

QNX[®] Neutrino[®] Realtime Operating System

Addon Interfaces Library Reference

For QNX[®] Neutrino[®] 6.2.1 or later

© 2004–2010, QNX Software Systems GmbH & Co. KG. All rights reserved.

Published under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: <http://www.qnx.com/>

Electronic edition published 2010

QNX, Neutrino, Photon, Photon microGUI, Momentics, Aviage, and related marks, names, and logos are trademarks, registered in certain jurisdictions, of QNX Software Systems GmbH & Co. KG. and are used under license by QNX Software Systems Co. All other trademarks belong to their respective owners.

About This Reference	v
What you'll find in this guide	vii
Typographical conventions	vii
Note to Windows users	viii
Technical support	ix
1 Overview	1
An Example	3
Library components	4
2 Addon Interfaces Library Reference	5
<i>AOIctrl_t</i>	9
<i>AOInterface_t</i>	10
<i>AoAdd()</i>	11
<i>AoAddDirectory()</i>	12
<i>AoAddStatic()</i>	13
<i>AoAddUnloadSignal()</i>	14
<i>AoGetInterface()</i>	15
<i>AoHold()</i>	17
<i>AoIterate()</i>	18
<i>AoIterateHoldGet()</i>	20
<i>AoRelease()</i>	22
<i>AoRemove()</i>	23
<i>AoFindExt()</i>	24
<i>AoFindFormats()</i>	25
<i>AoFindMime()</i>	26
<i>AoFindName()</i>	28
<i>AoFindStreams()</i>	30
<i>AoOpenFilespec()</i>	32
A Implemented Interfaces	35
Built-in Interfaces	37
<i>AODeConstructor</i>	38

<i>Create()</i>	38
<i>Destroy()</i>	38
AOExtInspector	39
<i>RateExtension()</i>	39
AOFormatInspector	39
<i>RateFormat()</i>	39
AOMimetypeInspector	40
<i>RateMimetype()</i>	40
AOStreamer	40
Open	41
Close	41
<i>Sniff()</i>	41
<i>Read()</i>	42
<i>Write()</i>	43
<i>Seek()</i>	43
<i>Tell()</i>	44
<i>Length()</i>	44
<i>SideInfo()</i>	44
AOStreamInspector	45
<i>RateStream()</i>	45
AOResourceAccess	46
<i>GetResources()</i>	46
<i>SetResource()</i>	46
Built-in interfaces	47
Unloading and InitializeInterface	47
Name	48

B Using Addon Resources 49

An Example	51
Using addon resources in your application	54

C Defined Structures 57

AOIStream_t	60
AOMimeInfo_t	61
AOResource_t	62
AOAudioFormat_t	64
ImageFormat_t	65
MediaFormat_t	66
VideoFormat_t	67

About This Reference

What you'll find in this guide

The *Addon Interfaces Library Reference* is intended for users who want to use the library to create addons that dynamically add functionality to applications. You can use addons to add functionality to your application without requiring redeployment of the entire application.

The new Multimedia Library uses the Addon Interfaces Library to build its interfaces. For more information about the Multimedia Library, and to see examples of interfaces implemented with the Addon Interfaces Library, see the *Multimedia Developer's Guide*.

This table may help you find what you need in the *Addon Interfaces Library Reference*:

When you want to:	Go to:
Read an overview of the Addon Interfaces Library, including a step-by-step example of how to write your own interface	Overview
See the list of basic API (structures and functions) that make up the Addon Interface Library, and a list of auxilliary interface-specific functions	Addon Interfaces Library Reference
See a list of existing interfaces that have been designed for use with the Addon Interfaces Library, which you can use in your applications	Appendix A: Existing Interfaces
Read about accessing data (resources) in an addon using the <code>AOResourceAccess</code> interface	Appendix B: Using Addon Resources
See the list of existing structures that have been designed for use with the Addon Interfaces Library, which you can use in your applications	Appendix C: Defined Structures

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

Reference	Example
Code examples	<code>if(stream == NULL)</code>
Command options	<code>-lR</code>
Commands	<code>make</code>
Environment variables	<code>PATH</code>
File and pathnames	<code>/dev/null</code>
Function names	<code>exit()</code>
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	<code>something you type</code>
Keyboard keys	Enter
Program output	<code>login:</code>
Programming constants	<code>NULL</code>
Programming data types	<code>unsigned short</code>
Programming literals	<code>0xFF, "message string"</code>
Variable names	<code>stdin</code>
User-interface components	Cancel

We use an arrow (→) in directions for accessing menu items, like this:

You'll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:



Notes point out something important or useful.



CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.



WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

To obtain technical support for any QNX product, visit the **Support** area on our website (www.qnx.com). You'll find a wide range of support options, including community forums.

Chapter 1

Overview

The Addon Interfaces Library, `libaoi.so`, contains functions for accessing *addon* interfaces. An addon is an implementation of a set of interfaces that provide some new functionality, and that can be dynamically loaded and unloaded by an application at runtime. An interface is a group of related functions and data. By using a known set of interfaces, you can add functionality to deployed applications without having to recompile and redeploy the application. Addons are typically contained in DLLs, and are sometimes also called *plugins*.

This approach to dynamically adding functionality is different from simply loading a DLL at runtime using a function such as `dlopen()`, as the application does not have to know the specific functionality contained by the DLL ahead of time. Rather, the application can search the DLL for a known interface, and if available, that functionality can be accessed.

An Example

For example, say that you want to write a screensaver application. You could define a screensaver interface that would allow you to write addons that contain screensaver functionality. Each screensaver addon would have a known interface that the application would use to start, stop, and set options for that particular screensaver. To write such an application, you would perform the following general steps to use the Addon Interfaces Library:

First, determine the mandatory functionality for the application. For each screensaver addon, you need a function that creates and populates an options pane. You may also want a function that initializes the screensaver addon, and another to uninitialized it to restore any resources acquired during the initialization of the addon.

You'll also need a function to start and stop the screensaver display. You may end up with an interface similar to:

```
typedef struct
{
    int32_t Initialize(void);
    int32_t Uninitialize(void);
    PtWidget_t *OptionsWidget(void);
    int32_t Start(void);
    int32_t Stop(void);
} ScreenSaver;
```

Within your addon a declaration of the interfaces available is provided that allows the AOI library to discover and make use of the interfaces.

```
AOInterface_t interfaces [] =
{
    { "Name", 0, "my_screensaver" },
    { "ScreenSaver", SS_VERSION, &SS2 },
    { "ScreenSaverPrefs", SS_PREFSVERSION, &SSP2 }
    { 0,0,0 }
};
```

Your screensaver would use the addon library to find all the available addons (loaded DLLs, which contain the screensavers) that have this set of interfaces, and allow the user to select which screensaver (and options for that screensaver) to use.

A screensaver addon might have user-set preferences. In this case, a “preferences” interface would exist for that addon. The application would only display preferences for those screensavers with a “ScreenSaverPrefs” interface. It might look like:

```
typedef struct
{
    int32_t LoadPrefs(void);
    int32_t SavePrefs(void);
} ScreenSaverPrefs;
```

Library components

The Addon Interfaces Library consists of:

- the basic elements (structures and functions) required to support interface creation and management
- some predefined interfaces and supporting predefined structures.

Chapter 2

Addon Interfaces Library Reference

The Addon Interfaces Library provides a framework for developing standard interfaces. Interfaces are used to build extensibility into an application without having to rewrite and redeploy the whole application. For example, because the new Multimedia Library implements standard addon interfaces, you can use it to write a video playback application that can handle new video formats as they become available, simply by adding a new filter or set of filters for that format.

The Addon Interfaces Library uses two structures to manage interfaces: **AOInterface_t**, which represents an interface, and **AOICtrl_t**, which represents an addon interface (AOI) control for one or more interfaces. Typically, addon code (including the interface(s) and control) is contained in a DLL, and is dynamically loaded by an application at runtime.

Each time an application requires an interface, it should hold the addon control containing the interface to prevent the addon from being unloaded while still in use. At this point, the addon control is loaded into memory, and its hold count is incremented. An addon control may be unloaded when there are no more holds on it.

This section lists the basic elements (structures and functions) of the Addon Interfaces Library (**libaoi.so**) that allow you to write and access your own interfaces.

Basic Addon Interfaces Library components:

AOI Structures

Structure	Description
AOICtrl_t	A structure that defines an interface control.
AOInterface_t	A structure that defines an interface.

AOI Functions

Function	Description
<i>AoAdd()</i>	Add the interfaces contained in a DLL to the global list of interfaces.
<i>AoAddStatic()</i>	Add a static list of interfaces to the global list of interfaces.
<i>AoAddDirectory()</i>	Add the interfaces contained in all DLLs in a directory to the global list of interfaces.
<i>AoRemove()</i>	Remove a control from the global list of interfaces.
<i>AoHold()</i>	Hold a control; ensure it's loaded if necessary.
<i>AoRelease()</i>	Release a control; unload it if necessary.

continued...

Function	Description
<i>AoGetInterface()</i>	Get a specific interface for a control.
<i>AoIterate()</i>	Iterate through the global list of controls and return the one that meets some criteria.
<i>AoIterateHoldGet()</i>	Iterate through the global list of controls to find a control that meets some criteria, hold, and then return the control.
<i>AoAddUnloadSignal()</i>	Add a signal handler to an application to unload an add-on.

Additional interface-specific functions:

Element	Description
<i>AoFindExt()</i>	Find a control with a AOExtInspector interface, and which is best suited for a specific file extension.
<i>AoFindFormats()</i>	Find a control with the best rating for a specific media format.
<i>AoFindMime()</i>	Find a control with the best rating for a mimetype.
<i>AoFindName()</i>	Find a control by its “Name” interface.
<i>AoFindStreams()</i>	Find a control with the best rating for a specific stream.
<i>AoOpenFileSpec()</i>	Find the first control that can open a filespec in a given mode.

Synopsis:

See below.

Description:

A structure that defines an interface control. It contains at least the following members:

char **name* The name of the interfaces that the control is for.

char **fname* The path and filename for the DLL that contains the control's interfaces.

Use this structure to pass the control for an interface.

Classification:

QNX Neutrino

See also:

AOInterface_t

The structure used to define an interface

Synopsis:

```
typedef struct
{
    char * name;
    int32_t version;
    void *interface;
} AOInterface_t;
```

Description:

The `AOInterface_t` structure defines an interface and contains at least the following members:

<i>name</i>	The name of the interface.
<i>version</i>	The version number of the interface.
<i>interface</i>	A pointer to the interface.

The interface itself can be anything you want. Typically it's a pointer to an array of function pointers, a pointer to a function, or a pointer to a string.

Classification:

QNX Neutrino

See also:

`AOICtrl_t`

Synopsis:

```
#include <aoi.h>

const AOIctrl_t *AoAdd(const char *path);
```

Arguments:

path The path and filename of the DLL you want to add.

Library:

```
libaoi.so
```

Description:

This function loads the DLL at the given *path*, registers all the interfaces contained in the DLL, and then unloads the DLL. The function returns the `AOIctrl_t` control for the registered interfaces. You can then access the control's interfaces with `AoGetInterface()`, or search for a specific set of interfaces with one of the `AoFind*()` functions.

Returns:

A pointer to the interfaces control of type `AOIctrl_t` for the added DLL's interfaces.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOIctrl_t`, `AoRemove()`, `AoAddStatic()`, `AoAddDirectory()`

AoAddDirectory()

© 2010, QNX Software Systems GmbH & Co. KG.

Add the interfaces contained in all DLLs in a directory to the global list of interfaces

Synopsis:

```
#include <aoi.h>

int32_t AoAddDirectory(const char *path,
                      const char *pattern);
```

Arguments:

path The directory that contains the DLLs with interfaces you want to make available within your application.

pattern A filename pattern, such as `.so` or `decoder`, that limits the type of files added as DLLs. Set to `NULL` to match all files.

Library:

`libaoi.so`

Description:

This function attempts to add all the interfaces in the DLLs that match a *pattern* at the given directory *path* to the global list of interfaces.

Returns:

0 if successful.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

AoAdd(), *AoAddStatic()*

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoAddStatic(AOInterface_t *interfaces);
```

Arguments:

interfaces An array of **AOInterface_t** structures that is the static list of interfaces you want to make available to your application.

Library:

`libaoi.so`

Description:

This function makes statically defined (that is, not loaded from a DLL) interfaces available to the AOI API. Use this function when you want to directly link your application with a set of interfaces.

Returns:

NULL

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

AOInterface_t, *AoAdd()*, *AoAddDirectory()*

AoAddUnloadSignal()

Add an unload signal for an addon to an application

Synopsis:

```
#include <aoi.h>

int32_t AoAddUnloadSignal(int sig);
```

Arguments:

sig The unload signal you want to add.

Library:

`libaoi.so`

Description:

This function allows you to add signals on which the addons should be unloaded. By default, the addons are automatically unloaded when an application exits normally, but not if the application is killed in some way.

Returns:

0 if successful.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

Synopsis:

```
#include <aoi.h>

void *AoGetInterface(const AOICtrl_t *control,
                    const char *name,
                    const int32_t version,
                    const int32_t nth);
```

Arguments:

control A pointer to an `AOICtrl_t` structure for the control that contains the interface you want to retrieve.

name Optional. The name of the interface you want to retrieve, or NULL if you don't want to find an interface using the "Name" interface.

version Optional. The minimum version of the interface you want to retrieve. This argument is used only if *name* is specified.

nth The instance number of the interface you want to retrieve.

Library:

`libaoi.so`

Description:

This function searches for the *nth* (starting at 0) interface with the given *name* and minimum *version* number in the given control, if specified.

You must hold the control (using `AoHold()`) before calling this function.

Returns:

The *nth* interface control, if found, or NULL if no interface control is found.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOICtrl_t`, `AoHold()`, `AoIterate()`, `AoIterateHoldGet()`

Synopsis:

```
#include <aoi.h>

int32_t AoHold(const AOICtrl_t *ctrl);
```

Arguments:

ctrl A pointer to the `AOICtrl_t` structure for the AOI control you want to hold.

Library:

`libaoi.so`

Description:

This function increments the hold counter for a control. If the control was previously not held, and it's a DLL, the DLL is loaded and initialized if necessary. You must hold a control before you attempt to get one of its interfaces, and release it with `AoRelease()` when you're finished using the interface.

Returns:

0 if successful.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOICtrl_t`, `AoGetInterface()`, `AoIterateHoldGet()`, `AoRelease()`

Find a list of controls for an interface name and version

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoIterate(const char *name,
                           const int32_t version,
                           int32_t * const cookie);
```

Arguments:

name The name of the interface that the returned control contains.

version The minimum version of the interface that the returned control contains.

cookie An opaque variable used to iterate through available AOI controls. Set the value to 0 on the first call to this function.

Library:

libaoi.so

Description:

This function iterates through all available AOI controls, returning each AOI control that has the given interface *name* and minimum version number *version*. The first time you call this function, you should set the value in **cookie* to 0. You can keep calling this function until it returns NULL. If *name* is NULL, *AoIterate()* iterates through all the controls in the global list.

Returns:

A pointer to an `AOIControl_t` structure for the control containing an interface that matches *name* and *version*. Subsequent calls return the next matched control, until there are no more matches. When there are no more matches, the function returns NULL.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOICtrl_t`, `AoGetInterface()`, `AoIterateHoldGet()`, `AoHold()`

Search for a specific interface and hold its control

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoIterateHoldGet(const char *name,
                                  const int32_t version,
                                  int32_t *cookie,
                                  void **interface);
```

Arguments:

<i>name</i>	The name of the interface in the control you want to find and hold.
<i>version</i>	The minimum version of the interface in the control you want to find and hold.
<i>cookie</i>	An opaque variable used to control the iteration through the list of controls. Set this parameter to 0 on the first call to this function.
<i>interface</i>	The returned interface within the held control that meets the search criteria.

Library:

libaoi.so

Description:

This function iterates through the global list of controls, returning each control that has the given interface *name* with the minimum *version* number. Unlike *AoIterate()*, it also holds the control returned, and sets **interface* to the interface you're looking for. The first time that you call this function, you should set the value in **cookie* to 0. You can keep calling this function until it returns NULL. If *name* is NULL, *AoIterate()* iterates through all available controls.

AoIterateHoldGet() is a convenience function that combines *AoIterate()*, *AoHold()*, and *AoGetInterface()*. You *must* release each **AOICtrl_t** at some point, or you will end up with an incorrect hold count, and the DLLs won't be automatically unloaded.

Returns:

A pointer to an **AOICtrl_t** structure for each control that contains the interface *name* with a minimum *version*. The *interface* parameter is set to the interface that meets the search criteria.

When there are no remaining controls that contain a matching interface, this function returns NULL.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOICtrl_t`, *AoRelease()*, *AoIterate()*, *AoHold()*

AoRelease()

Decrement the hold counter for a control

Synopsis:

```
#include <aoi.h>

int32_t AoRelease(const AOICtrl_t *control);
```

Arguments:

control A pointer to an `AOICtrl_t` structure for the control you want to release.

Library:

`libaoi.so`

Description:

This function decrements the hold counter for the given control. If the control represents a DLL, once the hold counter for a control reaches 0, that DLL is unloaded,

Returns:

0 if successful.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOICtrl_t`, `AoHold()`, `AoIterateHoldGet()`

Synopsis:

```
#include <aoi.h>

int32_t AoRemove(const AOICtrl_t *control);
```

Arguments:

control A pointer to an `AOICtrl_t` structure for the control you want to remove.

Library:

`libaoi.so`

Description:

This function removes the given control *control* from the global list of interfaces. The DLL is unloaded if necessary. You should be certain that all holds are released before this function is called by calling *AoRelease()* for every hold you place.

Returns:

0 if successful.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOICtrl_t`, *AoRelease()*

AoFindExt()

© 2010, QNX Software Systems GmbH & Co. KG.

Find a control with a **AOExtInspector** interface, and which is best suited for a specific file extension

Synopsis:

```
#include <aoi.h>
const AOICtrl_t *AoFindExt(const char *extension,
                           int32_t *rating,
                           const char *interface,
                           int32_t version);
```

Arguments:

extension The file extension you want to inspect.

rating A pointer to where the function stores the returned rating for how well the control can inspect the given *ext*.

interface The name of the interface the control must have.

version The minimum interface version the control must have.

Library:

libaoi.so

Description:

This function finds the control that has an **AOExtInspector** interface which returns the best rating for the given *extension*, and has the given *interface* and *version*, if specified.

Returns:

A pointer to an **AOICtrl_t** control, or NULL if no control is found.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

AOICtrl_t, *AoFindFormats()*, *AoFindMime()*, *AoFindName()*, *AoFindStreams()*

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoFindFormats(const AODataFormat_t *format,
                               int32_t *rating,
                               const char *interface,
                               int32_t version);
```

Arguments:

- format* A pointer to a `AODataFormat_t` structure that specifies the media format you want to find the best interface control for.
- rating* A pointer to where the function stores a returned rating, from 0 to 100, for the returned AOI control.
- interface* The interface required in the control.
- version* The minimum version of the interface required in the control.

Library:

```
libaoi.so
```

Description:

This function finds the control that has an `AOFormatInspector` interface that returns the best rating for the given *format* and has the given *interface* and *version*, if specified.

Returns:

The AOI control with the best rating for *format*, or NULL if no control is found.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AODataFormat_t`, `AoFindExt()`, `AoFindMime()`, `AoFindName()`, `AoFindStreams()`

Find a control with the best rating for a mimetype

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoFindMime(const char *mimetype,
                             int32_t *rating,
                             const char *interface,
                             int32_t version);
```

Arguments:

mimetype The mimetype for which you want to find the best rated control.

rating A pointer to where the function stores the rating, from 0 to 100, of how well the returned control can handle the *mimetype*.

interface The interface that the returned control must contain.

version The minimum version of the interface that the returned control must contain.

Library:

libaoi.so

Description:

This function finds the control that has an **AOMimetypeInspector** interface that returns the best rating for the given *mimetype*, and has the given *interface* and *version*, if specified.

Returns:

A **AOICtrl_t** control that meets the search criteria, or NULL if no AOI control is found.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

AoFindExt(), *AoFindFormats()*, *AoFindName()*, *AoFindStreams()* **AOIctrl_t**

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoFindName(const char *name,
                           const char *interface,
                           int32_t version);
```

Arguments:

name The name of the control you want to find, set in a control’s “Name” interface.

interface The interface contained in the control you want to find.

version The version of the interface in the control you want to find.

Library:

```
libaoi.so
```

Description:

This function finds a control with the *name*, that also has the *interface* and *version* specified. Controls are named if they have a string interface called “Name” declared in their interfaces list.

Returns:

A pointer to an `AOICtrl_t` structure for a control with a matched *name*, if one exists, and NULL if no control is found.

Examples:

Here’s an example of a “Name” interface declaration:

```
AOInterface_t pnm_idecoder_interface[] =
{
    {"Name", 0, "pnm_idecoder"},
    {"Description", 0, "PNM Image Reader"},
    ... (other interfaces)
    {0, 0, 0},
};
```

If the above interfaces were already added to the list of available interfaces, and you wanted to find the AOI control for the `pnm_idecoder` interface, you would write code like:

```
AOICtrl_t *ctrl;

ctrl=AoFindName("pnm_idecoder",NULL,0);

// now we can use the ctrl to find specific interfaces, etc.
```

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

AoFindExt(), *AoFindMime()*, *AoFindFormats()*, *AoFindStreams()*, `AOICtrl_t`

AoFindStreams()

Find a control with the best rating for a specific stream

Synopsis:

```
#include <aoi.h>

const AOICtrl_t *AoFindStreams(AOISStream_t *stream,
                               int32_t *rating,
                               const char *interface,
                               int32_t version);
```

Arguments:

- stream* A pointer to an **AOISStream_t** structure for the stream you want to find the best rated control for.
- rating* A pointer to where the function stores the rating, from 0 to 100, of how well the returned control can handle the *stream*.
- interface* The name of the interface the returned control must contain.
- version* The minimum version of the interface the returned control must contain.

Library:

libaoi.so

Description:

This function finds the control that has an **AOStreamInspector** interface that returns the best *rating* for the given *stream* and has the given *interface* and *version*, if specified.

Returns:

A pointer to an **AOICtrl_t** structure for the control with the best rating for the given *stream*, or NULL if no streamer addons are found.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

AoFindExt(), *AoFindMime()*, *AoFindFormats()*, *AoFindName()*, **AOIctrl_t**

Finds the first control that can open a file in a given mode

Synopsis:

```
#include <aoi.h>

AOIStream_t *AoOpenFilespec(const char *filename,
                           const char *mode);
```

Arguments:

filename The file name you want to open

mode The mode for opening the file; one of:

- rb** Read binary.
- wb** Write binary.

Library:

`libaoi.so`

Description:

This function iterates through all the available addons (addons that have been added using an *AoAdd*()* function) that export a **AOIStreamer** interface and returns the **AOIStream_t** for the first addon that successfully opens the given *filespec* in the given *mode*. In this case, the *control* element of the **AOIStream_t** structure is filled in with the control, and the control is held. When the stream has been closed, you should release this control.

Returns:

A pointer to an **AOIStream_t** for the successfully opened stream, or NULL if no streamer addons are found.

Classification:

QNX Neutrino

Safety

Interrupt handler	No
Signal handler	No
Thread	No

See also:

`AOIStream_t`, *AoRelease()*

Implemented Interfaces

In this appendix...

AODeConstructor	38
AOExtInspector	39
AOFormatInspector	39
AOMimetypeInspector	40
AOStreamer	40
AOStreamInspector	45
AOResourceAccess	46
Built-in interfaces	47

This chapter provides an overview of interfaces that have been defined for use with the Addon Interfaces Library. These interfaces define functions for commonly required functionality. To use these interfaces to create your own addon, you must implement the functions they define. If you use an interface, you should implement all its defined functions.

AOStreamer An interface containing all the functions necessary to implement a one-way byte stream.

AODestructor

A context constructor/destructor interface.

AOStreamInspector

An interface that allows your addon to return a rating as to how well it can process a given stream.

AOFormatInspector

An interface that allows your addon to return a rating as to how well it can process data in the given media format.

AOExtInspector

An interface that allows your addon to return a rating as to how well it can process the given file extension.

AOMimetypeInspector

An interface that allows your addon to return a rating as to how well it can process data with the given mimetype.

AOResourceAccess

An interface that allows an application to access your addon's resources.

Built-in Interfaces

There are three built-in interfaces:

Unloading This interface gives controls that access hardware a chance to leave that hardware in a stable state before being unloaded.

InitializeInterface

This interface allows DLLs to create or choose certain interfaces at runtime instead of at compile time.

Name An interface pointer that points to a string. You need to declare this interface to use *AoFindName()*.

AODeConstructor

A context constructor/destructor interface.

It defines these functions:

- *Create()*
- *Destroy()*

Create()

Synopsis

```
#include <aoi.h>

void *(*Create)(const AOICtrl_t *interfaces);
```

Arguments

interfaces A pointer to an `AOICtrl_t` structure for the control for the created interface.

Description

This function should create and return a new context for the addon.

Returns

A new context for the addon.

Destroy()

Synopsis

```
#include <aoi.h>

int32_t (*Destroy)(void *context);
```

Arguments

context The addon context you want to destroy.

Description

This function should free the *context* for an addon.

Returns

0 if successful.

AOExtInspector

An interface that allows your addon to return a rating as to how well it can process the given file extension. It defines one function:

RateExtension()

Synopsis

```
#include <aoi.h>

int32_t (*RateExtension)(const char *ext);
```

Arguments

ext The file extension you want to rate.

Description

This function should return a rating for the given extension.

Returns

A rating from 0 to 100 of how well the addon can handle the given file extension. 100 is the best rating.

AOFormatInspector

An interface that allows your addon to return a rating as to how well it can process data in the given media format. It defines one function:

RateFormat()

Synopsis

```
#include <aoi.h>

int32_t (*RateFormat)(const AODataFormat_t *format);
```

Arguments

format The media format you want to rate.

Description

This function should return a rating for the given media *format*. Usually this function checks the media type and four character code (fourcc). If it has more stringent requirements, it checks the `AODataFormat_t`'s corresponding union members.

Returns

A rating from 0 to 100 of how well the addon can handle the given media format. 100 is best rating.

AOMimetypeInspector

An interface that allows your addon to return a rating as to how well it can process data with the given mimetype. It defines one function:

RateMimetype()

Synopsis

```
#include <aoi.h>

int32_t (*RateMimetype)(const char *mimetype);
```

Arguments

mimetype The mimetype you want to rate.

Description

This function should return a rating for the given mimetype.

Returns

A rating between 0 and 100 of how well your addon can handle a given mimetype. 100 is the best rating.

AOStreamer

An interface that defines all the functions necessary to implement a one-way byte stream. It defines these functions:

- *Open()*
- *Close()*
- *Sniff()*
- *Read()*
- *Write()*
- *Seek()*
- *Tell()*
- *Length()*
- *SideInfo()*

Open

Synopsis

```
#include <aoi.h>

AOIStream_t *(*Open)(const char *name,
                    const char *mode);
```

Arguments

name The name of the stream you want to open.

mode The mode you want to open the stream in. The mode string should match modes for *fopen()*, such as:

- rb** Read binary.
- wb** Write binary.

Description

This function should open the stream with the given *name* in the *mode*.

Returns

A pointer to an **AOIStream_t** instance, or NULL if the function can't open the given stream in the given mode.

Close

Synopsis

```
#include <aoi.h>

int32_t (*Close)(AOIStream_t *stream);
```

Arguments

stream The stream you want to close.

Description

This function should close the *stream* and free any data allocated at open.

Returns

0 if successful.

Sniff()

Synopsis

```
#include <aoi.h>

int64_t (*Sniff)(void *ctx,
                void *buf,
                int64_t num);
```

Arguments

ctx The context for the stream you want to read from.

buf The buffer into which you want to put read bytes.

num The number of bytes to be read from the beginning of the stream.

Description

This function should nondestructively read *num* bytes from the beginning of a stream. All streamers should implement this function. Once stream data is read with *Read()*, you can no longer use *Sniff()*.

Returns

The number of bytes successfully sniffed from the stream.

Read()

Synopsis

```
#include <aoi.h>

int64_t (*Read)(void *ctx,
                void *buf,
                int64_t num);
```

Arguments

ctx The context for the stream you want to read from.

buf The buffer into which you want to read data.

num The number of bytes to be read from the stream (the length of *buf*).

Description

This function should read *num* bytes from the stream at the stream's current file position.

Returns

The number of bytes successfully read from the stream.

Write()

Synopsis

```
#include <aoi.h>

int64_t (*Write)(void *ctx,
                 const void *buf,
                 int64_t num);
```

Arguments

ctx The context of the stream you want to write to.

buf The buffer containing the data you want to write to the stream.

num The number of bytes to be written to the stream (the length of *buf*).

Description

This function should write *num* bytes to the stream at the stream's current file position.

Returns

The number of bytes successfully written to the stream.

Seek()

Synopsis

```
#include <aoi.h>

int64_t (*Seek)(void *ctx,
                int64_t offset,
                int32_t whence);
```

Arguments

ctx The context for the stream you want to seek in.

offset The offset, in bytes, to which you want to seek.

whence The position from which to apply the offset; one of:

- SEEK_SET Compute the new file position relative to the start of the file. The value of *offset* must not be negative.
- SEEK_CUR Compute the new file position relative to the current file position. The value of *offset* may be positive, negative or zero.
- SEEK_END Compute the new file position relative to the end of the file.

Description

This function should seek to the given position in the stream.

Returns

The new stream position.

Tell()

Synopsis

```
#include <aoi.h>

int64_t (*Tell)(void *ctx);
```

Arguments

ctx The context for the stream you're querying.

Description

This function should return the current position in the stream.

Returns

The current position in the stream.

Length()

Synopsis

```
#include <aoi.h>

int64_t (*Length)(void *ctx);
```

Arguments

ctx The context for the stream.

Description

This function should return the length of the stream, in bytes, if known.

Returns

The length of the stream.

SideInfo()

Synopsis

```
#include <aoi.h>

int32_t (*SideInfo)(void *context,
                    char **sideinfo,
                    int32_t *length);
```

Arguments

context A pointer to the context for the stream you want to retrieve side information from.

sideinfo The address of a pointer to space where the function can store the side information for the stream.

length A pointer to a space where the function can store the returned length of the *sideinfo* parameter.

Description

This function should store the current side information for a stream in the space provided by *sideinfo*, and set the *sideinfo length*. Side information can change any time, and often does, as in the case for inline information in streaming audio.

Returns

0 if successful.

AOSTreamInspector

An interface that allows your addon to return a rating as to how well it can process a given stream. It defines one function:

RateStream()

Synopsis

```
#include <aoi.h>

int32_t (*RateStream)(AOIStream_t *stream);
```

Arguments

stream A pointer to the **AOIStream_t** structure for the stream you want to rate.

Description

This function should return a rating for the given stream. This function should only ever call the *Sniff()* function in the given stream's **AOSTreamer** interface.

Returns

A rating from 0 to 100 of how well the addon can handle the stream. 100 is the best rating.

AOResourceAccess

An interface that allows an application to access to your addon's resources. It defines these functions:

- *GetResources()*
- *SetResource()*

GetResources()

Synopsis

```
#include <aoi.h>

const AOResource_t *(*GetResources)(void *ctx);
```

Arguments

ctx The context for the control you want a list of resources for.

Description

This function should return all the resources of a DLL for the given context.



The returned resources should be read only.

Returns

An `AOResource_t` list of resources.

SetResource()

Synopsis

```
#include <aoi.h>

int32_t (*SetResource)(void *ctx,
                       const char *resource,
                       const void *data);
```

Arguments

ctx A pointer to the control that contains the resource you want to set.

resource The resource you want to set.

data A pointer to the data you want to set the *resource* to.

Description

This function should set the value in a specific resource. You should be sure of the type of values you can set.

Returns

0 if successful.

Built-in interfaces

There are three built-in interfaces in the Addon Interfaces Library: **Unloading**, **InitializeInterface**, and **Name**.

Unloading and InitializeInterface

These interfaces define functions that manage different hardware configurations when the application initializes and unloads.

InitializeInterface is an interface pointer that points to a `(void (*)(AOInterface_t *))` function. This function is called automatically to determine the actual value of any interface pointer whose initial value was NULL. This allows DLLs to create or choose certain interfaces at runtime instead of at compile time.

Unloading is an interface pointer that points to a `(void (*)(void))` function. If you call `AoRelease()`, and the count reaches zero, the unloading function is called before the DLL is unloaded. This gives controls that access hardware a chance to leave that hardware in a stable state before being unloaded.

Let's say we have an addon that supports two slightly different pieces of hardware. In this case, we want two different **HardwareControl** interfaces, and we want to use the one that the hardware the user has installed is for. In this case, we set the interface pointer for the **HardwareControl** interface to NULL, and create a **InitializeInterface** interface that returns the appropriate interface. Also, we want to do some cleanup on the hardware when its done, so we implement a **Unloading** interface as well:

```
static void *InitializeInterface(AOInterface_t *i)
{
    // we only initialize the "HardwareControl" interface
    if (strcmp(i->name, "HardwareControl") != 0) return 0;

    // return the HardwareControlA/BInterface if either type of
    // hardware is found.
    if (hardware_is_type_a)
        return HardwareControlAInterface;
    else
    if (hardware_is_type_b)
        return HardwareControlBInterface;

    // neither piece of hardware found? return 0
}
```

```
        return 0;
    }

    static void Unloading(void)
    {
        // release the hardware, whatever it is
        release_hardware();
    }

    AOInterface_t my_hardware_interface[] =
    {
        {"Name", 0, "my_hardware"},
        {"Description", 0, "Plugin for my hardware"},
        {"HardwareControl", 0, NULL},
        {"InitializeInterface", 0, InitializeInterface},
        {"Unloading", 0, Unloading},
        ... (other interfaces)
        {0, 0, 0},
    };
};
```

The first time an application requests the **HardwareControl** interface for the above addon, the Addon Interfaces Library sees that the interface pointer is 0, and calls the *InitializeInterface()* function with the **HardwareControl** **AOInterface_t** as its parameter. The *InitializeInterface()* function recognizes the **HardwareControl** interface, checks which hardware is available, and returns the appropriate interface pointer.

Later, when the DLL is unloading, or the application is exiting, the Addon Interfaces Library checks to see if the addon has an **Unloading** interface, and because it does, that function in the addon is called.

Name

An interface pointer that points to a string. You need to declare this interface to use *AoFindName()*.

Using Addon Resources

In this appendix...

An Example	51
Using addon resources in your application	54

This appendix contains information about using resources in your addon. A resource is any piece of data that you want to have access to. For example, in a multimedia interface, you may want to specify “volume” as a resource. You can then write functions to get and set the volume, using *GetResource()* and *SetResource()* as defined in the **AOResourceAccess** interface.

An Example

Here’s an example of working with a resource in an addon. Lets start with creating resources in your addon. One approach is to create a **const** structure containing your resources definitions, and then use that as a template to create a resources structure for each context, if your addon uses contexts. You would use contexts if you needed more than one instance of the same addon. Lets create a context structure first:

```
typedef struct my_context
{
    int32_t volume;          // volume, 0-100
    int64_t position;       // position, in microseconds
    AOResource_t *resources;
};
```

We’ll need typing info for our volume and position resources, such as minimum, maximum, and increment values. In an **AOResource_t** there is pre-defined typing info in the *type* element flags. When the *type* flag for an **AOResource_t** is **AOR_TYPE_LONG**, then *value* is an **int32_t** and the resource *info* is a pointer to **int32_t** *min*, *max*, and *step* values:

```
// volume range from 0 to 100, in increments of one
static const int32_t volumerange[] = {0,100,1};
```

When *type* is **AOR_TYPE_LONGLONG**, the resource *value* is an **int64_t** and *info* is a pointer to **int64_t** *min*, *max*, and *step* values:

```
// position range from 0 to 86400000000 (86400 seconds, or 24 hours.)
static const int64_t posrange[] = {0,86400000000,1};
```

The **AOResource_t** structure is defined as:

```
typedef struct
{
    char *name;
    char *description;
    void *value;
    void *info;
    int32_t type;
} AOResource_t;
```

For more information about the **AOResource_t** structure, see **AOResource_t**.

Now we can define our **const AOResource_t** resources structure:

```
static const AOResource_t resources[] =
{
  { "Volume", "Current Volume", (void*)offsetof(my_context, volume),
    &volumerange, AOR_TYPE_LONG | AOR_TYPE_READABLE | AOR_TYPE_WRITABLE },
  { "Position", "Current Position", (void*)offsetof(my_context, position),
    &posrange, AOR_TYPE_LONGLONG | AOR_TYPE_READABLE | AOR_TYPE_WRITABLE },
  { 0 }
};
```

As you can see, the pointer to the current value of the resource is not valid; its an offset into the context. When we create a new context, we'll have to adjust this pointer accordingly. Assuming we use the `AODestructor` interface, our create function might look like:

```
static void *Create(const AOICtrl_t *interfaces)
{
  my_context *ctx=(my_context*)calloc(1, sizeof(my_context));
  int32_t n;
  AOResource_t *res;

  // allocate new resource structure, and copy const version
  // into it:
  ctx->res=(AOResource_t*)malloc(sizeof(resources));
  memcpy(ctx->res, &resources, sizeof(resources));

  for (res=ctx->res; res->name; res++)
  {
    char *p=(char *)ctx;

    // Add the address of the context to the offset, making
    // the value pointer now point to the correct location
    // in our context:
    res->value=(void*)(&p[(int32_t)res->value]);
  }

  // initialize our context elements, if necessary
  ctx->volume=50;

  return ctx;
}

static void Destroy(void *p)
{
  my_context *ctx=(my_context*)p;

  free(ctx->res);
  free(ctx);
}

static AODestructor media_filter =
{
  Create,
  Destroy
}
```

```
};
```

If we want the outside world to be able to access our resources, we'll need to implement the `AOResourceAccess` interface. This is quite easy as well:

```
static const AOResource_t *GetResources(void *handle)
{
    my_context *ctx=(my_context*)handle;

    return handle->resources;
}

static int32_t SetResource(void *handle,const char *res,
                          const void *data)
{
    my_context *ctx=(my_context*)handle;

    // first resource is volume
    if (strcmp(res,ctx->resources[0].name)==0)
    {
        ctx->volume=*((int32_t*)data);

        // do any other volume control stuff here

        // return success
        return 0;
    }
    else

    // second resource is position
    if (strcmp(res,ctx->resources[1].name)==0)
    {
        ctx->position=*((int64_t*)data);

        // do any other positioning stuff here

        // return success
        return 0;
    }

    // no matching resource, return error
    return -1;
}

static AOResourceAccess resource_access =
{
    GetResources,
    SetResource,
};
```

At the end of our addon, we put them all together in our interfaces list:

```
#ifdef VARIANT_dll
```

```

AOInterface_t interfaces[] =
#else
AOInterface_t my_addon_interfaces[] =
#endif
{
  { "Name", 1, "my_addon" },
  { "AODestructor", AODECONSTRUCTOR_VERSION, &media_filter },
  { "AOResourceAccess", AORESOURCEREACCESS_VERSION, &resource_access },
  { 0, 0, 0 },
};

```

We use the `#ifdef` in order to build a shared (DLL) and static (library) version of our addon with the same source code. This way we can link directly with the static version if we want our application to be completely self contained, and use the DLL if not.

Using addon resources in your application

To use the our addon's resources in an application, we simply get the addon's `AOResourceAccess` interface using the `AOGetInterface()` function, call its `GetResources()` function, and iterate through the resources until we find one we want to look at. If we want to change one of the resources, and its `AOR_TYPE_WRITABLE` type flag is set, we call the interfaces `SetResource()` function. Here are examples for getting and setting the volume:

```

int32_t GetVolume(AOICtrl_t *ctrl, void *ctx)
{
  AOResource_t *res;
  AOInterface_t *i;

  // does it have resource access?
  if (i=AOGetInterface(ctrl, "AOResourceAccess", AORESOURCEREACCESS_VERSION, 0))
  {
    // does it have resources?
    if (res=i->GetResources(ctx))
    {
      // iterate through the resources
      for (;res->name;res++)
      {
        // is the current resource the volume?
        if (strcmp(res->name, "Volume")==0) return *((int32_t*)res->value);
      }
    }
  }
  return -1;
}

int32_t SetVolume(AOICtrl_t *ctrl, void *ctx, int32_t volume)
{
  AOInterface_t *i;

  // does it have resource access?
  if (i=AOGetInterface(ctrl, "AOResourceAccess", AORESOURCEREACCESS_VERSION, 0))

```

```
{
    // try to set its Volume resource.
    return i->SetResource(ctx, "Volume", &volume);
}
return -1;
}
```


Appendix C

Defined Structures

This chapter lists the structures that are defined in the Addon Interface Library headers:

- `AOImageFormat_t`
- `AOVideoFormat_t`
- `AOAudioFormat_t`
- `AODataFormat_t`
- `AOResource_t`
- `AOMimeInfo_t`
- `AOIStream_t`

Synopsis:

See below.

Description:

This structure defines a stream object, and is used to simplify **AOStreamer** usage. It contains at least the following members:

const char **filespec*

The name of the open file/stream.

const AOStreamer **streamer*

The actual **AOStreamer** being used to stream the data.

const AOICtrl_t **control*

A pointer to the **AOICtrl_t** structure for the streamer's control, if it is an interface from an add-on.

void **ctx*

The streamer context; the data it points to is specific to the streamer.

Classification:

QNX Neutrino

See also:

AOICtrl_t

Synopsis:

See below.

Description:

This structure defines mimetype information. The structure contains at least the following members:

char * <i>mimetype</i>	The mimetype (type and subtype) supported (e.g. image/jpeg).
char * <i>extensions</i>	A comma-separated list of file extensions (e.g. jpg, jpeg).
char * <i>description</i>	A description of the mimetype.

Classification:

QNX Neutrino

Synopsis:

```
typedef struct
{
    char *name;
    char *description;
    void *value;
    void *info;
    int32_t type;
} AOResource_t;
```

Description:

This structure defines an addon's resources. It contains at least the following members:

<code>char *name</code>	The name of the resource.
<code>char *description</code>	A short description of the resource.
<code>void *parent</code>	The parent control for the resource.
<code>void *value</code>	A pointer to the actual value of the resource.
<code>void *info</code>	A pointer to typing information (such as a range, list of items, etc.).
<code>int32_t type</code>	The resource type flags, which is one of: <ul style="list-style-type: none">• <code>AOR_TYPE_LONG</code> — a long integer; <i>value</i> points to an <code>int32_t</code>, and <i>info</i> points to an array of three <code>int32_t</code> numbers containing minimum, maximum, and increment values.• <code>AOR_TYPE_LONGLONG</code> — a long long integer; <i>value</i> points to an <code>int64_t</code>, and <i>info</i> points to an array of three <code>int64_t</code> numbers containing minimum, maximum, and increment values.• <code>AOR_TYPE_FLOAT</code> — a float; <i>value</i> points to a <code>float</code>, and <i>info</i> points to an array of three <code>float</code> numbers containing minimum, maximum, and increment values.• <code>AOR_TYPE_STRING</code> — a string; <i>value</i> points to an allocated string buffer, and <i>info</i> points to an <code>int32_t</code> that contains the maximum length of the string.• <code>AOR_TYPE_RADIO</code> — a radio button; <i>value</i> points to an <code>int32_t</code>, and <i>info</i> points to a structure containing an <code>int32_t</code> for the count value, followed by count <code>char*</code> pointers.• <code>AOR_TYPE_TOGGLE</code> — a toggle button; <i>value</i> points to an <code>int32_t</code>. There is no <i>info</i> pointer requirement.

- AOR_TYPE_POINTER — a pointer; *value* is the actual pointer.

You can OR the *type* member with one or more of the following permission values:

- AOR_TYPE_READABLE — readable using resource functions.
- AOR_TYPE_WRITABLE — writable using resource functions.
- AOR_TYPE_ENABLED — enabled.
- AOR_TYPE_VISIBLE — visible.

These values are used when automatically generating a GUI for a DLL's resources, for example.

Classification:

QNX Neutrino

See also:

`AOIStream_t`, `AOMimeInfo_t`

A structure that defines an audio format

Synopsis:

See below.

Description:

This structure defines an audio format. It contains at least the following members:

`uint32_t channels`

The number of audio channels. For example, a stereo signal has 2 channels.

`uint32_t depth`

The audio depth (sample rate) in bytes.

`int32_t frame_rate`

The frame rate (frequency), in frames per second. This value may be divided by *scale* to represent a floating-point frame rate.

`int32_t scale`

A scaling variable to convert *frame_rate* into an actual rate. For example: $2997/100=29.97$.

`int32_t duration`

The duration of the audio, in frames.

Classification:

QNX Neutrino

See also:

`AOImageFormat_t`, `AOVideoFormat_t`, `AODataFormat_t`

Synopsis:

See below.

Description:

The `ImageFormat_t` structure describes an image format. It contains at least the following members:

<code>uint32_t width</code>	The width of the image, in pixels.
<code>uint32_t height</code>	The height of the image, in pixels.
<code>uint16_t depth</code>	The color depth of the image, in bits.
<code>int16_t transparent</code>	If this image is transparent, this value is the transparency index + 1. If the image isn't transparent, this value is 0.
<code>uint8_t pal[256][3]</code>	The image palette.

Classification:

QNX Neutrino

See also:

`AudioFormat_t`, `VideoFormat_t`, `MediaFormat_t`

Synopsis:

See below.

Description:

This structure defines a generic media type, which can be audio, video, or image, and includes compression information. It contains the following members:

<code>uint32_t mtype</code>	<p>A flag indicating media type, which can be one of:</p> <ul style="list-style-type: none">• MEDIA_TYPE_IMAGE• MEDIA_TYPE_VIDEO• MEDIA_TYPE_AUDIO <p>These flags can be ORed with MEDIA_TYPE_COMPRESSED if the data is compressed.</p>
<code>uint32_t fourcc</code>	<p>A standard “four character code” that describes the media type. This is the standard FOURCC value used in avi and quicktime files. A number of additional values are defined:</p> <ul style="list-style-type: none">• RGB6 — 16 bit RGB• RGB5 — 15 bit RGB• RGB4 — 24 bit RGB• RGB2 — 32 bit RGB
<code>u</code>	<p>A straight union for the above media formats. The union contains members <i>image</i>, <i>audio</i>, and <i>video</i>, of type <code>ImageFormat_t</code>, <code>AudioFormat_t</code>, and <code>VideoFormat_t</code> respectively.</p>

Classification:

QNX Neutrino

See also:

`ImageFormat_t`, `VideoFormat_t`, `AudioFormat_t`

Synopsis:

See below.

Description:

This structure defines a video format. It contains at least the following members:

<code>uint32_t width</code>	The width of the video image, in pixels.
<code>uint32_t height</code>	The height of the video image, in pixels.
<code>uint32_t depth</code>	The color depth (number of bits per pixel).
<code>int32_t frame_rate</code>	The scaled frame rate. This value is divided by <i>scale</i> for the actual frame rate.
<code>int32_t scale</code>	A scaling value for the frame rate. This value is required if the frame rate isn't an integer. For example, if the frame rate is 29.97, set <i>frame_rate</i> to 2997 and <i>scale</i> to 100.
<code>int32_t duration</code>	The duration of the video, in frames. Set to 0 if unknown.

Classification:

QNX Neutrino

See also:

`ImageFormat_t`, `AudioFormat_t`, `MediaFormat_t`