ExportFile Xtra 0.4

By Anthony Kleine

Special Thanks

- Nosamu, for his Director assistance.
- JrMasterModelBuilder, for his useful Shockwave research.
- Binzy, for the cover suggestion.

Image Credit

geralt, “Explosion Pop Big Bang,” 2016, via PixaBay.

Table of Contents

About ExportFile Xtra

ExportFile Xtra exports the content of Director cast members out to files. It can export all built-in cast member types, as well as Xtra Media.

ExportFile Xtra is an open source software, created by Anthony Kleine. It is not a commercial software. There is no trial period or license key. It is not shareware or donationware. You will never be asked to pay money for it. However, because it is MIT licensed, the code may be freely used and modified, including in commercial products.

Currently, ExportFile Xtra is only available for Windows. However, it may be available for Mac in the future, so any platform specific behaviour will still be noted. ExportFile is compatible with Windows XP to Windows 11.

Basic Usage

The most basic usage of ExportFile is to pass a single argument - the member to export - to the `exportFileOut` handler. If it succeeds, it returns a property list. If an error occurs, it returns a void value. Then the `exportFileStatus` handler may be called to get an error code.

```
exportFileOut(member("ExportFile Demo"))

if the ilk of the result <> #propList then
  alert("ExportFile Error! (" & string(exportFileStatus()) & ")")
end if
```

You may also specify a path in the second argument.

```
exportFileOut(member("Bitmap Member"), "Bitmap Path.bmp")
```

If this is all you need to be able to do, then you'll never need to use any other arguments or handlers from the ExportFile Xtra - but otherwise, ExportFile provides a flexible way to specify exactly how you would like your file to be exported.

Paths, Labels, and Agents

To make exporting files possible, ExportFile uses three concepts that are interlinked: paths, labels and agents. Understanding these concepts is essential to understanding how ExportFile works.

The problem that ExportFile aims to solve is that there may be multiple reasonable ways to represent the member's content as a file. For example, consider a text member. You might want to export the member as styled text (such as RTF,) or as plain text. You may even want to export the member's script, or export the member as an image. If the member were the only argument provided, ExportFile would have to assume what you want. Furthermore, not only are there multiple ways to represent the member's content, but you may wish to have that same representation in a different format - for instance, when exporting an image you may want a JPEG, PNG, or TIFF file.

The easiest way to solve this problem is by specifying the path. Based on the path, ExportFile will produce different results. The following will export the member as RTF.

```
exportFileOut(member("Text"), "Rich Text Document File.rtf")
```

The following will export the member as JPEG.

```
exportFileOut(member("Text"), "JPEG File.jpg")
```

This is all fine, but what if you want to use a file extension that isn't recognized? For example, let's say that our text member actually contains a plain text format, such as XML. If we do the following...

```
-- exports styled text
exportFileOut(member("XML Data"), "XML Data.xml")
```

The XML extension is not one of the extensions that ExportFile knows. In the absence of an extension that is known to ExportFile, it assumes the default behaviour of exporting text members as RTF. The result file contains styled text when all we want is plain text.

To solve this issue, ExportFile introduces the concept of labels, which are represented as symbols. As opposed to the default label - which in this case would normally be `#rtf` - we instead specify the `#text` label, signifying that we want plain text.

```
-- exports plain text
exportFileOut(member("XML Data"), "XML Data.xml", #text)
```

For text members, there is also the `#html` label to export the member as HTML, the `#image` label to export it as an image, the `#lingo` and `#javaScript` labels to export the member's script, and more. You can get the full list of labels for a member with `getExportFileLabelList`, which is discussed later in this document.

If we're only concerned about the label used, we don't need to specify the path. By passing a void value for the path argument, ExportFile Xtra will automatically name the file based on the member's name or number. Let's say we wish to export our text member as an image. All we need to specify is the `#image` label, and a default path such as Text.bmp or Text.pic will be used.

```
exportFileOut(member("Text"), void, #image)
```

Now we have a new problem, however. By default, our image will export to a standard format (BMP on Windows, or PICT on Mac.) What if we'd like to export to PNG? To solve this problem, ExportFile introduces the concept of agents, which are represented by strings.

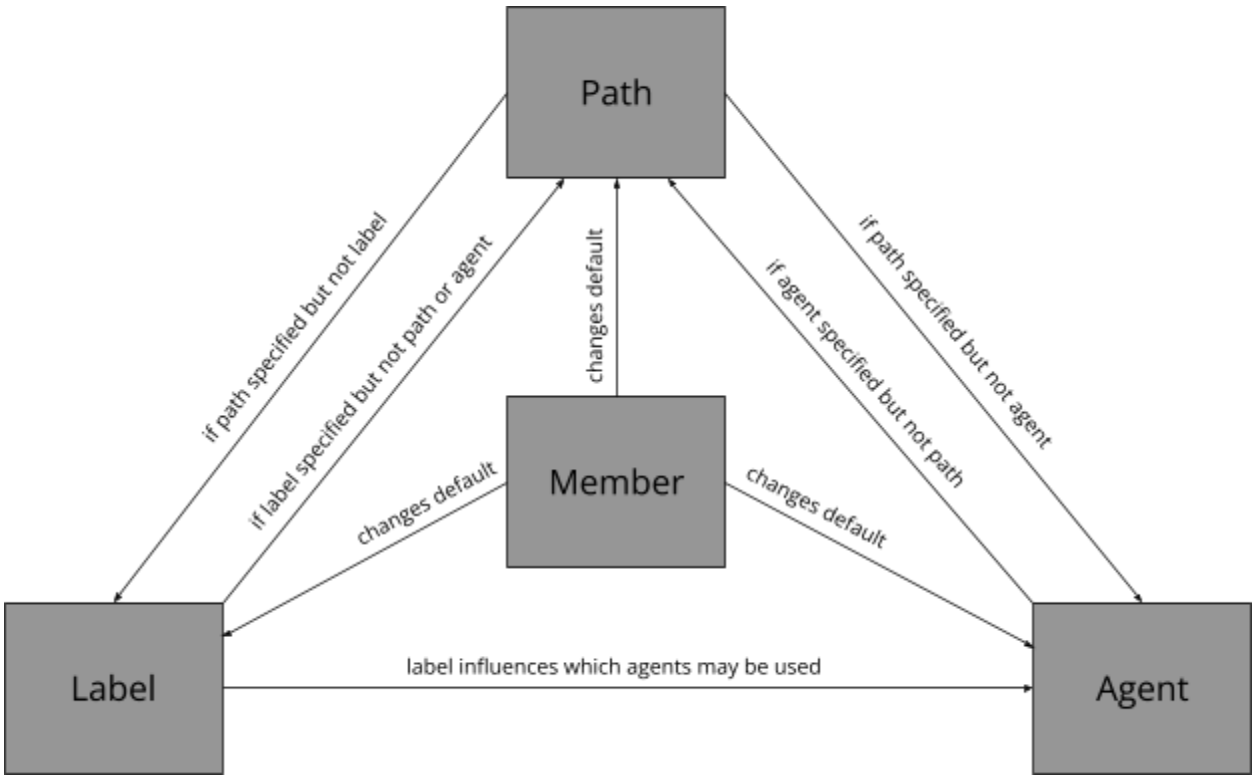
```
exportFileOut(member("Text"), void, #image, "kMoaCfFormat_PNG")
```

Which agents are available depends on which Agent Xtras you have. If the PNG Import Export Xtra is available, then the PNG agent may be used. If the Targa Import Export Xtra is available, then the Targa agent may be used, and so on.

This also means that ExportFile's format support is extensible. By writing your own Agent Xtra, not only do you extend Director's format support, but a new format will also be added to ExportFile. Currently, ExportFile only supports image and sound agents, because these are the only agent types included with Director that can export files, but support for additional agent types such as text may be added in future versions.

You can get the full list of agents for a member by calling `getExportFileAgentPropList`, which will be discussed later in this document. Note that these strings are case-insensitive, though may not be converted to symbols (they may contain characters such as spaces that would be invalid in symbols.)

Notice how the concepts of paths, labels, and agents interact with each other. If only a path is specified, the label and agent to use are decided based on the path. If the label is specified explicitly, that overrides the label that would've been decided from the path. And agents may only be used with specific labels - the PNG agent may be used with the `#image` label, but not with the `#text` label, because PNG files are image files, not text files. ExportFile treats void values the same as if they were not specified.



A diagram illustrating the relationship between the path, label, and agent for a given member.

Paths In Depth

ExportFile offers two methods of passing the path argument: a path string - which is a string containing the absolute or relative path to a file, as was seen in prior examples - or a path info property list. A path info property list is a Lingo property list with any

combination of the `#dirname`, `#basename`, `#extension`, and `#filename` properties. An example of a path info property list may look like this.

```
[#dirname: "C:\Directory Name\", #basename: "basename", #extension:
"ext", #filename: "basename.ext"]
```

Using Paths

For any properties that aren't specified, defaults are used. For example, you may pass a path info property list like this.

```
[#extension: "abc"]
```

In this case, the default directory name and base name (the same ones that would have been used if the path were not specified) will be used, but the extension will be `"abc"`. Do not include the period (.) in the extension. You may get the defaults by calling `getExportFileDefaultPath`, discussed later in this document.

If you would like the filename to be based on the member's name, you should prefer this method of setting the extension over string concatenation. By default, ExportFile will automatically filter any invalid pathname characters in the member's name, and use the member's number instead if it doesn't have a name or if the name would cause the path to be too long. If the member's name property is instead explicitly passed to ExportFile, it won't apply these measures, and an error will occur if that causes the path to become invalid.

Another potential use for a path info property list is to specify the directory name to use without string concatenation. This will cause the file to be written to the movie path explicitly.

```
[#dirname: the moviePath]
```

You should generally prefer explicitly specifying `the moviePath` or `the applicationPath` properties as the directory name to use instead of using a relative path. By default, when using a relative path, the file is exported to the current directory (unless the `#location` option is used, which is discussed later in this document.) The current directory is not consistent: it may be the movie path, the application path, or somewhere

else; it may be different depending on whether or not you are in author mode, and its behaviour in Director 11 is different from older versions.

If the `#dirname` property is specified, it may be relative or absolute, but it must not be empty or consist of only whitespace characters. Otherwise, an error occurs.

If the `#filename` property is specified together with the `#basename` or `#extension` properties, they MUST be consistent.

```
set valid to [#filename: "same.yes", #basename: "same", #extension:
"yes"]
set invalid to [#filename: "different.no", #basename: "distinct",
#extension: "bad"]
```

If the valid list were passed as the path, it would be accepted because the filename is consistent with the basename and extension. If the invalid list were passed as the path, an error would occur because it is not internally consistent. To ensure validity, delete or don't include the `#filename` property if you plan on using the `#basename` or `#extension` properties, and vice-versa.

The filename must not be empty or consist of only period or whitespace characters. Otherwise, an error occurs.

Paths in ExportFile do not support Director's @ symbol filename notation. Paths must be local, not remote - URLs are not supported.

Paths influence both labels and agents. If the label and agent were not specified, and the path `"file.aiff"` was specified, it would typically cause the `#sound` label and the `"kMoaCfFormat_AIFF"` agent to be used.

Although the default path is based on the member's name, the member's name will not influence labels or agents. A member with the name `"name.aiff"` will not cause the `#sound` label or the `"kMoaCfFormat_AIFF"` agent to be used. The label or agent will only be influenced by the path if it is specified.

Although paths influence agents, they do not influence the file the agent writes. If the path `"file.aifc"` was specified, it would write an identical file to if the path

`"file.aiff"` was specified, even though the extension `"aifc"` is intended for compressed AIFF files. To change the file the agent writes, you must specify the agent options as described later in this document.

Labels In Depth

The following table is the full list of labels and their descriptions.

Not every label can be used with every member. To get the list of labels for a member, call `getExportFileLabelList`.

Label	Description
`#rtf`	Rich Text Document (Styled Text.) Requires the Text Asset Xtra. Director 7 or newer.
`#html`	HTML Document (Styled Text.) Requires the Text Asset Xtra. Director 7 or newer.
`#text`	Text Document (Plain Text.) Using this label with members that are not Fields requires the Text Asset Xtra and is only supported in Director 7 and newer.
`#textStyles`	Text Styles. Similar to Mac TextEdit Styles.
`#pict`	PICT. Using this label is only supported in Director 7 or newer.
`#image`	Bitmap Image in a standard format (BMP on Windows, PICT on Mac.) Using this label with members that are not bitmaps is only supported in Director 8 or newer. On Mac, using this label is only supported in Director 7 or newer.
`#sound`	Sound Samples in a standard format (WAVE Sound on Windows, snd Resource Sound on Mac.)

Label	Description
	Requires the Sound Import Export Xtra.
`#palette`	Palette Entries in a standard format (Microsoft Palette on Windows, Photoshop CLUT on Mac.)
`#score`	VideoWorks Score. Score data for a Film Loop.
`#swf`	Flash, as an SWF. Requires the Flash Asset Xtra. Director 7 or newer.
`#w3d`	Shockwave 3D, as a W3D. Requires the Shockwave 3D Asset Xtra. Director 8.5 or newer.
`#gif`	Animated GIF, as a GIF Image. Requires the Animated GIF Asset Xtra. Director 7 or newer.
`#pfr`	Font, as a Bitstream Portable Font Resource (PFR,) Netscape Navigator 4.x's webfont format. Requires the Font Asset Xtra. Director 7 or newer.
`#xtraMedia`	Xtra Media, with an assumed extension based on the extensions the Xtra registers for import. Requires the Asset Xtra for the member.
`#wavAsync`	WAV (Audio Mixer, for mixer members only - for sound members, use the `#sound` label instead.) Using this label causes the file to be exported asynchronously. To receive an event when the mixer is saved, call `tellExportFileMixerSaved`. Requires the AudioMixer Xtra. Director 11.5 or newer.
`#mp4Async`	MP4 (Audio Mixer, for mixer members only - for sound members, use the `#sound` label instead.) Using this label causes the file to be exported asynchronously. To receive

Label	Description
	an event when the mixer is saved, call `tellExportFileMixerSaved`. Requires the AudioMixer Xtra. Director 11.5 or newer.
`#lingo`	Lingo. Files exported with this label may be empty if the movie is protected, because the scripts in protected movies are compiled - decompiling scripts is outside of the scope of ExportFile.
`#javaScript`	JavaScript. Files exported with this label may be empty if the movie is protected, because the scripts in protected movies are compiled - decompiling scripts is outside of the scope of ExportFile. Director MX 2004 or newer.
`#composite`	Composite. Cast member media data in a portable (byte-swapped) opaque format (a format only intended for Director itself to use.) Files exported with this label are compatible with the DrAccess Xtra (included in the Director XDK,) though the DrAccess Xtra is not required. To export to a standard format, use one of the other labels instead of this one if possible.

Using Labels

To tell if a label can be used for a member, call `getExportFileLabelList` and call the Lingo `getOne` handler on the list. If `getOne` returns `0`, the label cannot be used with this member.

```
set labelList to getExportFileLabelList(member("Bench"))

if listP(labelList) then
  if labelList.getOne(#image) then
    alert("The #image label is in the list!")
  end if
```

```
end if
```

The `#xtraMedia` label allows for exporting Xtra Media from Xtras that ExportFile does not have specific support for (such as Havok Physics.) To allow this to be possible, the label makes assumptions about Xtra Media:

1. It is assumed that the extension that the Xtra registers for import is the same extension that should be used for the exported file.
2. It is assumed that the format in which the Xtra stores its media is the same format that it imports.

These assumptions may not always be true, so other labels should be used instead of `#xtraMedia` if available. For example, Flash has an `#swf` label. Although it is possible to use the `#xtraMedia` label with Flash members, and the exported file will have a `"swf"` extension in the path, it will not be a valid SWF file because the Flash Asset Xtra stores its media in a different format than it imports.

The `#composite` label is compatible with the DrAccess Xtra (included in the Director XDK.) In Director 5 - 6.5 on Windows, the DrAccess Xtra is unable to open the composite files exported by ExportFile. This is due to a bug in the DrAccess Xtra's `readMedia` handler - it is not a bug in ExportFile, which exports the file correctly. In Director 5 - 6.5 on Mac, and in Director 7 or newer, the bug does not occur.

Labels influence which agents may be used. For example, the PNG agent may be used with the `#image` label, but not the `#text` label.

Agents In Depth

As of Director 12, the following table is the full list of agents that are included with Director. Some agents may not be in older versions. To use an agent, the Agent Xtra must be available.

Not every agent can be used with every member. To get the list of agents for a member, call `getExportFileAgentPropList`.

Image Agents

Agent	Name	Path Extensions	Hidden	Xtra(s)
"kMoaCfFormat_JPEG"	JPEG	["jpg"]	FALSE	JPEG Export
"kMoaCfFormat_PNG"	PNG	["png"]	FALSE	PNG Import Export
"kMoaCfFormat_TGA"	Targa	["tga"]	FALSE	Targa Import Export
"kMoaCfFormat_TIFF"	TIFF	["tif", "tiff"]	FALSE	TIFF Import Export
"kMoaCfFormat_TranslateBMP"	JavaTranslator ForBMP	["nuu"]	TRUE	Image Translator Helper

Sound Agents

Agent	Name	Path Extensions	Hidden	Xtra(s)
"kMoaCfFormat_AAC"	AAC Audio	["aac"]	FALSE	• MP4Asset • F4VAsset
"kMoaCfFormat_AIFF"	AIFF Sound	["aif", "aiff", "aifc"]	FALSE	Sound Import Export
"kMoaCfFormat_AU"	Sun AU Sound	["au"]	FALSE	Sun AU Import Export
"kMoaCfFormat_MPEG3"	MP3 Audio	["mp3", "mp2"]	FALSE	• MPEG 3 Import Export • FLVAsset
"kMoaCfFormat_MPEG4"	MP4 Audio	["m4a", "mp4", "f4v"]	FALSE	• MP4Asset • F4VAsset

Agent	Name	Path Extensions	Hidden	Xtra(s)
"kMoaCfFormat_snd"	snd Resource Sound	["snd_"]	FALSE	Sound Import Export
"kMoaCfFormat_SWA"	SWA Sound	["swa"]	FALSE	SWA Import Export
"kMoaCfFormat_WAVE"	WAVE Sound	["wav", "wave"]	FALSE	Sound Import Export

Using Agents

To tell if an agent can be used for a member/path/label, call `getExportFileAgentPropList` and call the Lingo `findPos` handler on the property list. If `findPos` returns a void value, the agent cannot be used with this member/path/label.

```
set agentPropList to getExportFileAgentPropList(member("Techno"),
void, #sound)

if the ilk of agentPropList = #propList then
  if integerP(agentPropList.findPos("kMoaCfFormat_SWA")) then
    alert("The SWA Sound agent is in the property list!")
  end if
end if
```

When specifying an agent to use, passing a void value has a different meaning than passing an empty string. A void value is the same as not specifying an agent, so the default agent will be used. An empty string forces no agent to be used.

This exports a TIFF image, because the default agent is the TIFF agent because of the path.

```
exportFileOut(member("Telescope"), "Telescope.tiff", #image, void))
```

This exports a BMP image with a `"tiff"` extension, because the empty string forces no agent to be used, so the image is exported as if there was no agent.

```
exportFileOut(member("Telescope"), "Telescope.tiff", #image, EMPTY))
```

You should typically prefer passing a void value over passing an empty string as the agent, unless you received the empty string as the return value from `getExportFileDefaultAgent`, which returns an empty string to indicate when no agent will be used by default.

You may get the names of the agents (like Targa, Sun AU Sound, etc.) by calling `getExportFileAgentPropList`. You should not store these names and should only use them for display in a user interface, because they are friendly names that may be localized or differ from system to system. Use the agent strings (like `"kMoaCfFormat_TGA"`, `"kMoaCfFormat_AU"`) for storing preferences instead.

Agents may be hidden. ExportFile exposes hidden agents so that they may be used like any other agent, but hidden agents should not be displayed in any user interface.

Agents do not influence labels. For example, if exporting a Shockwave 3D member, specifying `"kMoaCfFormat_JPEG"` as the agent without specifying the `#image` label will cause an error to occur because the default label is `#w3d`, which cannot be used with the JPEG agent. It is assumed that you will typically know which label you would like to use for a particular agent: if you are using an agent, you should specify the label to use explicitly. This is also why the `getExportFileDefaultLabel` handler does not accept an agent as an argument.

It is possible to loop over every label for the member, passing them to `getExportFileAgentPropList` to tell which agents may be used for each label. However, for performance reasons you should avoid doing this because it is slow.

Options

Many of the handlers in ExportFile accept a property list of options. Here are the options that may be in the list. To get the default options, call `getExportFileDefaultOptions`.

#incrementFilename

An integer specifying to increment the filename instead of replacing an existing file. For instance: if `TRUE`, when exporting a file called File.txt, and File.txt already exists, then the file is instead exported to File (2).txt, File (3).txt and so on. The default is `FALSE`.

#replaceExistingFile

An integer specifying to replace the existing file instead of erroring if a file with the same name already exists. The default is `TRUE`.

Only files are replaced. If a folder exists with the same name, an error occurs.

This option is ignored if the `#incrementFilename` option is `TRUE`.

#newFolder

An integer specifying to create a new folder if it does not exist instead of erroring. For instance: if `TRUE`, when exporting to Nonexists\File.txt, and the Nonexists folder does not exist, it is automatically created. The default is `FALSE`.

#alternatePathExtension

An integer specifying to use an alternate extension in the path. For instance: if `1`, when exporting a file with the `#html` label, the `"html"` extension will be used in the path instead of `"htm"`. The default is `0`.

An alternate path extension is used if the integer is non-zero. There may be more than one alternate path extension, or there may be none. The value of the integer specifies which alternate path extension to use. If the value is too high, an error occurs.

This option is useful for Xtra Media where an assumed extension is used. If you know the first assumed extension is incorrect, this option may be specified in order to use a different one.

This option is ignored if the extension is specified in the path.

#location

A symbol specifying where paths should be relative to. It may be one of the values in the following table. The default is `#current`.

Location	Description
`#current`	Current directory.
`#documents`	Documents folder. Windows only (this requirement may be removed in future versions.)
`#pictures`	Pictures folder. Windows only (this requirement may be removed in future versions.)
`#music`	Music folder. Windows only (this requirement may be removed in future versions.)
`#videos`	Videos folder. Windows only (this requirement may be removed in future versions.)

This option is useful for projectors which are installed in Program Files. In this scenario, the current directory, the movie path and the application path may all be locations in Program Files, which is not writable without admin rights. Setting this option allows convenient access to user folders to export the file into instead.

The option is ignored if the path is absolute.

#writerClassID

A Lingo list specifying the GUID of the writer to use. The default is `IID_NULL`.

Multiple Xtras can provide writers for agents. The `"kMoaCfFormat_MPEG3"` agent has writers provided by both the MPEG 3 Import Export and FLVAsset Xtras. Typically, when the file is exported, ExportFile will go through each writer in order until one of them successfully exports the file, effectively abstracting them away.

It may occasionally be useful to break this abstraction. Say that somebody creates a new MP3 Audio writer with a new feature, such as the ability to change the volume, so you would like to use that writer specifically. Setting this option allows the specific writer to be used and the others to be ignored.

This option is ignored if it is set to `IID_NULL`. Setting this option does not change which agent will be used - only the writer. For example, to export MP3 Audio, the `"kMoaCfFormat_MPEG3"` agent must still be specified, even if you set this option to the GUID of an MP3 Audio writer.

In Lingo, `IID_NULL`, which represents a null GUID, is represented as a Lingo list with sixteen zeroes.

```
set IID_NULL to [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

#agentOptions

A property list specifying options for the agent. The default is an empty property list if no agent is used, or the default options for the agent if an agent is used.

Agents allow you to set options such as the colour depth of images, whether or not the file will be compressed, the JPEG quality factor, the frequency of a sound, and so on. Which options are available for an agent are defined by the Agent Xtra. The default values may be different depending on the content - for example, the default colour depth agent option may be 8-bit if the member is a 8-bit bitmap, or 32-bit if the member is a 32-bit bitmap. However, the properties in the list should always be the same for a particular agent, even if their default values change.

In practice, the Agent Xtras included with Director rarely use the content to determine the default agent options, instead defaulting to the same value for every member - for example, the image agents included with Director default to 24-bit colour depth regardless of the member passed in. It is nonetheless possible for Xtras to provide smarter defaults if they choose to implement it.

The properties in this list are case-sensitive strings, which may contain characters that are invalid in symbols, such as spaces. You may not convert them to symbols.

Setting this option is taken as a strong hint that you would like to use an agent. For example, if the `#sound` label is used on Windows and the agent options are specified, the WAVE Sound agent is preferred even though it is not typically required to export WAVE Sound on Windows, in order to honour the agent options. However, if no agent is available,

no error occurs. You should generally specify the agent explicitly if setting this option so that you can know it will be interpreted as expected.

Handlers

exportFileStatus

```
* exportFileStatus
```

Returns the error code of the last method called.

The last error code is a runtime value - if an ExportFile handler that causes an error to occur is called in a movie, and then the `exportFileStatus` handler is called in a second movie, it receives the last error code for the first movie. The last error code is not reset by the Lingo `clearGlobals` handler.

The following table is a list of error codes and their descriptions.

Error Code	Description
`0`	Everything is fine.
`1`	Bad parameter.
`2`	Not enough memory to export the file.
`4`	Bad MOA Class. This generally means that the Mix Services Xtra is missing.
`5`	Bad MOA Interface. Same as Bad MOA Class in symptoms and remedies.
`6`	This error occurs if the folder the file is being exported to does not exist and the `#newFolder` option is `FALSE`.
`20`	Internal error. This error should never occur.
`84`	Read past end of stream. The agent failed to read the member's content.
`85`	Creating the file requires elevation.
`88`	Wrote past end of stream. Out of disk space.
`900`	File to be written to is read-only.

Error Code	Description
902	File to be created exists and the #replaceExistingFile option is FALSE.
905	Path is too long, or a folder with the same name exists and cannot be replaced.
1101	SWA Compressor configuration error.
2003	Xtra Asset Type unknown. The Asset Xtra required for this member is missing.
2005	Property not found. The Director version is too old to be used with this label, or a required Xtra is missing.
2008	Argument out of range. This error occurs if the value of the #alternatePathExtension option is too high.
2009	Integer expected.
2010	Symbol expected.
2012	String expected.
2024	Stream out failed. The Asset Xtra failed to save the member.
2031	Value Type mismatch.
3000	Cast member expected.
3003	Media label not supported. ExportFile does not support this label.
3004	Media format not supported. ExportFile does not support this format.
3007	Media data nonexistent. The member has no media attached.
3008	Label not found. The label is not in the label list.
3017	Handler not defined. This error occurs when an unknown handler symbol is passed to the Alternate Syntax handlers.
3030	Media not ready. The member could not be loaded.
4800	No such agent. The agent is not in the agent property list.
4802	This writer is not compatible with ExportFile.
4803	Bad data source type. The media data for the member is null.

Error Code	Description
16385	Not implemented.

exportFileOut

```
* exportFileOut object member, * path, label, agent, options
```

Exports the content of a member out to a file.

If no error occurs, it returns a property list with `#member`, `#path`, `#pathInfo`, `#label`, `#agent`, and `#options` properties. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `member`: The cast member to export the content of out to a file.
- `path`: An optional path string or path info property list specifying the path to a file. The default is returned by `getExportFileDefaultPath`.
- `label`: An optional label symbol from the label list. The default is returned by `getExportFileDefaultLabel`.
- `agent`: An optional agent string which is a property in the agent property list. The default is returned by `getExportFileDefaultAgent`.
- `options`: An optional options property list with the options to use. The default is returned by `getExportFileDefaultOptions`.

Some members do not have any media attached, such as bitmap members that have been newly created with `new(#bitmap)` or `_movie.newMember(#bitmap)`. An error occurs if `exportFileOut` is used to export such a member.

If the member is not loaded, `exportFileOut` will typically load the member in order to export it. Because loading can be a slow process, it may be desirable to load the member in advance (by calling the Lingo `preloadMember` handler) at a more ideal time, such as during a loading screen.

The returned values may not be the same as those passed in as the arguments. For example, if the filename is specified in the path passed as an argument to `exportFileOut`, and the `#incrementFilename` option is used, the returned path will have the incremented filename, rather than the one that was passed in as an argument.

Audio Mixers

The `#wavAsync` and `#mp4Async` labels, used for exporting mixer members, are the only labels that export the file asynchronously instead of synchronously. You may tell when the mixer has been saved by polling the `isSaving` property of the member. However, you MUST NOT poll the `isSaving` property in a continuous repeat loop. Because the file is exported asynchronously, the mixer can only continue saving when execution is idle. A repeat loop blocks execution, which will prevent the mixer from being saved, deadlocking the application. To prevent this, you must wait a frame each time before you can poll the `isSaving` property again.

Alternatively, you may turn on the `mixerSaved` handler (by using `tellExportFileMixerSaved`) and wait for the event to be fired instead of polling `isSaving`. If an error occurs, so `exportFileOut` returns a void value, the mixer will not be saved and the `mixerSaved` event will not be fired.

While the mixer is saving, other operations on the mixer like `play` and `pause` fail.

You may call the `stop` handler on the mixer member in order to cancel the file export. You may call the `stop` handler with a Timeout in order to impose a maximum time for the file to be exported within.

tellExportFileMixerSaved

```
* tellExportFileMixerSaved integer mixerSavedHandler
```

Turns the `mixerSaved` handler on or off.

If no error occurs, it returns `TRUE` if the `mixerSaved` handler was turned on, or `FALSE` if the `mixerSaved` handler was turned off. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `mixerSavedHandler`: an integer specifying whether to turn the `mixerSaved` handler on (`TRUE`) or off (`FALSE`.)

Before calling `exportFileOut` with an mixer member, call `tellExportFileMixerSaved` to enable or disable the firing of the `mixerSaved` event. If

the mixerSaved handler is turned on, and the exportFileOut handler is called from a movie, the mixerSaved event may be called in that movie at any time. If the mixerSaved handler is not found, no error occurs.

The mixerSaved handler

The mixerSaved handler takes three arguments.

- mixerMember: The mixer member whose content was exported out to a file.
- result: A value which is TRUE if the Audio Mixer was successfully exported, or FALSE if the Audio Mixer failed to be exported.
- errCode: An integer which is the error code. These are the same error codes returned by exportFileStatus.

When the mixerSaved handler is called, use the errCode argument to get the error code instead of calling exportFileStatus. The error code returned by exportFileStatus is not set when the mixerSaved handler is called.

Calling the stop handler on the mixer member causes the mixerSaved event to be fired early.

getExportFileLabelList

```
* getExportFileLabelList object member, * options
```

Gets the label list.

If no error occurs, it returns a linear list of label symbols. If an error occurs, it returns a void value. Call exportFileStatus to get the last error code.

- member: The cast member to get the label list for.
- options: An optional options property list with the options to use. The default is returned by getExportFileDefaultOptions.

The label list is generally, but not always, the same for any member of a given type. The label list may be empty, such as if the member is linked. ExportFile does not export linked media.

getExportFileAgentPropList

```
* getExportFileAgentPropList object member, * path, label, options
```

Gets the agent property list.

- `member`: The cast member to get the agent property list for.
- `path`: An optional path string or path info property list specifying the path to a file. The default is returned by `getExportFileDefaultPath`.
- `label`: An optional label symbol from the label list. The default is returned by `getExportFileDefaultLabel`.
- `options`: An optional options property list with the options to use. The default is returned by `getExportFileDefaultOptions`.

Note that the `path` argument is only used to determine the label if it is not specified, saving you a call to `getExportFileDefaultLabel`. It does not narrow down or change the order of the agent property list. Passing `"image.tif"` as the path argument will cause the `#image` label to be used, and therefore the property list of image agents to be returned, including both the TIFF agent and other image agents, and with no guarantee that the TIFF agent will be the first property in the list. To get the default agent given a specific path, call `getExportFileDefaultAgent` instead.

If no error occurs, it returns a property list of agents. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

The properties in the agent property list are agent strings, and the values are agent info property lists.

The `#name` property of an agent info property list should only be used for display in a user interface, because they are friendly names that may be localized or differ from system to system. The `#hidden` property of an agent info property list indicates that the agent should never be displayed in a user interface.

Avoid opening and closing Agent Xtras by calling the Lingo `openXlib` and `closeXlib` handlers. When the `getExportFileAgentPropList` handler is called for the first time, it enumerates the Agent Xtras, and the agent property list is cached by ExportFile. If an Agent Xtra is opened or closed, ExportFile recognizes that its cache is

invalid, and must enumerate the list of Agent Xtras again. For optimal performance, include Agent Xtras in the Xtras folder instead.

getExportFileDefaultPath

```
* getExportFileDefaultPath object member, * info, label, agent,
options
```

Gets the default path.

If no error occurs, it returns a path string or path info property list. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `member`: The cast member to get the default path for.
- `info`: An optional integer specifying to return a path info property list (`TRUE`) instead of a path string (`FALSE`.) The default is `FALSE`.
- `label`: An optional label symbol from the label list. The default is returned by `getExportFileDefaultLabel`.
- `agent`: An optional agent string which is a property in the agent property list. The default is returned by `getExportFileDefaultAgent`.
- `options`: An optional options property list with the options to use. The default is returned by `getExportFileDefaultOptions`.

The `getExportFileDefaultPath` handler does not perform file IO. It returns the default path that ExportFile will attempt to use, which is not necessarily the path the file will be exported to. The `#incrementFilename`, `#replaceExistingFile` and `#newFolder` options are ignored by `getExportFileDefaultPath` because they are only used when the file is created. To get the path of the exported file, get the `#path` or `#pathInfo` property in the property list returned by `exportFileOut`.

The path info property list returned by `getExportFileDefaultPath` has `#dirname`, `#basename`, `#extension`, and `#filename` properties. Changing the `#filename` property without changing the `#basename` or `#extension` to match, or vice-versa, will cause the path info property list to become invalid. To ensure the path info property list remains valid, delete the `#filename` property if changing the `#basename` or `#extension` properties, or vice-versa.

getExportFileDefaultLabel

```
* getExportFileDefaultLabel object member, * path, options
```

Gets the default label.

If no error occurs, it returns a label symbol. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `member`: The cast member to get the default label for.
- `path`: An optional path string or path info property list specifying the path to a file. The default is returned by `getExportFileDefaultPath`.
- `options`: An optional options property list with the options to use. The default is returned by `getExportFileDefaultOptions`.

If the label list is empty, an error occurs.

getExportFileDefaultAgent

```
* getExportFileDefaultAgent object member, * path, label, options
```

Gets the default agent.

If no error occurs, it returns an agent string. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `member`: The cast member to get the default agent for.
- `path`: An optional path string or path info property list specifying the path to a file. The default is returned by `getExportFileDefaultPath`.
- `label`: An optional label symbol from the label list. The default is returned by `getExportFileDefaultLabel`.
- `options`: An optional options property list with the options to use. The default is returned by `getExportFileDefaultOptions`.

The `getExportFileDefaultAgent` handler may return an empty string to indicate that no agent will be used by default. The empty string does not indicate that an error has occurred. Only a void value indicates that an error has occurred.

getExportFileDefaultOptions

```
* getExportFileDefaultOptions object member, * path, label, agent
```

Gets the default options.

If no error occurs, it returns an options property list. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `member`: The cast member to get the default options for.
- `path`: An optional path string or path info property list specifying the path to a file. The default is returned by `getExportFileDefaultPath`.
- `label`: An optional label symbol from the label list. The default is returned by `getExportFileDefaultLabel`.
- `agent`: An optional agent string which is a property in the agent property list. The default is returned by `getExportFileDefaultAgent`.

The `getExportFileDefaultOptions` handler is provided primarily for reference, to be called from the Message Window to see what the default options are. There is typically no need to call it from a script.

Although the items in the options property list are always in the same order, do not access the values in the list by their position. In future versions of ExportFile, additional options may be included, which may not be added to the end of the list, causing the order of the list to change. Access the values by property, such as by calling the Lingo `getaProp` handler, instead.

Getting the Interfaces in #agentOptions

The agent options may contain `PIMoaUnknown` interface values. However, Lingo does not provide any method of interacting with MOA interface pointers. If ExportFile added to their reference count, then it would leak memory because Lingo has no way to free them. In contrast, if it did not add to their reference count, then they could be wild pointers by the time the return value is used. Therefore, ExportFile does not include these values in the Lingo property list returned by `getExportFileDefaultOptions`.

If you are an Xtra developer who would like to leverage the functionality of ExportFile, you can get interface pointer values that are not otherwise included. To do this,

include the exportfile.h header provided with ExportFile in your project. Call `MoaCreateInstance` to create an instance of ExportFile's `IMoaMmXScript` interface. Then, use its `Call` method with `m_getExportFileDefaultOptionsX`, a handler which is not in the Lingo Message Table and is only available to Xtras.

Do not call the ExportFile Lingo handlers via this method. Use the `CallHandler` method of the `IMoaDrPlayer` or `IMoaDrMovie` interfaces instead.

```
/*
calls getExportFileDefaultOptionsX with the member value passed in
and gets a result value, which the caller is expected to release
*/
MoaError
TStdXtra_IMoaMmXScript::CallGetExportFileDefaultOptionsX(PMoaMmValue
memberValuePointer, PMoaMmValue resultValuePointer) {
      PIMoaMmXScript mmXScriptInterfacePointer = NULL;
      MoaDrCallInfo callInfo = {};

      moa_try

      ThrowNull(memberValuePointer);
      ThrowNull(resultValuePointer);

      ThrowErr(pObj->pCallback->MoaCreateInstance(&IID_IExportFile,
&IID_IMoaMmXScript, (PPMoaVoid)&mmXScriptInterfacePointer));
      ThrowNull(mmXScriptInterfacePointer);

      callInfo.methodSelector = m_getExportFileDefaultOptionsX;
      callInfo.nargs = 1;
      callInfo.pArgs = memberValuePointer;

      ThrowErr(mmXScriptInterfacePointer->Call(&callInfo));

      *resultValuePointer = callInfo.resultValue;

      moa_catch

      if (resultValuePointer) {
            *resultValuePointer = kVoidMoaMmValueInitializer;
      }
```

```
	moa_catch_end

	if (mmXScriptInterfacePointer) {
		mmXScriptInterfacePointer->Release();
		mmXScriptInterfacePointer = NULL;
	}

	moa_try_end
}
```

The result value is a property list with the options, including an `#agentOptions` property, which includes the interface pointers cast to integer values. ExportFile AddRef's these, and you are expected to know which properties are interfaces and Release them when done with them. You are also expected to release the result value when done with it.

If you are not sure which properties are interfaces, use the `Call` method with `m_getExportFileDefaultOptions`, then again with `m_getExportFileDefaultOptionsX`. The resulting values will be identical, except that the `#agentOptions` property list returned by `m_getExportFileDefaultOptionsX` includes the interface pointers. Then use the `GetPropertyNameByIndex` and `GetValueByProperty` methods of the `IMoaMmList` interface to iterate each property in the `#agentOptions` list from the call with `m_getExportFileDefaultOptionsX` and test if it is in the `#agentOptions` list from the call with `m_getExportFileDefaultOptions`. If it is not, then the value is an interface pointer.

getExportFileDisplayName

```
* getExportFileDisplayName object member, * path, label, agent,
options
```

Gets a display name.

If no error occurs, it returns a display name string. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

- `member`: The cast member to get the display name for.
- `path`: An optional path string or path info property list specifying the path to a file. The default is returned by `getExportFileDefaultPath`.

- `label`: An optional label symbol from the label list. The default is returned by `getExportFileDefaultLabel`.
- `agent`: An optional agent string which is a property in the agent property list. The default is returned by `getExportFileDefaultAgent`.
- `options`: An optional options property list with the options to use. The default is returned by `getExportFileDefaultOptions`.

The display name describes the filetype for display in a user interface, such as a file save dialog. If the label is `#xtraMedia`, the display name is the type display name of the member's type. If an agent is used, the display name is the agent's name. You should not store these names and should only use them for display in a user interface, because they are friendly names that may be localized or differ from system to system.

getExportFileTypePropList

```
* getExportFileTypePropList
```

Gets a property list of types.

If no error occurs, it returns a property list. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

The `getExportFileTypePropList` handler can be called to get the full property list of member types that may be created with `new` or `_movie.newMember`. The properties in the list are type symbols, and the values in the list are type display name strings.

getExportFileIconPropList

```
* getExportFileIcon object member
```

Gets a property list with any combination of `#color`, `#bw` and `#mask` properties (Director 8 or newer.)

If no error occurs, it returns a property list. If an error occurs, it returns a void value. Call `exportFileStatus` to get the last error code.

The `#color`, `#bw`, or `#mask` items may be void values if there is no colour icon, black and white icon, or mask, respectively. Otherwise, they are Lingo image objects.

The images are not guaranteed to have a consistent resolution. If you expect them to be the same size, call the Lingo `copyPixels` handler to resize them. The colour image will have a 32-bit colour depth. The black and white image, and the mask image, will have a 1-bit colour depth.

The icons are not necessarily guaranteed to be the same for any member of a given type. For instance, linked members can have distinct icons, script members can have different icons depending on the script's type, and the icons for the same member can be different from one Director version to another.

Alternate Syntax

The ExportFile Xtra provides an alternate syntax which may be used to call all of its handlers. The alternate syntax does not provide any additional functionality that is impossible to otherwise attain, but is ideal if you would like to use ExportFile in a more object oriented fashion. Do not call `new` on the Xtra object.

Here is how to call `exportFileOut` and `exportFileStatus` in the alternate syntax.

```
set exportFile to xtra("ExportFile")
exportFile.Out(member("Example"))

if the ilk of the result <> #propList then
  alert("ExportFile Error! (" & string(exportFile.Status()) & ")")
end if
```

Here is how to call `tellExportFileMixerSaved` in the alternate syntax.

```
set exportFile to xtra("ExportFile")
callTell ExportFile, #MixerSaved, TRUE

if integerP(the result) = FALSE then
  alert("ExportFile Error! (" & string(exportFile.Status()) & ")")
end if
```

Here is how to call `getExportFileAgentPropList` in the alternate syntax, specifying the `#sound` label.

```
set exportFile to xtra("ExportFile")
callGet ExportFile, #AgentPropList, member("Example"), void, #sound

set agentPropList to the result

if the ilk of agentPropList <> #propList then
  alert("ExportFile Error! (" & string(exportFile.Status()) & ")")
end if
```

FAQ

Which Xtras should I include with my projector (aside from the ExportFile Xtra itself?)

You should always include the Mix Services Xtra. Although some features of ExportFile may work without Mix Services, including it will enable ExportFile to work optimally.

Include the Asset Xtras for any members you'd like to export. If you're exporting a text member, include the Text Asset Xtra. If you're exporting a Shockwave 3D member, include the Shockwave 3D Asset Xtra, and so on.

Include the Agent Xtras for the agents you wish to use. If you are exporting to PNG, include the PNG Import Export Xtra. If you are exporting to MP3 Audio, include the MPEG 3 Import Export Xtra, and so on. If multiple Agent Xtras provide the same agent, you only need to include at least one.

If you are exporting sounds, you should include the Sound Import Export Xtra, regardless of which agents you are using or whether or not you use an agent at all. This is because for the `#sound` label, Director itself uses the Sound Import Export Xtra internally.

If you are exporting to MP3 Audio or SWA Audio, include the MPEG 3 Import Export Xtra, SWA Import Export Xtra, and SWA Compressor Xtra, which are required for compressing the sound.

Which Director versions are compatible with ExportFile?

ExportFile will work in Director 6 or newer, with some limitations.

- Exporting Audio Mixers is only supported in Director 11.5 or newer.
- Exporting JavaScript is only supported in Director MX 2004 or newer.
- Exporting Shockwave 3D is only supported in Director 8.5 or newer.
- The `getExportFileIconPropList` handler is only supported in Director 8 or newer.
- Using the `#image` label with members that are not bitmaps is only supported in Director 8 or newer.
- Using the `#pict` label is only supported in Director 7 or newer.
- Agents are only supported in Director 7 or newer.
- Exporting Text is only supported in Director 7 or newer (this does not apply to field members: exporting Fields is supported in Director 6 or newer.)
- Exporting Flash is only supported in Director 7 or newer.
- Exporting Animated GIFs is only supported in Director 7 or newer.
- Exporting Fonts is only supported in Director 7 or newer.

Why are my exported script files empty?

In protected movies, scripts are compiled and the script text is made empty. ExportFile is not able to recover the original script text.

The `#lingo` or `#javaScript` labels will be in the label list if either of two conditions is met:

1. The member type is `#script`. In this case, the `#lingo` or `#javaScript` label takes priority over the `#composite` label.
2. The member type is not `#script` but `the scriptText` property of the member is not `EMPTY`. In this case, the `#composite` label takes priority over the `#lingo` or `#javaScript` labels.

ExportFile can be used to export scripts on linked members, but cannot export linked scripts. For example, ExportFile may be used to export the script of a linked QuickTime Movie member, but not a script that has been linked using the Link As button in Director.

Exporting JavaScript is only supported in Director MX 2004 or newer. If you would like to export a member's script regardless of its syntax, pass the `scriptSyntax` property of the member as the label argument to exportFileOut.

What is the difference between the #image and #pict labels?

The `#image` label is generally preferred. You should use the `#image` label if you are exporting to a raster image format, such as JPEG or PNG. The `#pict` label is most useful for exporting PICT members, which may contain vector shapes.

Note that on Mac, PICT is the standard image format, so there is no distinction between the `#image` and `#pict` labels on Mac. To enable your code to work cross platform, you should explicitly use the `#image` label if using agents to export image files.

What is PICT?

PICT is a metafile format which is often used for rasters but may also be used for vectors. Director supports importing PICT files by going to File > Import... selecting "Import PICT as PICT" from the Media dropdown, and then selecting a PICT file. This creates a PICT member, which is distinct from a bitmap member because vector shapes in the picture are preserved.

The `#pict` label may be used to export PICT files with the shapes in the picture intact. If the `#pict` label is used on a bitmap member, then a raster PICT file will be exported.

On Windows, PICT files may be opened in LibreOffice Draw.

Why don’t my PNG images have transparency?

By default, the PNG agent included with Director exports images with 24-bit colour depth, meaning they will not have any transparency. If you expect 32-bit images to be exported with transparency, you need to specify the colour depth in the agent options.

```
set transparentImageMember to member("Ghost")

exportFileOut(transparentImageMember, void, #image,
"kMoaCfFormat_PNG", [#agentOptions: ["PixelFormat_BitsPerPixel": the
depth of transparentImageMember]])
```

Why can't I re-import my PNG images into Director?

There is a bug in Director that causes it to crash when reading a file that was exported using the PNG agent included with Director. The files can be opened in other applications without issue, so it is unknown why this crash occurs. The crash does not occur with files exported using other agents.

Why is my 32-bit image exported all transparent?

In Director, an all-zero alpha channel has the same effect as no alpha channel. In other words, your image *is* completely transparent, but Director ignores the alpha channel in completely transparent images. If you re-import the exported bitmap into Director you can see this for yourself: the imported result is identical.

You can fix this by calling the `setAlpha` handler in Lingo, although any partially transparent images will effectively have their alpha channel removed.

```
set alphaImage to the image of member("Alpha")
setAlpha(alphaImage, 255)
set the image of member("Alpha") to alphaImage
```

Why am I unable to export sound with the MP3 Audio or SWA Audio agents?

In Director 11.5, there is a bug in the MP3 Audio and SWA Audio agents included with Director that causes them not to work in a projector. This bug is fixed in Director 12.

There is a bug in the MP3 Audio and SWA Audio agents included with Director that causes them to fail exporting stereo sound. In Director MX 2004, this bug may be fixed by unchecking the Convert Mono to Stereo checkbox under File > Publish Settings... > Shockwave, but only in Director itself: projectors still have the bug. In newer Director versions, there is no fix.

Why doesn't setting the frequency work for the MP3 Audio or SWA Audio agents?

Although the MP3 Audio and SWA Audio agents included with Director appear to provide the ability to set the frequency by setting the `"PrefFrameRate"` agent option, this option is not implemented by the agent and the value is ignored. Sounds exported by the MP3 Audio and SWA Audio agents are always 22050 Hz.

Why do I get an empty file from the AAC Audio and MP4 Audio agents?

There is a bug in the AAC Audio and MP4 Audio agents included with Director that causes them to export an empty file. No error occurs when an empty file is exported, because for some formats (such as text files) an empty file is a valid result, and ExportFile has no way of differentiating if an empty file is valid for a particular format. As such, avoid using the AAC Audio and MP4 Audio agents in your movie.

Because the Lingo `importFileInto` handler uses agents in order to import files, you may call `importFileInto` to test that a file that was exported with an agent is valid. Call the Lingo `findEmpty` handler to find an empty member, then call `importFileInto` to import a file that was previously exported with `exportFileOut` into that member. If the member's type is not `#empty`, then the file is valid, but if the member's type is `#empty`, then the file is not valid. Optionally erase the member by calling the Lingo `erase` handler when you're done.

In Director 11.5 and newer, you may call the Lingo `createSoundObject` handler on a new mixer member, then export the member with ExportFile, in order to export sounds to MP4 without using the MP4 Audio agent.

Why does the agentPropList for palette members have image formats?

This is not a bug with ExportFile - rather, the Palette agent really does claim itself to be compatible with image agents. On some level, this makes sense. A palette could be represented as an image where each pixel of the image represents a colour in the palette.

Unfortunately, in practice it appears that the other agents included with Director are not compatible with the Palette agent, and if they are used with it an error occurs. As of the time of writing, it is uncertain why this is the case.

Why are there no letters in the text I exported as an image?

The shapes of the letters in text members are defined by the alpha channel. If the image is exported with 24-bit colour depth, the alpha channel will be missing, rendering them invisible. To solve this, specify 32-bit colour depth in the agent options.

```
set loremIpsumMember to member("Lorem Ipsum")

exportFileOut(loremIpsumMember, void, #image, "kMoaCfFormat_PNG",
[#agentOptions: ["PixelFormat_BitsPerPixel": 32]])
```

How can I export every cast member from my movie?

This simple Lingo handler may be used to export every cast member from your movie.

```
on exportFilesOut
  set milliseconds to the milliseconds

  set options to [#newFolder: TRUE, #incrementFilename: TRUE]

  repeat with i = 1 to the number of castLibs
    set pathInfo to [#dirname: the moviePath & "Exported Files\" & the
name of castLib i]

    repeat with j = 1 to the number of members of castLib i
      set exportMember to member j of castLib i

      if the type of exportMember <> #empty then
        exportFileOut(exportMember, pathInfo, void, void, options)

        if the ilk of the result <> #propList then
          put("ExportFile Error!")
          put("Member: " & string(exportMember))
          put("Error Code: " & string(exportFileStatus()))
        end if
      end if
```

```
    end repeat
  end repeat

  put("Elapsed Milliseconds: " & string(the milliseconds -
milliseconds))
end
```

This Lingo handler will create a new folder called Exported Files, then export every cast member in your movie into that folder using the default label and agent. The files will be sorted into subfolders that are named after the casts that the members are in. If a member has the same name as a previous one in the same cast, the filename is incremented. If an error occurs, it is printed to the Message Window.

“Pull the lever Donga! The lever!”