airsoft Softwair to obtain a unique value that is still vacant. This won't cost you anything; it's just needed to make sure that plugin sample savers don't use conflicting identifiers. Also, once you have published your sample saver plugin, the `FormatID` you have specified must not be changed or you will break compatibility with applets or executables that have been compiled with previous versions. If you are registering more than one sample saver using `HWSAVESMPCAPS_MORE`, you can look at the `FormatID` member to tell how many times Hollywood has already called `RegisterSampleSaver()` because `FormatID` will contain the identifier of the last sample saver you registered. If `FormatID` is 0, then this is the first call to `RegisterSampleSaver()`. Note that it is not recommended to keep your own counter because Hollywood might call `RegisterSampleSaver()` multiple times, i.e. it might first loop over `RegisterSampleSaver()` to determine how many sample savers there are in total and then it might loop over `RegisterSampleSaver()` again to actually register their names.

`FormatName:`

This must be set to a string that should form the second half of the constant that Hollywood registers for your sample saver. This string you specify here must follow the naming restrictions for Hollywood constants, i.e. only alphabetical characters, numbers and very few special characters like the underscore character are allowed. The `#SMPFMT_` prefix must not be included in the string you pass. Hollywood will add this automatically, i.e. if you pass the string "TESTFORMAT" here, Hollywood will make your sample saver available under the constant `#SMPFMT_TESTFORMAT`.

INPUTS

`reg` pointer to a `struct SaveSampleReg` to be filled out by your implementation

18.3 SaveSample

NAME

SaveSample – save sample to disk (V5.0)

SYNOPSIS

```
int ok = SaveSample(STRPTR filename, struct SaveSampleCtrl *ctrl);
```

FUNCTION

This function must save the sample provided by the pointer in the second parameter to the filename specified in the first parameter. Hollywood passes a pointer to a `struct SaveSampleCtrl` to this function. This structure looks like this:

```
struct SaveSampleCtrl
{
```

```
        APTR Data;        // [in]
        int DataSize;     // [in]
        int Samples;      // [in]
        int Channels;     // [in]
        int Bits;         // [in]
        int Frequency;    // [in]
        ULONG Flags;      // [in]
        ULONG FormatID;   // [in] -- V5.3
    };
```

In this structure Hollywood passes the following information to your `SaveSample()` function:

`Data:` This contains the raw PCM data to save to the file.

`DataSize:`
This contains the size of the `Data` buffer in bytes.

`Samples:` The total number of PCM frames to save. Note that this is specified in PCM frames, not in bytes.

`Channels:`
The number of channels used by the sound data. This will be either 1 (mono) or 2 (stereo).

`Bits:` The number of bits per PCM sample. This will be either 8 or 16.

`Frequency:`
The number of PCM frames that should be played per second.

`Flags:` A combination of the following flags describing additional properties of the sample:

`HWSNDFLAGS_BIGENDIAN`
The PCM samples are stored in big endian format. This flag is only meaningful if the bit resolution is 16.

`HWSNDFLAGS_SIGNEDINT`
The PCM samples are stored as signed integers. This is typically set for 16-bit samples.

`FormatID:`
This member contains the identifier of the sample format the file should be saved in. You only need to look at this member if your plugin supports more than one output sample format. But be careful, you are only allowed to look at this member if the user is running at least Hollywood 5.3. Otherwise, you must not access this member because older versions of Hollywood don't support it. (V5.3)

This function has to return `True` if the sample has been successfully saved or `False` in case of an error.

INPUTS

`filename` path to a destination file

`ctrl` pointer to a `struct SaveSampleCtrl` containing the sample to be saved

RESULTS

`ok` `True` or `False` indicating success or failure

19 Sound plugins

19.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_SOUND` set will be called whenever Hollywood has to load a sample or music. The plugin can check then whether the file is in a format that the plugin recognizes and if it is, it can open the sound file and stream the raw PCM data to Hollywood. This makes it possible to load custom sound file formats with Hollywood.

By default, sound plugins are automatically activated when Hollywood loads them. Starting with Hollywood 6.0 this behaviour can be changed by setting the `HWEXT_SOUND_NOAUTOINIT` extension bit. If this bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you will have to manually call `hw_AddLoaderAdapter()` to activate your plugin. For example, you could call `hw_AddLoaderAdapter()` from your `RequirePlugin()` implementation. In that case, the sound plugin would only be activated if the user called `@REQUIRE` on it. If you do not call `hw_AddLoaderAdapter()` on a plugin that has auto-initialization disabled, it will only be available if the user addresses it directly through the `Loader` tag. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

The SDK distribution comes with an example sound plugin which contains a loader for the AIFF sound format. Feel free to study this example code to learn how sound plugins are written in practice.

19.2 CloseStream

NAME

CloseStream – close sound handle (V5.0)

SYNOPSIS

```
void CloseStream(APTR handle);
```

FUNCTION

This function must close the specified sound stream handle that has been allocated by your plugin's `OpenStream()` function. Hollywood will call `CloseStream()` when it is done with your sound file.

INPUTS

`handle` handle returned by `OpenStream()`

19.3 GetFormatName

NAME

GetFormatName – get human-readable stream format name (V5.0)

SYNOPSIS

```
STRPTR name = GetFormatName(APTR handle);
```

FUNCTION

This function must return a human-readable name for the stream format of the sound file handle passed as parameter 1. This string can be retrieved from Hollywood by querying the `#ATTRFORMAT` constant on `#MUSIC` objects.

INPUTS

`handle` handle returned by `OpenStream()`

RESULTS

`name` stream format name

19.4 OpenStream

NAME

OpenStream – open a sound file (V5.0)

SYNOPSIS

```
APTR handle = OpenStream(STRPTR filename, struct LoadSoundCtrl *ctrl);
```

FUNCTION

This function has to open the specified filename, check if it is in a sound file format that the plugin wants to handle, and, if it is, return a handle to the sound file back to Hollywood. Otherwise it has to return `NULL`. The handle returned by `OpenStream()` is an opaque datatype that only your plugin knows about. Hollywood will simply pass this handle back to your `StreamSamples()` function when it wants to have the raw PCM data of the sound file.

This function also has to provide certain information about the sound file it has just opened. This information has to be written to the `struct LoadSoundCtrl` that is passed in the second parameter. This structure looks like this:

```
struct LoadSoundCtrl
{
    ULONG Samples;      // [out]
    int Channels;       // [out]
    int Bits;           // [out]
    int Frequency;      // [out]
    ULONG Flags;        // [out]
    int SubSong;        // [in]  -- V5.3
    int NumSubSongs;    // [out] -- V5.3
    STRPTR Adapter;     // [in]  -- V6.0
};
```

The following information has to be written to the `struct LoadSoundCtrl` pointer by `OpenStream()`:

`Samples:` The total number of PCM frames in the sound file.

`Channels:`
The number of channels used by the sound file. This must be either 1 (mono) or 2 (stereo).

`Bits:` The number of bits per PCM sample. This must be either 8 or 16.

`Frequency:`
The number of PCM frames that should be played per second. Usually 44100 or 48000.

`Flags:` A combination of the following flags describing additional properties of the stream:

`HWSNDFLAGS_BIGENDIAN`
The PCM samples are stored in big endian format. This flag is only meaningful if the bit resolution is 16. This flag is unsupported on Windows, AROS x86, and Android.

`HWSNDFLAGS_SIGNEDINT`
The PCM samples are stored as signed integers. This flag must always be set for 16-bit samples. For 8-bit samples this flag is unsupported on Windows, Linux and Android, i.e. 8-bit samples must always be unsigned on these three platforms. On all other platforms this flag must be set for 8-bit samples and the samples must be signed.

`HWSNDFLAGS_CANSEEK`
Plugin supports seeking directly to PCM frames using `SeekStream()`. Note that even if this flag isn't set, your plugin still needs to be able to rewind the stream, i.e. seek it back to the very beginning when 0 is passed to `SeekStream()`. See Section 19.5 [SeekStream], page 158, for details.

`SubSong:` This member will be set by Hollywood to the sub-song within the sound file that Hollywood wants your plugin to open. Only useful for old tracker modules. (V5.3)

`NumSubSongs:`
This must be set by your plugin to the total number of sub-songs in the sound file. This is typically set to 1 because most sound file formats do not support sub-songs. This feature is only here to allow support for old tracker module formats that can have various sub-songs in the same file. (V5.3)

`Adapter:` Starting with Hollywood 6.0 users can specify the file adapter that should be used to open certain files. If this member is non-`NULL`, Hollywood wants your plugin to use the file adapter specified in `Adapter` to open the sound file. This means that you have to use `hw_FOpenExt()` instead of `hw_FOpen()` to open the sound file. Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions. See Section 25.16 [hw_FOpenExt], page 202, for details. (V6.0)

Please note that you should not use ANSI C functions like `fopen()` to open the file that is passed to this function because the filename that is passed to this function can also be a specially formatted filename specification that Hollywood uses to load files that have been linked to applets or executables. In order to be able to load these files correctly, you have to use special IO functions provided by Hollywood. See Section 2.12 [File IO information], page 15, for details.

INPUTS

`filename` filename to open

`ctrl` pointer to a `struct LoadSoundCtrl` for storing information about the sound file

RESULTS

`handle` a handle that identifies this sound file or `NULL` if plugin doesn't want to handle this sound file

19.5 SeekStream

NAME

SeekStream – seek sound stream to new position (V5.0)

SYNOPSIS

```
void SeekStream(APTR handle, ULONG pos);
```

FUNCTION

This function must seek the specified sound stream handle to the position passed as parameter 2. The new position is specified in PCM frames. You only need to implement direct PCM frame seeking if you have set the flag `HWSNDFLAGS_CANSEEK` in `OpenStream()`. However, seeking back to position 0 has to be implemented by all plugins - no matter whether `HWSNDFLAGS_CANSEEK` has been set or not. If `SeekStream()` is called with 0 as the new position, you need to rewind the stream so that the next call to `StreamSamples()` returns PCM frames right from the start of the stream.

Warning: Due to a bug in Hollywood 5.x this function is called with non-zero positions even if `HWSNDFLAGS_CANSEEK` has not been set. This has been fixed in Hollywood 6.0. In Hollywood 6.0 and up `SeekStream()` is only ever called with a zero position if `HWSNDFLAGS_CANSEEK` hasn't been set.

This function must be implemented in a thread-safe way.

INPUTS

`handle` handle returned by `OpenStream()`

`pos` new stream position in PCM frames

19.6 StreamSamples

NAME

StreamSamples – get raw PCM frames from sound stream (V5.0)

SYNOPSIS

```
int error = StreamSamples(APTR handle, struct StreamSamplesCtrl *ctrl);
```

FUNCTION

This function has to read the number of raw PCM frames requested from the specified sound stream and copy them to the memory buffer that is passed to this function.

Hollywood will pass a pointer to a `struct StreamSamplesCtrl` to this function. This structure looks like this:

```
struct StreamSamplesCtrl
{
    APTR Buffer;    // [in]
    int Request;    // [in]
    int Written;    // [out]
    int Done;       // [out]
};
```

Here is the meaning of the individual members:

`Buffer:` This is a pointer to a memory buffer. You have to copy the PCM frames to this buffer.

`Request:` Contains the number of PCM frames that Hollywood wants you to copy to the memory buffer. Note that this value is specified in PCM frames, not in bytes. So if the request is 1024 and your PCM samples are formatted as 16-bit wide stereo frames, you would have to copy 4096 bytes to the memory buffer.

`Written:` This must be set by your implementation to the number of PCM frames that has actually been written to `Buffer`. Once again, the value is specified in PCM frames, not in bytes.

`Done:` Set this to `True` if the stream end has been reached. Otherwise set it to `False`.

`StreamSamples()` must return an error code or 0 for success.

This function must be implemented in a thread-safe way. Hollywood will usually call this function from a helper thread so make sure that your implementation is thread-safe.

INPUTS

`handle` handle returned by `OpenStream()`

`ctrl` pointer to a `struct StreamSamplesCtrl` containing Hollywood's request

RESULTS

`error` error code or 0 for success

20 Timer adapter plugins

20.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_TIMERADAPTER` set can replace Hollywood's inbuilt timer handler with their customized version. Timers are used to control Hollywood functions like `SetInterval()` or `SetTimeout()`, for instance. If you're writing a display adapter using an alternative toolkit, it might be good idea to also use the timer facilities of this toolkit to get a better performance.

Please note that timer adapters are not automatically initialized when Hollywood loads the plugin. Instead, you have to manually call `hw_SetTimerAdapter()` in your `RequirePlugin()` function to activate the timer adapter. The timer adapter will then only be activated if the user calls `@REQUIRE` on your plugin. Otherwise, Hollywood will use its default timer handler. See Section 2.5 [Auto and manual plugin initialization], page 8, for details.

See Section 34.41 [hw_SetTimerAdapter], page 326, for information on how to install your timer adapter.

This plugin type is supported since Hollywood 6.0.

20.2 FreeTimer

NAME

FreeTimer – free a timer (V6.0)

SYNOPSIS

```
void FreeTimer(APTR handle);
```

FUNCTION

This function must free the specified timer that has been allocated by `RegisterTimer()`. See Section 20.3 [RegisterTimer], page 161, for details.

INPUTS

`handle` timer handle allocated by `RegisterTimer()`

20.3 RegisterTimer

NAME

RegisterTimer – register a new timer (V6.0)

SYNOPSIS

```
APTR handle = RegisterTimer(lua_State *L, int ms, int oneshot);
```

FUNCTION

This function has to register a new timer that triggers once the time specified in parameter 2 has elapsed. If the third parameter is set to `True`, Hollywood wants to have a one-shot timer. Otherwise, the timer is expected to regard the time in parameter 2 as an interval time and fire continuously in these intervals.

To make Hollywood run the user callbacks associated with timer events you need to call `hw_RunTimerCallback()` when a timer has fired. See Section 34.38 [hw_RunTimerCallback], page 324, for details.

Hollywood will call `FreeTimer()` to free timers registered by this function.

INPUTS

`L`	pointer to the `lua_State`
`ms`	time in milliseconds
`oneshot`	`True` if this timer should fire only once

RESULTS

`handle`	handle to the timer or `NULL` in case of an error

21 Vectorgraphics plugins

21.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_VECTOR` set can provide support for drawing vector-based paths with Hollywood. Hollywood comes with a vectorgraphics library which can use an internal vectorgraphics backend or an external one provided by a plugin. The inbuilt vectorgraphics backend is platform-independent but has some restrictions when it comes to filling complex shapes. That's why it might be better to write a vectorgraphics plugin for certain advanced tasks. Vectorgraphics plugins might want to use the vector-based drawing functionality offered by the host operating systems, e.g. Quartz on Mac OS X or GDI+ on Windows, or they could use platform-independent vector drawing backends like Cairo.

Starting with Hollywood 6.0 library plugins support the `HWEXT_VECTOR_EXACTFIT` extension. If this extension is set, `GetPathExtents()` must take the transformation matrix it is passed into account when computing the path's extents. If `HWEXT_VECTOR_EXACTFIT` is not set, Hollywood will compute the extents of the transformed path but this is not recommended since it is your plugin that knows best about the real extents. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

21.2 CloseFont

NAME

CloseFont – close FreeType2 font (V5.0)

SYNOPSIS

```
void CloseFont(FT_Face face);
```

FUNCTION

This function is used to free FreeType2 fonts that have been allocated by `OpenFont()`. It's only necessary to implement this function on WarpOS. On all other systems, this function can be a dummy. See Section 21.8 [OpenFont], page 172, for details.

INPUTS

`face` `FT_Face` font handle allocated by `OpenFont()`

21.3 CreateVectorFont

NAME

CreateVectorFont – create a vector font (V5.0)

SYNOPSIS

```
APTR handle = CreateVectorFont(FT_Face face);
```

FUNCTION

This function has to create a vector font from the `FT_Face` passed to this function. `FT_Face` is a data type for describing font handles allocated by the FreeType2 library which Hollywood uses to render text. `CreateVectorFont()` has to return a handle which is

then passed in the `CCMD_TEXT` path command whenever Hollywood wants you to draw some vector text. See Section 21.4 [DrawPath], page 164, for details.

Hollywood will call `FreeVectorFont()` to free handles allocated by this function.

INPUTS

`face` pointer to an `FT_Face` describing a FreeType2 font

RESULTS

`handle` handle to a vector font to be used with `DrawPath()` or `NULL` in case of an error

21.4 DrawPath

NAME

DrawPath – draw a vector path (V5.0)

SYNOPSIS

```
int ok = DrawPath(struct DrawPathCtrl *ctrl);
```

FUNCTION

This function has to draw a vector path to a bitmap. The path itself and all other parameters that you need to know are passed in the `struct DrawPathCtrl` pointer that this function receives. `struct DrawPathCtrl` looks like this:

```
struct DrawPathCtrl
{
    void *Path;                 // [in]
    struct PathStyle *Style;    // [in]
    int Fill;                   // [in]
    int Thickness;              // [in]
    ULONG Color;                // [in]
    UBYTE *Buf;                 // [in]
    int LineWidth;              // [in]
    int Width;                  // [in]
    int Height;                 // [in]
    int Pad;                    // [unused]
    double TX;                  // [in]
    double TY;                  // [in]
    double MinX;                // [in]
    double MinY;                // [in]
    struct hwMatrix2D *Matrix; // [in]
};
```

Hollywood will pass the following data in this structure:

`Path:` A buffer containing the actual path data. This buffer contains the individual commands and their parameters in a number of disparate items. The command is stored in an `int` and is always first. The number of parameters

that follow the command `int` and their sizes depend on the actual command that has been passed. The following commands are currently recognized:

`CCMD_STACKTOP:`
: This is the terminator command. This will always be at the end of the path buffer. You must break out of your command loop when encountering `CCMD_STACKTOP`.

`CCMD_NEWSUBPATH:`
: This has to start a new sub-path and set the current point to undefined.

`CCMD_CLOSEPATH:`
: This has to close the current path by drawing a line from the current point to the first point in the sub-path. After that the current point has to be set to this start and end point of the sub-path.

`CCMD_MOVETO:`
: This command has to begin a new sub-path. The sub-path's current point must be set to the specified position. `CCMD_MOVETO` receives the following three arguments:

  `rel (int)` 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 (double)`
  : The x coordinate of the new position.

  `y (double)`
  : The y coordinate of the new position.

`CCMD_LINETO:`
: This command has to draw a line from the current point to the specified position. Additionally, it must change the current point to the line's end point when it is done. `CCMD_LINETO` receives the following three arguments:

  `rel (int)` 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 (double)`
  : The x coordinate of the new position.

  `y (double)`
  : The y coordinate of the new position.

  If there is no current point, `CCMD_LINETO` must behave as if it was `CCMD_MOVETO`, i.e. it must simply set the current point to the specified vertex.

CCMD_CURVETO:

This command has to draw 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 must change the current point to the curve's end point when it is done. CCMD_CURVETO receives the following seven arguments:

rel (int) 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 (double)
The x coordinate of the first control point.

y1 (double)
The y coordinate of the first control point.

x2 (double)
The x coordinate of the first control point.

y2 (double)
The y coordinate of the first control point.

x3 (double)
The x coordinate of the end point.

y3 (double)
The y coordinate of the end point.

If there is no current point, CCMD_CURVETO must use the point passed in (x1,y1) as the current point.

CCMD_ARC:

This command has to draw an elliptical arc. CCMD_ARC must open a new subpath for the new arc only in case there is no currently active subpath. If there is already a subpath, CCMD_ARC must connect its starting vertex with the current vertex of the subpath. CCMD_ARC also must not close the subpath when it has finished adding its vertices. CCMD_ARC must not connect the start and end angles of the arc with its center point automatically. The user has to explicitly request this by issuing separate CCMD_MOVETO and CCMD_LINETO commands before and after CCMD_ARC. CCMD_ARC receives the following arguments:

xc (double)
The x center point of the arc.

yc (double)
The y center point of the arc.

ra (double)
Arc's radius on the x axis.

`rb (double)`
: Arc's radius on the y axis.

`start (double)`
: Start angle in degrees.

`end (double)`
: End angle in degrees.

`clockwise (int)`
: Whether or not the angles should be connected in clockwise direction.

When `CCMD_ARC` is done, it needs to set the current point to the position of the end angle.

`CCMD_BOX:`
: This command has to draw a rectangle. `CCMD_BOX` must 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. `CCMD_BOX` receives the following arguments:

  `x (double)`
  : X position of the rectangle.

  `y (double)`
  : Y position of the rectangle.

  `width (double)`
  : Rectangle width.

  `height (double)`
  : Rectangle height.

  `rnd1 (int)`
  : Integer value in the range of 0 to 100 specifying the degree of rounding for the first corner of the rectangle. 0 for no rounding.

  `rnd2 (int)`
  : Integer value in the range of 0 to 100 specifying the degree of rounding for the second corner of the rectangle. 0 for no rounding.

  `rnd3 (int)`
  : Integer value in the range of 0 to 100 specifying the degree of rounding for the third corner of the rectangle. 0 for no rounding.

  `rnd4 (int)`
  : Integer value in the range of 0 to 100 specifying the degree of rounding for the fourth corner of the rectangle. 0 for no rounding.

CCMD_TEXT:

This command has to draw vector text relative to the current point. The individual characters should be added as closed subpaths. CCMD_TEXT receives the following arguments:

ptr (APTR)

This is set to a pointer to a vector font created by CreateVectorFont().

size (int)

Desired font size.

text (varies)

The text to draw is passed directly after the integer specifying the font size. It is a null-terminated string encoded in the UTF-8 format. To read the next command following the string data, you need to pad the pointer address after the terminating NULL to a multiple of 4, i.e. a long-aligned address. Commands are always long-aligned so be sure to pad to a long-aligned address after the string end.

When CCMD_TEXT is done, it needs to set the current point to where the next character would be displayed.

Style: This will be set to a pointer to a struct PathStyle containing information about the line style that shall be used when drawing this path. struct PathStyle looks like this:

```
struct PathStyle
{
    int LineJoin;
    int LineCap;
    int FillRule;
    int AntiAlias;
    double DashOffset;
    double *Dashes;
    int NumDashes;
};
```

Here's an explanation of the individual member's function:

LineJoin:

Contains the desired line join style for the path. This can be one of the following constants:

HWLINEJOIN_MITER:

Join lines using a sharp corner.

HWLINEJOIN_ROUND:

Join lines using round edges.

HWLINEJOIN_BEVEL:

Join lines using cut-off edges.

`LineCap:` Determines how line endings should be drawn. This can be one of the following constants:

`HWLINECAP_BUTT:`
Line should stop exactly at the end point without any further decoration.

`HWLINECAP_ROUND:`
Line ending should be round.

`HWLINECAP_SQUARE:`
Line ending should be squared.

`FillRule:`
Determines the fill rule for overlapping sections of the path. This can be one of the following constants:

`HWFILLRULE_WINDING:`
Fill all overlapping paths only if they are not winding.

`HWFILLRULE_EVENODD:`
Fill overlapping paths if the total number of intersections is odd.

`AntiAlias:`
This indicates whether Hollywood wants you to draw anti-aliased shapes or monochrome shapes. `True` means anti-aliasing should be used.

`DashOffset:`
If `NumDashes` is greater than zero, this specifies the offset at which dashing should start.

`Dashes:` If `NumDashes` is greater than zero, this is set to a `double` array which contains `NumDashes` entries, specifying alternate on and off sections that define the dash style.

`NumDashes:`
If this is greater than zero, Hollywood wants you to draw dashed lines. The dashing pattern is specified in `DashOffset` and `Dashes` (see above).

`Fill:` This member specifies whether Hollywood wants you to draw filled shapes or just the stroke outlines. If this is `True`, you have to fill all shapes.

`Thickness:`
This member sets the line thickness.

`Color:` This is currently unused since `DrawPath()` doesn't draw into color channels at all. Ignore this.

`Buf:` This contains a pointer to an 8-bit bitmap which you have to draw to. See `LineWidth` to find out about the bitmap's alignment.

`LineWidth:`
This contains the bytes per row of the bitmap passed in `Buf`.

`Width:` Contains the width of the bitmap in pixels. This can be less than `LineWidth`.

`Height:` Contains the height of the bitmap in pixels.

`TX:` If this does not equal 0, Hollywood wants you to translate every pixel that you draw by this many pixels on the x-axis. Note that the translation must be done after applying the transformation matrix passed in `Matrix`.

`TY:` If this does not equal 0, Hollywood wants you to translate every pixel that you draw by this many pixels on the y-axis. Note that the translation must be done after applying the transformation matrix passed in `Matrix`.

`MinX:` You have to translate the path by this many pixels on the x-axis before drawing it. Note that this value is inverted. A positive value indicates translation to the left whereas a negative value requires you to translate the shape to the right.

`MinY:` You have to translate the path by this many pixels on the y-axis before drawing it. Note that this value is inverted. A positive value indicates upwards translation whereas a negative value requires you to translate the shape in downward direction.

`Matrix:` Contains a 2D transformation matrix that should be applied to this path. If there is no transformation, you'll get a pointer to an identity matrix consisting of (1,0,0,1).

INPUTS

`ctrl` pointer to a `struct DrawPathCtrl`

RESULTS

`ok` `True` for success, `False` otherwise

21.5 FreeVectorFont

NAME
FreeVectorFont – free a vector font (V5.0)

SYNOPSIS
```
void FreeVectorFont(APTR handle);
```

FUNCTION
This function is used to free vector fonts that have been allocated by `CreateVectorFont()`.

INPUTS

`handle` vector font handle allocated by `CreateVectorFont()`

21.6 GetCurrentPoint

NAME

GetCurrentPoint – get path's current point (V5.0)

SYNOPSIS

```
void GetCurrentPoint(void *path, struct PathStyle *style, double *curx,
        double *cury);
```

FUNCTION

This function has to return the current point of the specified path. Your implementation has to write the x and y position of the current point to the `curx` and `cury` pointers passed as parameters 3 and 4. See Section 21.4 [DrawPath], page 164, for a detailed description on how the path buffer is formatted.

INPUTS

`path` pointer to a disparate array containing the path; See Section 21.4 [DrawPath], page 164, for details.

`style` pointer to a `struct PathStyle` containing the path's style; See Section 21.4 [DrawPath], page 164, for details.

`curx` pointer to a `double` that receives the path's current x point

`cury` pointer to a `double` that receives the path's current y point

21.7 GetPathExtents

NAME

GetPathExtents – compute path's extents (V5.0)

SYNOPSIS

```
void GetPathExtents(struct PathExtentsCtrl *ctrl);
```

FUNCTION

This function has to compute the extents of the specified path if drawn using the specified style. Hollywood needs this information to determine the size of the bitmap it allocates and passes to `DrawPath()`. Hollywood passes a `struct PathExtentsCtrl` pointer which contains all information to this function. `struct PathExtentsCtrl` looks like this:

```
struct PathExtentsCtrl
{
    void *Path;                 // [in]
    struct PathStyle *Style;    // [in]
    int Fill;                   // [in]
    int Thickness;              // [in]
    double TX;                  // [in]
    double TY;                  // [in]
    double X1;                  // [out]
    double Y1;                  // [out]
    double X2;                  // [out]
```

```
    double Y2;                  // [out]
    struct hwMatrix2D *Matrix;  // [in]  -- V6.0
};
```

Here's a description of each structure member's function:

`Path:` The path's whose extents are to be calculated. This is a pointer to a disparate array consisting of a variable number of path commands and their arguments. See Section 21.4 [DrawPath], page 164, for details.

`Style:` This will be set to a pointer to a `struct PathStyle` containing information about the line style that shall be used when drawing this path. See Section 21.4 [DrawPath], page 164, for details.

`Fill:` This member specifies whether Hollywood wants the extents of filled shapes or just the stroke outlines. If this is `True`, you have to determine the extents of filled shapes.

`Thickness:` This member contains the line thickness for the path.

`TX:` If this does not equal 0, Hollywood wants you to translate the path by this many pixels on the x-axis before determining the extents.

`TY:` If this does not equal 0, Hollywood wants you to translate the path by this many pixels on the y-axis before determining the extents.

`X1:` Your implementation must set this member to the start x position of the path.

`Y1:` Your implementation must set this member to the start y position of the path.

`X2:` Your implementation must set this member to the end x position of the path.

`Y2:` Your implementation must set this member to the end y position of the path.

`Matrix:` This member must only be handled if the `HWEXT_VECTOR_EXACTFIT` extension bit has been set. In that case, this member contains a 2D transformation matrix that should be applied to this path before computing the path extents. If there is no transformation, you'll get a pointer to an identity matrix consisting of (1,0,0,1). (V6.0)

INPUTS

`ctrl` pointer to a `struct PathExtentsCtrl` which contains information about the path and receives details about its extents

21.8 OpenFont

NAME

OpenFont – create FreeType2 font from memory buffer (V5.0)

SYNOPSIS

```
FT_Face face = OpenFont(UBYTE *data, int datalen);
```

FUNCTION

This function has to create an `FT_Face` handle from the raw font data passed to it. You only need to implement this function if you build plugins for WarpOS. On all other systems, your `OpenFont()` implementation can just return `NULL` because it is never called.

The reason why this function is needed on WarpOS is that the WarpOS version of Hollywood contains two builds of FreeType2: one for 68020 and one for the PowerPC architecture. Plugins will always have to work with the PPC native version to avoid context switches. Hollywood itself, however, uses the 68020 build of FreeType2 because it's faster in most cases because Hollywood doesn't have to do any context switches. Using the `FT_Face` handle allocated by the 68020 version of FreeType2 is not going to work with the PowerPC version of FreeType2, though, due to different structure alignments, etc. That is why you need to use the PowerPC version of FreeType2 on WarpOS to create a compatible `FT_Face` handle which is then passed to `CreateVectorFont()`. On all other systems `OpenFont()` is not necessary because there's only a single-architecture FreeType2 in Hollywood.

Basically, all this function has to do on WarpOS is the following:

```
HW_EXPORT FT_Face OpenFont(UBYTE *data, int datalen)
{
    FT_Face face;

    if(hwcl->FT2Base->FT_New_Memory_Face(freetype_library, data,
           datalen, 0, &face)) return NULL;

    return face;
}

HW_EXPORT void CloseFont(FT_Face face)
{
    hwcl->FT2Base->FT_Done_Face(face);
}
```

Hollywood will call `CloseFont()` to free handles allocated by this function.

INPUTS

`data` pointer to raw font data (usually pointer to TrueType font data)

`datalen` size of the `data` buffer in bytes

RESULTS

`face` handle to an `FT_Face` allocated by the WarpOS version of FreeType2 or `NULL` in case of an error

21.9 TranslatePath

NAME

TranslatePath – translate a path (V5.0)

SYNOPSIS

```
void TranslatePath(struct TranslatePathCtrl *ctrl);
```

FUNCTION

This function has to translate the specified path by the specified offsets. `TranslatePath()` receives a pointer to a `struct TranslatePathCtrl` which looks like this:

```
struct TranslatePathCtrl
{
    void *Path;  // [in/out]
    int Pad;     // [unused]
    double TX;   // [in]
    double TY;   // [in]
};
```

Here's a description of the individual structure members:

`Path:` The path which has to be translated. This is a pointer to a disparate array consisting of a variable number of path commands and their arguments. See Section 21.4 [DrawPath], page 164, for details. Your implementation of `TranslatePath()` has to retrieve all coordinates from this buffer and update them with the translated coordinates.

`TX:` Amount of pixels this path should be translated on the x-axis. This can also be negative.

`TY:` Amount of pixels this path should be translated on the y-axis. This can also be negative.

INPUTS

`ctrl` pointer to a `struct TranslatePathCtrl` which contains information about the path and receives the translated path

22 Video plugins

22.1 Overview

Plugins that have the capability flag `HWPLUG_CAPS_VIDEO` set will be called whenever Hollywood has to open a video file. The plugin can check then whether the file is in a format that the plugin recognizes and if it is, it can open the video file and stream the raw pixel and audio data to Hollywood. This makes it possible to load custom video file formats with Hollywood.

By default, video plugins are automatically activated when Hollywood loads them. Starting with Hollywood 6.0 this behaviour can be changed by setting the `HWEXT_VIDEO_NOAUTOINIT` extension bit. If this bit is set, Hollywood will not automatically activate your plugin at load time. Instead, you will have to manually call `hw_AddLoaderAdapter()` to activate your plugin. For example, you could call `hw_AddLoaderAdapter()` from your `RequirePlugin()` implementation. In that case, the video plugin would only be activated if the user called `@REQUIRE` on it. If you do not call `hw_AddLoaderAdapter()` on a plugin that has auto-initialization disabled, it will only be available if the user addresses it directly through the `Loader` tag. See Section 11.1 [Extension plugins], page 101, to learn how to use plugin extension bits.

22.2 CloseVideo

NAME

CloseVideo – close video handle (V5.0)

SYNOPSIS

```
void CloseVideo(APTR handle);
```

FUNCTION

This function must close the specified video stream handle that has been allocated by your plugin's `OpenVideo()` function. Hollywood will call `CloseVideo()` when it is done with your video file.

INPUTS

`handle` handle returned by `OpenVideo()`

22.3 DecodeAudioFrame

NAME

DecodeAudioFrame – get audio frame from packet (V5.0)

SYNOPSIS

```
int status = DecodeAudioFrame(APTR handle, APTR packet,
                struct DecodeAudioFrameCtrl *ctrl);
```

FUNCTION

This function must decode the specified audio packet into PCM audio frames. Hollywood will pass a pointer to a `struct DecodeAudioFrameCtrl` structure to this function. This structure looks like the following:

```
struct DecodeAudioFrameCtrl
{
    WORD *Buffer;     // [in]
    int BufferSize;   // [in]
    int Written;      // [out]
    int Done;         // [out]
};
```

Your `DecodeAudioFrame()` implementation has to write to the following members of this structure:

`Buffer:` This contains a pointer to a memory buffer allocated for you by Hollywood. You have to write the individual PCM frames to this buffer. Please note that audio must always be written as 16-bit PCM frames in native endian byte order. In case the video uses stereo sound, the PCM samples must be stored in interleaved order.

`BufferSize:` Contains the size of the memory buffer in bytes. This should always be enough to hold one second of audio PCM data but be sure to check against this value when writing PCM frames to the buffer allocated by Hollywood or you might trash innocent memory. If there's not enough space, simply set `Done` to `False` and Hollywood will call you again.

`Written:` This must be set to the number of bytes that you have copied to the memory buffer. Please note that this is specified in bytes, not in PCM frames or samples.

`Done:` This must be set to `False` if there is more audio data to decode in this packet. In that case, Hollywood will call you again. If you've decoded all audio data that is in this packet, set this member to `True`. Hollywood will then free the packet using `FreePacket()`.

This function must return a status code: 0 indicates success, any other value indicates failure.

This function must be implemented in a thread-safe way.

INPUTS

`handle` handle returned by `OpenVideo()`

`packet` pointer to a packet allocated by `NextPacket()`

`ctrl` pointer to a `struct DecodeAudioFrameCtrl` to be filled out by the function

RESULTS

`status` a status code (see above)

22.4 DecodeVideoFrame

NAME

DecodeVideoFrame – get video frame from packet (V5.0)

SYNOPSIS

```
int more = DecodeVideoFrame(APTR handle, APTR packet,
               struct DecodeVideoFrameCtrl *ctrl);
```

FUNCTION

This function must decode the specified packet into a video frame. For most video formats several packets need to be decoded in order to get a single video frame. If your video decoder needs more packets in order to finish decoding the video frame, then you have to return `True` here. This indicates that `DecodeVideoFrame()` wants to be fed more packets before handing out a decoded frame.

Hollywood will pass a pointer to a `struct DecodeVideoFrameCtrl` structure to this function. This structure looks like the following:

```
struct DecodeVideoFrameCtrl
{
    UBYTE **Buffer;     // [in/out]
    int *BufferWidth;   // [in/out]
    int Delay;          // [out]
    ULONG Offset;       // [out]
    double PTS;         // [out]
    double DTS;         // [out]
};
```

If you return `True` to indicate that you are not able to decode a full frame just yet, you don't have to write anything to the `struct DecodeVideoFrameCtrl` pointer passed to you by Hollywood. Otherwise you have to set the following members:

`Buffer:` This is already initialized to an array of three `UBYTE*` pointers. You need to set these pointers to the chunks of memory containing the actual pixel data of the decoded frame. If you have set the `PixFmt` member in `OpenVideo()` to `HWVIDPIXFMT_ARGB32`, you only need to set the first pointer because ARGB32 frames are stored as chunky pixels. If you use `HWVIDPIXFMT_YUV420P`, though, you need to set all three pointers to point to the individual YUV planes, i.e.

```
ctrl->Buffer[0] = y_plane;
ctrl->Buffer[1] = u_plane;
ctrl->Buffer[2] = v_plane;
```

Make sure that you do not modify the base `Buffer` pointer! You must only write to the individual three pointers already allocated for you by Hollywood.

`BufferWidth:`
This is already initialized to an array of three integers. You need to store the byte length of one row of pixel data for each pixel map you pass. If you have set the `PixFmt` member in `OpenVideo()` to `HWVIDPIXFMT_ARGB32`, you only need to set the first integer to the byte width of a single row of ARGB32 pixel data because ARGB32 frames are stored as chunky pixels. If you use `HWVIDPIXFMT_YUV420P`, though, you need to set all three integers to point to the byte width of a single row in each of the three individual YUV planes, i.e.

```
ctrl->BufferWidth[0] = y_plane_bytewidth;
```

```
ctrl->BufferWidth[1] = u_plane_bytewidth;
ctrl->BufferWidth[2] = v_plane_bytewidth;
```

Make sure that you do not modify the base `BufferWidth` pointer! You must only write to the individual integer array items already allocated for you by Hollywood.

`Delay:` This member can be used to specify an additional delay for the video frame. The delay you specify here is multiplied by the `FrameTime` specified in `OpenVideo()` divided by two. So to delay the current frame for one `FrameTime` unit, you'd have to set this member to 2. This should normally be set to 0.

`Offset:` This must be set to the current offset in the video stream in bytes. This is only required if `SeekMode` in `OpenVideo()` has been set to `HWVIDSEEKMODE_BYTE`. If that is not the case, you can set this member to 0.

`PTS:` This must be set to the video frame's presentation time stamp, i.e. the time when this frame should be presented to the viewer, relative to the beginning of the video stream. You have to specify this time stamp as a floating point number in seconds, i.e. a presentation time stamp of 10.2 means that the frame is to be shown 10.2 seconds after the start of the video stream. The value you pass in `PTS` must be a multiple of the `FrameTime` fraction specified in `OpenVideo()`.

`DTS:` This should normally be set to -1. You only need to set this to a time stamp if the frame's presentation order is different from the encoding order. In that case you need to set this member to the decoding time stamp. As above, `PTS` also needs to be specified as a floating point number in seconds.

This function must be implemented in a thread-safe way.

INPUTS

`handle` handle returned by `OpenVideo()`

`packet` pointer to a packet allocated by `NextPacket()`

`ctrl` pointer to a `struct DecodeVideoFrameCtrl` to be filled out by the function

RESULTS

`more` `True` if function needs more packets to decode a frame, `False` if it has successfully decoded a frame

22.5 FlushAudio

NAME

FlushAudio – flush audio decoder (V5.0)

SYNOPSIS

```
void FlushAudio(APTR handle);
```

FUNCTION

This function must flush the audio decoder, i.e. it must reset itself to a clean state so that `DecodeAudioFrame()` can be fed a fresh packet without the risk of confusing the decoder because it was expecting a different packet. This function is called by Hollywood before seeking the audio stream.

Depending on how flexible your decoder is, this function may not need to do anything at all.

This function must be implemented in a thread-safe way.

INPUTS

`handle` handle returned by `OpenVideo()`

22.6 FlushVideo

NAME

FlushVideo – flush video decoder (V5.0)

SYNOPSIS

```
void FlushVideo(APTR handle);
```

FUNCTION

This function must flush the video decoder, i.e. it must reset itself to a clean state so that `DecodeVideoFrame()` can be fed a fresh packet without the risk of confusing the decoder because it was expecting a different packet. This function is called by Hollywood before seeking the video stream.

Depending on how flexible your decoder is, this function may not need to do anything at all.

This function must be implemented in a thread-safe way.

INPUTS

`handle` handle returned by `OpenVideo()`

22.7 FreePacket

NAME

FreePacket – free stream packet (V5.0)

SYNOPSIS

```
void FreePacket(APTR packet);
```

FUNCTION

This function must free the specified stream packet that has been allocated by your plugin's `NextPacket()` function. Hollywood will call `FreePacket()` when it is done with this packet.

This function must be implemented in a thread-safe way.

INPUTS

`packet` packet allocated by `NextPacket()`

22.8 GetVideoFormat

NAME
GetVideoFormat – get human-readable stream format name (V5.0)

SYNOPSIS
```
STRPTR name = GetVideoFormat(APTR handle);
```

FUNCTION
This function must return a human-readable name for the stream format of the video file handle passed as parameter 1. This string can be retrieved from Hollywood by querying the `#ATTRFORMAT` constant on `#VIDEO` objects.

INPUTS

`handle` handle returned by `OpenVideo()`

RESULTS

`name` video format name

22.9 GetVideoFrames

NAME
GetVideoFrames – get number of video frames in stream (V5.0)

SYNOPSIS
```
ULONG count = GetVideoFrames(APTR handle);
```

FUNCTION
This function must return the total number of video frames in the specified video stream handle. If you don't have this information, you might return 0 here.

INPUTS

`handle` handle returned by `OpenVideo()`

RESULTS

`count` total number of video frames

22.10 NextPacket

NAME
NextPacket – get next packet from stream (V5.0)

SYNOPSIS
```
int status = NextPacket(APTR handle, struct VideoPacketStruct *p);
```

FUNCTION

This function must read the next packet from the video stream and return it to Hollywood. `NextPacket()` is passed a `struct VideoPacketStruct` pointer that looks like the following:

```
struct VideoPacketStruct
{
    APTR Packet;  // [out]
    int Type;     // [out]
    int Size;     // [out]
    int Pad;      // [unused]
    double PTS;   // [out]
};
```

You have to fill in the following members of the structure:

`Packet:` Must be set to a pointer to the actual packet data. This is an opaque data type only understood by your plugin. The packet pointer you specify here will be passed to your plugin's `DecodeVideoFrame()` or `DecodeAudioFrame()` function, depending on the packet type. To free the packet data passed here, Hollywood will call `FreePacket()` on it.

`Type:` This must be set to the type of data that is in the packet. This must be one of the following predefined types:

`HWVIDPKTTYPE_VIDEO:`
Packet contains video data.

`HWVIDPKTTYPE_AUDIO`
Packet contains audio data.

`Size:` This must be set to the packet's size in bytes.

`PTS:` This must be set to the packet data's presentation time stamp, i.e. the time when this packet's data should be presented to the viewer, relative to the beginning of the video stream. You have to specify this time stamp as a floating point number in seconds, i.e. a presentation time stamp of 10.2 means that the packet's data is to be shown 10.2 seconds after the start of the video stream. The value you pass in `PTS` must be a multiple of the `FrameTime` fraction specified in `OpenVideo()`.

`NextPacket()` must return a special status code: 0 means success, -1 means `NextPacket()` has reached the stream end, and -2 means there was an error.

This function must be implemented in a thread-safe way.

INPUTS

`handle` handle returned by `OpenVideo()`

`v` pointer to a `struct VideoPacketStruct` to be filled out by the function

RESULTS

`status` status code (see above)

22.11 OpenVideo

NAME

OpenVideo – open a video file (V5.0)

SYNOPSIS

```
APTR handle = OpenVideo(STRPTR filename, struct OpenVideoCtrl *ctrl);
```

FUNCTION

This function has to open the specified filename, check if it is in a video file format that the plugin wants to handle, and, if it is, return a handle to the video file back to Hollywood. Otherwise it has to return `NULL`. The handle returned by `OpenVideo()` is an opaque datatype that only your plugin knows about. Hollywood will simply pass this handle back to your `NextPacket()` function when it wants to have the individual packets of the video file.

This function also has to provide certain information about the video file it has just opened. This information has to be written to the `struct OpenVideoCtrl` that is passed in the second parameter. This structure looks like this:

```
struct OpenVideoCtrl
{
    int Width;          // [out]
    int Height;         // [out]
    ULONG Duration;     // [out]
    int Frequency;      // [out]
    int Channels;       // [out]
    int SeekMode;       // [out]
    int BitRate;        // [out]
    int PixFmt;         // [out]
    ULONG Flags;        // [out]
    int Pad;            // [unused]
    double FrameTime;   // [out]
    DOSINT64 FileSize; // [out] -- V6.0
    STRPTR Adapter;     // [in]  -- V6.0
};
```

The following information has to be written to the `struct OpenVideoCtrl` pointer by `OpenVideo()`:

`Width:` Frame width in pixels.

`Height:` Frame height in pixels.

`Duration:`
Duration of the video stream in milliseconds.

`Frequency:`
The number of PCM frames that should be played per second. Usually 44100 or 48000. If there is no audio stream accompanying the video stream, set this to 0.

`Channels:`
: The number of channels used by the sound file. This must be either 1 (mono) or 2 (stereo). If there is no audio stream accompanying the video stream, set this to 0.

`SeekMode:`
: This member specifies the unit in which `SeekVideo()` receives its seek destination position. This can be one of the following constants:

  `HWVIDSEEKMODE_TIME:`
  : Seek destination position will be passed as a time stamp in milliseconds. This is the most common seek mode but not all video formats support it.

  `HWVIDSEEKMODE_BYTE:`
  : Seek destination position will be passed as an absolute position in bytes which is calculated by multiplying the `BitRate` parameter of this structure with the target time stamp. This will obviously only work with video streams that use a constant bit rate. If you choose this mode, make sure to set `BitRate` to the correct video bit rate.

  Note that this member doesn't have any effect if the `HWVIDFLAGS_CANSEEK` flag isn't set.

`BitRate:`
: Set this to the number of bytes this video stream needs to store one second of video data. You only need to provide this information if you've set `SeekMode` to `HWVIDSEEKMODE_BYTE`. Otherwise this information is not needed. Please note that in contrast to its name, this member actually needs to be set to a value in bytes, not in bits!

`PixFmt:`
: This member specifies the pixel format in which you want to provide the individual video frame data to Hollywood. This can be one of the following constants:

  `HWVIDPIXFMT_YUV420P:`
  : Pixel data is provided in planar YUV 4:2:0 format.

  `HWVIDPIXFMT_ARGB32:`
  : Pixel data is provided as 32-bit ARGB pixel data.

`Flags:`
: A combination of the following flags describing additional properties of the stream:

  `HWVIDFLAGS_CANSEEK`
  : This plugin's `SeekVideo()` function supports seeking according to the mode specified in `SeekMode`. Note that even if this flag isn't set, your plugin still needs to be able to rewind the video stream, i.e. seek it back to the very beginning in case 0 is passed to `SeekVideo()`. See Section 22.12 [SeekVideo], page 184, for details.

`FrameTime:`
Specifies how long each video frame should be presented in seconds which can then be used to compute the frame rate. For example, a `FrameTime` of 0.04 means that the video is meant to play at 25 frames per second.

`FileSize:`
If Hollywood 6.0 or higher is calling your function, you should set this member to the size of the video file in bytes or -1 if you don't know the size. Setting this member is optional but doing so allows Hollywood to open your video faster since Hollywood won't have to query the file size on its own if you can provide it. Make sure to check for Hollywood 6.0 before touching this member because it wasn't there in previous versions. (V6.0)

`Adapter:` Starting with Hollywood 6.0 users can specify the file adapter that should be used to open certain files. If this member is non-`NULL`, Hollywood wants your plugin to use the file adapter specified in `Adapter` to open the video file. This means that you have to use `hw_FOpenExt()` instead of `hw_FOpen()` to open the video file. Make sure to check for Hollywood 6.0 before trying to access this member because it isn't there in previous versions. See Section 25.16 [hw_FOpenExt], page 202, for details. (V6.0)

Please note that you should not use ANSI C functions like `fopen()` to open the file that is passed to this function because the filename that is passed to this function can also be a specially formatted filename specification that Hollywood uses to load files that have been linked to applets or executables. In order to be able to load these files correctly, you have to use special IO functions provided by Hollywood. See Section 2.12 [File IO information], page 15, for details.

INPUTS

`filename` filename to open

`ctrl` pointer to a `struct OpenVideoCtrl` for storing information about the video file

RESULTS

`handle` a handle that identifies this video file or `NULL` if plugin doesn't want to handle this video file

22.12 SeekVideo

NAME
SeekVideo – seek video stream to new position (V5.0)

SYNOPSIS

```
int status = SeekVideo(APTR handle, ULONG pos, int mode);
```

FUNCTION
This function must seek the specified video stream handle to the position passed as parameter 2. The new position is specified either in milliseconds or as a byte offset relative to the beginning of the file. This depends on the `SeekMode` you have set in

`OpenVideo()`. The 'mode' parameter passed to this function is actually redundant. It will always be the same as the `SeekMode` passed in `OpenVideo()`.

Before calling `SeekVideo()`, Hollywood will always flush your audio and video decoder states using `FlushAudio()` and `FlushVideo()`.

Note that you only need to implement seeking if you have set the flag `HWVIDFLAGS_CANSEEK` in `OpenVideo()`. However, seeking back to position 0 has to be implemented by all plugins - no matter whether `HWVIDFLAGS_CANSEEK` has been set or not. If `SeekVideo()` is called with 0 as the new position, you need to rewind the stream so that the next call to `NextPacket()` returns packets right from the start of the stream.

This function must be implemented in a thread-safe way.

INPUTS

`handle`	handle returned by `OpenVideo()`
`pos`	new stream position in a unit that depends on "mode"
`mode`	seek mode

RESULTS

`status`	status code indicating error or success (see above)

23 AudioBase functions

23.1 Overview

`AudioBase` contains a number of functions to deal with the audio related functionality of Hollywood.

`AudioBase` is available since Hollywood 5.0.

23.2 hw_LockSample

NAME

hw_LockSample – gain access to the raw PCM data of a sample (V5.0)

SYNOPSIS

```
APTR handle = hw_LockSample(lua_ID *id, int readonly, struct hwTagList *
                  tags, struct hwos_LockSampleStruct *smplock);
```

FUNCTION

This function locks the specified sample and allows you to access its raw PCM data. You have to pass the object identifier of the sample you want to lock as a `lua_ID`. See Section 2.17 [Object identifiers], page 21, for details.

You also have to pass a pointer to a `struct hwos_LockSampleStruct` which will be filled with all the information you need by `hw_LockSample()`. `struct hwos_LockSampleStruct` looks like this:

```
struct hwos_LockSampleStruct
{
    APTR Buffer;        // [out]
    int BufferSize;     // [out]
    int Samples;        // [out]
    int Channels;       // [out]
    int Bits;           // [out]
    int Frequency;      // [out]
    ULONG Flags;        // [out]
};
```

`hw_LockSample()` will write to the structure members as follows:

`Buffer:` This will point to a buffer which contains the raw PCM samples.

`BufferSize:`
This will be set to the total size of the PCM buffer passed in `Buffer` in bytes.

`Samples:` The total number of PCM frames in the sample. Note that this value is specified in PCM frames, not in bytes.

`Channels:`
The number of channels used by the sample. This will be either 1 (mono) or 2 (stereo).

`Bits:` The number of bits per PCM sample. This will be either 8 or 16.

`Frequency:`
The number of PCM frames that should be played per second. Usually 44100 or 48000.

`Flags:` A combination of the following flags describing additional properties of the sample:

`HWSNDFLAGS_BIGENDIAN`
The PCM samples are stored in big endian format. This flag is only meaningful if the bit resolution is 16.

`HWSNDFLAGS_SIGNEDINT`
The PCM samples are stored as signed integers. This is typically set.

If you do not want to write to the raw PCM buffer, pass `True` in the `readonly` parameter. `hw_LockSample()` will be faster than as it does not have to update the sample data in the sound card memory.

Do not hold sample locks longer than necessary. In particular, do not return control to the script while holding a sample lock because the script might try to modify the sample then and this will lead to trouble in case the sample is still locked. You should call `hw_UnLockSample()` as soon as possible.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`id`	object identifier of sample to be locked
`readonly`	`True` for read-only access, `False` for read/write access
`tags`	reserved for future use; pass `NULL` for now
`smplock`	pointer to a `struct hwos_LockSampleStruct` that is to be filled by this function

RESULTS

`handle`	handle to the locked sample or `NULL` on error

23.3 hw_SetAudioAdapter

NAME
hw_SetAudioAdapter – install an audio adapter (V6.0)

SYNOPSIS

```
int error = hw_SetAudioAdapter(hwPluginBase *self, ULONG flags,
                struct hwTagList *tags);
```

FUNCTION
This function can be used to activate a plugin that has the `HWPLUG_CAPS_AUDIOADAPTER` capability flag set. This function must only be called from inside your `RequirePlugin()`

implementation. If this function succeeds, Hollywood's inbuilt audio driver will be completely replaced by the audio driver provided by your plugin and Hollywood will call into your plugin whenever it needs to output audio. In the first parameter, you have to pass a pointer to the `hwPluginBase` that Hollywood has passed to your plugin's `InitPlugin()` function. The second parameter must be set to a combination of flags. The following flags are currently defined:

`HWSAAFLAGS_PERMANENT:`
: If this flag is set, the audio adapter will be made permanent. This means that other plugins won't be able to overwrite this audio adapter with their own one. If `HWSAAFLAGS_PERMANENT` is set, all subsequent calls to `hw_SetAudioAdapter()` will fail and your audio adapter will persist.

`HWSAAFLAGS_UPDATE:`
: If this flag is set, Hollywood won't install an audio adapter but update the parameters of an existing audio adapter. This is only supported if the `hwPluginBase` passed to `hw_SetAudioAdapter()` equals the currently installed audio adapter. The only parameter that can currently be updated is the `HWSAATAG_BUFFERSIZE` attribute. Many audio drivers might not know the optimal buffer size before actually opening the audio device so they might need to adjust the audio adapter's buffer size later. This is what the `HWSAAFLAGS_UPDATE` tag is here for.

Additionally, `hw_SetAudioAdapter()` accepts a tag list which recognizes the following tags:

`HWSAATAG_BUFFERSIZE:`
: This must be set to size of your audio adapter's playback buffer in bytes. This value should be really small for low latency audio. If your buffer size is too large, it will take a long time until changes in volume or pitch take effect and pause is not very accurate. If you don't know about the optimal buffer size for your audio device at the time you call `hw_SetAudioAdapter()`, you may also adjust the buffer size later on by calling `hw_SetAudioAdapter()` again, but this time with the `HWSAAFLAGS_UPDATE` flag set (see above). By default, Hollywood uses an audio buffer size of 2048 bytes.

`HWSAATAG_CHANNELS:`
: This must be set to the number of channels your audio driver wants to offer. This defaults to 8.

See Section 6.1 [Audio adapter plugins], page 43, for information on how to write audio adapter plugins.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`self`	`hwPluginBase` pointer passed to `InitPlugin()`
`flags`	combination of flags (see above)
`tags`	tag list specifying further parameters (see above)

RESULTS

`error` error code or 0 for success

23.4 hw_UnLockSample

NAME

hw_UnLockSample – release sample lock (V5.0)

SYNOPSIS

```
void hw_UnLockSample(APTR handle);
```

FUNCTION

This function releases the specified sample lock. You need to call this function as soon as you're finished with accessing a sample's raw PCM data. After the call to `hw_UnLockSample()` you must no longer access the PCM data pointers returned by `hw_LockSample()`. See Section 23.2 [hw_LockSample], page 187, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` sample lock obtained by `hw_LockSample()`

24 CRTBase functions

24.1 Overview

`CRTBase` contains many functions from the ANSI C runtime library. This is mostly only useful if you compile plugins for AmigaOS and compatibles because you won't be able to use many of the C runtime library functions directly because they require constructor and destructor code which is not supported by AmigaOS modules loaded via `LoadSeg()`. See Section 3.2 [C runtime limitations], page 26, for details.

On all other systems you can just use the C runtime functions provided by your compiler directly. You should not use the ones from `CRTBase`. They are really just here to workaround AmigaOS module limitations.

Note that all functions that deal with files and expect a string will use Hollywood's default string encoding which is UTF-8 since Hollywood 7.0. This means that functions like `fopen()`, `stat()`, `rename()`, `remove()`, etc. will expect strings to be in UTF-8 encoding unless the Hollywood version handling your plugin is older than 7.0 or the script explicitly requests to be run in ISO 8859-1 compatibility mode. Note that this behaviour is different from the standard behaviour of functions like `fopen()`. Normally, they will expect strings to be in the system's default locale encoding. While this is typically UTF-8 on Linux, it certainly isn't UTF-8 on Windows or AmigaOS, for example.

`CRTBase` is available since Hollywood 5.0.

25 DOSBase functions

25.1 Overview

`DOSBase` contains commands for working with files and directories. As Hollywood can deal with virtual files as well as with files linked into other files like applets or executables you must make sure that you only use IO functions provided by Hollywood in the `DOSBase` pointer to deal with files. If you use functions like `fopen()` from the ANSI C library instead, your plugin will only work with normal files that are physically existent on a system drive.

For example, when writing plugins that provide loaders for additional file formats like images, sounds, or videos it can often happen that the filename that is passed to your plugin is a specially formatted specification that Hollywood uses to load files that have been linked to applets or executables. If you do not use Hollywood's IO functions to open this file, your plugin won't be able to load files that have been linked to applets or executables. This can be quite annoying for the end-user because the ability to link data files into applets and executables is a key functionality of Hollywood and thus your plugin should strive to be compatible with it. If you use `fopen()` instead, it will just fail whenever your function is passed a specially formatted specification to open one of Hollywood's virtual files.

`DOSBase` is available since Hollywood 5.0.

25.2 hw_AddPart

NAME

hw_AddPart – append file name to path specification (V5.0)

SYNOPSIS

```
int ok = hw_AddPart(STRPTR dirname, STRPTR filename, int size);
```

FUNCTION

This function appends the file name specified in parameter 2 to the directory name passed in parameter 1. Depending on the operating system in use, the components are joined by slash or backslash characters.

This function returns `True` on success, `False` otherwise.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`dirname`	directory name to append to
`filename`	file name to append
`size`	size of the `dirname` buffer in bytes

RESULTS

`ok`	`True` on success, `False` otherwise

25.3 hw_BeginDirScan

NAME

hw_BeginDirScan – start iteration over all directory entries (V5.0)

SYNOPSIS

```
int error = hw_BeginDirScan(APTR handle, APTR *dirhandle);
```

FUNCTION

This function initiates a directory scanning operation on the handle specified in parameter 1. This handle must have been obtained by `hw_Lock()`. You have to pass a pointer to an `APTR` to this function in parameter 2. This pointer will receive a special directory handle that you have to pass to `hw_EndDirScan()` once you are finished with the directory scanning. See Section 25.7 [hw_EndDirScan], page 196, for details.

To iterate over the single directory entries, call `hw_NextDirEntry()`. See Section 25.26 [hw_NextDirEntry], page 209, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` handle returned by `hw_Lock()`

`dirhandle`
pointer to an `APTR` to receive the scan handle

RESULTS

`error` error code or 0 on success

25.4 hw_ChunkToFile

NAME

hw_ChunkToFile – save virtual file to real file (V6.0)

SYNOPSIS

```
int error = hw_ChunkToFile(STRPTR dest, APTR src, DOSINT64 pos,
                DOSINT64 len, ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function can be used to save a virtual file to a physical file. This is useful in connection with the `hw_TranslateFileName()` and `hw_TranslateFileNameExt()` commands which break down a virtual file specification into their individual constituents. `hw_ChunkToFile()` uses these individual constituents to save the virtual file to the new physical file specified in `dest`.

If the `HWCTFFLAGS_MEMORYSOURCE` flag isn't set, you have to pass a `STRPTR` to the filename that contains the virtual file in `src`. The `pos` and `len` parameters must be set to the respective values returned by `hw_TranslateFileName()` or `hw_TranslateFileNameExt()`. If `HWCTFFLAGS_MEMORYSOURCE` is set, you have to pass a pointer to a memory block in `src`. The `pos` argument is ignored in this case but the `len` argument must contain the virtual file length in bytes. Once again, this value is returned by `hw_TranslateFileName()` and

`hw_TranslateFileNameExt()`. See Section 25.33 [hw_TranslateFileNameExt], page 216, for details.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`dest`	name of physical file to save virtual file to
`src`	pointer to a filename or memory block depending on whether `HWCTFFLAGS_MEMORYSOURCE` is set (see above)
`pos`	virtual file offset inside the container file
`len`	virtual file length inside the container file
`flags`	flags for the operation (see above)
`tags`	currently unused, pass `NULL`

RESULTS

`error`	error code or 0 on success

25.5 hw_CreateDir

NAME
hw_CreateDir – create a directory (V5.0)

SYNOPSIS
```
int error = hw_CreateDir(STRPTR name);
```

FUNCTION
This function creates the specified directory.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`name`	directory to create

RESULTS

`error`	error code or 0 on success

25.6 hw_DeleteFile

NAME
hw_DeleteFile – delete a file system object (V5.0)

SYNOPSIS
```
int ok = hw_DeleteFile(STRPTR name);
```

FUNCTION

This function deletes the specified file system object. It can be either a file or a directory. If `name` specifies a directory, then this directory must be empty or `hw_DeleteFile()` will fail.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`name` file system object to delete

RESULTS

`ok` `True` on success, `False` otherwise

25.7 hw_EndDirScan

NAME

hw_EndDirScan – stop directory scan (V5.0)

SYNOPSIS

```
void hw_EndDirScan(APTR dirhandle);
```

FUNCTION

This function stops a directory scanning operation initiated by a call to `hw_BeginDirScan()`. You have to pass the handle that was returned to you by `hw_BeginDirScan()` to this function. After you have called `hw_EndDirScan()` it is no longer allowed to call `hw_NextDirEntry()`.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`dirhandle`
handle returned by `hw_BeginDirScan()`

25.8 hw_ExLock

NAME

hw_ExLock – examine a file system object (V5.0)

SYNOPSIS

```
int ok = hw_ExLock(APTR handle, struct hwos_ExLockStruct *exlock);
```

FUNCTION

This function returns information about a file system object that has been locked using `hw_Lock()`. The information is written to the `struct hwos_ExLockStruct` which has to be passed in parameter 2. `struct hwos_ExLockStruct` looks like this:

```
struct hwos_ExLockStruct
```

```
{
    int nStructSize;    // [in]
    STRPTR Name;        // [out]
    int Type;           // [out]
    ULONG Size;         // [out]
    ULONG Flags;        // [out]
};
```

Here's a description of the individual structure members:

`nStructSize:`
: This must be set by you to `sizeof(struct hwos_ExLockStruct)` before calling `hw_ExLock()`.

`Name:`
: This is currently always set to `NULL`. Use `hw_NameFromLock()` to get the fully-qualified path to this file system object.

`Type:`
: This will be set to one of the following types:

  `HWEXLOCKTYPE_FILE:`
  : The file system object is a file.

  `HWEXLOCKTYPE_DIRECTORY:`
  : The file system object is a directory.

`Size:`
: Size of object in bytes if it is a file, otherwise 0.

`Flags:`
: Combination of flags describing the file system object attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle`
: handle returned by `hw_Lock()`

`exlock`
: pointer to a `struct hwos_ExLockStruct` for storing information about the file system object

RESULTS

`ok`
: `True` on success, `False` otherwise

25.9 hw_FClose

NAME

hw_FClose – close a file handle (V5.0)

SYNOPSIS

```
int ok = hw_FClose(APTR handle);
```

FUNCTION

This function closes the specified file handle, finishing all pending writes. `hw_FClose()` returns `True` on success, `False` otherwise.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`handle` file handle returned by `hw_FOpen()`

RESULTS

`ok` `True` on success, `False` on failure

25.10 hw_FEof

NAME
hw_FEof – check if end-of-file marker has been reached (V5.0)

SYNOPSIS

```
int ok = hw_FEof(APTR handle);
```

FUNCTION
This function returns `True` if the end-of-file marker has been reached for the specified file handle.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`handle` file handle returned by `hw_FOpen()`

RESULTS

`ok` `True` or `False`

25.11 hw_FFlags

NAME
hw_FFlags – get flags of an open file (V6.0)

SYNOPSIS

```
ULONG flags = hw_FFlags(APTR fh);
```

FUNCTION
This function returns the flags associated with the file handle passed in parameter 1. The following flags are currently defined:

`HWFOPENFLAGS_STREAMING:`
: If this flag is set, the file is being streamed from a network source. This means that you should try to avoid operations that are inefficient on streaming sources like excessive seeking operations.

`HWFOPENFLAGS_NOSEEK:`
: If this flag is set, you won't be able to seek the file. This means that most calls to `hw_FSeek()` will fail. The only operations that are still supported by `hw_FSeek()` are rewinding (i.e. reverting the read/write cursor to the beginning of the file) and querying the current file cursor position. If you want `hw_FSeek()` to work on files with `HWFOPENFLAGS_NOSEEK` set too, you may want to set the `HWFOPENMODE_EMULATESEEK` flag, although this can be very inefficient. See Section 25.15 [hw_FOpen], page 200, for details.

DESIGNER COMPATIBILITY
Supported since Designer 4.5

INPUTS

`handle` file handle returned by `hw_FOpen()`

RESULTS

`flags` combination of file flags

25.12 hw_FFlush

NAME
hw_FFlush – flush all pending writes (V6.0)

SYNOPSIS
```
int ok = hw_FFlush(APTR handle);
```

FUNCTION
This function will flush any pending buffered write operations to the specified file handle and return `True` on success, `False` otherwise.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.5

INPUTS

`handle` file handle returned by `hw_FOpen()`

RESULTS

`ok` `True` on success, `False` on failure

25.13 hw_FGetC

NAME
hw_FGetC – read a single character from a file (V5.0)

SYNOPSIS
```
int c = hw_FGetC(APTR handle);
```

FUNCTION

This function reads a single character from the specified file handle and returns it. In case the end-of-file marker has been reached or an error has occurred, -1 is returned.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` file handle returned by `hw_FOpen()`

RESULTS

`c` character read or -1 on error or EOF

25.14 hw_FilePart

NAME

hw_FilePart – get last component of a path specification (V5.0)

SYNOPSIS

```
STRPTR f = hw_FilePart(STRPTR name);
```

FUNCTION

This function returns a pointer to the last component in the specified path. If the path points to a file, then the last component is always the file name. If there is only one component in the path specification, `hw_FilePart()` will return a pointer to the beginning of the string.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`name` path specification

RESULTS

`f` pointer to the last component in the path

25.15 hw_FOpen

NAME

hw_FOpen – open a file (V5.0)

SYNOPSIS

```
APTR handle = hw_FOpen(STRPTR name, int mode);
```

FUNCTION

This function opens the specified file for reading and/or writing. The second parameter specifies the IO mode to be used on this file. It can be a combination of the following flags:

`HWFOPENMODE_READ_LEGACY:`
: File should be opened for reading. This is mutually exclusive with `HWFOPENMODE_WRITE` and `HWFOPENMODE_READWRITE`. Please note that this flag is actually set to 0 for compatibility reasons. This means that `hw_FOpen()` cannot use the bitwise AND-operator to check if it is set. Instead, it will check if either `HWFOPENMODE_WRITE` or `HWFOPENMODE_READWRITE` is set. If both aren't set, then it will assume that `HWFOPENMODE_READ_LEGACY` is set and will open the file for reading. You should only use this flag if you also need to target Hollywood 5. For Hollywood 6.0 and higher, use `HWFOPENMODE_READ_NEW` instead (see below).

`HWFOPENMODE_WRITE:`
: File should be opened for writing. If it doesn't exist, `hw_FOpen()` will create it first. This is mutually exclusive with `HWFOPENMODE_READ_NEW` and `HWFOPENMODE_READWRITE`.

`HWFOPENMODE_READWRITE:`
: File should be opened for reading and writing. This is mutually exclusive with `HWFOPENMODE_WRITE` and `HWFOPENMODE_READ_NEW`.

`HWFOPENMODE_READ_NEW:`
: File should be opened for reading. This is mutually exclusive with `HWFOPENMODE_WRITE` and `HWFOPENMODE_READWRITE`. Please note that this flag requires Hollywood 6.0. If you want to open files for reading with earlier Hollywood versions, use `HWFOPENMODE_READ_LEGACY`. See above for details. (V6.0)

`HWFOPENMODE_NOFILEADAPTER:`
: If this flag is set, Hollywood will skip all file adapters that are currently active and use its inbuilt file handler to open the file. Use this only if you have a good reason to bypass the file adapters. (V6.0)

`HWFOPENMODE_EMULATESEEK:`
: If this flag is set, Hollywood will emulate seeking for files that have the `HWFOPENFLAGS_NOSEEK` flag set. Emulation of the seek functionality is done by simply reading bytes from the file until the desired seek position has been reached. This is of course highly inefficient for large seek distances so it should only be used on small files or to bridge small seek distances. (V6.0)

`HWFOPENMODE_FORCEUTF8:`
: If this flag is set, Hollywood will expect the string passed in `name` to always be in UTF-8 encoding. By default, Hollywood expects the string to be in the encoding of the current script. You can override this behaviour by setting `HWFOPENMODE_FORCEUTF8`. (V7.0)

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`name` file to open

`mode` desired access mode (see above)

RESULTS

`handle` handle to refer to this file later or `NULL` on error

25.16 hw_FOpenExt

NAME

hw_FOpenExt – open a file (V6.0)

SYNOPSIS

```
APTR handle = hw_FOpenExt(STRPTR name, int mode, struct hwTagList *tags);
```

FUNCTION

This function does the same as `hw_FOpen()` but accepts an additional tag list that can be used to specify additional parameters for the open operation. The following tags are currently recognized:

`HWFOPENTAG_ADAPTER:`
: This tag allows you to specify one or more file adapters that should be used to open the file. The `pData` member of this tag must be set to a string containing the name of at least one file adapter or a special keyword (see the Hollywood documentation for more information). Multiple names or keywords must be separated by the vertical bar character (`|`). If this tag is set, `hw_FOpenExt()` will fail in case the specified file adapter refuses to open the file.

See Section 25.15 [hw_FOpen], page 200, for a detailed description of the other parameters.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`name` file to open

`mode` desired access mode (see above)

`tags` tag list containing additional options (see above)

RESULTS

`handle` handle to refer to this file later or `NULL` on error

25.17 hw_FPutC

NAME

hw_FPutC – write single character to file (V5.0)

SYNOPSIS

```
int ok = hw_FPutC(APTR handle, int ch);
```

FUNCTION

This function writes the specified character to the specified file handle. It returns `True` on success or `False` on failure.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle`	file handle returned by `hw_FOpen()`
`ch`	character to write to file (0-255)

RESULTS

`ok`	`True` to indicate success, `False` on failure

25.18 hw_FRead

NAME

hw_FRead – read file data into memory buffer (V5.0)

SYNOPSIS

```
int read = hw_FRead(APTR handle, APTR buf, ULONG size);
```

FUNCTION

This function reads the specified number of bytes into the memory buffer specified in parameter 2. It returns the number of bytes actually read.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle`	file handle returned by `hw_FOpen()`
`buf`	pointer to memory buffer to receive the data read
`size`	number of bytes to read from file handle

RESULTS

`read`	number of bytes actually read

25.19 hw_FSeek

NAME

hw_FSeek – seek file to new position (V5.0)

SYNOPSIS

```
ULONG oldpos = hw_FSeek(APTR handle, ULONG pos, int mode);
```

FUNCTION

This function seeks the file handle's read/write cursor to the specified position. Additionally, it returns the position of the read/write cursor before the seek operation. The specified position is relative to the seek mode passed in parameter 3. This can be one of the following modes:

`HWFSEEKMODE_CURRENT:`
: New seek position is relative to the current position.

`HWFSEEKMODE_BEGINNING:`
: New seek position is relative to the beginning of the file.

`HWFSEEKMODE_END:`
: New seek position is relative to the end of the file.

To find out the current position of the read/write cursor, call `hw_FSeek()` with a 0 zero position and `HWFSEEKMODE_CURRENT`.

If there was an error, `hw_FSeek()` return -1.

Note that `hw_FSeek()` currently isn't able to handle negative seek positions. Thus, the value you pass in parameter 2 must always be positive.

Starting with Hollywood 6.0 there is also a 64-bit version of this command: See Section 25.20 [hw_FSeek64], page 204, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle`	file handle returned by `hw_FOpen()`
`pos`	destination seek position; this must not be negative!
`mode`	seek mode (see above)

RESULTS

`oldpos`	previous position of file cursor or -1 on error

25.20 hw_FSeek64

NAME

hw_FSeek64 – seek file to new position (V6.0)

SYNOPSIS

```
DOSINT64 oldpos = hw_FSeek64(APTR handle, DOSINT64 pos, int mode);
```

FUNCTION

This function does the same as `hw_FSeek()` but uses 64-bit integers to be able to deal with large files. See Section 25.19 [hw_FSeek], page 203, for details.

Note that `hw_FSeek64()` currently isn't able to handle negative seek positions. Thus, the value you pass in parameter 2 must always be positive.

Please note that the AmigaOS 3 and WarpOS versions of Hollywood don't support large file handling. On these systems, the `DOSINT64` type will be mapped to a 32-bit integer automatically.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`handle` file handle returned by `hw_FOpen()`

`pos` destination seek position; this must not be negative!

`mode` seek mode (see above)

RESULTS

`oldpos` previous position of file cursor or -1 on error

25.21 hw_FStat

NAME

hw_FStat – obtain information about open file (V6.0)

SYNOPSIS

```
int ok = hw_FStat(APTR fh, ULONG flags, struct hwos_StatStruct *st,
                        struct hwTagList *tags);
```

FUNCTION

This function writes information about the file handle passed in parameter 1 to the structure pointer passed in parameter 3. `struct hwos_StatStruct` looks like this:

```
struct hwos_StatStruct
{
    int Type;                                // [out]
    DOSINT64 Size;                           // [out]
    ULONG Flags;                             // [out]
    struct hwos_DateStruct Time;             // [out]
    struct hwos_DateStruct LastAccessTime;   // [out]
    struct hwos_DateStruct CreationTime;     // [out]
    STRPTR FullPath;                         // [out]
    STRPTR Comment;                          // [out]
    int LinkMode;                            // [out]
    STRPTR Container;                        // [out]
};
```

The following information is written to the individual structure members:

`Type:` This will always be set to `HWSTATTYPE_FILE`.

`Size:` This will be set to the size of the file in bytes or -1 if the size is not known, for example because the file is being streamed from a network source.

`Flags:` Combination of flags describing the file's attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

`Time:` Time stamp indicating when this file was last modified.

`LastAccessTime:`
Time stamp indicating when this file was last accessed.

`CreationTime:`
Time stamp indicating when this file was created.

`FullPath:`
This will be set to a fully qualified path to the file. The string pointer used here will stay valid until the next call to `hw_FStat()`. If you set the `HWSTATFLAGS_ALLOCSTRINGS` flag, `hw_FStat()` will not use a static string buffer but allocate a new private string pointer for this structure member. You will have to call `hw_TrackedFree()` on this string when you're done with it in that case. This is useful if you need to use `hw_FStat()` in a multithreaded environment.

`Comment:` Comment stored for this file in the file system. The string pointer returned here will stay valid until the next call to `hw_FStat()`. This may be `NULL` if the file system does not support comments for its objects. If you set the `HWSTATFLAGS_ALLOCSTRINGS` flag, `hw_FStat()` will not use a static string buffer but allocate a new private string pointer for this structure member. You will have to call `hw_TrackedFree()` on this string when you're done with it in that case. This is useful if you need to use `hw_FStat()` in a multithreaded environment.

`LinkMode:`
Currently unused. May contain random data.

`Container:`
Currently unused. May contain random data.

The following flags are supported by `hw_FStat()`:

`HWSTATFLAGS_ALLOCSTRINGS:`
If you set this flag, `hw_FStat()` will not use static string buffers for the `FullPath` and `Comment` structure members but allocate new private string buffers for them. You will have to call `hw_TrackedFree()` on these buffers once you're done with them in that case. This flag is useful if you need to use `hw_FStat()` in a multithreaded environment.

`hw_FStat()` returns `True` on success or `False` on failure.

This function is only thread-safe if you set the `HWSTATFLAGS_ALLOCSTRINGS` flag.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`handle` file handle returned by `hw_FOpen()`

`flags` combination of flags (see above)

`st` pointer to a `struct hwos_StatStruct` for storing information about the file

`tags` reserved for future use (pass `NULL`)

RESULTS

`ok` `True` on success, `False` otherwise

25.22 hw_FWrite

NAME

hw_FWrite – write data to file handle (V5.0)

SYNOPSIS

```
int written = hw_FWrite(APTR handle, APTR buf, ULONG size);
```

FUNCTION

This function writes the specified number of bytes from the memory buffer specified in parameter 2 to the file handle passed in parameter 1. It returns the number of bytes actually written.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` file handle returned by `hw_FOpen()`

`buf` source memory buffer

`size` number of bytes to write to file handle

RESULTS

`written` number of bytes actually written

25.23 hw_GetCurrentDir

NAME

hw_GetCurrentDir – get path to current directory (V5.0)

SYNOPSIS

```
void hw_GetCurrentDir(STRPTR buf, int len);
```

FUNCTION

This function copies a fully-qualified path specification to the current directory to the specified memory buffer.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`buf` buffer to receive current directory

`len` size of that buffer

25.24 hw_Lock

NAME

hw_Lock – lock a file system object for examination (V5.0)

SYNOPSIS

```
APTR handle = hw_Lock(STRPTR name, int mode);
```

FUNCTION

This function can be used to lock a file system object for further examination via the `hw_ExLock()` call. If the file system object is a directory, you may also iterate through its entries using `hw_NextDirEntry()`.

The following bits are currently supported by the `mode` parameter:

`HWLOCKMODE_READ_LEGACY:`
: Open the file system object for reading. This must be set. Please note that this flag is actually set to 0 for compatibility reasons. This means that `hw_Lock()` cannot use the bitwise AND-operator to check if it is set. Instead, it will check if the `HWLOCKMODE_WRITE` flag is set and in case it isn't set, the file system object will be opened in read mode. You should only use this flag if you also need to target Hollywood 5. For Hollywood 6.0 and higher, use `HWLOCKMODE_READ` instead (see below).

`HWLOCKMODE_WRITE:`
: Open the file system object for writing. Currently unsupported. Do not use this.

`HWLOCKMODE_READ:`
: File system object should be opened for reading. Please note that this flag requires Hollywood 6.0. If you want to open file system objects for reading with earlier Hollywood versions, use `HWLOCKMODE_READ_LEGACY`. See above for details. (V6.0)

`HWLOCKMODE_NOADAPTER:`
: If this flag is set, `hw_Lock()` will skip all file and directory adapters and use Hollywood's inbuilt handlers. Use this only if you have a good reason to skip the file and directory adapters. (V6.0)

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`name` file system object to open

`mode` locking mode (see above)

RESULTS

`handle` handle to refer to this object later or `NULL` on error

25.25 hw_NameFromLock

NAME

hw_NameFromLock – get fully qualified path to file system object (V6.0)

SYNOPSIS

```
int ok = hw_NameFromLock(APTR handle, STRPTR buf, int size);
```

FUNCTION

This function copies the fully-qualified path of the specified file system object opened by `hw_Lock()` to the memory buffer provided in parameter 2.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` object handle returned by `hw_Lock()`

`buf` pointer to memory buffer to receive the path specification

`size` size of the buffer in bytes

RESULTS

`ok` `True` on success, `False` otherwise

25.26 hw_NextDirEntry

NAME

hw_NextDirEntry – return next directory object (V5.0)

SYNOPSIS

```
int ok = hw_NextDirEntry(APTR handle, APTR dirhandle, struct
               hwos_ExLockStruct *exlock);
```

FUNCTION

This function reads the next file system object from the specified directory handle. This handle must have been opened by `hw_Lock()`. You also have to pass the handle returned to you by `hw_BeginDirScan()`. If `hw_NextDirEntry()` returns `False`, then all entries have been read. If it returns `True`, then you can read information about the file system object from the `struct hwos_ExLockStruct` which has to be passed in parameter 2. `struct hwos_ExLockStruct` looks like this:

```
struct hwos_ExLockStruct
{
```

```
        int nStructSize;    // [in]
        STRPTR Name;        // [out]
        int Type;           // [out]
        ULONG Size;         // [out]
        ULONG Flags;        // [out]
    };
```

Here's a description of the individual structure members:

`nStructSize:`
: This must be set by you to `sizeof(struct hwos_ExLockStruct)` before calling `hw_NextDirEntry()`.

`Name:`
: This contains the name of the file system object without any path specification. This pointer will be valid until the next call to `hw_NextDirEntry()`.

`Type:`
: This will be set to one of the following types:

  `HWEXLOCKTYPE_FILE:`
  : The file system object is a file.

  `HWEXLOCKTYPE_DIRECTORY:`
  : The file system object is a directory.

`Size:`
: Size of object in bytes if it is a file, otherwise 0.

`Flags:`
: Combination of flags describing the file system object attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle`
: handle returned by `hw_Lock()`

`dirhandle`
: handle returned by `hw_BeginDirScan()`

`exlock`
: pointer to a `struct hwos_ExLockStruct` for storing information about the file system object

RESULTS

`ok`
: `True` if an object could be read, `False` if there are no more objects

25.27 hw_PathPart

NAME

hw_PathPart – return a pointer to the end of the penultimate path component (V5.0)

SYNOPSIS

```
STRPTR p = hw_PathPart(STRPTR path);
```

FUNCTION

This function returns a pointer to the end of the penultimate path component. This is usually a slash or backslash. The idea is that you can write a 0 to the pointer that

is returned by `hw_PathPart()` in order to separate file and path components in a path specification. If there is only one component in the path specification, `hw_PathPart()` will return a pointer to the beginning of the string.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`path` path specification

RESULTS

`p` pointer to the end of the penultimate path component

25.28 hw_Rename

NAME
hw_Rename – rename a file system object (V5.0)

SYNOPSIS
```
int ok = hw_Rename(STRPTR oldname, STRPTR newname);
```

FUNCTION
This function renames the specified file system object to the new name. It can be used to rename either a file or a directory.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`oldname` file system object to rename

`newname` new name for file system object

RESULTS

`ok` `True` on success, `False` otherwise

25.29 hw_Stat

NAME
hw_Stat – examine a file system object (V6.0)

SYNOPSIS
```
int ok = hw_Stat(STRPTR name, ULONG flags, struct hwos_StatStruct *st,
                 struct hwTagList *tags);
```

FUNCTION

This function examines the file system object specified in parameter 1 and writes information about it to the structure pointer passed in parameter 3. `struct hwos_StatStruct` looks like this:

```
struct hwos_StatStruct
{
    int Type;                                   // [out]
    DOSINT64 Size;                              // [out]
    ULONG Flags;                                // [out]
    struct hwos_DateStruct Time;                // [out]
    struct hwos_DateStruct LastAccessTime;      // [out]
    struct hwos_DateStruct CreationTime;        // [out]
    STRPTR FullPath;                            // [out]
    STRPTR Comment;                             // [out]
    int LinkMode;                               // [out]
    STRPTR Container;                           // [out]
};
```

The following information is written to the individual structure members:

`Type:` This will be set to one of the following types:

`HWSTATTYPE_FILE:`
The file system object examined is a file.

`HWSTATTYPE_DIRECTORY:`
The file system object examined is a directory.

`Size:` Size of object in bytes if it is a file, 0 for directories. Note that this can also be set to -1 in case the file size isn't know, for example because the file is being streamed from a network source.

`Flags:` Combination of flags describing the file system object attributes. See Section 2.13 [File attributes], page 16, for a list of supported attributes.

`Time:` Time stamp indicating when this file system object was last modified.

`LastAccessTime:`
Time stamp indicating when this file system object was last accessed.

`CreationTime:`
Time stamp indicating when this file system object was created.

`FullPath:`
Fully qualified path to the file system object. The string pointer used here will stay valid until the next call to `hw_Stat()`. If you set the `HWSTATFLAGS_ALLOCSTRINGS` flag, `hw_Stat()` will not use a static string buffer but allocate a new private string pointer for this structure member. You will have to call `hw_TrackedFree()` on this string when you're done with it in that case. This is useful if you need to use `hw_Stat()` in a multithreaded environment.

`Comment:` Comment stored for this object in the file system. The string pointer returned here will stay valid until the next call to `hw_Stat()`. This may be

NULL if the file system does not support comments for its objects. If you set the HWSTATFLAGS_ALLOCSTRINGS flag, hw_Stat() will not use a static string buffer but allocate a new private string pointer for this structure member. You will have to call hw_TrackedFree() on this string when you're done with it in that case. This is useful if you need to use hw_Stat() in a multithreaded environment.

LinkMode:
Currently unused. May contain random data.

Container:
Currently unused. May contain random data.

The following flags are supported by hw_Stat():

HWSTATFLAGS_NOFILEADAPTER:
If this flag is set, Hollywood will skip all file adapters and use its inbuilt file handler for examining this file system object. Use only if you have a good reason for overriding file adapters.

HWSTATFLAGS_ALLOCSTRINGS:
If you set this flag, hw_Stat() will not use static string buffers for the FullPath and Comment structure members but allocate new private string buffers for them. You will have to call hw_TrackedFree() on these buffers once you're done with them in that case. This flag is useful if you need to use hw_Stat() in a multithreaded environment.

hw_Stat() returns True on success or False on failure.

hw_Stat() can be used to find out whether a certain file system object is a file or a directory or to resolve relative file name specifications into absolute, fully-qualified paths.

This function is only thread-safe if you set the HWSTATFLAGS_ALLOCSTRINGS flag.

DESIGNER COMPATIBILITY
Supported since Designer 4.5

INPUTS

name	name of file system object to examine
flags	combination of flags (see above)
st	pointer to a struct hwos_StatStruct for storing information about the file system object
tags	reserved for future use (pass NULL)

RESULTS

ok	True on success, False otherwise

25.30 hw_TmpNam

NAME

hw_TmpNam – generate temporary file (V5.0)

SYNOPSIS

```
void hw_TmpNam(STRPTR buf);
```

FUNCTION

This function will create a temporary file for you and copy its path to the specified buffer. Make sure that this buffer is at least 4096 bytes in size.

Before Hollywood 6.0, `hw_TmpNam()` only returned a file name and didn't actually create it. Since 6.0 the file is created now to prevent other applications from trying to use a temporary file of the same name.

Note that on AmigaOS and compatibles this function might create a temporary filename in RAM. If you don't want this because you need to write large portions of data to the file, use the new `hw_TmpNamExt()` function. See Section 25.31 [hw_TmpNamExt], page 214, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`buf` pointer to a buffer receiving the file name; must be at least 4096 bytes

25.31 hw_TmpNamExt

NAME

hw_TmpNamExt – generate temporary file with options (V6.0)

SYNOPSIS

```
void hw_TmpNamExt(STRPTR buf, int useram);
```

FUNCTION

This function will create a temporary file for you and copy its path to the specified buffer. Make sure that this buffer is at least 4096 bytes in size. If the argument `useram` is set to `False`, `hw_TmpNamExt()` will never create a file in RAM on AmigaOS and compatibles. On all other systems there is no difference between `hw_TmpNamExt()` and `hw_TmpNam()`.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`buf` pointer to a buffer receiving the file name; must be at least 4096 bytes

`useram` `True` to allow temporary files in RAM, `False` to forbid them; this is only respected on AmigaOS and compatibles

25.32 hw_TranslateFileName

NAME

hw_TranslateFileName – translate a virtual file name (V5.3)

SYNOPSIS

```
int ok = hw_TranslateFileName(STRPTR name, STRPTR buf, int bufsize,
             ULONG *offset, ULONG *size);
```

FUNCTION

This function can be used to translate a virtual file specification into a physical file name. Hollywood supports special virtual file specifications in order to be able to load files that have been linked to other files, for example applets or executables. Only Hollywood functions like `hw_FOpen()` will be able to deal with these special virtual file name specifications transparently. If you pass them to a function like `fopen()` instead, it will fail to open the file. That's why you should always use the functions from `DOSBase` when dealing with files. See Section 2.12 [File IO information], page 15, for details.

If you cannot use the functions from `DOSBase` to do your file IO for some particular reason, you can use `hw_TranslateFileName()` to break down a virtual file specification into a physical one. You'll then be able to open the virtual file using functions like `fopen()` as well. Hollywood's virtual files are always part of a physical file or memory block. `hw_TranslateFileName()` only supports the first type, i.e. virtual files that are part of a physical file. If you want your plugin to be able to handle memory block-based virtual files as well, you will have to use the `hw_TranslateFileNameExt()` function which has been available since Hollywood 6.0.

`hw_TranslateFileName()` will return the name of the physical container file as well as the offset and size of the virtual file within that physical file. For example, there might be a virtual file named `intro.png` inside the physical file `gamedata.bin` at offset 1048576 from the start of the file taking up 65536 bytes inside `gamedata.bin`.

If the specified name does not describe a virtual file, `hw_TranslateFileName()` will return `0xFFFFFFFF` in both `offset` and `size` and simply copy the specified file name to the buffer specified in parameter 2.

Note that this function does not support all features of Hollywood's virtual files. If you need fine-tuned control over virtual file specification analysis, you might want to use `hw_TranslateFileNameExt()` instead. See Section 25.33 [hw_TranslateFileNameExt], page 216, for details.

You can use `hw_ChunkToFile()` to easily save a virtual file to a physical file. See Section 25.4 [hw_ChunkToFile], page 194, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`name`	file name specification containing either a virtual or a physical file
`buf`	memory buffer to receive the physical file
`bufsize`	size of the memory buffer

`offset` pointer to a `ULONG` to receive the offset in bytes where the virtual file starts within the physical file returned in `buf`

`size` pointer to a `ULONG` to receive the size in bytes of the virtual file

RESULTS

`ok` `True` on success, `False` otherwise

25.33 hw_TranslateFileNameExt

NAME

hw_TranslateFileNameExt – translate a virtual file name with extended options (V6.0)

SYNOPSIS

```
int ok = hw_TranslateFileNameExt(STRPTR name, struct hwTranslateFileInfo
             *tf, struct hwTagList *tags);
```

FUNCTION

This function does the same as `hw_TranslateFileName()` but supports additional options. Like `hw_TranslateFileName()`, it can be used to translate a virtual file specification into a physical file name. Hollywood supports special virtual file specifications in order to be able to load files that have been linked to other files, for example applets or executables. Only Hollywood functions like `hw_FOpen()` will be able to deal with these special virtual file name specifications transparently. If you pass them to a function like `fopen()` instead, it will fail to open the file. That's why you should always use the functions from `DOSBase` when dealing with files. See Section 2.12 [File IO information], page 15, for details.

If you cannot use the functions from `DOSBase` to do your file IO for some particular reason, you can use `hw_TranslateFileNameExt()` to break down a virtual file specification into a physical one. You'll have to pass a pointer to a `struct hwTranslateFileInfo` structure to this function. `struct hwTranslateFileInfo` looks like this:

```
struct hwTranslateFileInfo
{
    STRPTR File;       // [in/out]
    int FileLen;       // [in]
    STRPTR FileExt;    // [in/out]
    int FileExtLen;    // [in]
    STRPTR RealFile;   // [in/out]
    int RealFileLen;   // [in]
    APTR MemoryBlock;  // [out]
    DOSINT64 Offset;   // [out]
    DOSINT64 Length;   // [out]
};
```

Here's a description of the individual structure members:

`File:` If this is non-`NULL`, Hollywood will copy the name of the virtual file to this string buffer. You will also have to provide the size of this buffer in the `FileLen` argument.

`FileLen:` If `File` is non-`NULL`, you'll have to set this member to the size of the buffer passed in `File`.

`FileExt:` If this is non-`NULL`, Hollywood will copy the extension of the virtual file to this string buffer. You will also have to provide the size of this buffer in the `FileExtLen` argument.

`FileExtLen:`
If `FileExt` is non-`NULL`, you'll have to set this member to the size of the buffer passed in `FileExt`.

`RealFile:`
If this is non-`NULL`, Hollywood will copy the name of the file that contains the virtual file to this string buffer. For example, there might be a virtual file named `intro.png` inside the physical file `gamedata.bin` at offset 1048576 from the start of the file taking up 65536 bytes inside `gamedata.bin`. In case the virtual file doesn't have a container file but is stored within a memory block, Hollywood will set this member to an empty string and will store a pointer to the memory block that contains the virtual file in `MemoryBlock`. Note that if you set `RealFile`, you will also have to provide the size of the buffer in the `RealFileLen` argument.

`RealFileLen:`
If `RealFile` is non-`NULL`, you'll have to set this member to the size of the buffer passed in `RealFile`.

`MemoryBlock:`
In case the virtual file doesn't have a container file but is stored within a memory block, Hollywood will set this structure member to a pointer to the memory block that contains the virtual file.

`Offset:` Hollywood will store the offset of the virtual file inside the container file here. This is always 0 in case the virtual file is memory-based.

`Length:` Hollywood will store the length of the virtual file here.

You can use `hw_ChunkToFile()` to easily save a virtual file to a physical file. See Section 25.4 [hw_ChunkToFile], page 194, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`name` file name specification containing a virtual file

`tf` pointer to a `struct hwTranslateFileInfo` initialized as described above

`tags` reserved for future use, pass `NULL` for now

RESULTS

`ok` `True` on success, `False` otherwise

25.34 hw_UnLock

NAME

hw_UnLock – close a file system object lock (V6.0)

SYNOPSIS

```
void hw_UnLock(APTR handle);
```

FUNCTION

This function closes the specified file system object lock allocated by `hw_Lock()`.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` object handle returned by `hw_Lock()`

26 FontBase functions

26.1 Overview

`FontBase` contains functions to work with font files.

`FontBase` is available since Hollywood 5.0.

26.2 hw_FindTTFFont

NAME

hw_FindTTFFont – find a TrueType font file (V5.0)

SYNOPSIS

```
STRPTR file = hw_FindTTFFont(STRPTR name, int weight, int slant, int
                  fileonly, int *offset, int *len, int *tmp);
```

FUNCTION

This function can be used to find a matching TrueType font file for the specified font attributes. You have to pass a font family name in parameter 1. Additionally, you have to pass one of the following font weight constants in parameter 2:

```
#define HWFONTWEIGHT_THIN       0
#define HWFONTWEIGHT_EXTRALIGHT 40
#define HWFONTWEIGHT_ULTRALIGHT 40
#define HWFONTWEIGHT_LIGHT      50
#define HWFONTWEIGHT_BOOK       75
#define HWFONTWEIGHT_NORMAL     80
#define HWFONTWEIGHT_REGULAR    80
#define HWFONTWEIGHT_MEDIUM     100
#define HWFONTWEIGHT_SEMIBOLD   180
#define HWFONTWEIGHT_DEMIBOLD   180
#define HWFONTWEIGHT_BOLD       200
#define HWFONTWEIGHT_EXTRABOLD  205
#define HWFONTWEIGHT_ULTRABOLD  205
#define HWFONTWEIGHT_HEAVY      210
#define HWFONTWEIGHT_BLACK      210
#define HWFONTWEIGHT_EXTRABLACK 215
#define HWFONTWEIGHT_ULTRABLACK 215
```

If you pass -1 in the `weight` argument, `hw_FindTTFFont()` will ignore both the `weight` and the `slant` parameters. The `slant` parameter may be set to one of the following special constants:

```
#define HWFONTSLANT_ROMAN       0
#define HWFONTSLANT_ITALIC      100
#define HWFONTSLANT_OBLIQUE     110
```

If the `fileonly` parameter is set to `True`, `hw_FindTTFFont()` will only return stand-alone TrueType font files. If `fileonly` is set to `False`, `hw_FindTTFFont()` might also return TrueType fonts that have been linked into other files. In that case, `hw_FindTTFFont()`

will write the TrueType font's location and its size inside the container file to the `offset` and `len` integer pointers that you have to pass as parameters 5 and 6.

Finally, if `fileonly` is `False`, `hw_FindTTFFont()` might also extract a TrueType font from a TTC file, write it to a temporary file and return this temporary file to you. If that is the case, `hw_FindTTFFont()` will write `True` to the `tmp` integer pointer that is the last parameter of this function. So if you find out that `hw_FindTTFFont()` has written `True` to the integer pointer you've passed as the last parameter, you will have to delete the temporary file when you're done with it.

INPUTS

`name` family name of the font file to find

`weight` desired font weight or -1 (see above)

`slant` desired font slant (see above)

`fileonly` set this to `True` if you only want to have stand-alone files and no files-within-files or temporary TrueType fonts extracted by Hollywood (see above)

`offset` must be set to an integer pointer that receives the offset of linked fonts in bytes

`len` must be set to an integer pointer that receives the length of linked fonts in bytes

`tmp` must be set to an integer pointer that is set to `True` or `False` depending on whether this function has created a temporary file

DESIGNER COMPATIBILITY

Supported since Designer 4.0

RESULTS

`file` fully-qualified path to a TrueType font file that matches your request or `NULL` if no font could be found

27 FT2Base functions

27.1 Overview

`FT2Base` contains most functions from the FreeType2 library. If you want to use any of these functions, you'll need the FreeType2 header files. Make sure to use compatible header files only. Hollywood uses version 2.3.12 of FreeType2 so you need to use the header files from exactly this version if you plan to use functions from the `FT2Base` library.

`FT2Base` is available since Hollywood 5.0. The WarpOS-native version of `FT2Base`, however, was added in Hollywood 5.3. So if you are developing a WarpOS plugin and want to make calls into `FT2Base`, you have to check for Hollywood 5.3 first.

Please consult your FreeType2 documentation for information on the individual functions supported by `FT2Base`.

The implementations of `FT2Base` in Hollywood Designer and Hollywood are identical.

28 GfxBase functions

28.1 Overview

`GfxBase` contains a number of functions to deal with the graphics-oriented functionality of Hollywood.

`GfxBase` is available since Hollywood 5.0.

28.2 hw_AddBrush

NAME

hw_AddBrush – create a new brush (V5.3)

SYNOPSIS

```
int error = hw_AddBrush(lua_State *L, lua_ID *id, int width, int height,
                struct hwAddBrush *ctrl);
```

FUNCTION

This function can be used to create a new brush and make it available to the Hollywood script. You have to pass the desired object identifier for the brush as a `lua_ID`. See Section 2.17 [Object identifiers], page 21, for details. Additionally, you have to specify the brush's width and height as well as a pointer to a `struct hwAddBrush` which contains further information. `struct hwAddBrush` looks like this:

```
struct hwAddBrush
{
    ULONG *Data;
    int LineWidth;
    ULONG Transparency;
    ULONG Flags;
    APTR Image;
    ULONG *(*GetImage)(APTR handle, struct LoadImageCtrl *ctrl);
    void (*FreeImage)(APTR handle);
    int (*TransformImage)(APTR handle, struct hwMatrix2D *m,
                          int width, int height);
};
```

You need to provide the following information in this structure:

`Data:` This is only used if you want to create a raster brush, i.e. the `HWABFLAGS_VECTORBRUSH` flag is not set. In that case, you can set this structure member to a pointer to an array of 32-bit ARGB pixels that contain the image data for the new brush. `hw_AddBrush()` will only take the alpha byte into account if the `HWABFLAGS_USEALPHA` flag has been set. Otherwise the alpha byte is ignored. The pixel array specified here must contain exactly as many pixels per row as passed in the `LineWidth` member. If you set `Data` to `NULL` and `HWABFLAGS_VECTORBRUSH` isn't set, `hw_AddBrush()` will create an uninitialized raster brush for you, i.e. it will be filled with random pixel data.

`LineWidth:`
: This must only be set if you want to create a raster brush and `Data` is not `NULL`. In that case, you have to set this structure member to the number of pixels per row in the `Data` pixel array. This can be different from the value passed in the `width` parameter of `hw_AddBrush()` in case the pixel array you specified in `Data` contains some padding bytes. Please note that `LineWidth` must be specified in pixels, not in bytes.

`Transparency:`
: If the `HWABFLAGS_USETRANSPARENCY` flag has been set, this member contains a 24-bit RGB color that should be made transparent. Hollywood will scan through the pixel array passed in `Data` and create a monochrome mask for the new brush in which all pixels which match the RGB color specified here are transparent. This member is only supported for raster brushes.

`Flags:`
: This member controls several attributes for the new brush. It can be set to a combination of the following flags:

  `HWABFLAGS_USEALPHA:`
  : The brush uses alpha channel transparency. If this flag is set, the pixel array specified in `Data` has to contain transparency information in the alpha byte. If a vector brush is created, your `GetImage()` implementation must return transparency information in the alpha byte as well. If this flag is not set, `hw_AddBrush()` will ignore whatever is in the alpha byte. This flag and `HWABFLAGS_USETRANSPARENCY` are mutually exclusive.

  `HWABFLAGS_USETRANSPARENCY:`
  : This is only supported if you create a raster brush and a pixel array has been passed in `Data`. In that case, setting this flag indicates that you want `hw_AddBrush()` to create a mask in which all pixels that match the color specified in `Transparency` are made transparent. That is why you also need to set the `Transparency` member if you set this flag. This flag and `HWABFLAGS_USEALPHA` are mutually exclusive.

  `HWABFLAGS_VECTORBRUSH:`
  : If this flag is set, `hw_AddBrush()` will create a vector brush for you. Vector brushes can be transformed without any quality loss and whenever the script wants to have a vector brush scaled, rotated, or transformed, Hollywood will call into your plugin to apply this transformation to your vector brush. That is why you need to provide several callbacks when creating a vector brush (see below).

`Image:`
: This must only be set if you want to create a vector brush. In that case, it must be set to a handle that you want Hollywood to pass to your vector brush callbacks whenever it needs something done.

`GetImage:`
: This must only be set if you want to create a vector brush. In that case, it must be set to a callback function that returns the raw pixel data of the vector brush. The function that you specify here has to work exactly like the `GetImage()` function of image plugins. See Section 13.3 [GetImage], page 117, for details.

`FreeImage:`
: This must only be set if you want to create a vector brush. In that case, it must be set to a callback function that frees any data that your plugin has allocated for your vector brush. The function that you specify here has to work exactly like the `FreeImage()` function of image plugins. See Section 13.2 [FreeImage], page 117, for details.

`TransformImage:`
: This must only be set if you want to create a vector brush. In that case, it must be set to a callback function that applies transformation to a vector brush. The function that you specify here has to work exactly like the `TransformImage()` function of image plugins. See Section 13.6 [TransformImage], page 122, for details.

You can free brushes by calling the `hw_FreeBrush()` function.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L`	pointer to the `lua_State`
`id`	object identifier for the new brush
`width`	desired pixel width for the new brush
`height`	desired pixel height for the new brush
`ctrl`	pointer to a `struct hwAddBrush` containing additional information

RESULTS

`error`	error code or 0 on success

28.3 hw_AttachDisplaySatellite

NAME

hw_AttachDisplaySatellite – create a new display satellite (V5.2)

SYNOPSIS

```
APTR handle = hw_AttachDisplaySatellite(lua_ID *id, int (*dispatcher)
                  (APTR handle, int op, APTR opdata, APTR userdata),
                  APTR userdata, struct hwTagList *tags);
```

FUNCTION

This function can be used to attach a new satellite to the specified display. A display satellite is an object which receives all the graphics output that is being done to its

root display so that the satellite always mirrors the graphics of its root display. The satellite can then choose what to do with these graphics: It could save them to a file, upload them to the internet or another application, display them using a custom device, etc. There are many possible use cases for display satellites. A big advantage of display satellites is that they also work while their root display is hidden. This makes it possible to create some sort of light display adapter using satellites by hiding the display opened by Hollywood and using a custom display managed by the satellite instead. In this case, however, Hollywood will still run its inbuilt event processor so you cannot switch to entirely different toolkits as you can do with a display adapter. See Section 10.1 [Display adapter plugins], page 59, for details. Furthermore, the satellite can also post events to its root display. This is done by calling the `hw_PostSatelliteEvent()` function. See Section 34.28 [hw_PostSatelliteEvent], page 303, for details.

You have to pass the object identifier of the display the new satellite shall attach to. The object identifier must be passed as a `lua_ID`. See Section 2.17 [Object identifiers], page 21, for details.

You also have to pass a pointer to a dispatcher function which will be called whenever Hollywood draws something to the satellite's root display. In that case, Hollywood will first draw to the root display and then immediately call your satellite's dispatcher so that it is informed about the draw operation. The prototype of this dispatcher function looks like this:

```
int dispatcher(APTR handle, int op, APTR opdata, APTR userdata);
```

Hollywood will pass a handle to the display satellite in the first parameter and it will pass the user data that you specified in your call to `hw_AttachDisplaySatellite()` in the fourth parameter. Parameters 2 and 3 contain the information about the operation that Hollywood wants your satellite to execute. The data passed in `opdata` depends on the actual operation passed in parameter 2. The following operations are currently recognized:

`HWSATOP_BLTBITMAP:`
: Hollywood wants your satellite to blit a bitmap to its graphics buffer. `opdata` will get a pointer to a `struct hwSatelliteBltBitMap` which contains all information you need to do the blit operation. `struct hwSatelliteBltBitMap` looks like this:

```
struct hwSatelliteBltBitMap
{
    APTR BitMap;        // [in]
    int BitMapType;     // [in]
    int BitMapWidth;    // [in]
    int BitMapHeight;   // [in]
    int BitMapModulo;   // [in]
    int BitMapPixFmt;   // [in]
    UBYTE *Mask;        // [in]
    int MaskModulo;     // [in]
    int SrcX;           // [in]
    int SrcY;           // [in]
    int DstX;           // [in]
```

```
        int DstY;           // [in]
        int Width;          // [in]
        int Height;         // [in]
    };
```

The structure members will be initialized as follows:

`BitMap:` This will be set to a pointer to the bitmap that shall be blitted. The actual type of the bitmap specified here is specified in the `BitMapType` member (see below).

`BitMapType:`
This contains the type of the bitmap pointer passed in the `BitMap` member. The following types are currently supported:

`HWSATBMTYPE_AMIGABITMAP:`
`HWSATBMTYPE_AMIGABITMAP` indicates that the bitmap is an AmigaOS bitmap, i.e. a `struct BitMap` allocated by `graphics.library/AllocBitMap()`. This can only happen on AmigaOS based systems.

`HWSATBMTYPE_PIXELBUFFER:`
The bitmap is a raw pixel buffer. The actual format of the raw pixels is specified in the `BitMapPixFmt` structure member.

`HWSATBMTYPE_VIDEOBITMAP:`
The bitmap is a video bitmap allocated by your plugin's `AllocVideoBitMap()` function. (V6.0)

`HWSATBMTYPE_BITMAP:`
The bitmap is a Hollywood bitmap. Use `hw_LockBitMap()` to access its pixels. (V6.0)

`BitMapWidth:`
Contains the bitmap's width in pixels.

`BitMapHeight:`
Contains the bitmap's height in pixels.

`BitMapModulo:`
Contains the bitmap's modulo width in pixels, i.e. the pixel of one row of image data. This is often more than what is passed in `BitMapWidth` because row padding is used. `BitMapModulo` only contains a meaningful value if `BitMapType` has been set to `HWSATBMTYPE_PIXELBUFFER`.

`BitMapPixFmt:`
Contains the pixel format of the raw pixels passed in the `BitMap` structure member. This only contains a meaningful value if `BitMapType` has been set to `HWSATBMTYPE_PIXELBUFFER`. See Section 2.15 [Pixel format information], page 18, for a list of pixel formats.

`MaskData:`

If this does not equal `NULL`, Hollywood wants you to take this mask into account when blitting. `MaskData` points to an array of raw mask bits then (1 bit per pixel). This array matches the size of the bitmap passed in the `BitMap` member. Hollywood masks only know two different states: visible (1) and invisible (0) pixels. The bits are stored from left to right in chunks of one byte, i.e. the most significant bit of the first byte describes the transparency setting of the first pixel. The number of bytes per row is stored in the `MaskModulo` member (see below).

`MaskModulo:`

If `MaskData` contains a mask pointer, this member will be set to the number of bytes that is used for one row of mask data. Note that this value is specified in bytes and often contains some padding.

`SrcX:` Contains the source x-offset of the blit operation.

`SrcY:` Contains the source y-offset of the blit operation.

`DstX:` Contains the destination x-offset of the blit operation.

`DstY:` Contains the destination y-offset of the blit operation.

`Width:` Contains the number of columns to blit.

`Height:` Contains the number of rows to blit.

Please note that you do not have to do any clipping. Hollywood will clip all coordinates against your satellite root display's boundaries before invoking your dispatcher.

`HWSATOP_RECTFILL:`

Hollywood wants your satellite to draw a rectangle to its graphics buffer. `opdata` will get a pointer to a `struct hwSatelliteRectFill` which contains all information you need to do this operation. `struct hwSatelliteRectFill` looks like this:

```
struct hwSatelliteRectFill
{
    int X;
    int Y;
    int Width;
    int Height;
    ULONG Color;
};
```

The structure members will be initialized as follows:

`X:` Start x-offset of the rectangle to fill.

`Y:` Start y-offset of the rectangle to fill.

`Width:` Width in pixels of the area to fill.

`Height:` Height in pixels of the area to fill.

`Color:` Filling color specified as a 24-bit RGB value.

Please note that you do not have to do any clipping. Hollywood will clip all coordinates against your satellite root display's boundaries before invoking your dispatcher.

`HWSATOP_LINE:`

Hollywood wants your satellite to draw a line to its graphics buffer. `opdata` will get a pointer to a `struct hwSatelliteLine` which contains all information you need to do this operation. `struct hwSatelliteLine` looks like this:

```
struct hwSatelliteLine
{
    int X1;
    int Y1;
    int X2;
    int Y2;
    ULONG Color;
};
```

The structure members will be initialized as follows:

`X1:` Start x-offset for the line.

`Y1:` Start y-offset for the line.

`X2:` End x-offset for the line.

`Y2:` End y-offset for the line.

`Color:` Desired line color specified as a 24-bit RGB value.

Please note that you do not have to do any clipping. Hollywood will clip all coordinates against your satellite root display's boundaries before invoking your dispatcher.

`HWSATOP_WRITEPIXEL:`

Hollywood wants your satellite to draw a single pixel to its graphics buffer. `opdata` will get a pointer to a `struct hwSatelliteWritePixel` which contains all information you need to do this operation. `struct hwSatelliteWritePixel` looks like this:

```
struct hwSatelliteWritePixel
{
    int X;
    int Y;
    ULONG Color;
};
```

The structure members will be initialized as follows:

`X:` Pixel's x-offset.

`Y:` Pixel's y-offset.

`Color:` Pixel color specified as a 24-bit RGB value.

Please note that you do not have to do any clipping. Hollywood will clip all coordinates against your satellite root display's boundaries before invoking your dispatcher.

`HWSATOP_RESIZE:`

Hollywood wants your display satellite to resize. `opdata` will get a pointer to a `struct hwSatelliteResize` which contains all information you need to do this operation. `struct hwSatelliteResize` looks like this:

```
struct hwSatelliteResize
{
    int Width;
    int Height;
};
```

The structure members will be initialized like this:

`Width:` This member contains the new display satellite width in pixels.

`Height:` This member contains the new display satellite height in pixels.

`HWSATOP_VWAIT:`

This opcode is only sent if your display satellite has explicitly requested to be notified whenever its root display is asked to wait for the vertical blank interrupt by setting the `HWADS_DISPATCHVWAIT` tag to `True`. If that is the case, you will receive this opcode whenever a vertical blank wait is executed on the satellite's root display. This notification can come in handy in case the root display is hidden and doesn't execute any vertical blank waits. You could then do this job in your satellite dispatcher to prevent Hollywood from running too fast. (V6.0)

Finally, `hw_AttachDisplaySatellite()` also accepts a tag list which allows you to configure some further options. The following tags are currently recognized:

`HWADS_WIDTH:`

If you specify this tag, you need to set its `pData` member to a pointer to an `int`. `hw_AttachDisplaySatellite()` will then write the root display's pixel width to this `int`.

`HWADS_HEIGHT:`

If you specify this tag, you need to set its `pData` member to a pointer to an `int`. `hw_AttachDisplaySatellite()` will then write the root display's pixel height to this `int`.

`HWADS_DISPATCHVWAIT:`

This tag allows you to control whether or not you want to be notified when the root display waits for the vertical blank. By default, you won't be notified about this event but if you set the `iData` member of this tag to `True`, Hollywood will dispatch the `HWSATOP_VWAIT` operation to your satellite dispatcher whenever its root display waits for the vertical blank interrupt (see above). (V6.0)

`HWADS_OPTIMIZEDREFRESH:`
Set this tag to `True` to force optimized refresh of this display satellite's parent. See Section 10.32 [OpenDisplay], page 87, for more information on optimized refresh. (V6.1)

Note that Hollywood versions prior to 6.0 did not check the tag list pointer against `NULL` so make sure to pass a tag list even if there are no tags in it.

To detach your satellite from its root display, call the `hw_DetachDisplaySatellite()` function. See Section 28.6 [hw_DetachDisplaySatellite], page 233, for details. The user won't be able to free the root display until all satellites have been detached.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`id`	object identifier of the display you want to attach to
`dispatcher`	function to handle satellite actions (see above)
`userdata`	userdata to be passed to the dispatcher function
`tags`	taglist containing additional arguments; should not be `NULL` (see above)

RESULTS

`handle`	handle to the display satellite or `NULL` on error

28.4 hw_BitMapToARGB

NAME
hw_BitMapToARGB – convert Hollywood bitmap to ARGB pixel buffer (V6.0)

SYNOPSIS
```
ULONG *rgb = hw_BitMapToARGB(APTR bmap, APTR mask, APTR alpha,
                 struct hwTagList *tags);
```

FUNCTION
This function can be used to get the pixels of a Hollywood bitmap as a 32-bit ARGB pixel buffer. Optionally, this function can also take an additional mask or alpha channel bitmap into account and combine its transparency information into the pixel buffer it returns. The advantage over functions like `hw_LockBitMap()` is that `hw_BitMapToARGB()` will convert the pixels to the 32-bit ARGB format automatically so you don't have to be able to deal with dozens of different pixel formats. This conversion, however, means overhead which makes `hw_BitMapToARGB()` slower than `hw_LockBitMap()`.

`hw_BitMapToARGB()` accepts a tag list that allows you to fine-tune its behaviour. The following tags are currently recognized:

`HWBM2ARGBTAG_X:`
If you set the `iData` member of this tag to an x-offset within your bitmap boundaries, `hw_BitMapToARGB()` will start fetching pixels at this offset. This tag allows you to convert only a part of the bitmap to a pixel buffer. Defaults to 0.

`HWBM2ARGBTAG_Y:`
: If you set the `iData` member of this tag to an y-offset within your bitmap boundaries, `hw_BitMapToARGB()` will start fetching pixels at this offset. This tag allows you to convert only a part of the bitmap to a pixel buffer. Defaults to 0.

`HWBM2ARGBTAG_WIDTH:`
: If you only want to convert a part of the bitmap to an ARGB pixel buffer, set the `iData` member of this tag to the number of columns to convert. Defaults to bitmap width.

`HWBM2ARGBTAG_HEIGHT:`
: If you only want to convert a part of the bitmap to an ARGB pixel buffer, set the `iData` member of this tag to the number of rows to convert. Defaults to bitmap height.

Please note that alpha byte will always be set, even if you didn't pass a mask or alpha channel bitmap. In that case the alpha byte for every pixel will be set to 255, i.e. fully opaque.

The memory buffer that is returned by this function must be freed by using the `hw_TrackedFree()` function. See Section 34.44 [hw_TrackedFree], page 328, for details.

Note that `hw_BitMapToARGB()` can only be used with software bitmaps. It is not possible to get the raw pixels of hardware (video) bitmaps.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`bmap`	source bitmap
`mask`	mask bitmap or `NULL`
`alpha`	alpha channel bitmap or `NULL`
`tags`	pointer to a taglist specifying additional options (see above)

RESULTS

`rgb`	raw 32-bit ARGB pixel buffer or `NULL` on error

28.5 hw_ChangeRootDisplaySize

NAME

hw_ChangeRootDisplaySize – change size of satellite's root display (V5.2)

SYNOPSIS

```
int error = hw_ChangeRootDisplaySize(APTR handle, int width, int height,
                struct hwTagList *tags);
```

FUNCTION

This function can be used to force a size change of the satellite's root display. Please note that this will not trigger a `HWSATOP_RESIZE` operation for your satellite's dispatcher. It is

assumed that the satellite has already been resized when it calls this function. Calling `hw_ChangeRootDisplaySize()`, however, will trigger a satellite refresh so your dispatcher will get some drawing events like `HWSATOP_BLTBITMAP`.

See Section 28.3 [hw_AttachDisplaySatellite], page 225, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle`	satellite handle allocated by `hw_AttachDisplaySatellite()`
`width`	desired new width for root display in pixels
`height`	desired new height for root display in pixels
`tags`	reserved for future use; pass `NULL` for the time being

RESULTS

`error`	error code or 0 on success

28.6 hw_DetachDisplaySatellite

NAME

hw_DetachDisplaySatellite – detach satellite from display (V5.2)

SYNOPSIS

```
void hw_DetachDisplaySatellite(APTR handle);
```

FUNCTION

This function can be used to detach the specified satellite handle from its root display. After that the satellite's dispatcher function will no longer be called. See Section 28.3 [hw_AttachDisplaySatellite], page 225, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle`	satellite handle allocated by `hw_AttachDisplaySatellite()`

28.7 hw_FindDisplay

NAME

hw_FindDisplay – find display handle or identifier (V6.0)

SYNOPSIS

```
APTR rhandle = hw_FindDisplay(lua_ID *id, APTR handle);
```

FUNCTION

This function can be used to get a display handle from an object identifier or an object identifier from a display handle. If you pass a display handle in the second parameter,

`hw_FindDisplay()` will return its object identifier in the `lua_ID` passed in the first parameter. If the second parameter is `NULL`, `hw_FindDisplay()` will return the display handle that matches the specified object identifier. See Section 2.17 [Object identifiers], page 21, for details.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`lua_ID` pointer to a `lua_ID` containing or receiving an object identifier

`handle` handle to a display or `NULL`

RESULTS

`rhandle` display handle

28.8 hw_FreeARGBBrush

NAME
hw_FreeARGBBrush – free raw brush pixels (V5.2)

SYNOPSIS

```
void hw_FreeARGBBrush(ULONG *buffer);
```

FUNCTION
This function must be used to free the pixel buffer returned by `hw_GetARGBBrush()`. See Section 28.12 [hw_GetARGBBrush], page 236, for details.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`buffer` pixel buffer allocated by `hw_GetARGBBrush()`

28.9 hw_FreeBrush

NAME
hw_FreeBrush – free a brush (V6.1)

SYNOPSIS

```
int error = hw_FreeBrush(lua_State *L, lua_ID *id);
```

FUNCTION
This function frees the Hollywood brush specified by `id`. All memory occupied by the brush will be released. You can create brushes from the Hollywood script or by calling the `hw_AddBrush()` function.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`L` pointer to the `lua_State`

`id` object identifier of the brush to free

RESULTS

`error` error code or 0 on success

28.10 hw_FreeIcons

NAME

hw_FreeIcons – free application icon list (V6.0)

SYNOPSIS

```
void hw_FreeIcons(struct hwIconList *list);
```

FUNCTION

This function frees an application icon list allocated by `hw_GetIcons()`. See Section 28.15 [hw_GetIcons], page 239, for details.

Note that this function must not be used to free individual icons. You must always free the complete list.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`list` icon list allocated by `hw_GetIcons()`

28.11 hw_FreeImage

NAME

hw_FreeImage – free image handle (V5.0)

SYNOPSIS

```
void hw_FreeImage(APTR handle);
```

FUNCTION

This function frees an image handle returned by `hw_LoadImage()`. See Section 28.19 [hw_LoadImage], page 242, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` image handle returned by `hw_LoadImage()`

28.12 hw_GetARGBBrush

NAME

hw_GetARGBBrush – get raw brush pixels (V5.2)

SYNOPSIS

```
ULONG *rgb = hw_GetARGBBrush(lua_ID *id, struct hwTagList *tags);
```

FUNCTION

This function can be used to get a copy of the specified brush's pixels. In contrast to `hw_LockBrush()` `hw_GetARGBBrush()` will convert the pixels to the 32-bit ARGB format automatically and it will also mix any potential alpha channel or monochrome transparency mask into the pixel map. All this is of course overhead which makes `hw_GetARGBBrush()` slower than `hw_LockBrush()`. You have to pass the object identifier of the brush whose pixels you want to obtain. The object identifier must be passed as a `lua_ID`. See Section 2.17 [Object identifiers], page 21, for details.

Additionally, you can specify a taglist in the second parameter. The following tags are currently recognized:

`HWGAB_WIDTH:`
: If you specify this tag, you must set its `pData` member to a pointer to an `int`. `hw_GetARGBBrush()` will then write the brush's width in pixels to this `int`.

`HWGAB_HEIGHT:`
: If you specify this tag, you must set its `pData` member to a pointer to an `int`. `hw_GetARGBBrush()` will then write the brush's height in pixels to this `int`.

`HWGAB_OPAQUE:`
: If you specify this tag, you must set its `pData` member to a pointer to an `int`. `hw_GetARGBBrush()` will then write `True` to this `int` if the brush doesn't have a mask or an alpha channel, or `False` otherwise.

Note that currently you always have to pass a taglist to this function as it does not check against `NULL`. So just pass an empty taglist if you don't need to use any of the tags above.

Please note that alpha byte will always be set, even if the brush doesn't have any transparency information. In that case the alpha byte for every pixel will be set to 255, i.e. fully opaque.

The memory buffer that is returned by this function must be freed by using the `hw_FreeARGBBrush()` function. See Section 28.8 [hw_FreeARGBBrush], page 234, for details.

Note that `hw_GetARGBBrush()` can only be used with software brushes. It is not possible to get the raw pixels of hardware brushes.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`id` object identifier of brush whose pixels you want to get

`tags` pointer to a taglist specifying additional options (see above); this must not be `NULL`

RESULTS

`rgb` raw 32-bit ARGB pixel buffer or `NULL` on error

28.13 hw_GetBitMapAttr

NAME

hw_GetBitMapAttr – query bitmap attribute (V6.0)

SYNOPSIS

```
int v = hw_GetBitMapAttr(APTR handle, int attr, struct hwTagList *tags);
```

FUNCTION

This function returns the requested information about the specified bitmap. The `attr` parameter specifies which information you want to have. The following attributes are currently recognized:

`HWBMATTR_WIDTH:`
Return the bitmap's width in pixels.

`HWBMATTR_HEIGHT:`
Return the bitmap's height in pixels.

`HWBMATTR_BYTESPERROW:`
Return the bitmap's bytes per row.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` handle to a bitmap

`attr` attribute to query (see above)

`tags` reserved for future use; pass `NULL`

RESULTS

`v` value of the attribute

28.14 hw_GetDisplayAttr

NAME

hw_GetDisplayAttr – query display attribute(s) (V6.0)

SYNOPSIS

```
int c = hw_GetDisplayAttr(APTR handle, struct hwTagList *tags);
```

FUNCTION

This function can be used to query the current state of one or more attributes from the specified display. You have to pass a handle to the display as well as a tag list. The following tags are currently recognized:

`HWDISPATTR_RAWWIDTH:`
: This tag will return the display's raw width, i.e. the physical width of the display on the screen in pixels. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_RAWHEIGHT:`
: This tag will return the display's raw height, i.e. the physical height of the display on the screen in pixels. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_BUFFERWIDTH:`
: This tag will return the width of the display's back buffer. This can be different from `HWDISPATTR_RAWWIDTH` in case autoscaling is active. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_BUFFERHEIGHT:`
: This tag will return the height of the display's back buffer. This can be different from `HWDISPATTR_RAWHEIGHT` in case autoscaling is active. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_FLAGS:`
: This tag will return all flags that are set for this display. See Section 10.32 [OpenDisplay], page 87, for a list of flags. The return value will be written to the `pData` member of this tag. You must set `pData` to an `ULONG` pointer for this purpose.

`HWDISPATTR_SCALEWIDTH:`
: This tag will return the current scale width set for the display. Note that a value different than 0 here doesn't automatically mean that auto or layer scaling is active. You still need to check the respective flag to tell that scaling is active. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_SCALEHEIGHT:`
: This tag will return the current scale height set for the display. Note that a value different than 0 here doesn't automatically mean that auto or layer scaling is active. You still need to check the respective flag to tell that scaling is active. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_SCALEMODE:`
: This tag will return the current scale mode set for the display. Currently supported are 0 for hard scaling and 1 for interpolated scaling. The return value will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose.

`HWDISPATTR_CANDROPFILE:`
: This tag will return a boolean value indicating whether or not the display supports dropping files on it. The return value will be either `True` or `False` and it will be written to the `pData` member of this tag. You must set `pData` to an `int` pointer for this purpose. (V7.0)

`HWDISPATTR_USESATELLITE:`
: If the `iData` member of this tag is set to `True`, Hollywood will assume that the `handle` parameter passed to `hw_GetDisplayAttr()` is a display satellite handle allocated by `hw_AttachDisplaySatellite()` instead of a normal display handle. `hw_GetDisplayAttr()` will then query the specified attributes from the satellite's root display. This tag allows you to query attributes from a display through a display satellite handle. (V7.0)

`hw_GetDisplayAttr()` returns the number of attributes successfully handled.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` handle to a display

`tags` pointer to a taglist specifying the attributes to query

RESULTS

`c` number of attributes successfully queried

28.15 hw_GetIcons

NAME

hw_GetIcons – get application icons (V6.0)

SYNOPSIS

```
struct hwIconList *list = hw_GetIcons(void);
```

FUNCTION

This function returns a list of all application icons that are currently available. This list may contain user-defined icons that have been specified using the `@APPICON` preprocessor command, inbuilt default Hollywood icons or icons that have been linked to an applet or executable.

This function is especially useful for display adapters which redirect Hollywood's output to a custom display device and want to register the script's icons with this custom display device or toolkit. By calling `hw_GetIcons()` display adapters can easily obtain a list with all icons currently defined by the script. `hw_GetIcons()` will return a pointer to a `struct hwIconList` which looks like this:

```
struct hwIconList
{
    struct hwIconList *Succ;
    ULONG *Data;
```

```
        int Width;
        int Height;
        ULONG Flags;
    };
```

For each node in the list, `struct hw_IconList` will be initialized as follows:

`Succ:` Contains a pointer to the next list node or `NULL` if this node is the last one.

`Data:` This will be set to a 32-bit ARGB pixel buffer containing the icon's image data. The alpha byte will always be set for every pixel. The pixel buffer's size will be exactly `width * height * 4`. No row padding will be used. The pixel buffer pointer will be valid until you call `hw_FreeIcons()`.

`Width:` Contains the icon's width.

`Height:` Contains the icon's height.

`Flags:` This contains a combination of flags for this icon. The following flags are currently supported:

`HWICONFLAGS_DEFAULT:`
This flag marks the default icon. Hollywood's `@APPICON` preprocessor command allows scripts to designate an icon as the default one. It's up to you how you interpret and handle the default icon. On most systems you can probably ignore this flag because icons are chosen depending on the current screen resolution.

`HWICONFLAGS_SELECTED:`
If this flag is set, the image in this list node describes a selected icon state. On AmigaOS and compatibles, icons usually have two states: normal and selected. You can get the image of the selected icon state by checking if this flag is set.

The list that is returned by this function must be freed using the `hw_FreeIcons()` function. See Section 28.10 [hw_FreeIcons], page 235, for details.

Do not expect this list to be sorted. The individual icons can be stored in an order that is completely random inside this list.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

none

RESULTS

`list` a list containing all available icons

28.16 hw_GetImageData

NAME

hw_GetImageData – get image pixel data (V5.0)

SYNOPSIS

```
ULONG *rgb = hw_GetImageData(APTR handle);
```

FUNCTION

This function returns the raw 32-bit pixel data of the image handle allocated by `hw_LoadImage()`. The alpha byte will always be set, even if the image doesn't contain an alpha channel. In that case the alpha byte will be set to 255 (i.e. fully opaque) for every pixel. The pointer returned by this function is valid until you call `hw_FreeImage()` on this image.

The pixel format can either be RGBA or ARGB, depending on the parameters passed to `hw_LoadImage()`.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`handle` image handle returned by `hw_LoadImage()`

RESULTS

`rgb` pointer to the raw 32-bit RGBA or ARGB pixel data

28.17 hw_GetRGB

NAME

hw_GetRGB – convert pixel format to RGB (V6.0)

SYNOPSIS

```
ULONG rgb = hw_GetRGB(ULONG color, int infmt);
```

FUNCTION

This function converts the specified color from the specified pixel format to RGB. See Section 2.15 [Pixel format information], page 18, for details.

To convert a color from RGB to an arbitrary pixel format, use `hw_MapRGB()`. See Section 28.22 [hw_MapRGB], page 247, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`color` color to convert

`infmt` pixel format of source color

RESULTS

`rgb` converted color

28.18 hw_IsImage

NAME

hw_IsImage – check if file is in a supported image format (V5.0)

SYNOPSIS

```
int ok = hw_IsImage(STRPTR file, int *width, int *height, int *alpha);
```

FUNCTION

This function checks if the specified file is in a supported image format and if it is, `hw_IsImage()` will return its dimensions in pixels as well as a boolean value that indicates whether or not the image uses alpha channel transparency.

`hw_IsImage()` is the preferred way of checking if a file is in a supported image format as it just scans the file header and is thus very fast.

Use `hw_LoadImage()` to load the image. See Section 28.19 [hw_LoadImage], page 242, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`file`	file to check
`width`	pointer to an `int` that receives the image's width in pixels on success
`height`	pointer to an `int` that receives the image's height in pixels on success
`alpha`	pointer to an `int` that receives the image's alpha channel setting on success (either `True` or `False`)

RESULTS

`ok`	`True` if the file was recognized as an image file, `False` otherwise

28.19 hw_LoadImage

NAME

hw_LoadImage – load image (V5.0)

SYNOPSIS

```
APTR handle = hw_LoadImage(STRPTR filename, struct hwTagList *tags,
                  int *width, int *height, int *alpha);
```

FUNCTION

This function loads the specified image file and returns a handle to it. If this function succeeds, you can call `hw_GetImageData()` to get a pointer to the raw 32-bit pixel data of this image. See Section 28.16 [hw_GetImageData], page 241, for details. On success, `hw_LoadImage()` will also return the image dimensions in pixels as well as a boolean value that indicates whether or not the image uses alpha channel transparency.

Note that `hw_LoadImage()` will load the pixel data as 32-bit RGBA bytes. If you want to have the pixel data in ARGB format, you have to explicitly request this by passing the respective tag (see below).

The following tags are currently supported by `hw_LoadImage()`:

`HWLDIMGTAG_USEARGB:`
: If you set the `iData` member of this tag item to `True`, the pixel data will be returned in ARGB format. By default, RGBA order is used. (V6.1)

When you're done with the image, call `hw_FreeImage()` on it. See Section 28.11 [hw_FreeImage], page 235, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`file` image file to load

`tags` additional options or `NULL` (see above)

`width` pointer to an `int` that receives the image's width in pixels on success or `NULL` if you don't want this information

`height` pointer to an `int` that receives the image's height in pixels on success or `NULL` if you don't want this information

`alpha` pointer to an `int` that receives the image's alpha channel setting on success (either `True` or `False`) or `NULL` if you don't want this information

RESULTS

`handle` handle to the image if it was successfully loaded, else `NULL`

28.20 hw_LockBitMap

NAME

hw_LockBitMap – gain access to the raw pixels of a bitmap (V6.0)

SYNOPSIS

```
APTR handle = hw_LockBitMap(APTR bmap, ULONG flags, struct
                hwos_LockBitMapStruct *bmlock, struct hwTagList *tags);
```

FUNCTION

This function locks the specified bitmap and allows you to access its raw pixel data. The bitmap can be either a color bitmap, a monochrome mask bitmap or an alpha channel bitmap. You have to pass a pointer to a `struct hwos_LockBitMapStruct` which will be filled with all the information you need by `hw_LockBitMap()`. `struct hwos_LockBitMapStruct` looks like this:

```
struct hwos_LockBitMapStruct
{
    APTR Data;           // [out]
    int Modulo;          // [out]
    int PixelFormat;     // [out]
    int BytesPerPixel;   // [out]
    int Width;           // [out]
```

```
        int Height;          // [out]
    };
```

`hw_LockBitMap()` will write the following values to the structure members:

`Data:` This member will be set to a pointer to the raw pixel data. The actual format used by the individual pixels is determined by the `PixelFormat` member. Please note that even if a 32-bit pixel format is used, `Data` will never contain any alpha channel information because Hollywood always stores the alpha channel separately in order to be compatible with 15-bit and 16-bit screenmodes. See Section 2.14 [Bitmap information], page 17, for details.

`Modulo:` This contains the bitmap's row modulo, i.e. the number of pixels or bytes in a single row of image data. This can be more than returned in `Width` because Hollywood might choose to allocate some padding bytes for optimized blitting. Please note that the value in `Modulo` is returned in pixels for color bitmaps and in bytes for mask and alpha channel bitmaps.

`PixelFormat:`
This member is set to the pixel format used by the pixel array written to the `Data` member. See Section 2.15 [Pixel format information], page 18, for details.

`BytesPerPixel:`
This will be set to the number of bytes that are needed to represent one pixel in the `Data` array. If the bitmap is a monochrome mask, this member will be set to 1 although in reality only 1 bit is needed for a single pixel in case of a monochrome mask.

`Width:` This will be set to the bitmap's actual width, without any row padding.

`Height:` This will be set to the bitmap's actual height.

The following tags can be passed in tag list parameter:

`HWLBMAPTAG_READONLY:`
If you set the `iData` member of this tag item to `True`, the bitmap will be locked for read-only access. This might be faster with some bitmap backends. By default, the bitmap is locked for read and write access.

Do not hold bitmap locks longer than necessary. In particular, do not return control to the script while holding a bitmap lock because the script might try to modify the bitmap then and this will lead to trouble in case the bitmap is still locked. You should call `hw_UnLockBitMap()` as soon as possible.

Note that `hw_LockBitMap()` can only be used with software bitmaps. It is not possible to access the raw pixels of hardware bitmaps.

If you only need to read a bitmap's raw pixel data, it might be more convenient to use the `hw_BitMapToARGB()` function instead. This will give you the pixels as readily formatted 32-bit ARGB values as it also takes potential mask and alpha channel bitmaps into account. The downside is that `hw_BitMapToARGB()` is slower because it needs to copy and convert the pixels first. See Section 28.4 [hw_BitMapToARGB], page 231, for details.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`bmap` bitmap to lock

`flags` reserved for future use; pass 0 for now

`bmlock` pointer to a `struct hwos_LockBitMapStruct` that is to be filled by this function

`tags` pointer to a taglist specifying additional options (see above) or `NULL`

RESULTS

`handle` handle to the locked bitmap or `NULL` on error

28.21 hw_LockBrush

NAME
hw_LockBrush – gain access to the raw pixels of a brush (V5.0)

SYNOPSIS

```
APTR handle = hw_LockBrush(lua_ID *id, struct hwTagList *tags,
                  struct hwos_LockBrushStruct *brlock);
```

FUNCTION
This function locks the specified brush and allows you to access its raw pixel data. You have to pass the object identifier of the brush you want to lock as a `lua_ID`. See Section 2.17 [Object identifiers], page 21, for details.

You also have to pass a pointer to a `struct hwos_LockBrushStruct` which will be filled with all the information you need by `hw_LockBrush()`. `struct hwos_LockBrushStruct` looks like this:

```
struct hwos_LockBrushStruct
{
    APTR RGBData;        // [out]
    int RGBModulo;       // [out]
    UBYTE *AlphaData;    // [out]
    int AlphaModulo;     // [out]
    UBYTE *MaskData;     // [out]
    int MaskModulo;      // [out]
    int PixelFormat;     // [out]
    int BytesPerPixel;   // [out]
    int Width;           // [out]
    int Height;          // [out]
};
```

`hw_LockBrush()` will write to the structure members as follows:

`RGBData:` This member will be set to a pointer to the raw RGB pixel data. The actual format used by the individual pixels is determined by the `PixelFormat`

member. Please note that even if a 32-bit pixel format is used, `RGBData` will never contain any alpha channel information because Hollywood always stores the alpha channel separately in order to be compatible with 15-bit and 16-bit screenmodes. See Section 2.14 [Bitmap information], page 17, for details.

`RGBModulo:`
This contains the number of pixels in a single row. This can be more than returned in `Width` because Hollywood might choose to allocate some padding bytes for optimized blitting. Note that the value returned in `RGBModulo` is specified in pixels, not in bytes.

`AlphaData:`
If the brush contains an alpha channel, this member is set to the raw alpha channel data as an array of unsigned bytes. Otherwise this member will be set to `NULL`.

`AlphaModulo:`
If the brush contains an alpha channel, this member will be set to the number of pixels stored in one row of the `AlphaData` array. This can be more than what has been returned in the `Width` member because Hollywood might use padding bytes for optimized blitting.

`MaskData:`
If the brush contains a mask, this member will be set to the raw mask bits. Otherwise it is set to `NULL`. Hollywood masks only know two different states: visible (1) and invisible (0) pixels. The bits are stored from left to right in chunks of one byte, i.e. the most significant bit of the first byte describes the transparency setting for the first pixel.

`MaskModulo:`
If the brush contains a mask, this member will be set to the number of bytes that is used for one row of mask data. Note that this value is specified in bytes and often contains some padding.

`PixelFormat:`
This member is set to the pixel format used by the pixel array written to the `RGBData` member. See Section 2.15 [Pixel format information], page 18, for details.

`BytesPerPixel:`
This will be set to the number of bytes that are needed to represent one pixel in the `RGBData` array.

`Width:` This will be set to the brush's actual width, without any row padding.

`Height:` This will be set to the brush's actual height.

The following tags can be passed in tag list parameter:

`HWLBRSHTAG_READONLY:`
If you set the `iData` member of this tag item to `True`, the brush will be locked for read-only access. This might be faster with some bitmap backends. By default, the brush is locked for read and write access. (V6.0)

Do not hold brush locks longer than necessary. In particular, do not return control to the script while holding a brush lock because the script might try to modify the brush then and this will lead to trouble in case the brush is still locked. You should call `hw_UnLockBrush()` as soon as possible.

Note that `hw_LockBrush()` can only be used with software brushes. It is not possible to access the raw pixels of hardware brushes.

If you only need to read a brush's raw pixel data, it might be more convenient to use the `hw_GetARGBBrush()` function instead. This will give you the pixels as readily formatted 32-bit ARGB values. The downside is that `hw_GetARGBBrush()` is slower because it needs to copy and convert the pixels first. See Section 28.12 [hw_GetARGBBrush], page 236, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`id`	object identifier of brush to be locked
`tags`	pointer to a taglist specifying additional options (see above) or `NULL`
`brlock`	pointer to a `struct hwos_LockBrushStruct` that is to be filled by this function

RESULTS

`handle`	handle to the locked brush or `NULL` on error

28.22 hw_MapRGB

NAME

hw_MapRGB – convert RGB color to pixel format (V6.0)

SYNOPSIS

```
ULONG color = hw_MapRGB(ULONG rgb, int outfmt);
```

FUNCTION

This function converts the specified RGB color to the specified pixel format. See Section 2.15 [Pixel format information], page 18, for details.

To convert a color from an arbitrary pixel format to RGB, use `hw_GetRGB()`. See Section 28.17 [hw_GetRGB], page 241, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`rgb`	RGB color to convert
`outfmt`	output pixel format

RESULTS

`color`	converted color

28.23 hw_RawBltBitMap

NAME

hw_RawBltBitMap – blit source to destination pixel buffer (V6.0)

SYNOPSIS

```
void hw_RawBltBitMap(APTR src, APTR dst, struct hwRawBltBitMapCtrl *ctrl,
        ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function can be used to blit data from a source to a destination raw pixel buffer. Note that this function does not accept Hollywood bitmaps, but expects you to pass raw pixel buffers only. This makes it possible to use `hw_RawBltBitMap()` in lots of different contexts. If you want to use `hw_RawBltBitMap()` on Hollywood bitmaps, you need to lock those bitmaps first using `hw_LockBitMap()` and then pass the raw pixel buffer pointer obtained by `hw_LockBitMap()` to `hw_RawBltBitMap()`.

Optionally, `hw_RawBltBitMap()` can take a mask or alpha channel pixel buffer into account. In case an alpha channel pixel buffer is specified, `hw_RawBltBitMap()` will also do the blending for you.

You have to pass source and destination pixel buffer pointers as well as a pointer to a `struct hwRawBltBitMapCtrl` to this function. `struct hwRawBltBitMapCtrl` looks like this:

```
struct hwRawBltBitMapCtrl
{
    int SrcX;           // [in]
    int SrcY;           // [in]
    int DstX;           // [in]
    int DstY;           // [in]
    int Width;          // [in]
    int Height;         // [in]
    int PixFmt;         // [in]
    UBYTE *MaskData;    // [in]
    UBYTE *AlphaData;   // [in]
    int SrcModulo;      // [in]
    int DstModulo;      // [in]
    int MaskModulo;     // [in]
    int AlphaModulo;    // [in]
};
```

Here's an explanation of the individual structure members:

`SrcX:` Contains the x position in the source buffer that marks the start offset for the copy operation. This is relative to the upper-left corner of the source buffer.

`SrcY:` Contains the y position in the source buffer that marks the start offset for the copy operation. This is relative to the upper-left corner of the source buffer.

`DstX:` Contains the destination x position relative to the upper-left corner.

`DstY:` Contains the destination y position relative to the upper-left corner.

`Width:` Contains the number of columns to copy.

`Height:` Contains the number of rows to copy.

`PixFmt:` Contains the pixel format used by the source and destination pixel buffers. Both buffers must use the same pixel format. See Section 2.15 [Pixel format information], page 18, for details.

`SrcModulo:`
This must be set to the number of pixels per row in the source buffer. This can be more than the actual image width in case there are padding pixels.

`DstModulo:`
This must be set to the number of pixels per row in the destination buffer. This can be more than the actual destination image width in case there are padding pixels.

`MaskData:`
This can be set to a pointer containing an array of raw mask bits. Hollywood masks only know two different states: visible (1) and invisible (0) pixels. The bits are stored from left to right in chunks of one byte, i.e. the most significant bit of the first byte describes the transparency setting for the first pixel. The buffer provided here must be exactly `MaskModulo` bytes wide and must match the source buffer's height. If you don't want to use masked blitting, set this to `NULL`.

`MaskModulo:`
If you specify a mask bitplane in `MaskData`, you need to set this member to the number of bytes that is used for one row of mask data. Note that this value is specified in bytes and often needs to use some padding. For example, if the source buffer is 123 pixels wide, the `MaskModulo` value would usually be set to 16 because 15 bytes are not enough for 123 pixels.

`AlphaData:`
This member can be set to an array containing alpha channel values for every pixel. This array must use one byte for every pixel and must match the source buffer's height. The width of the alpha channel array can be specified by setting the `AlphaModulo` member (see below). If this member is specified, `hw_RawBltBitMap()` will do blit the source pixel buffer to the destination pixel buffer with alpha blending. If you don't want to use alpha blending, set this to `NULL`.

`AlphaModulo:`
If `AlphaData` has been provided, this member must be set to the number of pixels stored in one row of the `AlphaData` array. This can be more than the source buffer's width in case you need padding.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`src` pointer to source raw pixel buffer

`dst`	pointer to destination raw pixel buffer
`ctrl`	pointer to a `struct hwRawBltBitMapCtrl` containing the blit parameters
`flags`	reserved for future use; pass 0
`tags`	reserved for future use; pass `NULL`

28.24 hw_RawLine

NAME

hw_RawLine – draw line to pixel buffer (V6.0)

SYNOPSIS

```
void hw_RawLine(APTR dst, int x1, int y1, int x2, int y2, ULONG color,
         ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function can be used to draw a line to a raw pixel buffer. Note that this function does not draw to a Hollywood bitmap, but to a raw pixel buffer only. This makes it possible to use `hw_RawLine()` in lots of different contexts. If you want to use `hw_RawLine()` on a Hollywood bitmap, you need to lock the bitmap first using `hw_LockBitMap()` and then pass the raw pixel buffer pointer obtained by `hw_LockBitMap()` to `hw_RawLine()`.

The following tags are recognized by `hw_RawLine()`:

`HWRLITAG_PIXFMT:`
: This tag can be used to set the pixel format `hw_RawLine()` should use when drawing into the pixel buffer. You have to pass a pixel format constant in the `iData` member of this tag. See Section 2.15 [Pixel format information], page 18, for details. This tag defaults to `HWOS_PIXFMT_ARGB32`.

`HWRLITAG_DSTWIDTH:`
: This tag must be specified. You have to pass the number of pixels per row in the destination buffer in the `iData` member of this tag. If you don't pass this tag, `hw_RawLine()` won't work.

Additionally, `hw_RawLine()` supports the following flags:

`HWRLIFLAGS_BLEND:`
: If this flag is set, then `hw_RawLine()` will draw a line with alpha-blending to the destination pixel buffer. The blend intensity is taken from the upper 8 bits in the `color` parameter that you've passed to `hw_RawLine()`. If this flag is not set, then `hw_RawLine()` will just draw with a static color.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`dst`	pointer to destination raw pixel buffer
`x1`	start x offset for the line
`y1`	start y offset for the line

`x2`	end x offset for the line
`y2`	end y offset for the line
`color`	ARGB color to use; the A component is only used if `HWRLIFLAGS_BLEND` has been set
`flags`	additional drawing flags (see above)
`tags`	additional drawing tags (see above)

28.25 hw_RawRectFill

NAME

hw_RawRectFill – draw filled rectangle to pixel buffer (V6.0)

SYNOPSIS

```
void hw_RawRectFill(APTR dst, int x, int y, int width, int height,
         ULONG color, ULONG flags, struct hwTagList *tags);
```

FUNCTION

This function can be used to draw a filled rectangle to a raw pixel buffer. Note that this function does not draw to a Hollywood bitmap, but to a raw pixel buffer only. This makes it possible to use `hw_RawRectFill()` in lots of different contexts. If you want to use `hw_RawRectFill()` on a Hollywood bitmap, you need to lock the bitmap first using `hw_LockBitMap()` and then pass the raw pixel buffer pointer obtained by `hw_LockBitMap()` to `hw_RawRectFill()`.

The following tags are recognized by `hw_RawRectFill()`:

`HWRRFTAG_PIXFMT:`
: This tag can be used to set the pixel format `hw_RawRectFill()` should use when drawing into the pixel buffer. You have to pass a pixel format constant in the `iData` member of this tag. See Section 2.15 [Pixel format information], page 18, for details. This tag defaults to `HWOS_PIXFMT_ARGB32`.

`HWRRFTAG_DSTWIDTH:`
: This tag must be specified. You have to pass the number of pixels per row in the destination buffer in the `iData` member of this tag. If you don't pass this tag, `hw_RawRectFill()` won't work.

Additionally, `hw_RawRectFill()` supports the following flags:

`HWRRFFLAGS_BLEND:`
: If this flag is set, then `hw_RawRectFill()` will draw a rectangle with alpha-blending to the destination pixel buffer. The blend intensity is taken from the upper 8 bits in the `color` parameter that you've passed to `hw_RawRectFill()`. If this flag is not set, then `hw_RawRectFill()` will just draw with a static color.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`dst`	pointer to destination raw pixel buffer
`x`	start x-offset of rectangle
`y`	start y-offset of rectangle
`width`	number of columns to draw
`height`	number of rows to draw
`color`	ARGB color to use; the A component is only used if `HWRRFFLAGS_BLEND` has been set
`flags`	additional drawing flags (see above)
`tags`	additional drawing tags (see above)

28.26 hw_RawWritePixel

NAME

hw_RawWritePixel – draw a single pixel to buffer (V6.0)

SYNOPSIS

```
void hw_RawWritePixel(APTR dst, int x, int y, ULONG color, ULONG flags,
        struct hwTagList *tags);
```

FUNCTION

This function can be used to draw a single pixel to a raw pixel buffer. Note that this function does not draw to a Hollywood bitmap, but to a raw pixel buffer only. This makes it possible to use `hw_RawWritePixel()` in lots of different contexts. If you want to use `hw_RawWritePixel()` on a Hollywood bitmap, you need to lock the bitmap first using `hw_LockBitMap()` and then pass the raw pixel buffer pointer obtained by `hw_LockBitMap()` to `hw_RawWritePixel()`.

The following tags are recognized by `hw_RawWritePixel()`:

`HWRWPTAG_PIXFMT:`
: This tag can be used to set the pixel format `hw_RawWritePixel()` should use when drawing into the pixel buffer. You have to pass a pixel format constant in the `iData` member of this tag. See Section 2.15 [Pixel format information], page 18, for details. This tag defaults to `HWOS_PIXFMT_ARGB32`.

`HWRWPTAG_DSTWIDTH:`
: This tag must be specified. You have to pass the number of pixels per row in the destination buffer in the `iData` member of this tag. If you don't pass this tag, `hw_RawWritePixel()` won't work.

Additionally, `hw_RawWritePixel()` supports the following flags:

`HWRWPFLAGS_BLEND:`
: If this flag is set, then `hw_RawWritePixel()` will plot a pixel with alpha-blending to the destination pixel buffer. The blend intensity is taken from the upper 8 bits in the `color` parameter that you've passed to

`hw_RawWritePixel()`. If this flag is not set, then `hw_RawWritePixel()` will just draw with a static color.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`dst`	pointer to destination raw pixel buffer
`x`	pixel's x-offset
`y`	pixel's y-offset
`color`	ARGB color to use; the A component is only used if `HWRWPFLAGS_BLEND` has been set
`flags`	additional drawing flags (see above)
`tags`	additional drawing tags (see above)

28.27 hw_RefreshDisplay

NAME
hw_RefreshDisplay – redraw display contents (V6.0)

SYNOPSIS
```
int error = hw_RefreshDisplay(APTR handle, ULONG flags,
                 struct hwTagList *tags);
```

FUNCTION
This function forces a complete refresh on the specified display handle. Useful to call when the operating system tells you that you have to redraw yourself.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`handle`	display handle
`flags`	currently unused; set to 0
`tags`	currently unused; set to `NULL`

RESULTS

`error`	error code or 0 for success

28.28 hw_RefreshSatelliteRoot

NAME
hw_RefreshSatelliteRoot – force refresh of satellite's root display (V5.2)

SYNOPSIS
```
void hw_RefreshSatelliteRoot(APTR handle);
```

FUNCTION

This function can be used to force a refresh of the specified satellite's root display. This, in turn, will lead to Hollywood calling the satellite's dispatcher to refresh as well. See Section 28.3 [hw_AttachDisplaySatellite], page 225, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` satellite handle allocated by `hw_AttachDisplaySatellite()`

28.29 hw_SetDisplayAdapter

NAME

hw_SetDisplayAdapter – install a display adapter (V6.0)

SYNOPSIS

```
int error = hw_SetDisplayAdapter(hwPluginBase *self, ULONG flags,
                struct hwTagList *tags);
```

FUNCTION

This function can be used to activate a plugin that has the `HWPLUG_CAPS_DISPLAYADAPTER` capability flag set. This function must only be called from inside your `RequirePlugin()` implementation. If this function succeeds, Hollywood's inbuilt display driver will be completely replaced by the display driver provided by your plugin and Hollywood will call into your plugin whenever it needs to deal with displays. In the first parameter, you have to pass a pointer to the `hwPluginBase` that Hollywood has passed to your plugin's `InitPlugin()` function. The second parameter must be set to a combination of flags. The following flags are currently defined:

`HWSDAFLAGS_PERMANENT:`
: If this flag is set, the display adapter will be made permanent. This means that other plugins won't be able to overwrite this display adapter with their own one. If `HWSDAFLAGS_PERMANENT` is set, all subsequent calls to `hw_SetDisplayAdapter()` will fail and your display adapter will persist.

`HWSDAFLAGS_TIEDVIDEOBITMAP:`
: Set this flag to indicate that your video bitmaps depend on your display, i.e. they cannot exist without the display. This is normally true for all device-dependent bitmaps. If this flag is set and the user closes the display, Hollywood will call your `ReadVideoPixels()` function to convert the device-dependent bitmaps (DDBs) into device-independent bitmaps (DIBs). Once the user opens the display again, Hollywood will call your `AllocVideoBitMap()` function to convert the DIBs back into DDBs. Obviously, this flag is only meaningful if you also set the `HWSDAFLAGS_VIDEOBITMAPADAPTER` flag.

`HWSDAFLAGS_SOFTWAREFALLBACK:`
: Set this flag if you want your display adapter to fall back to software bitmaps if hardware bitmaps (i.e. video bitmaps) cannot be allocated for some

reason, e.g. out of memory. If this flag is set and Hollywood cannot allocate a hardware bitmap, it will simply allocate a software bitmap instead. Obviously, this flag is only meaningful if you also set the `HWSDAFLAGS_VIDEOBITMAPADAPTER` flag.

`HWSDAFLAGS_CUSTOMSCALING:`
Set this flag to indicate that your display adapter wants to do scaling on its own in case autoscaling mode is active. If this flag is not set, Hollywood will do the autoscaling for you but this might be slower then. See Section 10.7 [BltBitMap], page 67, for details.

`HWSDAFLAGS_VIDEOBITMAPADAPTER:`
Set this flag to indicate that your display adapter supports custom hardware (video) bitmaps. Hollywood will then call your `AllocVideoBitMap()` function whenever it needs to allocate a hardware bitmap. See Section 10.5 [AllocVideoBitMap], page 63, for details. If you set this flag, you should also provide the `HWSDATAG_VIDEOBITMAPCAPS` tag to fine-tune the capabilities of your video bitmap adapter (see below).

`HWSDAFLAGS_BITMAPADAPTER:`
Set this flag to indicate that your display adapter wants to allocate all software bitmaps on its own. Hollywood will then call your `AllocBitMap()` function whenever it needs to allocate a software bitmap. Hollywood's inbuilt software bitmap handler will never be used if this flag is set. See Section 10.4 [AllocBitMap], page 62, for details.

`HWSDAFLAGS_DOUBLEBUFFERADAPTER:`
Set this flag if your display adapter wants to offer a custom hardware double buffering mode. If this flag is set, Hollywood will call your plugin's `BeginDoubleBuffer()` function when the user calls Hollywood's `BeginDoubleBuffer()` function with the optional argument set to `True`. See Section 10.6 [BeginDoubleBuffer], page 66, for details.

`HWSDAFLAGS_ALPHADRAW:`
Set this flag to indicate that your implementations of `RectFill()`, `Line()`, and `WritePixel()` support alpha-blending. If this flag isn't set, Hollywood will do any alpha-blending on its own and your implementations of functions like `RectFill()` will only have to be able to draw static colors. See Section 10.34 [RectFill], page 92, for details.

`HWSDAFLAGS_SLEEP:`
Set this flag to indicate that you want Hollywood to call your plugin's `Sleep()` function whenever it needs to sleep for a certain amount of time. See Section 10.41 [Sleep], page 96, for details.

`HWSDAFLAGS_VWAIT:`
Set this flag to indicate that you want Hollywood to call your plugin's `VWait()` function whenever it needs to wait for the vertical blank. See Section 10.43 [VWait], page 97, for details.

`HWSDAFLAGS_MONITORINFO:`
: Set this flag to indicate that your display adapter provides its own functions to query information about all available monitors. If this flag is set, Hollywood will call your plugin's `GetMonitorInfo()` function to obtain information about monitors available to the system. See Section 10.24 [GetMonitorInfo], page 79, for details.

`HWSDAFLAGS_GRABSCREEN:`
: Set this flag to indicate that your plugin provides custom routines for grabbing the desktop screen pixels. If this flag is set, Hollywood will call your plugin's `GrabScreenPixels()` function whenever it needs to grab the desktop screen's pixels. If this flag isn't set, Hollywood will use its inbuilt screen grabber. See Section 10.27 [GrabScreenPixels], page 82, for details.

`HWSDAFLAGS_DRAWALWAYS:`
: By default, Hollywood will never call any of your display adapter's functions that draw graphics to a display when that display is hidden or minimized. If you don't want this behaviour, set this flag and Hollywood will always call your display adapter's drawing functions, even when the display is hidden or minimized. (V6.1)

Additionally, `hw_SetDisplayAdapter()` accepts a tag list that allows you to configure further settings for the new display adapter. The following tags are currently recognized:

`HWSDATAG_PIXELFORMAT:`
: This tag allows you to set the pixel format that should be used when allocating bitmaps. Hollywood will allocate all of its software bitmaps in the pixel format specified in the `iData` member of this tag. See Section 2.15 [Pixel format information], page 18, for a list of available pixel formats. Please note that this tag doesn't have any effect if you provide your own bitmap adapter by setting the `HWSDAFLAGS_BITMAPADAPTER`. Obviously, the pixel format specified here doesn't have any effect on hardware (video) bitmaps either.

`HWSDATAG_BITMAPHOOK:`
: This tag can be used to provide custom routines for drawing to software bitmaps. You have to set the `iData` member of this tag to a combination of the following flags:

  `HWBMAHOOK_BLTBITMAP:`
  : Indicates that your `BltBitMap()` implementation wants to be called whenever Hollywood needs to blit a bitmap that doesn't have an accompanying mask or alpha channel to a software bitmap. Your `BltBitMap()` implementation has to do this blit operation then instead of Hollywood. See Section 10.7 [BltBitMap], page 67, for details. Hollywood will pass the `HWBBFLAGS_DESTBITMAP` flag to `BltBitMap()` so that it knows that the specified destination handle is a software bitmap.

`HWBMAHOOK_BLTMASKBITMAP:`
: Indicates that your `BltBitMap()` implementation wants to be called whenever Hollywood needs to blit a bitmap that has an accompanying mask to a software bitmap. Your `BltBitMap()` implementation has to do this blit operation then instead of Hollywood. See Section 10.7 [BltBitMap], page 67, for details. Hollywood will pass the `HWBBFLAGS_DESTBITMAP` flag to `BltBitMap()` so that it knows that the specified destination handle is a software bitmap.

`HWBMAHOOK_BLTALPHABITMAP:`
: Indicates that your `BltBitMap()` implementation wants to be called whenever Hollywood needs to blit a bitmap that has an an accompanying alpha channel to a software bitmap. Your `BltBitMap()` implementation has to do this blit operation then instead of Hollywood. See Section 10.7 [BltBitMap], page 67, for details. Hollywood will pass the `HWBBFLAGS_DESTBITMAP` flag to `BltBitMap()` so that it knows that the specified destination handle is a software bitmap.

`HWBMAHOOK_RECTFILL:`
: Indicates that your `RectFill()` implementation wants to be called whenever Hollywood needs draw a filled rectangle to a software bitmap. Your `RectFill()` implementation has to do this operation then instead of Hollywood. See Section 10.34 [RectFill], page 92, for details. Hollywood will pass the `HWRFFLAGS_DESTBITMAP` flag to `RectFill()` so that it knows that the specified destination handle is a software bitmap.

`HWBMAHOOK_WRITEPIXEL:`
: Indicates that your `WritePixel()` implementation wants to be called whenever Hollywood needs plot a single pixel to a software bitmap. Your `WritePixel()` implementation has to do this operation then instead of Hollywood. See Section 10.45 [WritePixel], page 99, for details. Hollywood will pass the `HWWPFLAGS_DESTBITMAP` flag to `WritePixel()` so that it knows that the specified destination handle is a software bitmap.

`HWBMAHOOK_LINE:`
: Indicates that your `Line()` implementation wants to be called whenever Hollywood needs draw a line to a software bitmap. Your `Line()` implementation has to do this operation then instead of Hollywood. See Section 10.29 [Line], page 84, for details. Hollywood will pass the `HWLIFLAGS_DESTBITMAP` flag to `Line()` so that it knows that the specified destination handle is a software bitmap.

`HWSDATAG_VIDEOBITMAPCAPS:`
: If you've enabled video bitmap support for this display adapter by setting the `HWSDAFLAGS_VIDEOBITMAPADAPTER` flag, the `iData` member of this tag

configures the exact capability set of your video bitmap adapter. This must be set to a combination of the following flags:

`HWVBMCAPS_SCALE:`
: Set this to indicate that your video bitmap adapter supports bitmap scaling. See Section 10.5 [AllocVideoBitMap], page 63, for details.

`HWVBMCAPS_TRANSFORM:`
: Set this to indicate that your video bitmap adapter supports bitmap transformation. See Section 10.5 [AllocVideoBitMap], page 63, for details.

`HWVBMCAPS_OFFSCREENCOLOR:`
: Set this to indicate that your `BltBitMap()`, `RectFill()`, `Line()`, and `WritePixel()` functions can draw to off-screen video bitmaps in color mode. Hollywood will pass one of the `HWxxFLAGS_DESTVIDEOBITMAP` flags to these functions to indicate that the destination is a video bitmap.

`HWVBMCAPS_OFFSCREENALPHA:`
: Set this to indicate that your `BltBitMap()`, `RectFill()`, `Line()`, and `WritePixel()` functions can draw to the alpha channel of off-screen video bitmaps. Hollywood will pass the `HWxxFLAGS_DESTVIDEOBITMAP` and the `HWxxFLAGS_ALPHAONLY` flags to these functions to indicate that the destination is a video bitmap and that the function must only draw to the alpha channel.

See Section 10.1 [Display adapter plugins], page 59, for information on how to write display adapter plugins.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`self`	`hwPluginBase` pointer passed to `InitPlugin()`
`flags`	combination of flags (see above)
`tags`	tag list specifying further options (see above)

RESULTS

`error`	error code or 0 for success

28.30 hw_UnLockBitMap

NAME

hw_UnLockBitMap – release bitmap lock (V6.0)

SYNOPSIS

```
void hw_UnLockBitMap(APTR handle);
```

FUNCTION

This function releases the specified bitmap lock. You need to call this function as soon as you're finished with accessing a bitmap's raw pixel data. After the call to `hw_UnLockBitMap()` you must no longer access the pixel data pointer returned by `hw_LockBitMap()`. See Section 28.20 [hw_LockBitMap], page 243, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` bitmap lock obtained by `hw_LockBitMap()`

28.31 hw_UnLockBrush

NAME

hw_UnLockBrush – release brush lock (V5.0)

SYNOPSIS

```
void hw_UnLockBrush(APTR handle);
```

FUNCTION

This function releases the specified brush lock. You need to call this function as soon as you're finished with accessing a brush's raw pixel data. After the call to `hw_UnLockBrush()` you must no longer access any of the pixel data pointers returned by `hw_LockBrush()`. See Section 28.21 [hw_LockBrush], page 245, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` brush lock obtained by `hw_LockBrush()`

29 JPEGBase functions

29.1 Overview

`JPEGBase` contains most functions from the libjpeg library. If you want to use any of these functions, you'll need the correct libjpeg header files. Make sure to use compatible header files only. Hollywood uses version 6b of libjpeg so you need to use the header files from exactly this version if you plan to use functions from the `JPEGBase` library.

Please consult your libjpeg documentation for information on the individual functions supported by `JPEGBase`.

Please note that you need to check for Hollywood version 5.3 or later before trying to access `JPEGBase`. It is not supported by earlier Hollywood versions.

The implementations of `JPEGBase` in Hollywood Designer and Hollywood are identical.

30 LuaBase functions

30.1 Overview

`LuaBase` contains functions to deal with the Lua VM that is at the heart of Hollywood. You will have to use these functions when writing plugins that add new commands and constants to Hollywood's script language. These plugins need to have the `HWPLUG_CAPS_LIBRARY` capability flag set. See Section 15.1 [Library plugins], page 129, for details.

Hollywood is based on Lua 5.0.2, although it isn't compatible with the Lua language. See Section 2.16 [Differences between Hollywood and Lua], page 19, for details.

Most functions that are offered by `LuaBase` are identical to their Lua counterparts. These ones aren't documented here. Please consult the Lua 5.0.2 manual for information on these functions. The following documentation only covers the functions that behave differently than their Lua counterparts or are Hollywood-specific additions.

Please note that many Lua functions jump directly into Hollywood's error handler in case something goes wrong. For example, functions like `luaL_checklstring()`, `luaL_checktable()`, or `luaL_checknumber()` will never return control to you if something goes wrong. They will always jump into Hollywood's error handler directly using the `longjmp()` API. Thus, you need take some care when it comes to managing resources that have been allocated by your function because you often do not get the chance to free them if an error occurs because the Lua functions jump directly into Hollywood's error handler. You need to find another way of making sure that resources get freed in case of an error as well so that you don't leak any memory.

`LuaBase` is available since Hollywood 5.0.

Note that `LuaBase` is not available in Hollywood Designer.

30.2 luaL_checkfilename

NAME

luaL_checkfilename – get filename from the stack (V5.3)

SYNOPSIS

```
const char *name =  luaL_checkfilename(lua_State *L, int numArg);
```

FUNCTION

This is not an official Lua API but a Hollywood extension. `luaL_checkfilename()` checks if there is a filename at the given stack index. When working with files, you should always use this function instead of `luaL_checklstring()` because `luaL_checkfilename()` is able to work with linked files as well. For example, if the user passes the filename `menulogo.png` to your function and this file does not physically exist, but has been linked to your applet or executable using Hollywood's `-linkfiles` option, `luaL_checkfilename()` will automatically set up a special virtual file name specification that is understood by `hw_FOpen()` and `hw_TranslateFileName()` and return it to you. `luaL_checklstring()`, however, would just return the string `menulogo.png` to you, leading to a failure as soon as you try to open this non-existent file.

Please note that like all other `luaL_checkxxx()` functions, this function will immediately jump into Hollywood's error handler if an error occurs. It will never return `NULL`. If `luaL_checkfilename()` returns, then it has been successful as well. If there is an error, `luaL_checkfilename()` won't return control to you at all.

INPUTS

`L` pointer to the `lua_State`

`numArg` stack index to examine

RESULTS

`name` file name specification

30.3 luaL_checknewid

NAME

luaL_checknewid – get new object identifier from the stack (V5.3)

SYNOPSIS

```
void luaL_checknewid(lua_State *L, int numArg, lua_ID *id);
```

FUNCTION

This is not an official Lua API but a Hollywood extension. `luaL_checknewid()` can be used from functions that add a new Hollywood object to a user-defined object list registered using the `hw_RegisterUserObject()` function. See Section 34.35 [hw_RegisterUserObject], page 317, for details.

As you might know, the Hollywood standard for functions that create new Hollywood objects is that the user can either pass a numerical value that the object should use or he can pass `Nil` to make the function choose a vacant identifier for the new object on its own. See Section 2.17 [Object identifiers], page 21, for details. What `luaL_checknewid()` does is simply to see if a numerical value is on the stack or `Nil`. In case `Nil` is at the specified stack index, `luaL_checknewid()` will set the `ptr` member of the `lua_ID` structure to `(void *) 1`. This is of course not a valid identifier. It is only a temporary set up to tell your function that the user has passed `Nil`. Your function then has to set the `ptr` member to the real object identifier and push it into the stack using `lua_pushlightuserdata()`. If `ptr` is `NULL`, however, your function simply has to use the identifier passed in the `num` member of the `lua_ID` structure and return nothing.

See Section 2.17 [Object identifiers], page 21, for details.

INPUTS

`L` pointer to the `lua_State`

`numArg` stack index to examine

`id` pointer to a `lua_ID` to receive the object identifier passed by the user

30.4 luaL_checkid

NAME

luaL_checkid – get object identifier from the stack (V5.0)

SYNOPSIS

```
void luaL_checkid(lua_State *L, int numArg, lua_ID *id);
```

FUNCTION

This is not an official Lua API but a Hollywood extension. `luaL_checkid()` checks if there is a Hollywood object identifier at the given stack index. If there is, it will be written to the `lua_ID` that has been passed to this function. Otherwise, `luaL_checkid()` will jump directly into Hollywood's error handler.

A Hollywood object identifier can either be a numerical value or a value of the `LUA_TLIGHTUSERDATA` type. See Section 2.17 [Object identifiers], page 21, for details.

INPUTS

`L`	pointer to the `lua_State`
`numArg`	stack index to examine
`id`	pointer to a `lua_ID` to receive the object identifier

30.5 lua_pcall

NAME

lua_pcall – run a Lua function in protected mode (V5.0)

SYNOPSIS

```
int error = lua_pcall(lua_State *L, int nargs, int nrvals, int errfunc);
```

FUNCTION

This function does the same as Lua's `lua_pcall()` function except that it returns a Hollywood error code if something goes wrong. On success, 0 is returned which is the same behaviour as in Lua's `lua_pcall()`.

Furthermore, the `errfunc` parameter is not supported and must be 0. The Hollywood version of `lua_pcall()` also won't push the error message on the stack.

Finally, you should always set `HWMCP_SETCALLBACKMODE` to `True` before calling `lua_pcall()` and reset it to `False` when `lua_pcall()` returns. See Section 34.24 [hw_MasterControl], page 291, for details.

Note that up to and including Hollywood 6.1 `lua_pcall()` didn't keep the stack balanced in case an error was returned. This has been fixed for Hollywood 6.2.

INPUTS

`L`	pointer to the `lua_State`
`nargs`	number of arguments on the stack
`nrvals`	number of return values
`errfunc`	unsupported; must be 0

RESULTS

`error` Hollywood error code or 0 on success

30.6 lua_throwerror

NAME

lua_throwerror – throw an error (V5.0)

SYNOPSIS

```
void lua_throwerror(lua_State *L, int error);
```

FUNCTION

This is not an official Lua API but a Hollywood extension. `lua_throwerror()` allows you to jump directly into Hollywood's error handler, forcing it to show the message associated with the specified error code. Internally, `lua_throwerror()` uses `longjmp()` to jump directly into the error handler.

Use this function only if you have very good reason to do so. The normal way of indicating an error is to have your function return its error code back to Hollywood. If this is not possible, maybe because the error has occurred in a callback that doesn't allow you to delegate an error code to the main function in a convenient way, you may call `lua_throwerror()` to cause an immediate error exit. Make sure that you free any resources that your function has allocated before calling `lua_throwerror()`, though, since this function never returns.

INPUTS

`L` pointer to the `lua_State`

`error` error code to throw; must not be 0

31 MiscBase functions

31.1 Overview

`MiscBase` contains various functions and data that doesn't fit anywhere else. It currently looks like this:

```
typedef struct _hwMiscBase
{
    UBYTE *VeraSans;
    UBYTE *VeraMono;
    UBYTE *VeraSerif;
    int sizeof_VeraSans;
    int sizeof_VeraMono;
    int sizeof_VeraSerif;
} hwMiscBase;
```

Here's a description of the individual structure members:

`VeraSans:`
: Contains a pointer to the raw data of the Bitstream Vera Sans TrueType font. The size of this buffer is stored in the `sizeof_VeraSans` member. Bitstream Vera Sans is a font that is free for commercial use as long as the font license is included.

`VeraMono:`
: Contains a pointer to the raw data of the Bitstream Vera Sans Mono TrueType font. The size of this buffer is stored in the `sizeof_VeraMono` member. Bitstream Vera Sans Mono is a font that is free for commercial use as long as the font license is included.

`VeraSerif:`
: Contains a pointer to the raw data of the Bitstream Vera Serif TrueType font. The size of this buffer is stored in the `sizeof_VeraSerif` member. Bitstream Vera Serif is a font that is free for commercial use as long as the font license is included.

`MiscBase` is available since Hollywood 5.0.

The implementations of `MiscBase` in Hollywood Designer and Hollywood are identical.

32 PluginBase functions

32.1 Overview

`PluginBase` contains functions to deal with plugins. For example, it allows plugins to expose interfaces to other plugins and disable plugins.

Please note that you need to check for Hollywood version 6.0 or later before trying to access `PluginBase`. It is not supported by earlier Hollywood versions.

32.2 hw_DisablePlugin

NAME

hw_DisablePlugin – enable or disable a plugin (V6.0)

SYNOPSIS

```
void hw_DisablePlugin(hwPluginBase *plugin, int disable);
```

FUNCTION

This function can be used to enable or 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.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`plugin`	plugin to be disabled or enabled
`disable`	`True` to disable the plugin, `False` to enable it again

32.3 hw_FreePluginList

NAME

hw_FreePluginList – free plugin list (V6.0)

SYNOPSIS

```
void hw_FreePluginList(struct hwPluginList *list);
```

FUNCTION

This function frees a plugin list allocated by `hw_GetPluginList()`. See Section 32.4 [hw_GetPluginList], page 270, for details.

Note that this function must not be used to free individual plugin nodes. You must always free the complete list.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`list` plugin list allocated by `hw_GetPluginList()`

32.4 hw_GetPluginList

NAME

hw_GetPluginList – get plugin list (V6.0)

SYNOPSIS

```
struct hwPluginList *list = hw_GetPluginList(struct hwTagList *tags);
```

FUNCTION

This function returns a list of all plugins that have been loaded by Hollywood. This is useful if you want to disable other plugins or get access to their user pointer which can be used to expose a public interface to other plugins.

`hw_GetPluginList()` will return a pointer to a `struct hwPluginList` which looks like this:

```
struct hwPluginList
{
    struct hwPluginList *Succ;
    hwPluginBase *Plugin;
};
```

For each node in the list, `struct hw_PluginList` will be initialized as follows:

`Succ:` Contains a pointer to the next list node or `NULL` if this node is the last one.

`Plugin:` Contains a pointer to the plugin's `hwPluginBase`. You can get all necessary information about the plugin by examining the members of this structure. See Section 7.3 [InitPlugin], page 47, for a description of the individual structure members.

The list that is returned by this function must be freed using the `hw_FreePluginList()` function. See Section 32.3 [hw_FreePluginList], page 269, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`tags` reserved for future use; pass `NULL`

RESULTS

`list` a list containing all loaded plugins

32.5 hw_GetPluginUserPointer

NAME

hw_GetPluginUserPointer – get custom data from plugin (V6.0)

SYNOPSIS

```
APTR userdata = hw_GetPluginUserPointer(hwPluginBase *plugin);
```

FUNCTION

This function allows you to get the value of a plugin's user pointer. This is the value that has been set by calling the `hw_SetPluginUserPointer()` function. The user pointer can be used to expose a public interface or custom data structures to other plugins. See Section 32.6 [hw_SetPluginUserPointer], page 271, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`plugin` plugin whose user pointer should be obtained

RESULTS

`userdata` user pointer of the specified plugin

32.6 hw_SetPluginUserPointer

NAME

hw_SetPluginUserPointer – associate custom data with plugin (V6.0)

SYNOPSIS

```
void hw_SetPluginUserPointer(hwPluginBase *plugin, APTR userdata);
```

FUNCTION

This function can be used to store a custom value in the plugin's user pointer. This value can later be obtained by other plugins by calling the `hw_GetPluginUserPointer()` function. This enables plugins to expose a public interface or custom data structures to other plugins. Other plugins can make use of your plugin by first looking for it using `hw_GetPluginList()` and, if found, accessing its interface using `hw_GetPluginUserPointer()`.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`plugin` plugin to use

`userdata` user data to store in the plugin's user pointer

33 RequesterBase functions

33.1 Overview

`RequesterBase` contains a number of functions that deal with Hollywood's support for several system requesters.

`RequesterBase` is available since Hollywood 5.0.

33.2 hw_EasyRequest

NAME

hw_EasyRequest – pop up a system requester (V5.0)

SYNOPSIS

```
int r = hw_EasyRequest(STRPTR title, STRPTR body, STRPTR gadgets,
            struct hwTagList *tags);
```

FUNCTION

This function will show a system requester, also known as a message box. You have to pass title and body text for the requester as well as a string containing the name of at least one button to show in the requester. If you want to have multiple buttons, you have to separate them by a vertical bar character, e.g. `"OK|Cancel"`.

`hw_EasyRequest()` will return the id of the button that has been pressed. The right-most button always has the id 0. If there is only one button, it will also have the id 0. The ids of the other buttons are counted from left to right starting at 1. This arrangement has been chosen so that in case there are two buttons like `"OK|Cancel"` or `"Yes|No"`, the affirmative button's id will correspond to `True` whereas the negative response button's id will correspond to `False`.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`title`	string to show in the requester's title
`body`	body text for the requester
`gadgets`	string containing the name of at least one button to show in the requester
`tags`	reserved for future use; pass `NULL`

RESULTS

`r`	id of the button pressed by the user

33.3 hw_FileRequest

NAME

hw_FileRequest – pop up a file requester (V5.0)

SYNOPSIS

```
int ok = hw_FileRequest(STRPTR title, STRPTR buf, int len,
             struct hwTagList *tags);
```

FUNCTION

This function will open a file requester (also known as an open dialog box or file chooser dialog) that prompts the user to select a file for opening. The function will then copy the path to the string buffer passed in the second parameter. If the user cancels the requester, `hw_FileRequest()` will return `False` and the string buffer won't be modified. If the user selects a file and acknowledges the requester, `True` is returned.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`title`	string to show in the requester's title
`buf`	string buffer to receive user's selection
`len`	size of the string buffer in bytes
`tags`	reserved for future use; pass `NULL`

RESULTS

`ok`	`True` or `False` indicating whether the user selected a file or not

33.4 hw_PathRequest

NAME

hw_PathRequest – pop up a path requester (V5.0)

SYNOPSIS

```
int ok = hw_PathRequest(STRPTR title, STRPTR buf, int len,
             struct hwTagList *tags);
```

FUNCTION

This function will open a path requester (also known as a browse for folder dialog) that prompts the user to select a directory. The function will then copy this directory's path to the string buffer passed in the second parameter. If the user cancels the requester, `hw_PathRequest()` will return `False` and the string buffer won't be modified. If the user selects a path and acknowledges the requester, `True` is returned.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`title`	string to show in the requester's title
`buf`	string buffer to receive user's selection
`len`	size of the string buffer in bytes
`tags`	reserved for future use; pass `NULL`

RESULTS

`ok` `True` or `False` indicating whether the user selected a path or not

33.5 hw_SetRequesterAdapter

NAME

hw_SetRequesterAdapter – install a requester adapter (V6.0)

SYNOPSIS

```
int error = hw_SetRequesterAdapter(hwPluginBase *self, ULONG flags,
                struct hwTagList *tags);
```

FUNCTION

This function can be used to activate a plugin that has the `HWPLUG_CAPS_REQUESTERADAPTER` capability flag set. This function must only be called from inside your `RequirePlugin()` implementation. If this function succeeds, Hollywood's inbuilt requester handler will be completely replaced by the requester handler provided by your plugin. In the first parameter, you have to pass a pointer to the `hwPluginBase` that Hollywood has passed to your plugin's `InitPlugin()` function. The second parameter must be set to a combination of flags.

Please note that Hollywood will only call your plugin for the requester types that your plugin has explicitly declared as supported by setting the respective flag (see below). This way it is possible that your plugin chooses to replace only a certain set of Hollywood's requesters and not all of them. For example, if your plugin just wants to override Hollywood's file and path requesters, then you would have to set the `HWSRAFLAGS_FILEREQUEST` and `HWSRAFLAGS_PATHREQUEST` flags below. In that case, Hollywood would only call your plugin when it has to show a file or path requester - all other types would be handled by Hollywood itself.

The following flags are currently recognized:

`HWSRAFLAGS_PERMANENT:`
: If this flag is set, the requester adapter will be made permanent. This means that other plugins won't be able to overwrite this requester adapter with their own one. If `HWSRAFLAGS_PERMANENT` is set, all subsequent calls to `hw_SetRequesterAdapter()` will fail and your requester adapter will persist.

`HWSRAFLAGS_SYSTEMREQUEST:`
: Set this flag if your plugin provides a custom implementation of `SystemRequest()`. See Section 16.9 [SystemRequest], page 146, for details.

`HWSRAFLAGS_FILEREQUEST:`
: Set this flag if your plugin provides a custom implementation of `FileRequest()`. See Section 16.3 [FileRequest], page 139, for details.

`HWSRAFLAGS_PATHREQUEST:`
: Set this flag if your plugin provides a custom implementation of `PathRequest()`. See Section 16.7 [PathRequest], page 143, for details.

`HWSRAFLAGS_STRINGREQUEST:`
: Set this flag if your plugin provides a custom implementation of `StringRequest()`. See Section 16.8 [StringRequest], page 144, for details.

`HWSRAFLAGS_LISTREQUEST:`
: Set this flag if your plugin provides a custom implementation of `ListRequest()`. See Section 16.6 [ListRequest], page 142, for details.

`HWSRAFLAGS_FONTREQUEST:`
: Set this flag if your plugin provides a custom implementation of `FontRequest()`. See Section 16.4 [FontRequest], page 140, for details.

`HWSRAFLAGS_COLORREQUEST:`
: Set this flag if your plugin provides a custom implementation of `ColorRequest()`. See Section 16.2 [ColorRequest], page 138, for details.

See Section 16.1 [Requester adapter plugins], page 137, for information on how to write requester adapter plugins.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`self`	`hwPluginBase` pointer passed to `InitPlugin()`
`flags`	combination of flags (see above)
`tags`	reserved for future use; set it to `NULL` for now

RESULTS

`error`	error code or 0 for success

34 SysBase functions

34.1 Overview

`SysBase` contains commands for controlling various lowlevel functionalities of Hollywood.

`SysBase` is available since Hollywood 5.0.

34.2 hw_AddLoaderAdapter

NAME

hw_AddLoaderAdapter – add a loader or an adapter (V6.0)

SYNOPSIS

```
int error = hw_AddLoaderAdapter(hwPluginBase *self, ULONG type);
```

FUNCTION

This function can be used to manually activate a file loader or adapter plugin. The following types are currently supported:

`HWPLUG_CAPS_IMAGE:`
: Image loader plugins. You only need to call `hw_AddLoaderAdapter()` on them if automatic initialization has been disabled by setting the `HWEXT_IMAGE_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_ANIM:`
: Anim loader plugins. You only need to call `hw_AddLoaderAdapter()` on them if automatic initialization has been disabled by setting the `HWEXT_ANIM_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_SOUND:`
: Sound loader plugins. You only need to call `hw_AddLoaderAdapter()` on them if automatic initialization has been disabled by setting the `HWEXT_SOUND_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_VIDEO:`
: Video loader plugins. You only need to call `hw_AddLoaderAdapter()` on them if automatic initialization has been disabled by setting the `HWEXT_VIDEO_NOAUTOINIT` extension bit.

`HWPLUG_CAPS_FILEADAPTER:`
: File adapter plugins. These are not initialized automatically. So you will have to call `hw_AddLoaderAdapter()` on them or they will only be available when the user directly addresses them using the `Adapter` tag.

`HWPLUG_CAPS_DIRADAPTER:`
: Directory adapter plugins. These are not initialized automatically. So you will have to call `hw_AddLoaderAdapter()` on them or they will only be available when the user directly addresses them using the `Adapter` tag

Please note that this function cannot be used to activate display, timer, requester, and audio adapters. These all have custom functions that are used for their activation, e.g. `hw_SetDisplayAdapter()` for display adapters.

This function should be called from inside your `RequirePlugin()` implementation. If this function succeeds, Hollywood will call your loader or adapter whenever it needs to open an object of the respective type and your plugin can then choose which objects it would like to handle. In the first parameter, you have to pass a pointer to the `hwPluginBase` that Hollywood has passed to your plugin's `InitPlugin()` function. The second parameter has to specify the type of the loader or adapter to add (see above for supported types). Note that this must not be a combination of types, but only a single loader or adapter type can be activated per call.

Loaders and adapters may be added and removed any time you want. They are not a one time setting, though traditionally you will want to install your loaders and adapters when your `RequirePlugin()` is called as a result of the user running `@REQUIRE` on your plugin. But this is not a must. You may also choose to add and remove loaders and adapters at any later time. When Hollywood opens an object, the loaders and adapters will be asked whether they want to handle this object in the order they were added into the system. First come, first served.

It is not necessary to remove the loader or adapter using `hw_RemoveLoaderAdapter()` when your plugin is closed. Hollywood will do this automatically for you.

Please note that even if this function hasn't been called, Hollywood can still call your loaders and adapters. This will happen if the user directly addresses a loader or an adapter in the script. Consider the following example:

```
LoadBrush(1, "a.tiff.pp", {Loader = "tiff", Adapter = "powerpacker"})
```

This Hollywood code will call into the file adapter of `powerpacker`.hwp and into the image loader of `tiff`.hwp directly, no matter if they have been activated or not. If loaders and adapters are addressed directly, Hollywood will always call them if the respective plugins are not disabled. Thus, it is advised that you call `hw_ConfigureLoaderAdapter()` in your `InitPlugin()` implementation because `RequirePlugin()` might not even be called if the user addresses the loader or adapter directly. See Section 34.7 [hw_ConfigureLoaderAdapter], page 280, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`self`	`hwPluginBase` pointer passed to `InitPlugin()`
`type`	type of the loader or adapter to add (see above)

RESULTS

`error`	error code or 0 for success

34.3 hw_AddTime

NAME

hw_AddTime – add two time stamps (V5.0)

SYNOPSIS

```
void hw_AddTime(struct hwos_TimeVal *dest, struct hwos_TimeVal *src);
```

FUNCTION
This functions adds the `dest` and `src` time stamps and stores the resulting time stamp in `dest`.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`dest`	pointer to a `struct hwos_TimeVal` containing the augend; the result of the addition will be written to this buffer
`src`	pointer to a `struct hwos_TimeVal` containing the addend

34.4 hw_AllocSemaphore

NAME
hw_AllocSemaphore – allocate a semaphore (V6.0)

SYNOPSIS
```
APTR handle = hw_AllocSemaphore(void);
```

FUNCTION
This function allocates and initializes a semaphore that can be used to protect certain data structures from access by multiple threads. Another name for semaphore is critical section (Windows) or mutex (POSIX).

DESIGNER COMPATIBILITY
Unsupported

INPUTS
none

RESULTS

`handle`	a semaphore handle or `NULL` on error

34.5 hw_CmpTime

NAME
hw_CmpTime – compare two time stamps (V5.0)

SYNOPSIS
```
int r = hw_CmpTime(struct hwos_TimeVal *t1, struct hwos_TimeVal *t2);
```

FUNCTION
This function compares the two time stamps and returns a value that indicates their relation. 0 is returned if both time stamps are identical, -1 is returned if `t2` is earlier than `t1` and 1 is returned if `t1` is earlier than `t2`.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`t1`	pointer to a `struct hwos_TimeVal`
`t2`	pointer to a `struct hwos_TimeVal`

RESULTS

`r`	relation of the two time stamps (see above)

34.6 hw_CompareString

NAME

hw_CompareString – compare two strings (V7.0)

SYNOPSIS

```
int r = hw_CompareString(STRPTR s1, STRPTR s2, ULONG flags,
                         struct hwTagList *tags);
```

FUNCTION

This function compares `s1` and `s2` and returns how the two strings are related. If `s1` is less than `s2`, -1 is returned. If `s1` is greater than `s2`, 1 is returned, otherwise, i.e. if the strings are equal, the return value is 0.

The `flags` parameter can be set to a combination of the following flags:

`HWCMPSTR_IGNORECASE`:
: If this flag is set, `hw_CompareString()` will compare the two strings in a case-insensitive manner.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`s1`	first string
`s2`	second string
`flags`	options for the comparison (see above)
`tags`	currently unused; pass `NULL`

RESULTS

`r`	relation of the two strings (see above)

34.7 hw_ConfigureLoaderAdapter

NAME

hw_ConfigureLoaderAdapter – configure a loader or adapter (V6.0)

SYNOPSIS

```
int hw_ConfigureLoaderAdapter(hwPluginBase *self, ULONG type, ULONG flags,
                              struct hwTagList *tags);
```

FUNCTION

This function allows you to configure your loader or adapter plugin. Since Hollywood might call into your loader or adapter even if it hasn't been activated, it is recommended to always call this function from inside your `InitPlugin()` implementation, i.e. before you call `hw_AddLoaderAdapter()`. This ensures that your loader or adapter plugin always uses the configuration you want it to use, no matter if it is manually activated by `hw_AddLoaderAdapter()` or if Hollywood is calling directly into it.

The second parameter specifies the type of the loader or adapter you want to configure. See Section 34.2 [hw_AddLoaderAdapter], page 277, for a list of supported types.

The third and fourth parameters depend on the `type` passed as parameter 2. The following flags and tags are currently recognized:

`HWPLUG_CAPS_FILEADAPTER:`

File adapter plugins currently support the following flags:

`HWCLAFAFLAGS_CHUNKLOADER:`

If this flag is set, you indicate that your file adapter's `FOpen()` implementation supports loading of chunked files. If you set this flag, your `FOpen()` implementation must support the `HWFOPENTAG_CHUNKXXX` tags. See Section 12.6 [FOpen], page 107, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`self`	`hwPluginBase` pointer passed to `InitPlugin()`
`type`	loader or adapter type (see above)
`flags`	desired flags for the loader or adapter (see above)
`tags`	currently unused; pass `NULL`

RESULTS

`error`	error code or 0 for success

34.8 hw_ConvertString

NAME

hw_ConvertString – convert string between character encodings (V6.0)

SYNOPSIS

```
STRPTR s = hw_ConvertString(STRPTR in, int infmt, int outfmt,
                            struct hwTagList *tags);
```

FUNCTION

This function converts the string specified from `infmt` to `outfmt` and returns it. `hw_ConvertString()` allocates the resulting string for you. You have to free this string using `hw_FreeString()` then.

The following formats are currently available:

`HWOS_ENCODING_ISO8859_1:`
: ISO 8859-1. This is the default encoding in Hollywood version older than 7.0.

`HWOS_ENCODING_UTF8:`
: UTF-8. This is the default encoding since Hollywood 7.0.

`HWOS_ENCODING_AMIGA:`
: The system charset on AmigaOS and compatibles. This is obviously only supported on AmigaOS and compatibles. Note that `HWOS_ENCODING_AMIGA` can only be converted to and from `HWOS_ENCODING_UTF8`. It must not be used together with `HWOS_ENCODING_ISO8859_1`. (V7.0)

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`in`	null-terminated string to convert
`infmt`	format of the source string (see above)
`outfmt`	desired destination format (see above)
`tags`	currently unused; pass `NULL`

RESULTS

`s`	converted string pointer or `NULL` on error

34.9 hw_Delay

NAME

hw_Delay – sleep for a certain amount of time (V5.0)

SYNOPSIS

```
void hw_Delay(int time);
```

FUNCTION

This function sleeps for the specified amount of milliseconds. Please note that this function will really put the complete application to sleep, i.e. no window handling will take place at all.

This function is not thread-safe on AmigaOS and compatibles. You must not call this function from threads on AmigaOS and compatibles. On all other platforms it is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`time`	number of milliseconds to sleep

34.10 hw_DisableCallback

NAME
hw_DisableCallback – disable a callback (V6.0)

SYNOPSIS
```
void hw_DisableCallback(APTR handle, int disable);
```

FUNCTION
This function can be used to temporarily disable a callback that has been registered using `hw_RegisterCallback()`. Pass `True` in parameter 2 to disable the callback, `False` to enable it again.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`handle`	callback handle allocated by `hw_RegisterCallback()`
`disable`	boolean flag indicating whether the callback should be disabled

34.11 hw_FreeObjectData

NAME
hw_FreeObjectData – free user object data (V5.3)

SYNOPSIS
```
void hw_FreeObjectData(lua_State *L, struct hwObjectList *item);
```

FUNCTION
This function frees any data that Hollywood has associated with your object. You have to call this function before freeing your object so that Hollywood gets a chance to free any data it has associated with your object, for example via the `SetObjectData()` call.

See Section 34.35 [hw_RegisterUserObject], page 317, for details.

DESIGNER COMPATIBILITY
Supported since Designer 4.5

INPUTS

`L`	pointer to the `lua_State`
`item`	user object whose data shall be freed

34.12 hw_FreeSemaphore

NAME
hw_FreeSemaphore – free semaphore (V6.0)

SYNOPSIS
```
void hw_FreeSemaphore(APTR sem);
```

FUNCTION
This function frees the specified semaphore handle.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`sem` semaphore handle allocated by `hw_AllocSemaphore()`

34.13 hw_FreeString

NAME
hw_FreeString – free converted string (V6.0)

SYNOPSIS

```
void hw_FreeString(STRPTR s);
```

FUNCTION
This function can be used to free a string that has been allocated by `hw_ConvertString()`.

DESIGNER COMPATIBILITY
Supported since Designer 4.5

INPUTS

`s` string allocated by `hw_ConvertString()`

34.14 hw_GetDate

NAME
hw_GetDate – get current date and time (V5.0)

SYNOPSIS

```
void hw_GetDate(STRPTR buf);
```

FUNCTION
This function copies the current date and time to the specified memory buffer. The string will be formatted as follows: `dd-mmm-yyyy hh:mm:ss`. All constituents are numbers except the month which is specified as a three letter code containing the first three letters of the English month name, e.g. `"Jan"`.

If you want to query the current date in an abstracted format, please use the `hw_GetDateStamp()` function instead. See Section 34.15 [hw_GetDateStamp], page 285, for details.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`buf` buffer large enough to hold the date and time string

34.15 hw_GetDateStamp

NAME

hw_GetDateStamp – get current date and time (V6.0)

SYNOPSIS

```
void hw_GetDateStamp(struct hwos_DateStruct *stamp);
```

FUNCTION

This function will retrieve the current date and time and copy it to the `struct hwos_DateStruct` pointer passed as parameter 1. `struct hwos_DateStruct` looks like this:

```
struct hwos_DateStruct
{
    int Seconds;
    int Minutes;
    int Hours;
    int Day;
    int Month;
    int Year;
};
```

`hw_GetDateStamp()` will initialize the individual members as follows:

`Seconds:` This will be set to a value between 0 and 59.

`Minutes:` This will be set to a value between 0 and 59.

`Hours:` This will be set to a value between 0 and 23.

`Day:` This will be set to a value between 1 and 31.

`Month:` This will be set to a value between 0 and 11.

`Year:` This will be set to the year number.

If you want to query the current date in a human-readable format, please use the `hw_GetDate()` function instead. See Section 34.14 [hw_GetDate], page 284, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`stamp` pointer to a `struct hwos_DateStruct`

34.16 hw_GetEncoding

NAME

hw_GetEncoding – get current script's encoding (V7.0)

SYNOPSIS

```
int hw_GetEncoding(struct hwTagList *tags);
```

FUNCTION

This function returns the encoding used by the script. By default, all scripts will use `HWOS_ENCODING_UTF8` unless they explicitly request to run in ISO 8859-1 mode. Note that even if a script requests ISO 8859-1 encoding, `hw_GetEncoding()` will still return `HWOS_ENCODING_UTF8` if you call it while being in either the `InitPlugin()` or `InitLibrary()` function because the encoding requested by the script will be set while parsing it and both `InitPlugin()` and `InitLibrary()` are called before that.

Thus, it is probably a better idea to install a callback of type `HWCB_ENCODINGCHANGE`. Such a callback will be run immediately whenever the encoding changes. See Section 34.30 [hw_RegisterCallback], page 308, for details.

This function can return the following values:

`HWOS_ENCODING_UTF8:`
: UTF-8. This is Hollywood's default encoding since version 7.0.

`HWOS_ENCODING_ISO8859_1:`
: ISO 8859-1. This was Hollywood's default encoding before version 7.0. It is still supported for compatibility reasons but it should not be used by new scripts.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`tags`	reserved for future use; pass `NULL` for the time being

34.17 hw_GetErrorName

NAME

hw_GetErrorName – compose error string into memory buffer (V5.0)

SYNOPSIS

```
void hw_GetErrorName(int error, STRPTR buf, int size);
```

FUNCTION

This function composes a full error message including extended error information set via `hw_SetErrorString()` or `hw_SetErrorCode()` into a memory buffer. You have to pass a pointer to a memory buffer in parameter 2 and the size of this buffer in parameter 3. Hollywood will then copy a null-terminated string to this memory buffer. Some error messages are quite lengthy so make sure that the buffer you pass here is at least 1 kilobyte in size.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`error`	error code whose message text you want to retrieve
`buf`	pointer to a memory buffer
`size`	total size of the memory buffer in bytes

34.18 hw_GetEventHandler

NAME

hw_GetEventHandler – get information about event handler (V6.1)

SYNOPSIS

```
int func = hw_GetEventHandler(STRPTR name, struct hwTagList *tags);
```

FUNCTION

This function can be used to obtain information about an event handler registered using `hw_RegisterEventHandler()` or `hw_RegisterEventHandlerEx()`. Precisely, `hw_GetEventHandler()` can be used to get the reference to the user function and data associated with the event handler by calling Hollywood's `InstallEventHandler()` function.

If the script currently doesn't listen to the event handler, i.e. the script didn't install a callback for this event using `InstallEventHandler()`, `hw_GetEventHandler()` will return -1. Otherwise, a reference value is returned which can be used to obtain the callback function from the Lua registry.

Additionally, you can pass a tag list containing the following tags:

`HWGEHTAG_USERDATA:`
: Set the `pData` of this tag to a pointer to an `int` and `hw_GetEventHandler()` will store the reference to the user data associated with this event handler in this pointer. If there is no user data associated with the event handler, `hw_GetEventHandler()` will write -1 to the pointer.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`name` name of the event type; must be identical to the name passed to`hw_RegisterEventHandler()` or `hw_RegisterEventHandlerEx()`

`tags` tag list containing further options or `NULL`

RESULTS

`func` reference to user callback function for this event handler or -1

34.19 hw_GetSysTime

NAME

hw_GetSysTime – get system time (V5.0)

SYNOPSIS

```
void hw_GetSysTime(struct hwos_TimeVal *tv);
```

FUNCTION

This function queries the system time and stores it in the `struct hwos_TimeVal` you pass to this function. The system time is counted from a platform-dependent start time defined as 0 and is monotonically increasing. You can use the related functions `hw_SubTime()`, `hw_AddTime()`, and `hw_CmpTime()` to work with the time stamps returned by this function.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`tv` pointer to a `struct hwos_TimeVal` to hold the current system time

34.20 hw_GetVMErrorInfo

NAME
hw_GetVMErrorInfo – get extended error information from VM (V6.1)

SYNOPSIS

```
int error = hw_GetVMErrorInfo(lua_State *L, struct hwGVMErrorInfo *vme,
                struct hwTagList *tags);
```

FUNCTION
This function can be used to obtain extended error information from the Lua VM in case `lua_pcall()` exited with an error code, i.e. `lua_pcall()` returned anything except 0. You have to pass the `lua_State` pointer and a pointer to a `struct hwGVMErrorInfo` to this function. `hw_GetVMErrorInfo()` will then fill the `struct hwGVMErrorInfo` with extended error information. `struct hwGVMErrorInfo` looks like this:

```
struct hwGVMErrorInfo
{
    int Line;
    STRPTR Function;
    STRPTR File;
};
```

`hw_GetVMErrorInfo()` will write the following information to the structure pointer passed to it:

`Line:` This will be set to the line number in the script where the error occurred.

`Function:`
This will be set to a string containing the name of the function in which the error occurred.

`File:` This will be set to a string containing the name of the file in which the error occurred.

Note that you should call `hw_GetVMErrorInfo()` right after `lua_pcall()` has returned with an error code. Otherwise there is no guarantee that the returned information is still valid.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`L` pointer to the `lua_State`

`vme` pointer to a `struct hwGVMErrorInfo` to be filled by the function

`tags` reserved for future use; pass `NULL` for the time being

RESULTS

`error` error code or 0 for success

34.21 hw_HandleEvents

NAME

hw_HandleEvents – handle events (V6.0)

SYNOPSIS

```
int error = hw_HandleEvents(lua_State *L, ULONG flags, int *quit);
```

FUNCTION

This function reads all events that are currently in the queue and processes them. If there is an event that tells Hollywood to quit, the `int` pointer in the third parameter is set to `True`. Together with `hw_WaitEvents()` this function can be used to set up a temporary modal event loop.

The following flags are currently defined:

`HWHEFLAGS_LINEHOOK:`
: This flag must be set if `hw_HandleEvents()` has been called from the Lua line hook. This should never be set by you.

`HWHEFLAGS_MODAL:`
: This flag signals that `hw_HandleEvents()` has been called from a temporary modal event loop. You should always set this flag.

`HWHEFLAGS_CHECKEVENT:`
: This flag is set if `hw_HandleEvents()` has been called as a result of the script calling Hollywood's `CheckEvent()` command. This should never be set by you.

`HWHEFLAGS_WAITEVENT:`
: This flag is set if `hw_HandleEvents()` has been called as a result of the script calling Hollywood's `WaitEvent()` command. This should never be set by you.

`HWHEFLAGS_RUNCALLBACKS:`
: This flag signals that `hw_HandleEvents()` should also run any event callbacks that have triggered. This is useful on plugins which don't install a display adapter and hence cannot use `HWMSFLAGS_RUNCALLBACKS` because `hw_MasterServer()` must only be called by display adapters. Use this flag only if you have a very good reason to do so. Normally, you should leave callback execution to Hollywood. It will run event callbacks whenever the script calls `CheckEvent()` or `WaitEvent()` and normally you shouldn't have intervene in this design. (V6.1)

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L`	pointer to the `lua_State`
`flags`	combination of flags (see above)
`quit`	pointer to an `int` that is set to `True` if Hollywood shall quit

RESULTS

`error`	error code or 0 for success

34.22 hw_LockSemaphore

NAME

hw_LockSemaphore – lock a semaphore (V6.0)

SYNOPSIS

```
void hw_LockSemaphore(APTR sem);
```

FUNCTION

This function attempts to lock the specified semaphore handle. If another thread has already locked the semaphore, `hw_LockSemaphore()` will wait until that thread releases the semaphore again.

Note that `hw_LockSemaphore()` contains a nesting count. Every call to `hw_LockSemaphore()` must be matched by a call to hw_UnLockSemaphore().

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`sem`	semaphore handle allocated by `hw_AllocSemaphore()`

34.23 hw_LogPrintF

NAME

hw_LogPrintF – print to debug device (V6.0)

SYNOPSIS

```
void hw_LogPrintF(const char *fmt, va_list argptr);
```

FUNCTION

This function can be used to print the specified string to the current debug device. If you intend to use `printf()` style formatting codes be warned that the internal representation of `va_list` is compiler-dependent, so you might get crashes if your compiler uses a representation that is different from the compiler that was used to build Hollywood.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`fmt`	`printf()` style format string

`argptr` `va_list` containing the values for the format fields of `fmt`

34.24 hw_MasterControl

NAME

hw_MasterControl – control various internal attributes (V5.0)

SYNOPSIS

```
int c = hw_MasterControl(struct hwTagList *tags);
```

FUNCTION

This function can be used to set/get various internal attributes. You have to pass a pointer to a `hwTagList` to this function. `hw_MasterControl()` will iterate through this taglist and set/get all the individual tags in the list. It will return the number of tags successfully handled.

The following tags are currently recognized:

`HWMCP_GETPOWERPCBASE:`
: In the WarpOS version of Hollywood, `pData` will be set to a pointer of `PowerPCBase`. In all other versions, `NULL` will be written to `pData`.

`HWMCP_GETAPPTITLE:`
: This tag returns the application's title as specified in the `@APPTITLE` preprocessor command. You have to pass a pointer to a `STRPTR` in `pData`. Hollywood will set this `STRPTR` to the application's title then. Hollywood will never write `NULL` to the `STRPTR` but it may return an empty string. (V5.2)

`HWMCP_GETAPPVERSION:`
: This tag returns the application's version as specified in the `@APPVERSION` preprocessor command. You have to pass a pointer to a `STRPTR` in `pData`. Hollywood will set this `STRPTR` to the application's version then. Hollywood will never write `NULL` to the `STRPTR` but it may return an empty string. (V5.2)

`HWMCP_GETAPPCOPYRIGHT:`
: This tag returns the application's copyright text as specified in the `@APPCOPYRIGHT` preprocessor command. You have to pass a pointer to a `STRPTR` in `pData`. Hollywood will set this `STRPTR` to the application's copyright text then. Hollywood will never write `NULL` to the `STRPTR` but it may return an empty string. (V5.2)

`HWMCP_GETAPPAUTHOR:`
: This tag returns the application's author as specified in the `@APPAUTHOR` preprocessor command. You have to pass a pointer to a `STRPTR` in `pData`. Hollywood will set this `STRPTR` to the application's author then. Hollywood will never write `NULL` to the `STRPTR` but it may return an empty string. (V5.2)

HWMCP_GETAPPDESCRIPTION:
: This tag returns the application's description as specified in the @APPDESCRIPTION preprocessor command. You have to pass a pointer to a STRPTR in pData. Hollywood will set this STRPTR to the application's description then. Hollywood will never write NULL to the STRPTR but it may return an empty string. (V5.2)

HWMCP_GETAPPIDENTIFIER:
: This tag returns the application's identifier as specified in the @APPIDENTIFIER preprocessor command. You have to pass a pointer to a STRPTR in pData. Hollywood will set this STRPTR to the application's identifier then. Hollywood will never write NULL to the STRPTR but it may return an empty string. (V6.1)

HWMCP_SETCALLBACKMODE:
: This tag enables or disables callback mode according to the value passed in iData. You have to pass True to enable callback mode or False to disable it. Callback mode should be enabled whenever one of your plugin's Lua functions runs another Lua function by calling lua_pcall(). Please note that HWMCP_SETCALLBACKMODE contains a nesting count. Thus, every enable operation must be matched by a disable operation. (V6.0)

HWMCP_GETGTKREADY:
: This tag returns a boolean value indicating whether GTK has been setup correctly on Linux. You have to set pData to a pointer to an int. Hollywood will then write either True or False to this int. This tag is only supported on Linux. (V6.0)

HWMCP_SETDISABLELINEHOOK:
: This tag can be used to enable or disable Hollywood's Lua line hook. By default, the line hook is called after executing one line of Lua code. This leads to some overhead because the line hook will handle window events and update video frames among other things. To increase performance in certain situations, your plugin can temporarily disable this line hook by setting iData to True in this tag. However, make sure that you enable it again as soon as possible because several key features of Hollywood won't work while the line hook is disabled. This tag expects either True or False in iData. (V6.0)

HWMCP_GETFPSLIMIT:
: This tag returns the FPS limit that has been set by a call to Hollywood's SetFPSLimit() command. You have to set pData to a pointer to an int. Hollywood will then write the FPS limit to this int. (V6.0)

HWMCP_GETDESIGNERVERSION:
: This tag returns the version of Hollywood Designer if your plugin has been opened by Hollywood Designer. You have to set pData to a pointer to a ULONG. The upper 16-bits of the ULONG will then receive Designer's version number whereas the revision number will be written to the lower 16-bits. Obviously, this tag is only recognized by Hollywood Designer and not by

Hollywood itself. Note that Designer 4.0 doesn't support this tag. If your plugin was opened by Designer and `hw_MasterControl()` fails to obtain this tag, you can be sure that Designer 4.0 is handling your plugin. (V6.0)

`HWMCP_SETGLOBALQUIT:`
This tag can be used to change the state of Hollywood's global quit flag. If the global quit flag is set to `True`, Hollywood will immediately shutdown. In contrast to posting `HWEVT_QUIT` using `hw_PostEvent()`, setting the global quit flag will cause an instant shut down of Hollywood. (V6.1)

`HWMCP_SETLIGHTCHKEVT:`
Set this tag to `True` to make Hollywood's `CheckEvent()` function only call into your display adapter's `HandleEvents()` function and do nothing else. Normally, Hollywood's `CheckEvent()` function will also run event callbacks if events have triggered. You can prohibit this behaviour by setting this flag to `True`. If `HWMCP_SETLIGHTCHKEVT` has been set to `True`, `CheckEvent()` will only call your display adapter's `HandleEvents()` function and do nothing else. (V6.1)

`HWMCP_RESETERRORFLAG:`
Whenever `lua_pcall()` exits with an error, Hollywood expects a program shutdown and sets several internal flags to prepare this complete shutdown. If you want to keep the program running even after `lua_pcall()` returned an error, you have to execute `HWMCP_RESETERRORFLAG` to reset all internal error flags. Then Hollywood can continue running without any issues. Note that this tag doesn't take any data. Both `iData` and `pData` elements are ignored. (V6.1)

`HWMCP_SETAMIGASIGNALERROR:`
This tag can be used from within an Amiga signal callback installed using `hw_RegisterCallback()`. It allows the Amiga signal callback to pass an error code back to Hollywood. This is a glue code feature which works around the design flaw that callbacks of type `HWCB_AMIGASIGNAL` can't pass an error code back to Hollywood. If you need to pass an error code back to Hollywood, use this tag and set the `iData` member of it to the error code that should be passed back to Hollywood. See Section 34.30 [hw_RegisterCallback], page 308, for details. (V6.1)

`HWMCP_SETDISABLERAISEONERROR:`
This tag can be used to temporarily disable an error handling function installed by Hollywood's `RaiseOnError()` function. Pass `True` in `iData` to disable the error handling function, `False` to enable it again. Normally it is not necessary to mess with error handlers installed by `RaiseOnError()` but in plugins doing really advanced things it might be convenient to have control over the error handler. Also see the `hw_RaiseOnError()` function made available by SysBase. See Section 34.29 [hw_RaiseOnError], page 308, for details. (V7.0)

DESIGNER COMPATIBILITY
Supported since Designer 4.

INPUTS

`tags` pointer to a `struct hwTagList` containing various tags (see above)

RESULTS

`c` the number of tags successfully handled

34.25 hw_MasterServer

NAME

hw_MasterServer – call into the master server (V6.0)

SYNOPSIS

```
int error = hw_MasterServer(lua_State *L, ULONG flags,
                struct hwTagList *tags);
```

FUNCTION

This function allows you to call into Hollywood's master server. This is necessary when you install a display adapter and do your own event processing to give Hollywood a chance to manage its asynchronous operations. The `flags` parameter determines which sections of the master server you want to enter. The following flags are currently defined:

`HWMSFLAGS_RUNCALLBACKS:`
: This will run all user callbacks for events that have triggered. It does the same as the Hollywood functions `CheckEvent()` and `WaitEvent()`. If for some reason your display adapter cannot delegate event handling to `CheckEvent()` or `WaitEvent()`, the `HWMSFLAGS_RUNCALLBACKS` flag allows you to manually force Hollywood to run user event callbacks when you need it. Use this flag with care. It's only needed under very special circumstances.

`HWMSFLAGS_DRAWVIDEOS:`
: This flag will update all videos that are currently playing, if necessary. Your display adapter needs to call `hw_MasterServer()` with this flag set whenever it does some event processing. Hollywood's video server worker threads will wake up your event loop using `ForceEventLoopIteration()` whenever a video needs updating. That's why you should always call `hw_MasterServer()` with `HWMSFLAGS_DRAWVIDEOS` whenever you process your window events, preferably in your `hw_HandleEvents()` implementation.

Please note that this function should only be used by display adapter plugins. If Hollywood is using its default display adapter, it will take care of calling into the master server on its own.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L` pointer to the `lua_State`

`flags` flags indicating what to do (see above)

`tags` currently unused; pass `NULL`

34.26 hw_PostEvent

NAME

hw_PostEvent – post a new event to queue (V5.2)

SYNOPSIS

```
void hw_PostEvent(int type, APTR userdata);
```

FUNCTION

This function posts a new event to Hollywood's event queue. Prior to Hollywood 6.0 `hw_PostEvent()` could only be used with event types that had been registered using `hw_RegisterEventHandler()`. Starting with Hollywood 6.0 `hw_PostEvent()` supports several new event types. Here is a list:

`HWEVT_QUIT:`

If you post this event, Hollywood will quit. The user data parameter is unused for this event. Note that posting `HWEVT_QUIT` will shut down Hollywood as soon as it runs its event handler. To make Hollywood quit immediately, you can set the global quit flag using `HWMCP_SETGLOBALQUIT`. See Section 34.24 [hw_MasterControl], page 291, for details. (V6.0)

`HWEVT_CALLFUNCTION:`

If you post this event, Hollywood will call the specified function the next time it enters a `WaitEvent()` or `CheckEvent()` cycle. You have to pass a pointer to a `struct hwEvtCallFunction` in the user data parameter. `struct hwEvtCallFunction` looks like this:

```
struct hwEvtCallFunction
{
    int (*Func)(lua_State *L, APTR userdata);
    APTR UserData;
};
```

The following members are part of `struct hwEvtCallFunction`:

`Func:` This must be set to the function that you want Hollywood to call. Hollywood will pass a pointer to the `lua_State` and the user data you specify below to your function.

`UserData:` Set this to the user data that Hollywood should pass to your function.

(V6.0)

`HWEVT_WAKEUP:`

This event can be used to wake up Hollywood from another thread. The user data parameter is unused for this event. (V6.0)

`HWEVT_MOUSE:`

This event can only be used from plugins that install a display adapter. `HWEVT_MOUSE` posts a mouse event to Hollywood's event queue. You have to pass a pointer to a `struct hwEvtMouse` in the user data parameter. `struct hwEvtMouse` looks like this:

```
struct hwEvtMouse
```

```
{
    APTR Handle;
    int X;
    int Y;
    int Button;
    int Down;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtMouse`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`X:` This must be set to the mouse cursor's current x position, relative to the upper-left corner of the display.

`Y:` This must be set to the mouse cursor's current y position, relative to the upper-left corner of the display.

`Button:` This must be set to one of the following predefined constants to define the mouse button that `Down` member of `struct hwEvtMouse` is referring to. The following button types are currently defined:

`HWMBTYPE_NONE:`
Structure member `Down` is unused.

`HWMBTYPE_LEFT:`
Structure member `Down` contains the current state of the left mouse button.

`HWMBTYPE_RIGHT:`
Structure member `Down` contains the current state of the right mouse button.

`HWMBTYPE_MIDDLE:`
Structure member `Down` contains the current state of the middle mouse button.

`Down:` If `Button` does not equal `HWBMTYPE_NONE`, this member needs to be set to either `True` or `False` depending on whether the corresponding mouse button is currently down or up.

`Flags:` Currently unused. Must be 0.

(V6.0)

`HWEVT_KEYBOARD:`
This event can only be used from plugins that install a display adapter. `HWEVT_KEYBOARD` posts a keyboard event to Hollywood's event queue. You have to pass a pointer to a `struct hwEvtKeyboard` in the user data parameter. `struct hwEvtKeyboard` looks like this:

```
struct hwEvtKeyboard
```

```
{
    APTR Handle;
    int ID;
    int Down;
    ULONG Qualifiers;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtKeyboard`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`ID:` This must be set to the identifier of the key this event is referring to. This can be either the 8-bit character code of a ISO 8859-1 key or one of the following special keys:

```
HWKEY_CURSOR_UP
HWKEY_CURSOR_DOWN
HWKEY_CURSOR_RIGHT
HWKEY_CURSOR_LEFT
HWKEY_HELP
HWKEY_F1
HWKEY_F2
HWKEY_F3
HWKEY_F4
HWKEY_F5
HWKEY_F6
HWKEY_F7
HWKEY_F8
HWKEY_F9
HWKEY_F10
HWKEY_F11
HWKEY_F12
HWKEY_F13
HWKEY_F14
HWKEY_F15
HWKEY_F16
HWKEY_BACKSPACE
HWKEY_TAB
HWKEY_ENTER
HWKEY_RETURN
HWKEY_ESC
HWKEY_SPACE
HWKEY_DEL
HWKEY_INSERT
HWKEY_HOME
HWKEY_END
HWKEY_PAGEUP
HWKEY_PAGEDOWN
```

```
HWKEY_PRINT
HWKEY_PAUSE
```

`Down:` This must be set to either `True` or `False` indicating whether the specified key is currently pressed.

`Qualifiers:`
This must be set to a combination of qualifiers like shift and alt that are currently pressed. See Section 10.26 [GetQualifiers], page 81, for a list of available qualifiers. Don't forget that the internal control bit `HWKEY_QUAL_MASK` must always be set.

`Flags:` Currently unused. Must be 0.

Note that `HWEVT_KEYBOARD` can only be used for ISO 8859-1 characters and control keys. To post a Unicode key to Hollywood's event queue, use `HWEVT_VANILLAKEY` instead (see below for details). (V6.0)

`HWEVT_CLOSEDISPLAY:`
This event can only be used from plugins that install a display adapter. `HWEVT_CLOSEDISPLAY` posts a close display event to Hollywood's event queue. This event is usually posted when the user presses the window's close widget. You have to pass a pointer to a `struct hwEvtCloseDisplay` in the user data parameter. `struct hwEvtCloseDisplay` looks like this:

```
struct hwEvtCloseDisplay
{
    APTR Handle;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtCloseDisplay`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`Flags:` Currently unused. Must be 0.

(V6.0)

`HWEVT_SIZEDISPLAY:`
This event can only be used from plugins that install a display adapter. `HWEVT_SIZEDISPLAY` posts a size display event to Hollywood's event queue. This event is usually posted when the user changes the size of the window. You have to pass a pointer to a `struct hwEvtSizeDisplay` in the user data parameter. `struct hwEvtSizeDisplay` looks like this:

```
struct hwEvtSizeDisplay
{
    APTR Handle;
    int Width;
    int Height;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtSizeDisplay`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`Width:` This must be set to the new display width in pixels.

`Height:` This must be set to the new display height in pixels.

`Flags:` Currently unused. Must be 0.

(V6.0)

`HWEVT_MOVEDISPLAY:`

This event can only be used from plugins that install a display adapter. `HWEVT_MOVEDISPLAY` posts a move display event to Hollywood's event queue. This event is usually posted when the user moves the window around. You have to pass a pointer to a `struct hwEvtMoveDisplay` in the user data parameter. `struct hwEvtMoveDisplay` looks like this:

```
struct hwEvtMoveDisplay
{
    APTR Handle;
    int X;
    int Y;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtMoveDisplay`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`X:` This must be set to the display's new x position in pixels. The position you specify here must be relative to the screen's upper-left corner.

`Y:` This must be set to the display's new y position in pixels. The position you specify here must be relative to the screen's upper-left corner.

`Flags:` Currently unused. Must be 0.

(V6.0)

`HWEVT_SHOWHIDEDISPLAY:`

This event can only be used from plugins that install a display adapter. `HWEVT_SHOWHIDEDISPLAY` posts a show/hide display event to Hollywood's event queue. This event is usually posted when the user minimizes the window or restores it from a minimized state. You have to pass a pointer to a `struct hwEvtShowHideDisplay` in the user data parameter. `struct hwEvtShowHideDisplay` looks like this:

```
struct hwEvtShowHideDisplay
{
    APTR Handle;
```

```
    int Show;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtShowHideDisplay`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`Show:` This must be set to `False` if the display has been minimized/hidden and `True` if it has been restored from a minimized/hidden state.

`Flags:` Currently unused. Must be 0.

(V6.0)

`HWEVT_FOCUSCHANGEDISPLAY:`

This event can only be used from plugins that install a display adapter. `HWEVT_FOCUSCHANGEDISPLAY` posts a focus change display event to Hollywood's event queue. This event is usually posted when the window loses or gets the focus. You have to pass a pointer to a `struct hwEvtFocusChangeDisplay` in the user data parameter. `struct hwEvtFocusChangeDisplay` looks like this:

```
struct hwEvtFocusChangeDisplay
{
    APTR Handle;
    int Focus;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtFocusChangeDisplay`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`Focus:` This must be set to `False` if the display has lost the focus and `True` if it has received the focus.

`Flags:` Currently unused. Must be 0.

(V6.0)

`HWEVT_VANILLAKEY:`

This event can only be used from plugins that install a display adapter. `HWEVT_VANILLAKEY` posts a keyboard event to Hollywood's event queue. In contrast to `HWEVT_KEYBOARD`, `HWEVT_VANILLAKEY` must only be used to post printable characters (including the space character). It must not be used for control keys. These should be posted as `HWEVT_KEYBOARD` events only (see above for details). In contrast to `HWEVT_KEYBOARD`, `HWEVT_VANILLAKEY` supports the full Unicode character range. You have to pass a pointer to a `struct hwEvtKeyboard` in the user data parameter. `struct hwEvtKeyboard` looks like this:

```
struct hwEvtKeyboard
```

```
{
    APTR Handle;
    int ID;
    int Down;
    ULONG Qualifiers;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtKeyboard`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`ID:` This must be set to the Unicode character code of the key event you want to post.

`Down:` This is ignored for `HWEVT_VANILLAKEY`.

`Qualifiers:`
This is ignored for `HWEVT_VANILLAKEY`.

`Flags:` Currently unused. Must be 0.

(V7.0)

`HWEVT_DROPFILE:`
This event can only be used from plugins that install a display adapter. `HWEVT_DROPFILE` can be used to post an event to Hollywood's event queue which indicates that files have been dropped on a Hollywood display. You have to pass a pointer to a `struct hwEvtDropFile` in the user data parameter. `struct hwEvtDropFile` looks like this:

```
struct hwEvtDropFile
{
    APTR Handle;
    int MouseX;
    int MouseY;
    STRPTR DropFiles;
    ULONG Flags;
};
```

The following members are part of `struct hwEvtDropFile`:

`Handle:` This must be set to a display handle obtained from `OpenDisplay()`.

`MouseX:` This must be set to the x-coordinate (relative to the window's upper-left corner) where the user dropped the file(s).

`MouseY:` This must be set to the y-coordinate (relative to the window's upper-left corner) where the user dropped the file(s).

`DropFiles:`
This must be set to a list of filenames that have been dropped on the window. The individual filenames need to contain fully

qualified paths and must be separated from one another by a single `NULL` terminator byte whereas the complete list is terminated by two `NULL` terminator bytes to signal the list end to Hollywood.

`Flags:` Currently unused. Must be 0.

(V7.0)

Alternatively, you can also specify a custom event type that you have registered through `hw_RegisterEventHandler()` in parameter 1. In that case, the user data pointer you pass in parameter 2 is directly forwarded to your custom event's handler function. See Section 34.32 [hw_RegisterEventHandler], page 311, for details.

Starting with Hollywood 6.0 this function has an extended version named `hw_PostEventEx()`. In contrast to `hw_PostEvent()`, the extended version will tell you whether the event was successfully posted. See Section 34.27 [hw_PostEventEx], page 302, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`type`	type of event to post (see above)
`userdata`	event-dependent data (see above)

34.27 hw_PostEventEx

NAME

hw_PostEventEx – post a new event to queue (V6.0)

SYNOPSIS

```
int error = hw_PostEventEx(lua_State *L, int type, APTR userdata,
                           struct hwTagList *tags);
```

FUNCTION

This function does the same as `hw_PostEvent()` but returns an error code that informs you whether or not the event could be added successfully. Additionally, it accepts a tag list parameter but there are currently no tags that are supported here. See Section 34.26 [hw_PostEvent], page 295, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L`	pointer to a `lua_State`
`type`	type of event to post
`userdata`	event-dependent data

`tags` currently unused; set this to `NULL`

34.28 hw_PostSatelliteEvent

NAME

hw_PostSatelliteEvent – post a satellite event to queue (V5.2)

SYNOPSIS

```
void hw_PostSatelliteEvent(APTR handle, int type, APTR typedata);
```

FUNCTION

This function posts an event that has happened in a display satellite to the satellite's root display. You have to pass the satellite handle as returned by `hw_AttachDisplaySatellite()` in parameter 1 and the event type and its data in parameters 2 and 3. The following event types are currently supported:

`HWSATEVT_MOUSEMOVE:`

This event indicates that the mouse has been moved in the display satellite. You need to pass a pointer to a `struct hwSatelliteEventMouse` in `typedata`. The structure looks like this:

```
struct hwSatelliteEventMouse
{
    int MouseX;
    int MouseY;
    int ButtonDown;
};
```

The following members need to be initialized for `HWSATEVT_MOUSEMOVE`:

`MouseX:` This must be set to the mouse cursor's current x position, relative to the upper-left corner of the display satellite.

`MouseY:` This must be set to the mouse cursor's current y position, relative to the upper-left corner of the display satellite.

`HWSATEVT_LEFTMOUSE:`

This event indicates that the left mouse button has been pressed or released in the display satellite. You need to pass a pointer to a `struct hwSatelliteEventMouse` in `typedata`. The structure looks like this:

```
struct hwSatelliteEventMouse
{
    int MouseX;
    int MouseY;
    int ButtonDown;
};
```

The following members need to be initialized for `HWSATEVT_LEFTMOUSE`:

`MouseX:` This must be set to the mouse cursor's current x position, relative to the upper-left corner of the display satellite.

`MouseY:` This must be set to the mouse cursor's current y position, relative to the upper-left corner of the display satellite.

`ButtonDown:`
This must be set to `True` or `False` depending on whether the left mouse button is down or not.

`HWSATEVT_RIGHTMOUSE:`
This event indicates that the right mouse button has been pressed or released in the display satellite. You need to pass a pointer to a `struct hwSatelliteEventMouse` in `typedata`. The structure looks like this:

```
struct hwSatelliteEventMouse
{
    int MouseX;
    int MouseY;
    int ButtonDown;
};
```

The following members need to be initialized for `HWSATEVT_RIGHTMOUSE`:

`MouseX:` This must be set to the mouse cursor's current x position, relative to the upper-left corner of the display satellite.

`MouseY:` This must be set to the mouse cursor's current y position, relative to the upper-left corner of the display satellite.

`ButtonDown:`
This must be set to `True` or `False` depending on whether the right mouse button is down or not.

`HWSATEVT_MIDMOUSE:`
This event indicates that the middle mouse button has been pressed or released in the display satellite. You need to pass a pointer to a `struct hwSatelliteEventMouse` in `typedata`. The structure looks like this:

```
struct hwSatelliteEventMouse
{
    int MouseX;
    int MouseY;
    int ButtonDown;
};
```

The following members need to be initialized for `HWSATEVT_MIDMOUSE`:

`MouseX:` This must be set to the mouse cursor's current x position, relative to the upper-left corner of the display satellite.

`MouseY:` This must be set to the mouse cursor's current y position, relative to the upper-left corner of the display satellite.

`ButtonDown:`
This must be set to `True` or `False` depending on whether the middle mouse button is down or not.

HWSATEVT_MOUSEWHEEL:
This event indicates that the mouse wheel has been rotated in the display satellite. You need to pass a pointer to a struct hwSatelliteEventMouse in typedata. The structure looks like this:

```
struct hwSatelliteEventMouse
{
    int MouseX;
    int MouseY;
    int ButtonDown;
};
```

The following members need to be initialized for HWSATEVT_LEFTMOUSE:

MouseX: This must be set to the mouse cursor's current x position, relative to the upper-left corner of the display satellite.

MouseY: This must be set to the mouse cursor's current y position, relative to the upper-left corner of the display satellite.

ButtonDown:
This must be set to True if the wheel has been spinned downwards or False if it has been spinned in upwards direction.

HWSATEVT_KEYBOARD:
A keyboard event has occurred in the display satellite. You need to pass a pointer to a struct hwSatelliteEventKeyboard in typedata. The structure looks like this:

```
struct hwSatelliteEventKeyboard
{
    int KeyID;
    int KeyDown;
    ULONG Qualifiers;
};
```

The individual structure members need to be initialized like this:

KeyID: This must be set to the identifier of the key this event is referring to. This can be either the 8-bit character code of a ISO 8859-1 key or one of the following special keys:

```
HWKEY_CURSOR_UP
HWKEY_CURSOR_DOWN
HWKEY_CURSOR_RIGHT
HWKEY_CURSOR_LEFT
HWKEY_HELP
HWKEY_F1
HWKEY_F2
HWKEY_F3
HWKEY_F4
HWKEY_F5
HWKEY_F6
HWKEY_F7
```

```
HWKEY_F8
HWKEY_F9
HWKEY_F10
HWKEY_F11
HWKEY_F12
HWKEY_F13
HWKEY_F14
HWKEY_F15
HWKEY_F16
HWKEY_BACKSPACE
HWKEY_TAB
HWKEY_ENTER
HWKEY_RETURN
HWKEY_ESC
HWKEY_SPACE
HWKEY_DEL
HWKEY_INSERT
HWKEY_HOME
HWKEY_END
HWKEY_PAGEUP
HWKEY_PAGEDOWN
HWKEY_PRINT
HWKEY_PAUSE
```

`KeyDown:` This must be set to either `True` or `False` indicating whether the specified key is currently pressed.

`Qualifiers:`
This must be set to a combination of qualifiers like shift and alt that are currently pressed. See Section 10.26 [GetQualifiers], page 81, for a list of available qualifiers. Don't forget that the internal control bit `HWKEY_QUAL_MASK` must always be set.

Note that `HWSATEVT_KEYBOARD` can only be used for ISO 8859-1 characters and control keys. To post a Unicode key to the satellite's root display, use `HWSATEVT_VANILLAKEY` instead (see below for details).

`HWSATEVT_VANILLAKEY:`
Post a Unicode key event to the satellite's root display. In contrast to `HWSATEVT_KEYBOARD`, `HWSATEVT_VANILLAKEY` must only be used to post printable characters (including the space character). It must not be used for control keys. These should be posted as `HWSATEVT_KEYBOARD` events only (see above for details). In contrast to `HWSATEVT_KEYBOARD`, `HWSATEVT_VANILLAKEY` supports the full Unicode character range. You need to pass a pointer to a `struct hwSatelliteEventKeyboard` in `typedata`. The structure looks like this:

```
struct hwSatelliteEventKeyboard
{
    int KeyID;
```

```
        int KeyDown;
        ULONG Qualifiers;
    };
```

The individual structure members need to be initialized like this:

`KeyID:` This must be set to the Unicode character code of the key event you want to post.

`KeyDown:` This is ignored for `HWSATEVT_VANILLAKEY`.

`Qualifiers:`
This is ignored for `HWSATEVT_VANILLAKEY`.

(V7.0)

`HWSATEVT_DROPFILE:`
This event can be used to post an event to the satellite's root display which indicates that files have been dropped on a display satellite. You need to pass a pointer to a `struct hwSatelliteEventDropFile` in `typedata`. The structure looks like this:

```
struct hwSatelliteEventDropFile
{
    int MouseX;
    int MouseY;
    STRPTR DropFiles;
};
```

The individual structure members need to be initialized like this:

`MouseX:` This must be set to the x-coordinate (relative to the satellite's upper-left corner) where the user dropped the file(s).

`MouseY:` This must be set to the y-coordinate (relative to the satellite's upper-left corner) where the user dropped the file(s).

`DropFiles:`
This must be set to a list of filenames that have been dropped on the satellite. The individual filenames need to contain fully qualified paths and must be separated from one another by a single `NULL` terminator byte whereas the complete list is terminated by two `NULL` terminator bytes to signal the list end to Hollywood.

(V7.0)

See Section 28.3 [hw_AttachDisplaySatellite], page 225, for more information on display satellites.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`handle` display satellite handle allocated by `hw_AttachDisplaySatellite()`

`type`	type of event to post (see above)
`typedata`	event-dependent data (see above)

34.29 hw_RaiseOnError

NAME

hw_RaiseOnError – call user error handling function (V7.0)

SYNOPSIS

```
int r = hw_RaiseOnError(lua_State *L, int error, struct hwTagList *tags);
```

FUNCTION

This function can be used to run an error handling function installed by the Hollywood script using Hollywood's `RaiseOnError()` command. It returns 0 if there was an error handling function and it handled the error. Otherwise it simply returns the error code you passed in `error`.

This function is only here for people who need fine-tuned control over Hollywood's automatic error handler. Normally it is not needed to call this function manually. Under normal circumstances, managing error handlers should be left to Hollywood. If you need fine-tuned control over the error handler, you may also want to take a look at the `HWMCP_SETDISABLERAISEONERROR` tag for `hw_MasterControl()`. See Section 34.24 [hw_MasterControl], page 291, for details.

Note that certain error codes need additional information that you have to specify by calling either `hw_SetErrorString()` or `hw_SetErrorCode()`. See Section 2.7 [Error codes], page 11, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L`	pointer to the `lua_State`
`error`	error code you want to throw
`tags`	reserved for future use; set it to `NULL` for now

RESULTS

`r`	0 if there is an error handling function and it was run successfully, otherwise the error code you passed in

34.30 hw_RegisterCallback

NAME

hw_RegisterCallback – register a new callback (V5.2)

SYNOPSIS

```
APTR cb = hw_RegisterCallback(int type, APTR func, APTR userdata);
```

FUNCTION

This function can be used to register a new callback that will be run by Hollywood in a certain context which is specified by `type`. The following callback types are currently supported:

`HWCB_AMIGASIGNAL:`

This is only supported in the AmigaOS compatible versions of Hollywood. Hollywood will run callbacks of the type `HWCB_AMIGASIGNAL` whenever it is about to call `exec.library/Wait()` to wait on a set of signals. Your callback will then be asked for additional signal bits that should be included in the call to `Wait()`. The prototype for an Amiga signal callback looks like this:

```
ULONG AmigaSignal(APTR userdata);
```

Hollywood will pass the user data that you pass in parameter 3 of `hw_RegisterCallback()` to your `AmigaSignal()` callback. Your `AmigaSignal()` callback has to return a combination of signal bits that should be included in the call to `Wait()`. There are two special return values: If you return 0, Hollywood will run your `AmigaSignal()` again. If you return `0xFFFFFFFF`, no additional signals will be included in the call to `Wait()`. If your callback needs to return an error code to Hollywood, you have to use `HWMCP_SETAMIGASIGNALERROR` for that. See Section 34.24 [hw_MasterControl], page 291, for details.

`HWCB_LINEHOOK:`

Callbacks of type `HWCB_LINEHOOK` will be run whenever Hollywood runs its line hook, i.e. after running one line of Lua code. Line hooks are called very often, usually many times per second. Thus, you must make sure that your line hook callback doesn't do any expensive things or it will slow down the script's execution significantly. The prototype for a line hook callback looks like this:

```
int LineHook(lua_State *L, APTR userdata);
```

Your line hook callback will receive a pointer to the `lua_State` as well as the user data that has been passed to `hw_RegisterCallback()` when registering the callback. Your callback has to return a standard Hollywood error code or 0 for success. (V6.0)

`HWCB_SHOWHIDEAPP:`

This is only supported in the AmigaOS compatible versions of Hollywood. Hollywood will run callbacks of type `HWCB_SHOWHIDEAPP` whenever the user changes the visibility of an application via commodities' Exchange tool or application.library on AmigaOS4. The prototype for a show/hide app callback looks like this:

```
void ShowHideApp(int show, APTR userdata);
```

When the application is shown, Hollywood will pass `True` in the `show` parameter and when the application is hidden, this parameter will be set to `False`. (V6.1)

`HWCB_ENCODINGCHANGE:`

Callbacks of type `HWCB_ENCODINGCHANGE` will be run whenever the encoding of the current script changes. This can only happen once for each script. Since all scripts run in UTF-8 encoding by default starting in Hollywood 7.0, the only change that you can be notified of through this callback is an encoding change from UTF-8 to ISO 8859-1. This can only happen if a script explicitly requests ISO 8859-1 to run in compatibility mode. Normally, all scripts should run in UTF-8 encoding starting with Hollywood 7.0. The prototype for an encoding change callback looks like this:

```
void EncodingChange(int encoding, APTR userdata);
```

The `encoding` parameter will contain the new encoding. As described above, this can only ever be `HWOS_ENCODING_ISO8859_1` since the default encoding is `HWOS_ENCODING_UTF8`. Note that there is also a `hw_GetEncoding()` function to get the current encoding but this is not so convenient because the script encoding hasn't been set yet when the plugin is initialized. See Section 34.16 [hw_GetEncoding], page 285, for details. (V7.0)

`HWCB_DROPFILECHANGE:`

Callbacks of type `HWCB_DROPFILECHANGE` will be run whenever the user installs or deinstalls a listener on the `"OnDropFile"` event handler using Hollywood's `InstallEventHandler()` function. This is especially useful for display adapters which want to support drag'n'drop through Hollywood's standard event handlers and need to take some action whenever the user enables or disables drag'n'drop by installing or removing a listener on the `"OnDropFile"` event handler. The prototype for a drop file change callback looks like this:

```
void DropFileChange(APTR d, int enable, APTR userdata);
```

The callback will receive a pointer to the display for which drag'n'drop should be enabled or disabled in the `d` parameter and either `True` or `False` in the `enable` parameter. (V7.0)

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`type` callback type to register (see above for supported types)

`func` pointer to callback function; the actual format of this function depends on the specified callback type (see above)

`userdata` user data that should be passed to the callback function when Hollywood runs it

RESULTS

`cb` callback handle or `NULL` on error

34.31 hw_RegisterError

NAME

hw_RegisterError – register a new error code (V5.0)

SYNOPSIS

```
int error = hw_RegisterError(STRPTR msg);
```

FUNCTION

This function can be used to register custom Hollywood error codes. All functions that return a standard Hollywood error code may return this custom error code then to make Hollywood show the specified error message. The string you pass to this function may contain a `%s` or `%ld` wildcard. If that is the case, Hollywood will replace these wildcards with the values set using `hw_SetErrorString()` and `hw_SetErrorCode()`.

Hollywood will make a copy of the string you pass to this function so you may free it once `hw_RegisterError()` returns.

This function should only be called in your `InitPlugin()` function.

See Section 2.7 [Error codes], page 11, for more information on standard Hollywood error codes.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`msg`	null-terminated string describing the error message

RESULTS

`error`	new error code or `ERR_MEM` in case of an error

34.32 hw_RegisterEventHandler

NAME

hw_RegisterEventHandler – register a new event type (V5.2)

SYNOPSIS

```
int id = hw_RegisterEventHandler(STRPTR name, void (*evtfunc)
             (lua_State *L, int type, APTR userdata));
```

FUNCTION

This function allows you to register a new event type. The user will then be able to listen to events of this type by installing a new event handler for this type by using the Hollywood function `InstallEventHandler()`. Whenever you want this event handler to trigger, you have to post an event to Hollywood's event queue using the `hw_PostEvent()` or `hw_PostEventEx()` function.

You have to specify a name for the new event type. This name must follow the conventions for Hollywood variables because the user will need to use this name when installing a handler for this event type using Hollywood's `InstallEventHandler()` function. Thus, the name you specify here must not clash with any existing event handler names and

it must not use any spaces or special characters. It has to start with a letter from the English alphabet.

In parameter 2, you need to pass a function pointer to `hw_RegisterEventHandler()`. This function will be called whenever Hollywood is about to run the callback that the user has installed for the event using `InstallEventHandler()`. Your function may then push additional values on the stack. When `evtfunc` is called, Hollywood will have already pushed a table to the stack and it will have set the `Action` field of that table to the name of your event for consistency with inbuilt Hollywood events. All other values that your event handler may want to provide to the user's event callback should also be stored in that table. Note that you must implement this function even if you do not want to push any additional values on the stack. `evtfunc` must not be `NULL`. The user data that is passed to `evtfunc` as the third parameter is the same pointer that you provided in the call to `hw_PostEvent()` or `hw_PostEventEx()`.

This function is also available as an extended version. See Section 34.33 [hw_RegisterEventHandlerEx], page 312, for details.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`name` name for the new event type (see above for limitations in the event name format)

`evtfunc` function that should be called when running user callbacks for this event

RESULTS

`id` identifier of the new event or 0 on error

34.33 hw_RegisterEventHandlerEx

NAME

hw_RegisterEventHandlerEx – register a new event type (V6.0)

SYNOPSIS

```
int id = hw_RegisterEventHandlerEx(STRPTR name, int type, APTR data);
```

FUNCTION

This function allows you to register a new event type. The user will then be able to listen to events of this type by installing a new event handler for this type by using the Hollywood function `InstallEventHandler()`. Whenever you want this event handler to trigger, you have to post an event to Hollywood's event queue using the `hw_PostEvent()` or `hw_PostEventEx()` function.

You have to specify a name for the new event type. This name must follow the conventions for Hollywood variables because the user will need to use this name when installing a handler for this event type using Hollywood's `InstallEventHandler()` function. Thus, the name you specify here must not clash with any existing event handler names and it must not use any spaces or special characters. It has to start with a letter from the English alphabet.

In parameter 2, you need to pass the type of the event you would like to register. The following event types are currently recognized:

`HWEVTHANDLER_STANDARD:`

This registers a standard event. Standard events are the most basic event types available and they are also the ones that are registered by the `hw_RegisterEventHandler()` function. For standard events, the `data` parameter must be set to a pointer to a `struct hwStandardEventHandler` which looks like this:

```
struct hwStandardEventHandler
{
    void (*EvtFunc)(lua_State *L, int type, APTR userdata);
};
```

The following items are members of this structure:

`EvtFunc:` This member must be set to a function pointer which will be called whenever Hollywood is about to run the callback that the user has installed for the event using `InstallEventHandler()`. The function you specify here may then push additional values on the stack. When `evtfunc` is called, Hollywood will have already pushed a table to the stack and it will have set the `Action` field of that table to the name of your event for consistency with inbuilt Hollywood events. All other values that your event handler may want to provide to the user's event callback should also be stored in that table. Note that you must write this function even if you do not want to push any additional values on the stack. `EvtFunc` must not be `NULL`. The user data that is passed to `EvtFunc` as the third parameter is the same pointer that you provided in the call to `hw_PostEvent()` or `hw_PostEventEx()`.

`HWEVTHANDLER_STANDARD` is identical to the event handler registered by `hw_RegisterEventHandler()` so you might also use the latter to register standard events.

`HWEVTHANDLER_CUSTOM:`

This registers a custom event. Custom events differ from standard events in the way that your plugin is given more fine-tuned control about the way the events are handled. In contrast to standard events, Hollywood will call your plugin also after it has run a user callback for this event and it also supports event destructors. For custom events, the `data` parameter must be set to a pointer to a `struct hwCustomEventHandler` which looks like this:

```
struct hwCustomEventHandler
{
    int (*PushData)(lua_State *L, int type, APTR udata);
    void (*PostCall)(lua_State *L, int type, APTR udata);
    void (*FreeEvent)(lua_State *L, int type, APTR udata);
};
```

The following items are members of this structure:

`PushData:`
: This member must be set to a function pointer which will be called whenever Hollywood is about to run the callback that the user has installed for the event using `InstallEventHandler()`. The function you specify here may then push additional values on the stack. Note that in comparison to events of type `HWEVTHANDLER_STANDARD`, Hollywood won't push any values on the stack for custom events. There will also be no table on the stack as it is the case with `HWEVTHANDLER_STANDARD`. Instead, your plugin is given complete control over what the stack should look like when Hollywood runs the user callback. For consistency reasons, however, it is advised that you push a table, set the `Action` field to the name of your event and then store all additional values inside that table. You should only deviate from this standard if you have very good reason to do so.

`PostCall:`
: This member must be set to a function that should be called immediately after Hollywood has finished running the user callback registered for this event type.

`FreeEvent:`
: This member must be set to a function that should act as a destructor for this event type. Hollywood will call this function whenever it has to free an event of your type.

Note that you must provide functions for all of the structure members above, even if they don't do anything. No structure member must be `NULL`. The user data that is passed to all the functions above as the third parameter is the same pointer that you provided in the call to `hw_PostEvent()` or `hw_PostEventEx()` when you posted the event.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`name` name for the new event type (see above for limitations in the event name format)

`type` desired event type (see above for supported types)

`data` event type dependent data (see above for details)

RESULTS

`id` identifier of the new event or 0 on error

34.34 hw_RegisterFileType

NAME

hw_RegisterFileType – register a new file type (V5.3)

SYNOPSIS

```
int ok = hw_RegisterFileType(hwPluginBase *self, int type, STRPTR name,
                STRPTR mime, STRPTR ext, ULONG formatid, ULONG flags);
```

FUNCTION

This function can be used to register a new file type with Hollywood. Hollywood won't do anything with these file types except return information about them to the user if requested by a call to `GetPlugins()`. The information about the file types supported by the individual plugins can be useful to filter file name extensions when showing file requesters or it can be useful to ask the user to select a file format when saving an image, etc. This makes it possible for scripts to dynamically support all plugins that are currently available.

You have to pass a pointer to your plugin's `hwPluginBase` in parameter 1. Hollywood passes this pointer to you when it first calls your `InitPlugin()` function. You also have to specify the root type of the file format you want to register. The following types are currently supported:

`HWFILETYPE_IMAGE:`
: Register specified file type as an image file format.

`HWFILETYPE_ANIM:`
: Register specified file type as an animation file format.

`HWFILETYPE_SOUND:`
: Register specified file type as a sound file format.

`HWFILETYPE_VIDEO:`
: Register specified file type as a video file format.

The name you have to specify in parameter 3 must be a human-readable name of the file type you want to register, e.g. `"IFF ILBM"`. The name you specify here may use spaces. The fourth parameter is optional and allows you to specify the MIME type for the file format you want to register. If you don't want to provide this, pass `NULL` as the fourth parameter. Parameter number 5 must contain the extension(s) used by this file type. The extension(s) must be specified without the dot. If the file type supports more than one extension, separate the individual extensions using a vertical bar character (`|`), e.g. `"iff|ilbm|lbm"`. The `formatid` parameter is only used if `HWFILETYPEFLAGS_SAVE` is set in the `flags` parameter. In that case, you have to pass the identifier of this file format here. This must match the identifier that is returned by your implementations of functions like `RegisterImageSaver()` or `RegisterAnimSaver()`.

Finally, RegisterFileType() accepts the following flags:

`HWFILETYPEFLAGS_SAVE:`
: If this flag is set, you're registering a saver file type. You must register loaders and savers in two separate calls. It is not possible to register loader and saver within a single call to `hw_RegisterFileType()`.

`HWFILETYPEFLAGS_ALPHA:`
: This flag indicates that your plugin can load or save alpha channel information, depending on whether `HWFILETYPEFLAGS_SAVE` is set. This is only applicable for `HWFILETYPE_IMAGE` and `HWFILETYPE_ANIM`.

`HWFILETYPEFLAGS_QUALITY:`
: This flag indicates that your plugin supports lossy compression of image data. It is only supported by for `HWFILETYPE_IMAGE` and `HWFILETYPE_ANIM` and the `HWFILETYPEFLAGS_SAVE` flag must be set. The quality level is passed to your plugin saver as a value ranging between 0 (bad quality) to 100 (excellent quality).

`HWFILETYPEFLAGS_FPS:`
: This flag indicates that your plugin can save animations with different frames per second settings. It is only supported by `HWFILETYPE_ANIM` together with the `HWFILETYPEFLAGS_SAVE` flag set.

RegisterFileType() is usually called in your `InitPlugin()` function but make sure that you have at least Hollywood 5.3 before attempting to call this function!

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`self`	pointer to a `hwPluginBase` as passed to your `InitPlugin()` function
`type`	file type to register (see above)
`name`	human-readable name of this file type
`mime`	MIME type for this file format or `NULL`
`ext`	extension(s) for this file type; multiple extensions must be separated by a vertical bar (`|`)
`formatid`	when registering output file types, the unique identifier used by this format (see above)
`flags`	additional flags (see above)

RESULTS

`ok`	`True` for success, `False` on failure

EXAMPLE

```
hw_RegisterFileType(self, HWFILETYPE_IMAGE, "JPEG 2000", NULL,
                        "jp2|j2k", 0, HWFILETYPEFLAGS_ALPHA);

hw_RegisterFileType(self, HWFILETYPE_IMAGE, "JPEG 2000 (JP2)", NULL,
                        "jp2", outfmt[0], HWFILETYPEFLAGS_SAVE|
                        HWFILETYPEFLAGS_ALPHA|HWFILETYPEFLAGS_QUALITY);

hw_RegisterFileType(self, HWFILETYPE_IMAGE, "JPEG 2000 (J2K)", NULL,
                        "j2k", outfmt[1], HWFILETYPEFLAGS_SAVE|
                        HWFILETYPEFLAGS_ALPHA|HWFILETYPEFLAGS_QUALITY);
```

The code above registers a loader file type for the JPEG 2000 format and two savers for the JPEG 2000 format. The savers distinguish between the JP2 and the J2K container formats while the loader combines them into a single format.

34.35 hw_RegisterUserObject

NAME

hw_RegisterUserObject – register a new object type (V5.3)

SYNOPSIS

```
int id = hw_RegisterUserObject(lua_State *L, STRPTR name,
                struct hwObjectList **list,
                int (*attrfunc)(lua_State *L, int attr, lua_ID *id));
```

FUNCTION

This function allows you to register a new object type with Hollywood. The user will then be able to use all of Hollywood's object functions with your new type. For example, it is possible to associate custom data with Hollywood objects by using the `SetObjectData()` function and it is possible to query object attributes by calling the `GetAttribute()` Hollywood function. Additionally, all registered object types will also appear in Hollywood's resource monitor.

`hw_RegisterUserObject()` will return an identifier for the new object type or 0 in case of an error.

The name you pass to `hw_RegisterUserObject()` may contain spaces and non-ASCII characters as it is only used by Hollywood's resource monitor.

You have to pass a pointer to a pointer that marks the start of an object list to this function. Each list node must start with a `struct hwObjectList` item so that Hollywood can iterate the list and find objects in it on its own. `struct hwObjectList` looks like this:

```
struct hwObjectListHeader
{
    int type;
    lua_ID id;
    APTR reserved;
};

struct hwObjectList
{
    struct hwObjectListHeader hdr;
    struct hwObjectList *succ;
    // ... your private data must follow here ...
};
```

Whenever you add a new Hollywood object, you need to initialize the following members:

`type:` This member must be set to the object's identifier returned by `hw_RegisterUserObject()`.

`id:` This member must be set to the `lua_ID` of this object. Hollywood objects can use two different kinds of identifiers: They can either use a numerical identifier or an automatically chosen identifier that uses the `LUA_TLIGHTUSERDATA` object type. The user can request an automatically chosen identifier by passing `Nil` as the desired identifier when creating the object. In that case, the

plugin should automatically choose an identifier for the object and return it. This is usually done by using the raw memory pointer to the newly allocated object as an identifier because this guarantees its uniqueness. Internally, the two different kinds of identifiers are managed using the `lua_ID` structure which looks like this:

```
typedef struct _lua_ID
{
    int num;
    void *ptr;
} lua_ID;
```

When adding a new object, the two structure members must be initialized like this:

`num:` If the object is to use a numerical identifier, you need to write this identifier to `num` and set the `ptr` member to `NULL`. If the `ptr` member is not `NULL`, Hollywood will ignore whatever is in `num` so don't forget to set `ptr` to `NULL`.

`ptr:` If the object has been created using automatic ID selection, you need to set this member to the identifier that this object should use. This is typically set to the raw memory pointer of the newly allocated object. If `ptr` is `NULL`, Hollywood will automatically use the numerical identifier specified in `num`.

`reserved:`
Reserved for future use. Must be `NULL`.

`succ:` This must point to the next object in the list or it must be `NULL` in case the object is the last one. Whenever you create a new object, make sure to chain it into the list of objects that you passed to `hw_RegisterUserObject()`.

You also have to pass a pointer to a function that is called whenever the user calls `GetAttribute()` on your object type. Hollywood will handle the `#ATTRCOUNT` attribute automatically for your object type but for all other attributes, Hollywood will simply run the callback you specified when registering the new object type. The callback then has to push the return value(s) for this attribute on the stack and return the number of values actually pushed or an error code, just like a standard Lua function would do. See below for an example implementation.

Note that all of Hollywood's inbuilt objects use constant identifiers defined by inbuilt object constants like `#BRUSH`, `#ANIM`, or `#VIDEO`. User objects, however, use dynamic object identifiers that are determined at runtime by `hw_RegisterUserObject()`. They can be different every time Hollywood is run. That is why you should never create constants to refer to your user objects because constant values will be hard-coded in applets when scripts are compiled so there can be conflicts if `hw_RegisterUserObject()` returns an identifier that is different from the constant definition. The recommended way of dealing with user object identifiers is to implement a function named `GetObjectType()` which returns the dynamic object identifier to the script.

An example implementation could look like this:

```
struct myobj
```

```
{
    struct hwObjectList list;

    // store your object data here
    ...
};

// the actual identifier will be determined at runtime by
// hw_RegisterUserObject()
static int MY_OBJECT_TYPE = 0;

// our list of objects
static struct myobj *firstobj = NULL;

// this function is called whenever the user calls GetAttribute()
// on our user object
static SAVEDS int attrfunc(lua_State *L, int attr, lua_ID *id)
{
    struct myobj *o;

    // first find the object in our list
    for(o = firstobj; o; o = o->list.succ) {
        if(id->num == o->list.hdr.id.num &&
           id->ptr == o->list.hdr.id.ptr) break;
    }

    // not found? --> error out!
    if(!o) return ERR_FINDOBJECT;

    // check attribute that should be queried and push return values
    switch(attr) {
    case MYATTRONE:
        lua_pushnumber(L, ...);
        return 1;
    }

    // unknown attribute
    return ERR_UNKNOWNATTR;
}

HW_EXPORT int InitLibrary(lua_State *L)
{
    // register our new object type
    MY_OBJECT_TYPE = hw_RegisterUserObject(L, "MyObject",
                        (struct hwObjectList **) &firstobj, attrfunc);

    return 0;
```

```
}

HW_EXPORT void FreeLibrary(lua_State *L)
{
    struct myobj *o, *succ;

    // do not forget to see if there are any objects that
    // the user hasn't freed yet on exit --> otherwise you
    // will leak memory

    for(o = firstobj; o; o = succ) {
        o = o->list.succ;
        freeobject(L, o);
        free(o);
    }
}

// this function is important because the actual object identifier
// can be different each time Hollywood is run
static SAVEDS int my_GetObjectType(lua_State *L)
{
    lua_pushnumber(L, MY_OBJECT_TYPE);
    return 1;
}

static SAVEDS int my_CreateObject(lua_State *L)
{
    struct myobj *o, *prev = NULL;
    lua_ID id;

    // this will check whether the user passed a number in
    // parameter 1 or Nil if he passed Nil, luaL_checknewid()
    // will set id.ptr to ((void *) 1)
    luaL_checknewid(L, 1, &id);

    if(!id.ptr) {
        // must check if there already is an object with this
        // id and free it
        ...
    }

    for(o = firstobj; o; o = o->list.succ) prev = o;

    // allocate new object
    if(!(o = calloc(sizeof(struct myobj), 1))) return ERR_MEM;

    // additional initialization to be done here
```

```
    ...

    // make sure to chain our object into the list
    if(!prev) {
        firstobj = o;
    } else {
        prev->list.succ = o;
    }

    // if the user wants automatic id selection, we need to set id.ptr
    // to our object and push it as light user data on the stack
    if(id.ptr) {
        id.ptr = o;
        lua_pushlightuserdata(L, id.ptr);
    }

    // don't forget to initialize the hwObjectList header
    pdf->list.hdr.type = MY_OBJECT_TYPE;
    pdf->list.hdr.id = id;

    // returns 1 if the user wants automatic id selection because in
    // that case there will be one return value; otherwise there won't
    // be any return values
    return (id.ptr != NULL);
}

static SAVEDS int my_FreeObject(lua_State *L)
{
    struct myobj *o, *prev = NULL;
    lua_ID id;

    // check whether the user passed a number or a light userdata
    // parameter
    luaL_checkid(L, 1, &id);

    // find the object in our list
    for(o = firstobj; o; o = o->list.succ) {
        if(id.num == o->list.hdr.id.num &&
           id.ptr == o->list.hdr.id.ptr) break;
        prev = o;
    }

    // not found? exit!
    if(!o) return ERR_FINDOBJECT;

    // do your clean up here
    ...
```

```
    // important! ask Hollywood to free all data associated with this
    // object!
    hw_FreeObjectData(L, (struct hwObjectList *) o);

    // unchain object from our list
    if(prev) {
        prev->list.succ = o->list.succ;
    } else {
        firstobj = o->list.succ;
    }

    // and free it
    free(o);

    return 0;
}

static const struct hwCmdStruct plug_commands[] = {
    {"CreateObject", my_CreateObject},
    {"FreeObject", my_FreeObject},
    {"GetObjectType", my_GetObjectType},
    ...
    {NULL, NULL}
};

HW_EXPORT struct hwCmdStruct *GetCommands(void)
{
    return (struct hwCmdStruct *) plug_commands;
}
```

Note that you have to iterate through your object list in `FreeLibrary()` and free all objects that the user didn't free explicitly. Hollywood won't do this automatically. If you do not iterate through your object list in `FreeLibrary()`, you will leak memory. It's also very important that you call `hw_FreeObjectData()` on every object that you have allocated. See Section 34.11 [hw_FreeObjectData], page 283, for details.

If your plugin implements support for additional object types like above, the user will be able to do the following from the Hollywood script to work with these new object types:

```
MY_OBJECT_TYPE = myplug.GetObjectType()

obj1 = myplug.CreateObject(Nil, ...)

DebugPrint(GetAttribute(MY_OBJECT_TYPE, obj1, #ATTRYOURATTR))

SetObjectData(MY_OBJECT_TYPE, obj1, "test", "Hello")
DebugPrint(GetObjectData(MY_OBJECT_TYPE, obj1, "test"))
```

```
myplug.FreeObject(obj1)
```

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`L`	pointer to the `lua_State`
`name`	user object name to be displayed in the resource monitor
`list`	pointer to a `struct hwObjectList` pointer (see above)
`attrfunc`	function to be called when the user queries attributes for this object

RESULTS

`id`	identifier of the new object or 0 on error

34.36 hw_RemoveLoaderAdapter

NAME

hw_RemoveLoaderAdapter – remove a loader or adapter (V6.0)

SYNOPSIS

```
void hw_RemoveLoaderAdapter(hwPluginBase *self, ULONG type);
```

FUNCTION

This function can be used to remove a loader or adapter plugin. You may choose to activate it again later by calling `hw_AddLoaderAdapter()`. In the first parameter, you have to pass a pointer to the `hwPluginBase` that Hollywood has passed to your plugin's `InitPlugin()` function. The second parameter specifies the loader or adapter type. See Section 34.2 [hw_AddLoaderAdapter], page 277, for a list of supported types.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`self`	`hwPluginBase` pointer passed to `InitPlugin()`
`type`	loader or adapter type (see above)

34.37 hw_RunEventCallback

NAME

hw_RunEventCallback – run event user callback (V6.1)

SYNOPSIS

```
int error = hw_RunEventCallback(lua_State *L, int type, APTR userdata);
```

FUNCTION

This function runs the user callback associated with the event specified in `type`. This must be an event handler that has been registered using either the `hw_RegisterEventHandler()` or `hw_RegisterEventHandlerEx()` function. The `userdata`

you have to pass to this function is the same you would pass to `hw_PostEvent()` or `hw_PostEventEx()`. See Section 34.26 [hw_PostEvent], page 295, for details.

Normally, you should leave the execution of event user callbacks to Hollywood, i.e. you should just call `hw_PostEvent()` or `hw_PostEventEx()` when an event occurs and Hollywood will run the appropriate user callbacks whenever the script runs Hollywood's `WaitEvent()` or `CheckEvent()` command. However, there might be situations in which you'd like to run event callbacks immediately instead of in the next event cycle, e.g. if you're running a modal loop or something. In that case, `hw_RunEventCallback()` can be used to instantly run the user callback.

Note that you need to take care to call this function at a reasonable time. `hw_RunEventCallback()` will immediately call `lua_pcall()` so it must be called at a time when the Hollywood script expects to be interrupted by a user callback. For example, calling `hw_RunEventCallback()` in `HandleEvents()` when `HWHEFLAGS_LINEHOOK` is set is a bad idea because it can result in all sorts of unwanted behaviour since in that case user event callbacks might be entered after every single line of code that the Lua VM has executed. The best place to call `hw_RunEventCallback()` is when Hollywood is in a modal loop or in a `WaitEvent()` state.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L`	pointer to the `lua_State`
`type`	event handler type registered through `hw_RegisterEventHandler()` or `hw_RegisterEventHandlerEx()`
`userdata`	user data to pass to the event handler

RESULTS

`error`	error code or 0 for success

34.38 hw_RunTimerCallback

NAME

hw_RunTimerCallback – run timer user callback (V6.0)

SYNOPSIS

```
int error = hw_RunTimerCallback(lua_State *L, APTR handle);
```

FUNCTION

This function runs the user callback associated with the specified timer handle that has been allocated by `RegisterTimer()`. The user callback can be a function that has been installed either using Hollywood's `SetInterval()` or `SetTimeout()` command.

You will normally call `hw_RunTimerCallback()` whenever your timer has fired. However, you need to take care that you call this function at a reasonable time. `hw_RunTimerCallback()` will immediately call `lua_pcall()` so it must be called at a time when the Hollywood script expects to be interrupted by a user callback, preferably while

it is in a `WaitEvent()` state. Thus, it is advised that you call this function somewhere in your `HandleEvents()` implementation. When in `HandleEvents()` make sure to check that the `HWHEFLAGS_LINEHOOK` and `HWHEFLAGS_MODAL` flags aren't set. Then you can be sure that `HandleEvents()` has been called in response to the `WaitEvent()` or `CheckEvent()` Hollywood commands and you may safely call `hw_RunTimerCallback()`.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`L`	pointer to the `lua_State`
`handle`	timer handle allocated by `RegisterTimer()`

RESULTS

`error`	error code or 0 for success

34.39 hw_SetErrorCode

NAME

hw_SetErrorCode – set extended error information (V5.0)

SYNOPSIS

```
void hw_SetErrorCode(int c);
```

FUNCTION

This function can be used to set extended error information for errors containing a `%ld` wildcard. When composing the final error message, Hollywood will replace the `%ld` wildcard with the value passed to this function.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`c`	extended error information to store in Holywood

34.40 hw_SetErrorString

NAME

hw_SetErrorString – set extended error information (V5.0)

SYNOPSIS

```
void hw_SetErrorString(STRPTR s);
```

FUNCTION

This function can be used to set extended error information for errors containing a `%s` wildcard. When composing the final error message, Hollywood will replace the `%s` wildcard with the string passed to this function.

Hollywood will make a copy of the string you pass to this function so you may free it once `hw_SetErrorString()` returns.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`s` null-terminated string describing the extended error information

34.41 hw_SetTimerAdapter

NAME
hw_SetTimerAdapter – install a timer adapter (V6.0)

SYNOPSIS

```
int error = hw_SetTimerAdapter(hwPluginBase *self, ULONG flags,
                                    struct hwTagList *tags);
```

FUNCTION
This function can be used to activate a plugin that has the `HWPLUG_CAPS_TIMERADAPTER` capability flag set. This function must only be called from inside your `RequirePlugin()` implementation. If this function succeeds, Hollywood's inbuilt timer handler will be completely replaced by the timer handler provided by your plugin and Hollywood will call into your plugin whenever it needs to work with timers. In the first parameter, you have to pass a pointer to the `hwPluginBase` that Hollywood has passed to your plugin's `InitPlugin()` function. The second parameter must be set to a combination of flags. The following flags are currently defined:

`HWSTAFLAGS_PERMANENT:`
If this flag is set, the timer adapter will be made permanent. This means that other plugins won't be able to overwrite this timer adapter with their own one. If `HWSTAFLAGS_PERMANENT` is set, all subsequent calls to `hw_SetTimerAdapter()` will fail and your timer adapter will persist.

See Section 20.1 [Timer adapter plugins], page 161, for information on how to write timer adapter plugins.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`self` `hwPluginBase` pointer passed to `InitPlugin()`

`flags` combination of flags (see above)

`tags` reserved for future use; set it to `NULL` for now

RESULTS

`error` error code or 0 for success

34.42 hw_SubTime

NAME
hw_SubTime – subtract two time stamps (V5.0)

SYNOPSIS
```
void hw_SubTime(struct hwos_TimeVal *dest, struct hwos_TimeVal *src);
```

FUNCTION
This function subtracts the time stamp `src` from `dest` and stores the resulting time stamp in `dest`.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`dest` pointer to a `struct hwos_TimeVal` containing the minuend; the result of the subtraction will be written to this buffer

`src` pointer to a `struct hwos_TimeVal` containing the subtrahend

34.43 hw_TrackedAlloc

NAME
hw_TrackedAlloc – allocate memory buffer with tracking (V5.0)

SYNOPSIS
```
APTR buf = hw_TrackedAlloc(int size, ULONG flags, STRPTR name);
```

FUNCTION
The debug version of Hollywood supports memory tracking to make it easier to detect memory leaks and illegal free memory calls. If you are developing your plugin using a debug version of Hollywood, you can use `hw_TrackedAlloc()` to allocate a memory buffer that gets tracked by Hollywood. If you forget to free this memory buffer, Hollywood will issue a warning before it terminates. In order to be able to identify the memory buffer that hasn't been freed, you need to provide a name for every memory buffer you allocate using `hw_TrackedAlloc()`. In case you forget to free a buffer, Hollywood will tell you its name so that you can identify where in your code the allocation was made.

The following flags are currently supported by `hw_TrackedAlloc()`:

`HWMEMF_CLEAR:`
If this flag is set, `hw_TrackedAlloc()` will clear the memory buffer with zeros before returning control to your plugin.

In release versions of Hollywood this function does the same as `malloc()`.

You need to use `hw_TrackedFree()` to free memory allocated by `hw_TrackedAlloc()`.

This function is thread-safe.

DESIGNER COMPATIBILITY
Supported since Designer 4.0

INPUTS

`size` size of the buffer in bytes

`flags` combination allocation flags (see above)

`name` null-terminated string containing a name for this buffer; it need not be unique but should help you to identify the allocation; this must always be provided!

RESULTS

`buf` pointer to newly allocated memory or `NULL` on error

34.44 hw_TrackedFree

NAME

hw_TrackedFree – free a tracked memory buffer (V5.0)

SYNOPSIS

```
void hw_TrackedFree(APTR buf);
```

FUNCTION

This function must be used to free memory allocated by `hw_TrackedAlloc()`. See Section 34.43 [hw_TrackedAlloc], page 327, for details.

This function is thread-safe.

DESIGNER COMPATIBILITY

Supported since Designer 4.0

INPUTS

`buf` memory buffer allocated by `hw_TrackedAlloc()`

34.45 hw_UnLockSemaphore

NAME

hw_UnLockSemaphore – unlock a semaphore (V6.0)

SYNOPSIS

```
void hw_UnLockSemaphore(APTR sem);
```

FUNCTION

This function unlocks the specified semaphore handle. See Section 34.22 [hw_LockSemaphore], page 290, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`sem` semaphore handle allocated by `hw_AllocSemaphore()`

34.46 hw_UnregisterCallback

NAME
hw_UnregisterCallback – unregister a callback (V6.0)

SYNOPSIS
```
void hw_UnregisterCallback(APTR handle);
```

FUNCTION
This function can be used to unregister a callback registered by `hw_RegisterCallback()`. It is normally not necessary to call this function since Hollywood unregisters all callbacks before it exits.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`handle` callback handle allocated by `hw_RegisterCallback()`

34.47 hw_WaitEvents

NAME
hw_WaitEvents – wait for events (V6.0)

SYNOPSIS
```
int error = hw_WaitEvents(lua_State *L, ULONG flags);
```

FUNCTION
This function halts the program execution until one or more events come in. This can be useful if you need to set up a temporary modal event loop. The following flags are currently recognized by this function:

`HWWEFLAGS_MODAL`:
Signals that `hw_WaitEvents()` is called from a temporary modal event loop. This flag should always be set.

Once `hw_WaitEvents()` returns, you should immediately call `hw_HandleEvents()`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`L` pointer to the `lua_State`

`flags` combination of flags (see above)

RESULTS

`error` error code or 0 for success

35 UnicodeBase functions

35.1 Overview

`UnicodeBase` contains functions to deal with Unicode text. They are very similar to standard functions from the C runtime but have been modified to deal with UTF-8 strings and Unicode characters.

Please note that you need to check for Hollywood version 7.0 or later before trying to access `UnicodeBase`. It is not supported by earlier Hollywood versions.

35.2 composechar

NAME

composechar – write Unicode character to UTF-8 string (V7.0)

SYNOPSIS

```
void composechar(char *s, int ch);
```

FUNCTION

Converts the Unicode character specified in `ch` to UTF-8 and writes it to the string specified in `s`. It also appends a terminating `NULL` character. Thus, this function may write up to 5 bytes to `s` so make sure the buffer is large enough.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`s`	output string
`ch`	Unicode character to convert to UTF-8

35.3 getnextchar

NAME

getnextchar – get next Unicode character from string (V7.0)

SYNOPSIS

```
int ch = getnextchar(const char *s, int *idx);
```

FUNCTION

Reads the next Unicode character from `s` from the byte offset pointed to by `idx`. Upon return, the integer pointed to by `idx` is increased by the number of bytes read. In UTF-8, this can be up to 4 bytes per character. `s` must be a valid UTF-8 string.

This function can be used to easily read all individual Unicode characters from a UTF-8 string like so:

```
int i = 0;
while(s[i]) {
    printf("Got character: %d\n", getnextchar(s, &i));
}
```

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`s` input string

`idx` pointer to an integer containing the byte offset that defines where to start reading

RESULTS

`ch` Unicode character read from string

35.4 getbyteindex

NAME
getbyteindex – convert from character to byte index (V7.0)

SYNOPSIS
```
int idx = getbyteindex(const char *s, int charindex);
```

FUNCTION
Returns the byte index of the character specified in `charindex`. Character indices are counted from 0. String `s` must be a valid UTF-8 string and `charindex` must point to a valid character index.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`s` input string

`charindex`
character index to convert to bytes

RESULTS

`idx` byte index of the specified character

35.5 getcharindex

NAME
getcharindex – convert from byte to character index (V7.0)

SYNOPSIS
```
int idx = getcharindex(const char *s, int byteindex);
```

FUNCTION
Returns the character index of the byte position inside `s` specified by `byteindex`. Both indices are counted from 0. String `s` must be a valid UTF-8 string and `byteindex` must point to a valid byte index inside `s`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`s` input string

`byteindex`
byte index to convert to characters

RESULTS

`idx` character index at the specified byte offset

35.6 isalnum

NAME
isalnum – check if character is alphanumeric (V7.0)

SYNOPSIS
```
int r = isalnum(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is is either a decimal digit or an upper or lower case letter and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c` character to test

RESULTS

`r` test result

35.7 isalpha

NAME
isalpha – check if character is alphabetic (V7.0)

SYNOPSIS
```
int r = isalpha(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is an alphabetic letter and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c` character to test

RESULTS

`r` test result

35.8 iscntrl

NAME

iscntrl – check if character is a control character (V7.0)

SYNOPSIS

```
int r = iscntrl(int c);
```

FUNCTION

Checks if the Unicode character passed in `c` 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.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`c` character to test

RESULTS

`r` test result

35.9 isdigit

NAME

isdigit – check if character is a decimal digit (V7.0)

SYNOPSIS

```
int r = isdigit(int c);
```

FUNCTION

Checks if the Unicode character passed in `c` is a decimal digit and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`c` character to test

RESULTS

`r` test result

35.10 isgraph

NAME
isgraph – check if character has a graphical representation (V7.0)

SYNOPSIS
```
int r = isgraph(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` 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.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.11 islower

NAME
islower – check if character is a lower case letter (V7.0)

SYNOPSIS
```
int r = islower(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is a lower case letter and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.12 isprint

NAME
isprint – check if character is printable (V7.0)

SYNOPSIS
```
int r = isprintable(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is printable and returns `True` if it is, otherwise `False`. 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.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.13 ispunct

NAME
ispunct – check if character is a punctuation character (V7.0)

SYNOPSIS
```
int r = ispunct(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is a punctuation character and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.14 isspace

NAME
isspace – check if character is a white-space character (V7.0)

SYNOPSIS
```
int r = ispunct(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` 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.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.15 isupper

NAME
isupper – check if character is an upper case letter (V7.0)

SYNOPSIS
```
int r = isupper(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is an upper case letter and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.16 isxdigit

NAME
isxdigit – check if character is a hexadecimal digit (V7.0)

SYNOPSIS
```
int r = isxdigit(int c);
```

FUNCTION
Checks if the Unicode character passed in `c` is a hexadecimal digit and returns `True` if it is, otherwise `False`.

DESIGNER COMPATIBILITY
Unsupported

INPUTS

`c`	character to test

RESULTS

`r`	test result

35.17 stricmp

NAME

stricmp – compare two strings in a case-insensitive way (V7.0)

SYNOPSIS

```
int r = stricmp(const char *s1, const char *s2);
```

FUNCTION

Compares `s1` and `s2` in a case-insensitive way. Both strings must be valid UTF-8 strings which are terminated by a `NULL` character.

If `s1` is less than `s2`, -1 is returned. If `s1` is greater than `s2`, 1 is returned, otherwise the return value is 0.

Note that this function is mostly useful for checking the equality of two strings in a case-insensitive way. It is not appropriate for sorting because it currently doesn't handle all intricacies of Unicode collation. If you need to compare strings for sorting, please use `hw_CompareString()` instead. See Section 34.6 [hw_CompareString], page 280, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`s1`	first string to compare
`s2`	second string to compare

RESULTS

`r`	result of comparison (see above)

35.18 strlen

NAME

strlen – determine string length in characters (V7.0)

SYNOPSIS

```
size_t len = strlen(const char *s);
```

FUNCTION

Returns the string length (in characters) of the string specified in `s`. If the string isn't a valid UTF-8 string, -1 is returned.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`s`	input string

RESULTS

`len`	character length of string or -1 on error

35.19 strnicmp

NAME

strnicmp – compare string characters case-insensitively (V7.0)

SYNOPSIS

```
int r = strnicmp(const char *s1, const char *s2, size_t n);
```

FUNCTION

Compares a maximum of `n` bytes in strings `s1` and `s2` in a case-insensitive way. If a `NULL` character is encountered in either string before reaching `n` bytes, the comparison is stopped. Both strings must be valid UTF-8 strings.

If `s1` is less than `s2`, -1 is returned. If `s1` is greater than `s2`, 1 is returned, otherwise the return value is 0.

Note that `n` must be specified in bytes, not in characters.

Note that this function is mostly useful for checking the equality of two strings in a case-insensitive way. It is not appropriate for sorting because it currently doesn't handle all intricacies of Unicode collation. If you need to compare strings for sorting, please use `hw_CompareString()` instead. See Section 34.6 [hw_CompareString], page 280, for details.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`s1`	first string to compare
`s2`	second string to compare
`n`	number of bytes to compare

RESULTS

`r`	result of comparison (see above)

35.20 tolower

NAME

tolower – convert character to lower case (V7.0)

SYNOPSIS

```
int rc = tolower(int c);
```

FUNCTION

Converts the Unicode character specified in `c` to lower case and returns it.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`c`	character to convert

RESULTS

`rc` lower case character

35.21 toupper

NAME

toupper – convert character to upper case (V7.0)

SYNOPSIS

```
int rc = toupper(int c);
```

FUNCTION

Converts the Unicode character specified in `c` to upper case and returns it.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`c` character to convert

RESULTS

`rc` upper case character

35.22 validate

NAME

validate – validate UTF-8 string (V7.0)

SYNOPSIS

```
int r = validate(const char *s);
```

FUNCTION

Checks if the string passed in `s` is a valid UTF-8 string and returns `True` if it is and `False` otherwise.

DESIGNER COMPATIBILITY

Unsupported

INPUTS

`s` string to validate

RESULTS

`r` validation result

36 UtilityBase functions

36.1 Overview

`UtilityBase` contains some general purpose utility functions that are needed quite often.

Please note that you need to check for Hollywood version 6.0 or later before trying to access `UtilityBase`. It is not supported by earlier Hollywood versions.

36.2 hw_CRC32

NAME

hw_CRC32 – compute CRC32 checksum (V6.0)

SYNOPSIS

```
ULONG crc = hw_CRC32(UBYTE *data, int len);
```

FUNCTION

This function computes the CRC32 checksum of the data passed in the first parameter and returns it.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`data`	pointer to arbitrary memory data
`len`	number of bytes to read from pointer

RESULTS

`crc`	the data's CRC32 checksum

36.3 hw_DecodeBase64

NAME

hw_DecodeBase64 – decode Base64 data (V6.0)

SYNOPSIS

```
int ok = hw_DecodeBase64(UBYTE *src, int srclen, UBYTE *dst, int *dstlen,
                         struct hwTagList *tags);
```

FUNCTION

This function decodes Base64 data stored in the `src` memory buffer to the memory buffer passed in the `dst` parameter. Make sure that the `dst` buffer is large enough to hold the decoded data. On success, `hw_DecodeBase64()` will return `True`, otherwise `False`.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`src`	pointer to Base64 encoded data

`srclen`	length of Base64 data
`dst`	buffer to receive the decoded Base64 data
`dstlen`	pointer to an `int` which will receive the length of the decoded data
`tags`	reserved for future use, pass `NULL` for now

RESULTS

`ok`	`True` for success, `False` if there was a decoding error

36.4 hw_EncodeBase64

NAME

hw_EncodeBase64 – encode data into the Base64 format (V6.0)

SYNOPSIS

```
void hw_EncodeBase64(UBYTE *src, int srclen, UBYTE *dst, int *dstlen,
                     struct hwTagList *tags);
```

FUNCTION

This function reads arbitrary data from a memory buffer and encodes it into the Base64 format. The data will be read from `src` up until `srclen` bytes have been read. The encoded bytes will be written to the `dst` buffer. Make sure that this buffer is large enough to hold the encoded Base64 data. After encoding, `hw_EncodeBase64()` will also write the output length to `dstlen`.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`src`	pointer to arbitrary memory data
`srclen`	number of bytes to read from pointer
`dst`	buffer to receive the encoded Base64 data
`dstlen`	pointer to an `int` which will receive the length of the encoded data
`tags`	reserved for future use, pass `NULL` for now

36.5 hw_MD5

NAME

hw_MD5 – compute MD5 checksum (V6.0)

SYNOPSIS

```
void hw_MD5(UBYTE *data, int len, STRPTR dest);
```

FUNCTION

This function computes the MD5 checksum of the data passed in the first parameter and writes the MD5 checksum to the string buffer passed in `dest`. The MD5 checksum is written as a 32-byte hexadecimal hash so make sure that the buffer you pass in `dest` can store at least 33 bytes.

DESIGNER COMPATIBILITY

Supported since Designer 4.5

INPUTS

`data`	pointer to arbitrary memory data
`len`	number of bytes to read from pointer
`dest`	pointer to memory buffer to receive the MD5 hash

37 ZBase functions

37.1 Overview

`ZBase` contains most functions from the zlib library. If you want to use any of these functions, you'll need the correct zlib header files. Make sure to use compatible header files only. Hollywood uses version 1.2.5 of zlib so you need to use the header files from exactly this version if you plan to use functions from the `ZBase` library.

Please consult your zlib documentation for information on the individual functions supported by `ZBase`.

Please note that you need to check for Hollywood version 5.3 or later before trying to access `ZBase`. It is not supported by earlier Hollywood versions.

The implementations of `ZBase` in Hollywood Designer and Hollywood are identical.

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 LGPL license

GNU LESSER GENERAL PUBLIC LICENSE Version 3, 29 June 2007

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

H

I

L

M

N

O

P

R

S

T

U

V

W