QNX® Neutrino® Realtime Operating System

QNX Neutrino®
Multimedia Developer's Guide

For QNX® Neutrino® 6.3

Printed 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/`

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (`www.qnx.com`). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

Contents

List of Figures

About This Reference

The Multimedia *Developer's Guide* is intended for users who want to:

- use the library of multimedia interfaces and filters in their applications to work with multimedia data
- create their own multimedia filters.

☞ If you're familiar with earlier versions, you should read the "What's new" sections to find out how the multimedia API has changed in this release.

This table may help you find what you need in the *Multimedia Developer's Guide*:

When you want to:	**Go to:**
Read an overview of the Multimedia library	Multimedia Framework Overview
Learn about using the Multimedia library and QNX-provided components in your applications	Using the Multimedia library
See sample code of using the library	"Examples of using the Multimedia library" in Using the Multimedia Library
See a list of all *Mm*()* library functions your application can use	Multimedia Library
See more detailed information about the filters provided by QNX	Multimedia Filters reference
Learn about writing your own filters to extend the framework	Extending the Multimedia Framework
Find out more about interfaces you need to implement to write your own filters	Multimedia Interface Reference
Find out more about structures required to write your own filters	Multimedia library Structure Reference
Find out more about configuring MIDI with **`midi.cfg`**	The MIDI Configuration File

What's new in Photon for QNX Neutrino 6.3 Service Pack 2

The Multimedia Developer's Guide has been substantially reorganized and rewritten:

- The Using Graphs chapter is now Using the Multimedia library, and includes new information including an example of building an MPEG1 System graph.

- The Writing Filters chapter is now Extending the Multimedia library, and includes new information.
- Functions and structures are now separated onto their own reference pages.

What’s new in Photon for QNX Neutrino 6.3

The Multimedia Developer’s Guide has these changes:

- The Overview now lists additional supported formats, AIF, IFF, MIDI. There are two additional MPEG-1 parsers.
- The MIDI parser’s configuration file is described in a new appendix.

Chapter 1

Multimedia Framework Overview

In this chapter. . .

This chapter provides an overview of the Multimedia Framework, and covers:

- Multimedia architecture
- QNX-provided components
- Extending functionality

Multimedia architecture

The Multimedia library architecture is modular, meaning that it consists of units that can be interchanged with each other to provide only the functionality you need. In addition, because it uses the Addon Interfaces library to define its standard interfaces, you can write new components (such as codecs), and easily incorporate new components written by QNX or third parties as they become available, in some cases without having to recompile your application.

A filter is the basic building block of the Multimedia framework. Multimedia Filters:

- are compiled as DLLs or libraries. The DLL versions of the filters are located by default in **`lib/dll/mmedia`** (on the target). The library versions are used at compile-time if your application is linking against them statically.
- perform a specific task on multimedia data, such as reading a file from disk, parsing, decoding, or writing data to an audio device
- are *plugins*, and therefore implement a standard interface that the Multimedia library understands. The interfaces are defined in the Addon Interface library, **`aoi`**.

To process a multimedia file, your application uses only the filters it needs to get the multimedia data, process it, and send it to a target device. We classify filters based on their specific task. A filter can be a:

Reader	Reads data from a source. Examples are audio card readers, data stream readers, and video capture card readers.
Parser	Parses a stream of data into component parts. Examples are MPEG bitstream parsers, WAV format parsers, and AVI format parsers.
Decoder	Decodes compressed or encoded data. Examples are mpeg audio & video decoders, and divx decoders.
Encoder	Encodes raw data into a specific format. Examples are mpeg audio & video encoders.
Writer	Sends data to a destination. Examples are audio card writers, video output writers, and output file writers.

A filter is a generic object. Its functionality is determined by the way it processes data, and the interfaces it implements. A Multimedia library interface defines the set of methods that the library expects filters of a specific type to have. For example, all filters must implement the create and destroy methods defined in the **`AODeConstructor`** interface so that the library can create and destroy them, while only filters that read data from a source need to implement the **`MediaReader`** interface. The implementation of an interface is declared as a structure of pointers to implemented functions in a defined order.

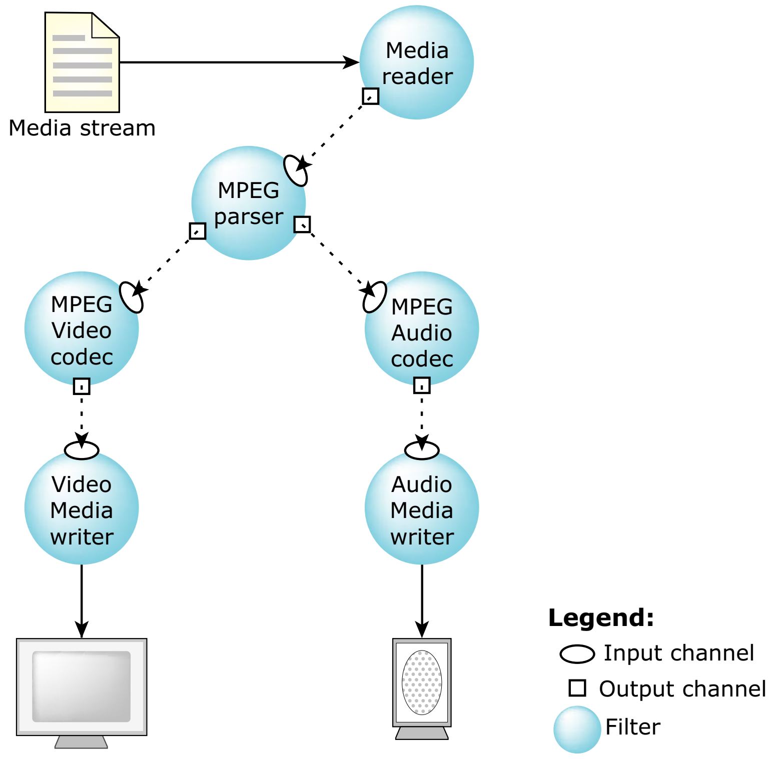
Each filter can have multiple input and output channels, so if you have a file format that's interleaved with more than one data stream, you can have as many output data channels as the input data stream contains. This is how the MPEG System parser works, for example — it parses MPEG System data into two streams, video and sound. An MPEG System encoder filter works the opposite way; it has separate input channels for sound and video, and one output channel.

We already provide a number of filters written for the Multimedia framework, but if you need a new one, you have to write only the decoder, parser, or other filter that hasn't already been written. Because the multimedia architecture uses a "plugin" framework, each filter component can easily be integrated (plugged in) with existing filters.

To use the Multimedia library, your application sets up a *graph*, an abstract object that encapsulates the multimedia filters needed to process a multimedia stream. A graph is stored as a **`MmGraph_t`** structure. It's built and destroyed at run time by the player application, and is specific to a particular media stream. Your application adds to the graph all the *filters* required to process multimedia data from source to destination.

If you know the specific filter required to process the multimedia stream, you can add it to the graph by name. However, a more flexible approach is to use functions that query the library and find the best filter for the data. In this way your application can handle any media type that the library's filters can handle, and use any new filters added to the library without being recompiled.

In the example below, six filters are used to read, parse, decode, and present an MPEG media stream. Each filter has an input channel and at least one output channel, except for special purpose "media reader" and "media writer" filters, which are at the beginning and end of the filter chain.



Multimedia graph example.

In the filters we provide, multimedia streams are synchronized by the audio stream, if present. In the MPEG example above, the writer filters don't communicate directly. Instead, they use the audio writer filter MediaClock interface to the Multimedia library (all filters have hooks to the library) to determine the correct media time. In this way, the video writer continuously checks its media time against the audio writer, and synchronizes its playback accordingly.

QNX components

The Multimedia library includes filters provided by QNX Software Systems that support the multimedia formats listed in the table below. You can write your own filters, or obtain filters from third parties to handle additional formats.

☞ Some filters are available only with the Multimedia Technology Development Kit.

Format	Filter(s)	Mimetype
MPEG-1 (audio)	`mpega_parser.so`, `xing_mpega_decoder.so`	`audio/mp1`, `audio/mp2`, `audio/mp3`, `audio/mpeg`

continued...

Format	Filter(s)	Mimetype
MPEG-1 (video)	`mpegv_parser.so`, `ff_mpegv_decoder.so`	`video/mpv`
MPEG-1 System (audio and video)	`mpegs_parser.so`, `ff_mpegv_decoder.so`	`video/mpeg`
AU (audio format)	`au_parser.so`	`audio/x-au`, `audio/x-basic`
IFF (audio format)	`iff_parser.so`	`audio/x-iff`
AIFF (audio format)	`aif_parser.so`	`audio/x-aif`
AVI (file format, parser filter only, no codec support)	`avi_parser.so`	`audio/x-avi`
OGG (open source audio format)	`ogg_decoder.so`, `oggi_decoder.so`	`audio/x-ogg`
WAV (audio format, most versions supported)	`wav_parser.so`	`audio/x-wav`
MIDI (audio format)	`midi_parser.so`*	`audio/midi`

* The MIDI filter requires a configuration file, **midi.cfg**, that defines the mapping of MIDI programs to instrument files. See the MIDI Configuration File appendix for more information.

For more detail about each of the filters included with the Multimedia library, see the Multimedia Filter Reference chapter.

The Multimedia library offers these benefits:

- In most cases it doesn't require hardware with an FPU to deliver acceptable performance (although hardware with an FPU may deliver better performance).
- Its modular design means you include only the media elements you need in your application, which reduces size and complexity.
- It allows you to easily write your own codecs to deal with new multimedia data formats, or to rewrite or replace existing codecs.
- It can be optimized for memory or CPU performance to suit a wide variety of hardware.
- It can handle both frame-based (such as MPEG-1 audio) and streaming (such as MPEG video) data streams.
- It's designed with a simple, intuitive API.

Extending functionality

You may want to extend the Multimedia Framework's functionality by writing filters for various media types not handled by the QNX-provided components. The chapter on Extending the Multimedia Framework covers this subject in detail, and shows examples of writing a parser and decoder.

☞ The source for the QNX-provided filters, as well as the source for the Addon Library and Multimedia library, are available in the Multimedia TDK. You require this TDK to write your own filters.

Chapter 2

Using the Multimedia Library

In this chapter. . .

This chapter describes how you can use the Multimedia library to create an application. To see a detailed description of any of the functions mentioned here, see the Multimedia library function reference.

To use the Multimedia library, an application has to do a couple of things:

- Initialize the library and create a graph. A *graph* is an abstract object that encapsulates the components needed to process a multimedia stream. A graph is stored as a **`MmGraph_t`** structure.
- Add the filters required to process the stream to the graph.
- Control the graph to provide the ability to start, stop, pause, and seek within the stream.

In addition:

- Your application may need to get or set resources for a graph, filter, or channel.
- You may wish to link your application statically if your target system has slow disk access or limited memory.
- The examples at the end of this chapter put all these steps together. The examples are:
  1. Create a graph, add a stream reader, and get its output channel.
  2. Find a filter for a specific output channel.
  3. Create a one stream audio playback graph.
  4. Play MP3s
  5. Play MPEG System 1

☞ Some of the code samples below are from the source code for **`mmplay`**, a sample application that's included with the Multimedia TDK.

Let's look at each step in more detail:

Initializing the library and creating a graph

Before you can create your graph and start working with it, you need to initialize the Multimedia library with a call to *MmInitialize()*. Typically you pass NULL to use the standard multimedia DLLs located in **`/lib/dll/mmedia`**. However, if you want to use a different set of filters, you can pass the path to those filters to *MmInitialize()*.

Next, you create a graph using *MmCreateGraph()*, passing it an arbitrary string to set the graph's ID. This returns a pointer to an opaque **`MmGraph_t`** structure that represents your multimedia graph. Make sure your application calls *MmDestroyGraph()* for each *MmCreateGraph()* call to clean up any resources associated with the graph.

Once you've created the graph and added filters to it, you need to set the sychronization clock for the Multimedia library by calling *MmSetDefaultClock()*.

Adding filters to the graph

Once you've created a graph, you need to add the filters that will process some multimedia data from the input stream to output. Generally you'll need at least a reader and writer filter, though you'll probably also require filters to parse and or decode the data as well. Filters are connected by *channels* — the output channel on one filter connects to the input channel on the next filter in the graph.

How you find the filters you need and attach their channels depends on how much you know about the input stream format, and how flexible you require your application to be.

In the case where your application needs to handle only a single data format, and you know all the filters required for end-to-end processing, you first find a reader filter for the file, and attach it to the graph with *MmFindMediaReader()*. Then:

1. Get the next filter by name and attach it to the graph with *MmFindFilter()*.
2. Get the output channel for the reader filter with *MmAcquireOutputChannel()* and the input channel for the next filter with *MmAcquireInputChannel()*.
3. Connect the two channels with *MmAttachChannels()*.
4. Repeat these steps until you've attached the input channel for the writer filter.

In the example of the MP3 player below, we know that the file format requires a file reader, Mpeg audio parser (**`mpega_parser.so`**), Xing Mpeg audio decoder (**`xing_mpega_decoder.so`**), and an audio writer.

In the case where your application should handle more than one media format (for example, an audio player that can play MP3 and WAV files), you build your graph a bit differently. You first find a reader filter for the file, and attach it to the graph with *MmFindMediaReader()*. Then:

1. Get an output channel from the filter with *MmAcquireOutputChannel()*.
2. Get the filter that has the best rating for the output channel with *MmFindChannelsFilter()*. This function finds the best-rated filter for an output channel, creates its input channel, attaches the two channels, and returns the filter.
3. Repeat these steps until you've connected a writer filter.

See Example 3 for an example of using this approach. Another example is the graph-creation function for **`mmplay`**, which can handle all the media formats supported by the Multimedia library.

☞ If you create your graph this way, and link your application dynamically, your application will be able to use any of the filters in the Multimedia library directory, which is **/lib/dll/mmedia** by default. If you link your application statically (see below), it will be able to use only the filters built into the application at compile time.

If your application acquires a channel but for some reason can't use it (and you want to continue building the graph), you should release it with *MmReleaseChannel()*, as illustrated in this example from **mmplay**:

```
if( (sc=MmAcquireOutputChannel(sf,MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_UNKNOWN)))
{
    MmFilter_t *tsf;
    if( !(tsf=MmFindChannelsFilter(graph,sc)) )
    {
        //printf("Couldn't find filter for compressed stream.\n");
        MmReleaseChannel(sc);
    }
    else
        sf=tsf;
}
```

Similarly, if you create a filter that the application won't use (and you want to continue to build the graph), you should destroy it with *MmDestroyFilter()*, as illustrated in this example from **mmplay**:

```
// if we failed to connect the audio output, destroy the
// audio_writer filter instance. (Otherwise it'll still be used.)
if( !mmplay.audioout )
{
    if( asc )
        MmReleaseChannel(asc);
    if( asf )
    MmDestroyFilter(asf);
    if( df )
        MmDestroyFilter(df);
}
```

Working with graphs

Once you've created the graph, added all the required filters, and connected them with channels, your application can start to control the multimedia playback. These are the Multimedia library functions that control graph playback:

- *MmStart()*
- *MmStop()*
- *MmPause()*
- *MmResume()*

- *MmSeek()*
- *MmStatus()*

Your application should call *MmStart()* once, after the graph is created and you want to start playback. Likewise, your application calls *MmStop()* once for a graph, before you call *MmDestroyGraph()* to destroy the graph and release its resources. *MmStart()* starts the graph in a paused state, so you need to call *MmResume()* to begin playback.

After the graph is playing, use *MmPause()*, *MmSeek()* and *MmResume()* to pause, change the playback time, and resume playback. For example, to "stop" playback, you could do something like this:

```
MmPause( graph );
MmSeek( graph, 0 );
```

Use *MmStatus()* to determine the status of a graph or filter. The most common use is to deterimine whether the graph has finished playing the input stream, in which case the status returned is MM_STATUS_EOF.

Getting information about a graph, filter, or channel

Use these functions to manipulate a graph's resources:

- *MmGetResourceValue()*
- *MmSetResourceValue()*

Every filter in a graph has a set of resources. When you call *MmGetResourceValue()* (or one of the convenience macros, such as *MmGetResourceINT32()*) on a filter, you get the value for that resource, if it exists. If you call these functions on a graph, the last filter in the graph is queried for the resource, and if it has the resource, the value is returned. If not, the next filter in the graph is queried.

Some resources are read-only, while others can be set by the application with *MmSetResourceValue()*.

QNX-provided filters have these resources:

Common:

- **`Duration`** — **`int64_t`** value containing the duration in nanoseconds. Read-only.
- **`Position`** — **`int64_t`** value containing the current position in nanoseconds. Read-only.

Audio writer:

- **`Volume`** — **`int32_t`** value containing the volume, from 0 to 100. Read / write.
- **`Balance`** — **`int32_t`** value containing the balance, from 0 to 100 (50 is "normal"). Read / write.

MPEG audio parser:

- **`IcyInfo`** — a string containing any icecast embedded information (streaming audio information). Read-only.
- **`ID3`** — an ID3 tag structure containing any ID3 embedded information. Read-only.

Video writer:

- **`Width`** — **`int32_t`** value containing the width of a video or image. Read-only.
- **`Height`** — **`int32_t`** value containing the height of a video or image. Read-only.
- **`DisplayArea`** — **`PhArea_t`** * value containing the current video widget position and dimension. Read / write.

CDDA reader:

- **`TracksCount`** — **`int32_t`** value containing the number of tracks a media stream contains. Read-only.
- **`Tracks`** — **`int32_t value`** containing the number of the current track. Read / write.
- **`CDDA_MILLISECONDS`** — **`int32_t value`** containing the amount of read-ahead buffer the CDDA reader is using. This value can be between 300 milliseconds and 1 minute. Read / write.
- **`Error`** — **`string`** value containing an error message. Read-only.

Miscellaneous functions

Use *MmPrintGraph()* to print a graph and all its filters to the display. This function is useful for debugging an application.

Linking an application statically

In general, dynamically linking an application has many benefits, including the ability to upgrade DLLs later without having to relink the application, and reduced memory footprint (provided that you unload the DLLs when you don't need them). In an embedded system where disk access is slow, however, you may find that statically linking an application against the Multimedia library is a better choice. This is because the library searches the DLL directory (by default **`/lib/dll/mmedia`**) and loads, queries, and unloads every DLL it finds. It does the same thing each time a new graph is constructed using *MmFindChannelsFilter()*. In addition, you avoid the additional memory overhead required for each dynamically loaded DLL, which can add up when you load several DLLs.

☞ If you want to link your application dynamically, you can also reduce its startup time by putting only the DLLs it requires into the Multimedia library directory.

If your application is linked statically, all the DLLs are loaded into memory, and the application never needs to read from the disk to query them after startup.

To link your multimedia application statically, you need to follow these steps:

1. In your makefile (`common.mk`), add these settings:

   ```
   LIBS += stream_reader   xing_mpega_decoder mpega_parser wav_parser\
   audio_writer fildes_streamer http_streamer  other_filters... \
   mmedia mmconvenienceS mmedia aoi  m  asound socket

   LIBPREF_mmedia= -Bstatic
   LIBPOST_mmedia= -Bdynamic
   ```

   This links the Multimedia filters statically.

2. Declare the external filters in your application source code. For example:

   ```
     extern AOInterface_t stream_reader_interfaces[];
     extern AOInterface_t ogg_decoder_interfaces[];
     extern AOInterface_t xing_mpega_decoder_interfaces[];
     extern AOInterface_t mpega_parser_interfaces[];
     extern AOInterface_t wav_parser_interfaces[];
     extern AOInterface_t audio_writer_interfaces[];
     extern AOInterface_t http_interfaces[];
     extern AOInterface_t fildes_interfaces[];
   ```

   Note that the names of the filters have a `_interfaces[]` added.

3. In your application's *main()* function, call *AoAddStatic()* for each imported filter to add its interfaces statically. For example:

   ```
       AoAddStatic(stream_reader_interfaces);
       AoAddStatic(mpega_parser_interfaces);
       AoAddStatic(wav_parser_interfaces);
       AoAddStatic(xing_mpega_decoder_interfaces);
       AoAddStatic(ogg_decoder_interfaces);
       AoAddStatic(audio_writer_interfaces);
       AoAddStatic(fildes_interfaces);
       AoAddStatic(http_interfaces);
   ```

The Multimedia Framework TDK ships with several sample applications. You can look at the `playaudio` example to see an application that is linked statically.

Examples of using the Multimedia library

This section contains code examples of using the Multimedia library.

Example 1: Initializing the Multimedia library, and creating a new multimedia graph

```
#include <aoi/aoi.h>
...
 // initialize the Multimedia library using default addon paths
MmInitialize(NULL);

 // create a new graph context
graph=MmCreateGraph("sample app");
...
 // Creating a filter for a given stream:
 // (assuming the multimedia has already been initialized, and a graph created.)

MmFilter_t *create_filter_for_stream(MmGraph_t *graph, AOIStream_t *stream)
{
        MmFilter_t *sf;
        MmFilter_t *df;
        MmChannel_t *sc;

        // Find the media reader to use with the stream
        // (usually turns out to be stream_reader)
        if (!(sf=MmFindMediaReader(graph, stream))) return 0;

        // Now grab a compressed output channel from the
        // stream_reader filter. (It always assumes it's
        // compressed, because it doesn't know anything
        // about the filter.)
        if (!(sc=MmAcquireOutputChannel(sf, MEDIA_TYPE_COMPRESSED)))
        {
                MmDestroyFilter(sf);
                return 0;
        }
        // Finally, find the best filter that can attach one
        // of its input channels to our output channel.
        if (!(df=MmFindChannelsFilter(graph, sc)))
        {
                MmReleaseChannel(sc);
                MmDestroyFilter(sf);
                return 0;
        }
        return df;
}
```

Example 2: Creating a filter for a given channel

This example assumes the Multimedia library is initialized, and a graph exists.

This task is easier than creating a filter for a given stream, since you skip the *MmFindMediaReader()* call. The simple approach is to call *MmFindChannelsFilter()* directly.

```
MmFilter_t *create_filter_for_channel(MmGraph_t *graph,
                                                MmChannel_t *channel)
{
        MmFilter_t *df;

        // Find the best filter that can attach one of its
        // input channels to our output channel.

        if (!(df=MmFindChannelsFilter(graph,channel)))
           return 0;

        return df;
}
```

Example 3: Creating a one-stream audio playback graph

In this example, the input is some type of compressed audio file, and the output is an audio device.

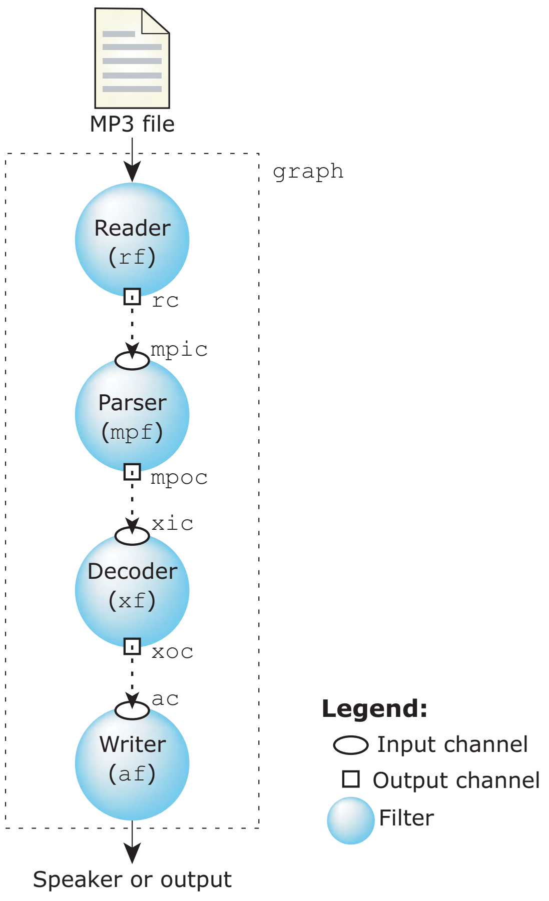
Start with a stream reader, and go from most complex (unknown compressed) to less complex (audio compressed), to least complex (raw audio). You put each complexity level in a loop to catch any multilevel complexity decoding.

```
MmGraph_t *create_graph_for_compressed_audio_stream(AOIStream_t *stream)
{
        MmGraph_t *graph;
        MmFilter_t *nf;
        MmChannel_t *sc;

        if (!(graph=MmCreateGraph("Audio Player")))
                return 0;

        if (!(sf=MmFindMediaReader(graph,stream)))
        {
                MmDestroyGraph(graph);
                return 0;
        }

        // connect compressed unknown channels
        while (sc=MmAcquireOutputChannel(sf,MEDIA_TYPE_COMPRESSED))
        {
                MmFilter_t *nf;

                // if we cannot find a filter for the channel,
                // release it, and break out of the loop.
                if (!(nf=MmFindChannelsFilter(graph,sc)))
                {
                        MmReleaseChannel(sc);
                        break;
                }
                sf=nf;
        }

        // connect compressed audio channels
        while (sc=MmAcquireOutputChannel(sf, MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_AUDIO))
        {
                MmFilter_t *nf;

                if (!(nf=MmFindChannelsFilter(graph,sc)))
                {
                        MmReleaseChannel(sc);
                        break;
                }
                sf=nf;
        }

        // connect uncompressed audio channels
        while (sc=MmAcquireOutputChannel(sf,MEDIA_TYPE_AUDIO))
        {
                MmFilter_t *nf;

                if (!(nf=MmFindChannelsFilter(graph,sc)))
                {
                        MmReleaseChannel(sc);
                        break;
                }
                sf=nf;
        }

        // return the graph
        return graph;
}
```

Example 4: Playing MP3s

This example shows the complete code for running an MP3.

The graph created in this example looks like this:



MP3 Graph.

```
#include <stdio.h>
#include <mmedia/mmedia.h>

int main(int argc,char *argv[])
{
    AOIStream_t *file;
    MmGraph_t *graph;
    MmFilter_t *rf,*mpf,*xf,*af;
    MmChannel_t *rc,*mpic,*mpoc,*xic,*xoc,*ac;

    // make sure we have one argument
    if (argc!=2)
    {
        printf("Usage: playmp3 <mp3 file>\n");
        exit(-1);
    }

    // initialize the Multimedia library
    MmInitialize(NULL);

    // open the streamer
    if (!(file=AoOpenFilespec(argv[1],"rb")))
    {
        printf("Unable to open '%s'.\n",argv[1]);
        exit(-2);
    }

    graph=MmCreateGraph("mp3 player");

    // we should always be able to find the MediaReader filter
    if (!(rf=MmFindMediaReader(graph,file)))
    {
        printf("Couldn't find the MediaReader filter.\n");
        MmDestroyGraph(graph);
        exit(-3);
    }

    // grab the compressed output channel
    if (!(rc=MmAcquireOutputChannel(rf,MEDIA_TYPE_COMPRESSED)))
    {
        printf("Couldn't get a compressed output channel from MediaReader.\n");
        MmDestroyGraph(graph);
        exit(-4);
    }

    // grab the mpeg audio parser
    if (!(mpf=MmFindFilter(graph,"mpega_parser")))
    {
        printf("Couldn't grab the mpeg audio parser.\n");
        MmDestroyGraph(graph);
        exit(-5);
    }

    // grab the mpeg audio parsers input channel
    if (!(mpic=MmAcquireInputChannel(mpf,MEDIA_TYPE_COMPRESSED)))
    {
        printf("Couldn't get a compressed input channel from mpega_parser.\n");
        MmDestroyGraph(graph);
        exit(-4);
    }

    // connect the two channels
    if (MmAttachChannels(rc,mpic)!=0)
    {
        printf("Couldn't attach MediaReader and mpega_parser channels.\n");
        MmDestroyGraph(graph);
        exit(-5);
    }

    // grab the mpega_parser filter output channel
    if (!(mpoc=MmAcquireOutputChannel(mpf,MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_AUDIO)))
    {
        printf("Couldn't get a compressed audio output channel from mpega_parser.\n");
        MmDestroyGraph(graph);
        exit(-6);
    }

    // grab the mpeg audio decoder
    if (!(xf=MmFindFilter(graph,"xing_mpega_decoder")))
    {
        printf("Couldn't grab the xing mpeg audio decoder.\n");
        MmDestroyGraph(graph);
        exit(-7);
    }

    // grab the mpeg audio decoder input channel
    if (!(xic=MmAcquireInputChannel(xf,MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_AUDIO)))
    {
        printf("Couldn't get a compressed input channel from xing_mpega_decoder.\n");
        MmDestroyGraph(graph);
        exit(-8);
    }

    // connect the two channels
    if (MmAttachChannels(mpoc,xic)!=0)
    {
        printf("Couldn't attach mpega_parser and xing_mpega_decoder channels.\n");
        MmDestroyGraph(graph);
        exit(-9);
    }

    // grab the xing_mpega_decoder filter output channel
    if (!(xoc=MmAcquireOutputChannel(xf,MEDIA_TYPE_AUDIO)))
    {
        printf("Couldn't get a audio output channel from xing_mpega_decoder.\n");
        MmDestroyGraph(graph);
        exit(-10);
    }

    // grab the audio_writer filter
    if (!(af=MmFindFilter(graph,"audio_writer")))
    {
        printf("Couldn't grab the audio_writer filter.\n");
        MmDestroyGraph(graph);
        exit(-11);
    }

    // grab the audio input channel
    if (!(ac=MmAcquireInputChannel(af,MEDIA_TYPE_AUDIO)))
    {
        printf("Couldn't get an audio input channel from audio_writer.\n");
        MmDestroyGraph(graph);
        exit(-12);
    }

    // connect the two channels
    if (MmAttachChannels(xoc,ac)!=0)
    {
        printf("Couldn't attach xing_mpega_decoder and audio_writer channels.\n");
        MmDestroyGraph(graph);
        exit(-13);
    }

    // set our default clock
    MmSetDefaultClock(graph);

    // start the graph playing
    MmStart(graph,0);
    MmResume(graph);

    // wait for the graph to finish playing
    while (MmStatus(graph)==MM_STATUS_PLAYING) delay(500);

    // stop destroy the graph
    MmStop(graph);
    MmDestroyGraph(graph);

    // close the input streamer
    file->streamer->Close(file);
```

```
    return 0;
}
```

Example 5: Playing MPEG-1 System

This example builds a graph specific to an MPEG-1 System stream, and plays the video on a Photon window. It adds a callback to the window to handle any movement or resize events.

```
#include <stdlib.h>
#include <stdio.h>
#include <mmedia/mmedia.h>
#include <Pt.h>

#define EVENT_SIZE (sizeof(PhEvent_t)+1000)

int OnWindowEvent(PtWidget_t *window, void *data, PtCallbackInfo_t *cbinfo) {

    // When the window moves, the window_writer filter needs
    // to be updated - otherwise the video won't resize or
    // move.

    PhArea_t rarea;
    short winx,winy;
    MmFilter_t *win_writer = (MmFilter_t*)data;
    PhWindowEvent_t *wev=(PhWindowEvent_t*)cbinfo->cbdata;
    PhDim_t *odim;

    PtGetResource(window,Pt_ARG_DIM,&odim,0);
    PtGetResource(window,Pt_ARG_AREA,&rarea,0);
    PtGetAbsPosition(window,&winx,&winy);

    rarea.pos.x=winx;
    rarea.pos.y=winy;
    rarea.size.w=odim->w;
    rarea.size.h=odim->h;
    MmSetResourceValue(win_writer,"DisplayArea", &rarea);


  return( Pt_CONTINUE );
}

int main(int argc,char *argv[])
{
    AOIStream_t *file;
    MmGraph_t *graph;
    MmFilter_t *media_reader,*mpegs_parser,*mpv_decoder,*win_writer, *mpa_decoder,
      *audio_writer;
    MmChannel_t *mr_out,*mps_in,*mpsv_out,*mpv_in,*mpv_out,*win_in, *mpsa_out,
      *mpa_in, *mpa_out, *audio_in;
    int32_t width, height;
    PhEvent_t *event;


   PtWidget_t *window;
   PtArg_t    args[3];

    // make sure we have one argument
    if (argc!=2)
    {
        printf("Usage: mpeg_play <mpeg file>\n");
        exit(-1);
    }

    // initialize the Multimedia library
    MmInitialize(NULL);

    // open the streamer
```

```
if (!(file=AoOpenFilespec(argv[1],"rb")))
{
    printf("Unable to open '%s'.\n",argv[1]);
    exit(-2);
}

graph=MmCreateGraph("mpeg player");

// we should always be able to find the MediaReader filter
if (!(media_reader=MmFindMediaReader(graph,file)))
{
    printf("Couldn't find the MediaReader filter.\n");
    MmDestroyGraph(graph);
    exit(-3);
}

// Get the MPEG system parser
if (!(mpegs_parser=MmFindFilter(graph,"mpegs_parser")))
{
    printf("Couldn't get the mpeg system parser.\n");
    MmDestroyGraph(graph);
    exit(-4);
}

// Get the reader output channel
if (!(mr_out=MmAcquireOutputChannel(media_reader,MEDIA_TYPE_COMPRESSED)))
{
    printf("Couldn't get an  output channel from MediaReader.\n");
    MmDestroyGraph(graph);
    exit(-5);
}

    // Get the mpeg system parser's input channel
if (!(mps_in=MmAcquireInputChannel(mpegs_parser,MEDIA_TYPE_UNKNOWN)))
{
    printf("Couldn't get a compressed input channel from mpega_parser.\n");
    MmDestroyGraph(graph);
    exit(-4);
}

  // connect the two channels
if (MmAttachChannels(mr_out,mps_in)!=0)
{
    printf("Couldn't attach MediaReader and mpegs_parser channels.\n");
    MmDestroyGraph(graph);
    exit(-5);
}

    // get the MPEG video decoder
if (!(mpv_decoder=MmFindFilter(graph,"ff_mpegv_decoder")))
{
    printf("Couldn't grab the mpeg video decoder.\n");
    MmDestroyGraph(graph);
    exit(-6);
}

    // Get the parser output channel
if (!(mpsv_out=MmAcquireOutputChannel(mpegs_parser,MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_VIDEO)))
{
    printf("Couldn't get a compressed output channel from MPEG system parser.\n");
    if (!(mpsv_out=MmAcquireOutputChannel(mpegs_parser,MEDIA_TYPE_VIDEO)))
    {
        printf("Couldn't get an uncompressed output channel from MPEG system parser either.\n");
        MmDestroyGraph(graph);
        exit(-7);
    }
}

    // Get the mpeg video decoder's  input channel
if (!(mpv_in=MmAcquireInputChannel(mpv_decoder,MEDIA_TYPE_VIDEO)))
{
    printf("Couldn't get a input channel from mpegv_decoder.\n");
    MmDestroyGraph(graph);
    exit(-8);
}
```

```
  // connect the two channels
if (MmAttachChannels(mpsv_out,mpv_in)!=0)
{
    printf("Couldn't attach MediaReader and mpega_parser channels.\n");
    MmDestroyGraph(graph);
    exit(-9);
}


// find the video writer filter and channel, and attach the two channels
if (PtInit("/dev/photon")!=0){
    printf("Error: couldn't connect to photon\n");
    PtExit(EXIT_FAILURE);
}
if( !(win_writer=MmFindFilter(graph,"window_writer")) )
{
    printf("Couldn't find video_writer.\n");
    MmDestroyGraph(graph);
    exit(-10);

}
// Create a window for the win_writer
// we just use an arbitrary size, but you could
// query the video for the right size to use

PtSetArg(&args[0], Pt_ARG_HEIGHT, 200, 0);
PtSetArg(&args[1], Pt_ARG_WIDTH, 300, 0);
PtSetArg(&args[2], Pt_ARG_WINDOW_NOTIFY_FLAGS, Ph_WM_RESIZE|Ph_WM_MOVE,
    Ph_WM_RESIZE|Ph_WM_MOVE);
window = PtCreateWidget(PtWindow, Pt_NO_PARENT, 3, args);

// this callback handles resize and move:
PtAddCallback(window, Pt_CB_WINDOW, OnWindowEvent, win_writer);
PtRealizeWidget(window);


MmSetResourceValue(win_writer, "PtWidget_t", window);

if( !(win_in=MmAcquireInputChannel(win_writer,MEDIA_TYPE_VIDEO)) )
{
    printf("Couldn't get video_writer's input channel.\n");
    MmDestroyGraph(graph);
    exit(-10);
}

if( !(mpv_out=MmAcquireOutputChannel(mpv_decoder,MEDIA_TYPE_VIDEO)) )
{
    printf("Couldn't get video_decoder's output channel.\n");
    MmDestroyGraph(graph);
    exit(-10);
}

if( MmAttachChannels(mpv_out,win_in) != 0 )
{
    printf("Couldn't attach to video_writer's input channel.\n");
    MmDestroyGraph(graph);
    exit(-10);
}

// Get the mpeg system parser audio output channel
if (!(mpsa_out=MmAcquireOutputChannel(mpegs_parser,MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_AUDIO)))
{
    printf("Couldn't get an audio output channel from MPEG system parser.\n");
    MmDestroyGraph(graph);
    exit(-7);
}

    // grab the mpeg audio decoder
if (!(mpa_decoder=MmFindFilter(graph,"xing_mpega_decoder")))
{
    printf("Couldn't grab the xing mpeg audio decoder.\n");
    MmDestroyGraph(graph);
    exit(-7);
}
```

```
// grab the mpeg audio decoder input channel
if (!(mpa_in=MmAcquireInputChannel(mpa_decoder,MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_AUDIO)))
{
    printf("Couldn't get a compressed input channel from xing_mpega_decoder.\n");
    MmDestroyGraph(graph);
    exit(-8);
}

// connect the two channels
if (MmAttachChannels(mpsa_out,mpa_in)!=0)
{
    printf("Couldn't attach system parser and mpega_parser channels.\n");
    MmDestroyGraph(graph);
    exit(-5);
}

// grab the xing_mpega_decoder filter output channel
if (!(mpa_out=MmAcquireOutputChannel(mpa_decoder,MEDIA_TYPE_AUDIO)))
{
    printf("Couldn't get a audio output channel from xing_mpega_decoder.\n");
    MmDestroyGraph(graph);
    exit(-10);
}

// grab the audio_writer filter
if (!(audio_writer=MmFindFilter(graph,"audio_writer")))
{
    printf("Couldn't grab the audio_writer filter.\n");
    MmDestroyGraph(graph);
    exit(-11);
}

// grab the audio input channel
if (!(audio_in=MmAcquireInputChannel(audio_writer,MEDIA_TYPE_AUDIO)))
{
    printf("Couldn't get an audio input channel from audio_writer.\n");
    MmDestroyGraph(graph);
    exit(-12);
}

// connect the two channels
if (MmAttachChannels(mpa_out,audio_in)!=0)
{
    printf("Couldn't attach xing_mpega_decoder and audio_writer channels.\n");
    MmDestroyGraph(graph);
    exit(-13);
}

// set our default clock
MmSetDefaultClock(graph);

// print graph, starting at reader
MmPrintGraph(media_reader,0);

// start the graph playing
MmStart(graph,0);
MmResume(graph);

// We're not using PtMainLoop(), because then we wouldn't know
// when to stop the app and destroy the graph
// Instead we do our own loop:

event = malloc(EVENT_SIZE);

while (MmStatus(graph)==MM_STATUS_PLAYING) {

    switch (PhEventPeek(event, EVENT_SIZE)){
        case Ph_EVENT_MSG:
            PtEventHandler(event);
            break;
        case 0:
            // give the video filter a chance to render:
            PtLeave(Pt_EVENT_PROCESS_PREVENT);
            delay(500);
```

```
                PtEnter(Pt_EVENT_PROCESS_PREVENT);
                PgFlush();
                break;

            case -1:
                printf("Ack! error.\n");
                exit(0);
                break;

        }
    }

    free(event);

    // stop and destroy the graph
    MmStop(graph);
    printf("Done playing, destroying the graph.\n");
    MmDestroyGraph(graph);

    // close the input streamer
    file->streamer->Close(file);

    return (EXIT_SUCCESS);

}
```

Chapter 3

Extending the Multimedia Framework

This chapter contains information about writing your own multimedia filters.

A filter is the basic building block of an application that uses the Multimedia library. There are many existing filters that handle a wide range of multimedia data formats. However, you may need to handle a new format, or want to use some hardware to process the data. In these cases, you'll need to write a new multimedia filter, or modify an existing one.

The QNX Multimedia library comes with many fully functional filters, but you may wish to write your own. This chapter shows you how to write filters, with two examples of filters for an application that handles MPEG data. Typically you won't need to create reader or writer filters, since the QNX-provided filters handle most input and output scenarios. It's more likely that you'll want to write a parser or decoder for formats that the standard filters don't handle.

The sample filters in this chapter are a parser for MPEG data, and a decoder for MPEG video data. Of course your application would also require (at least) a reader filter to read data from a stream (for example, an MPEG file), and a writer filter that writes the video data to an external device. You'd probably also want an MPEG audio decoder and audio writer.

How do I write a multimedia filter?

Let's look at the generic steps you need to take to write either a codec or parser filter, and compare the differences between them.

Whether you're writing a parser or codec filter, you need to implement some or all of the methods defined in these interfaces:

In this interface:	Implement the methods for:
`AODeConstructor`	Creating and destroying the filter
`MediaInput`	Managing input channels
`MediaOutput`	Managing output channels
`MediaControl`	Controlling processing
`MediaSeeker`	Seeking to a location in the media stream
`AOResourceAccess`	Exposing filter resources

In addition, the library needs a way to query the filters on how well they can handle a media stream. The parser filter provides this functionality by implementing the `AOStreamInspector` interface, which rates raw data, while the codec filter implements the `AOFormatInsector` interface, which rates parsed data.

The filters in this example use the default `MediaBufferAllocator` provided by the Multimedia library to create buffers. If you want to handle your own buffer allocation and management (for example, if you have some specific hardware requirements), you need to implement this interface as well.

Let's walk through the implementation of an MPEG system parser filter and codec filter and see how all of the pieces fit together. An MPEG has at least two channels, video and audio. This parser will parse the MPEG data stream into the audio and video components, and pass them on to decoder filters that handle MPEG audio and MPEG video. The decoder filter in this example decodes the parsed MPEG video data.

An *interface* is declared as a static array of function pointers. You'll notice in some of the examples below that some filters that don't implement every function. In these cases, they use pointers to convenience functions in the Multimedia convenience library.

Creating and destroying

The **AODeConstructor** interface defines the methods that create and destroy a filter. Both the parser and decoder filters need to implement these methods:

```
static AODeConstructor media_filter =
 {
   Create,
   Destroy
 };
```

The *Create()* method should:

- allocate our **MmFilter_t** filter data structure
- allocate our **MmFilterUser_t** user private data structure, and assign it to **MmFilter_t->***user*. This structure is declared as type **media_filter_user**, and can contain variables specific to the filter. It is available to any of the methods you implement.
- return the **MmFilter_t** filter on success, `0` if an error occurs

The *Destroy()* method should:

- free all resources allocated in *Create()*
- return `0`

Let's have a closer look at how we could implement these methods for the decoder.

```
struct media_filter_user
{
    // input channel variables
    const MediaOutput *mo;
    const MediaBufferAllocator *mba;
    MmChannel_t *mbac;

    // output channel variables
    MmFormat_t format;

    // codec variables and
    // other variables
...
};

...
```

```
static void *Create(const AOICtrl_t *interfaces)
{
  MmFilter_t *f;

  // allocate our MmFilter_t filter data structure
  if( !(f = (MmFilter_t*) calloc(1,sizeof(MmFilter_t))))
    return 0;

  // initialize and setup out identification string
  if( !(f->element.ID = strdup("MPEGVIDEO_Decoder")))
    return (MmFilter_t*) Destroy(f);

  // flag the structure as a filter
  f->element.type = MM_ELEMENT_FILTER;

  // allocate our MmFilterUser_t user private data structure
  if( !(f->user = (MmFilterUser_t*) calloc(1,sizeof(MmFilterUser_t))))
    return (MmFilter_t*) Destroy(f);

  // allocate and setup our input channel
  if( !(f->ichannels=(MmChannel_t*)calloc(1,sizeof(MmChannel_t))))
    return (MmFilter_t*) Destroy(f);
  if( !(f->ichannels[0].element.ID = strdup("MPEGVIn"))
    return (MmFilter_t*) Destroy(f);
  f->ichannels[0].element.type    = MM_ELEMENT_CHANNEL;
  f->ichannels[0].filter          = f;
  f->ichannels[0].direction       = MM_CHANNEL_INPUT;
  f->ichannels[0].format.mf.mtype = MEDIA_TYPE_COMPRESSED|
  MEDIA_TYPE_VIDEO;

  // allocate and setup our  output channel
  if( !(f->ochannels = (MmChannel_t*) calloc(1,sizeof(MmChannel_t))))
    return (MmFilter_t*) Destroy(f);
  if( !(f->ochannels[0].element.ID = strdup("RawVideoOut")))
    return (MmFilter_t*) Destroy(f);
  f->ochannels[0].element.type    = MM_ELEMENT_CHANNEL;
  f->ochannels[0].filter          = f;
  f->ochannels[0].direction       = MM_CHANNEL_OUTPUT;
  f->ochannels[0].format.mf.mtype = MEDIA_TYPE_VIDEO;

  /*
  ... mutex, condvar, etc.. initialization
  */

  //success  return the newly created filter
  return f;

}
```

This function frees the resources allocated in the *Create()* function.

```
static int32_t Destroy(void *obj)
{
  MmFilter_t *f = (MmFilter_t*) obj;
  // make sure we have a valid pointer
  if(!f)
    return -1;
  // free our filter private data structure
  if( f->user )
  free(f->user);
  //free our input channel
  if( f->ichannels )
  {
    if( f->ichannels[0].element.ID )
    free(f->ichannels[0].element.ID);
    free(f->ichannels);
  }
  // free our output channel
  if( f->ochannels )
  {
    if( f->ochannels[0].element.ID )
      free(f->ochannels[0].element.ID);
    free(f->ochannels);
  }
  //free our filter
  if( f->element.ID )
  free(f->element.ID);
  free(f);
  // return success
  return 0;
}
```

Inspecting the stream

The Multimedia library needs to be able to query both the parser and decoder for their ability to process some data. The two types of filter implement different interfaces to accomplish this task:

- The parser implements the **`AOStreamInspector`**, which rates raw, unparsed data (an **`AOIStream_t`**). You should always use the *sobj->streamer->Sniff()* method to nondestructively sniff the start of the stream.
- The decoder filter implements **`AOFormatInspector`**, which rates data that is already parsed (and described by an **`AODataFormat_t`**).

Both methods return a rating from `0` to `100` (where `100` is the best) of how well the data can be handled. Filters already implemented in the library generally return `80` when they can process the data.

☞ We use the *MmFOURCC()* macro in the decoder example to check whether the format four character code matches MPEG 1 video. This macro takes the four characters that make up a fourcc code, and returns an `int` representation of the fourcc.

Parser

```
static AOStreamInspector stream_inspector =
{
  SniffData
};



static int32_t SniffData(AOIStream_t *sobj)
{
  // Sniff the data  and return our rating for the stream
  // (0 to 100).
}
```

Decoder

```
static AOFormatInspector format_inspector =
{
  RateFormat,
};

static int32_t RateFormat(const AODataFormat_t *fmt)
{
  /*
     In the case of the mpegvideo decoder filter, we check
     that we have a matching format fourcc and format type
   */
  if( fmt->fourcc == MmFOURCC('M','P','1','V') &&
      fmt->mtype == (MEDIA_TYPE_COMPRESSED|MEDIA_TYPE_VIDEO) )
      return 95;
  return 0;
}
```

Managing input channels

The **MediaInput** interface defines methods that the Multimedia library uses to query, reserve, and release a filter's input channels. These methods contain all the functionality the library needs to connect our input channel to the output channel of the previous filter in the graph.

```
static MediaInput media_input =
{
  IterateChannels,
  AcquireChannel,
  ReleaseChannel,
  RateFormat,
  SetFormat,
  SetMediaOutput,
  SetInputStream
};
```

Both the parser and decoder filters can use convenience functions supplied by the Multimedia library for *IterateChannels()*, *AcquireChannel()*, and *ReleaseChannel()*. Since the parser accepts raw data, it doesn't need to rate or set its input format. On the other hand, the decoder takes parsed data, and therefore must do both. Parsed data must have a format (a **`MmFormat_t`**) associated with it, and the decoder sets its input format so the library can tell what sort of data it can handle.

The parser's input channel is connected to streaming data, so it implements *SetInputStream()*. The decoder's input channel is connected to buffered data, so it implements *SetMediaOutput()*. This is what these methods should do:

IterateChannels
: This method gives the Multimedia library access to all our input channels. Since both the parser and decoder have a single input channel, we can use the *singleIterateInputChannels()* convenience method from the library. It returns the input channel on the first call, and NULL thereafter.

AcquireChannel
: This method flags our input channels as being in use. Since both the parser and decoder have a single input channel, we can use the *singleAcquireInputChannel()* convenience method from the library.

ReleaseChannel
: This method flags our input channels as being released. Since both the parser and decoder have a single input channel, we can use the *singleReleaseInputChannel()* convenience method from the library.

RateFormat, SetFormat
: These methods apply to filters that get their input from the buffered output of another filter. *RateFormat()* inspects the format of the input channel, and returns a rating of how well the filter can handle it. *SetFormat()* negotiates the format.

  Since our MPEG system parser has direct access to an input stream, it doesn't need to implement these methods. We can use the *noRateInputFormat()* and *noSetInputFormat()* convenience methods from the library.

  The decoder does connect to the buffered output of the parser, so it needs to implement these methods.

SetMediaOutput, SetInputStream
: These methods connect the filter's input channels to the output channels of the previous filter in the chain. *SetMediaOutput()* applies to filters that connect to a buffered output, and *SetInputStream()* applies to filters that connect to streaming output. Our parser will implement *SetInputStream()* and use the *noSetMediaOutput()* convenience method. Our decoder will

implement *SetMediaOutput()*, and use the *noSetInputStream()* convenience method.

Parser

```
static MediaInput media_input =
{
  singleIterateInputChannels,
  singleAcquireInputChannel,
  singleReleaseInputChannel,
  noRateInputFormat,
  noSetInputFormat,
  noSetMediaOutput,
  SetInputStream
};
```

Let's take a closer look at how we would implement each of these functions:

```
// From the mmconvienience library:
  MmChannel_t *singleIterateInputChannels(const MmFilter_t *f,int32_t
* const cookie);

// From the mmconvienience library:
int32_t singleAcquireInputChannel(MmChannel_t *c);

// From the mmconvienience library:
int32_t singleReleaseInputChannel(MmChannel_t *c);

// From the mmconvienience library:
int32_t noRateInputFormat(MmChannel_t *c,MmFormat_t *f,int32_t *
const cookie);

// From the mmconvienience library:
int32_t noSetInputFormat(MmChannel_t *c,const MmFormat_t *f);

// From the mmconvienience library:
int32_t noSetMediaOutput(MmChannel_t *c,const MediaOutput *m);

/*
    This method is where a filter that connect its input
    to a stream save its streamer object , and decode
    enough of the input stream to figure out what the
    output channels are.
 */
static int32_t SetInputStream(MmChannel_t *c,AOIStream_t *sobj)
{
 /*
   In the case of the mpegsystem parser filter:
    - Find out how many audio and video streams
      are inside the system stream. We'll just
      sniff the mpeg system stream and extract the
      systemheader data, using the streamer Sniff()
      method.
    - Allocate and setup our output channels (audio/video).
    - Set up our output formats (audio/video, fourcc,
      buffer size, number of buffers).
    - Flag the provided channel as in use (set the
      MM_CHANNEL_INPUTSET bit in the channel's flags).
    - Return 0 on success, -1 on error.
  */
}
```

Decoder

```
static MediaInput media_input =
{
  singleIterateInputChannels,
  singleAcquireInputChannel,
  singleReleaseInputChannel,
  RateInputFormat,
  SetInputFormat,
  SetMediaOutput,
  noSetInputStream
};
```

Let's take a closer look at how we would implement each of these functions:

```
// From the mmconvienience library:
MmChannel_t *singleIterateInputChannels(const MmFilter_t *f,int32_t
* const cookie);

// From the mmconvienience library:
int32_t singleAcquireInputChannel(MmChannel_t *c);

// From the mmconvienience library:
int32_t singleReleaseInputChannel(MmChannel_t *c);


/*
    This is where a filter that connects to the
    buffered output of an other filter gives a
    rating on its ability to handle the format.
    Since our mpegvideo decoder filter only
    handles one specific input format,
    we just check that the proposed format is correct.
 */
int32_t RateInputFormat(MmChannel_t *c,MmFormat_t *f,int32_t * const
cookie);
{
  if( (*cookie) != 0 )
    return 0;
  (*cookie)++;

  if( f->mf.fourcc != MmFOURCC('M','P','1','V') || f->mf.mtype
!= (MEDIA_TYPE_VIDEO|MEDIA_TYPE_COMPRESSED))
    return 0;
  return 100;
}


/*
    This is where a filter that connects its input
    to the buffered output of an other filter would
    set a negotiated input format.
    In the case of the mpegvideo decoder, we just
    save the negotiated format.
 */
int32_t SetInputFormat(MmChannel_t *c,const MmFormat_t *f)
{
 memcpy(&c->format,fo,sizeof(MmFormat_t));
 ...
}


/*
    This is where a filter that connects its input
    to the buffered output of an other filter  saves the
    other filter's MediaOutput interface.
    This is also a good place to initialize the decoder.
 */
int32_t SetMediaOutput(MmChannel_t *c,const MediaOutput *mo);
{
  MmFilter_t *f = c->filter;
  f->user->mo   = mo;
  c->flags     |= MM_CHANNEL_INPUTSET;
  /*
  ... initialize the decoder
  */
  return 0;
}

// From the mmconvienience library:
static int32_t noSetInputStream(MmChannel_t *c,AOIStream_t *sobj)
```

Managing output channels

Methods that connect your filter's output channels to another filter's input channels, either buffered or unbuffered, are defined in the **MediaOutput** interface.

```
static MediaOutput media_output =
{
    IterateChannels,
    AcquireChannel,
    ReleaseChannel,
    GetStreamer,
    IterateFormats,
    VerifyFormat,
    SetFormat,
    NextBuffer,
    ReleaseBuffer,
    DestroyBuffers
};
```

This is what these methods should do:

IterateChannels	This method gives the library access to all of the filter's output channels by returning each channel in turn, each time the method is called. In the case of an MPEG system parser, we have two output channels (one audio and one video) so we need to implement this method. Since our decoder uses a single channel, we can use the convenience method *singleIterateOutputChannels()*.
AcquireChannel	This method flags our output channel as being in use. Both parser and decoder should implement this method.
ReleaseChannel	This method flags our output channel as being released. Both parser and decoder should implement this method.
GetStreamer	This method provides another filter with a streamed interface to one of our output channels. Neither our parser nor decoder has a streaming output (both are buffered), so we can use the *noGetStreamer()* convenience method.
IterateFormats	This method queries the filter for all possible output formats that a given channel has available. Both parser and decoder should implement this method.
VerifyFormat	This method lets the library give the filter a last chance to accept or reject a negotiated output format for a given channel. Our parser and decoder can simply use the convenience method *acceptVerifyOutputFormats()*.

SetFormat	This method sets the channel's negotiated output format. Both parser and decoder should implement this method.
NextBuffer	This method is called by the next filter in the graph when it needs a buffer for the given time. This method should acquire the buffer (in this case, using the default MediaBufferAllocator), but it doesn't need to release the buffer. This is the responsibility of the filter that calls the *NextBuffer()* method. Both parser and decoder should implement this method.
ReleaseBuffer	When the next filter in the graph is finished with the buffer we gave it in *NextBuffer()*, it calls this function to release the buffer back into its pool.
DestroyBuffers	This method releases any buffers. It's called by the Multimedia library when an output channel is being released.

Parser

Let's take a closer look at how we can implement this interface:

```
static MediaOutput media_output =
{
  IterateOutputChannels,
  AcquireOutputChannel,
  ReleaseOutputChannel,
  noGetStreamer,
  IterateOutputFormats,
  acceptVerifyOutputFormats,
  SetOutputFormat,
  NextBuffer,
  ReleaseBuffer,
  DestroyBuffers
};
```

```
/*  This is where we give the mmedia library access
    to all of our output channels. In the case
    of an mpegsystem parser, we have one audio and one
    video output channel. So we iterate through and
    return each channel, then NULL afterward.
 */
static MmChannel_t *IterateOutputChannels(const MmFilter_t *f,int32_t
* const cookie);
{
  int32_t cnum=*cookie;
  if( cnum >= f->user->nstreams )
    return 0;
  (*cookie)++;
  return &f->ochannels[cnum];
}


/*
    This is where the library flags our
    output channel as being in use.
    Acquire the given output channel if its
    available and mark it as acquired.
    Return -1 on error 0 on success.
*/
static int32_t AcquireOutputChannel(MmChannel_t *c)
{
  if( c->flags&MM_CHANNEL_ACQUIRED )
    return -1;
  // mark as acquired
  c->flags |= MM_CHANNEL_ACQUIRED;
  return 0;
}


/* This is where the mmedia library flags our
   output channel as being released.
   Mark the given channel as no longer acquired.
 */
static int32_t ReleaseOutputChannel(MmChannel_t *c)
{
  c->flags &= ~(MM_CHANNEL_ACQUIRED|MM_CHANNEL_OUTPUTSET);
  return 0;
}


// From the mmconvenience library:
AOIStream_t *noGetStreamer(MmChannel_t *c);


/*  This is where the library can query a
    filter for all possible output formats that
    a given channel has available. Since our
    mpegsystem parser has 2 output channels (audio
    and video) and just one output format per channel
    we return our output channel format the
    first time this function is called and NULL afterward.
 */
static int32_t IterateOutputFormats(MmChannel_t *c,MmFormat_t *fmt,int32_t * const cookie)
{
  if( (*cookie)!=0 )
    return 0;
  (*cookie)++;
  memcpy(fmt,&c->format,sizeof(MmFormat_t));
  return 100;
}

// From the mmconvenience library:
int32_t acceptVerifyOutputFormats(MmChannel_t *c,const MmFormat_t
*f);

/*  This is where the library sets the
    channel negotiated output format, and the
    MediaBufferAllocator interface to use to
    acquire and release buffers.
    Return -1 on error, 0 on success.
 */
static int32_t SetOutputFormat( MmChannel_t *c, const MmFormat_t
*fo,
const MediaBufferAllocator *mba, MmChannel_t *mbac )
{
  if( !(c->flags&MM_CHANNEL_ACQUIRED) )
    return -1;
  if( c->flags&MM_CHANNEL_OUTPUTSET )
    return -1;
  if( memcmp(fo,&c->format,sizeof(MmFormat_t))
!= 0 )
    return -1;
  c->user->mbac = mbac;
  c->user->mba  = mba;
  // mark the channel's output set
  c->flags     |= M_CHANNEL_OUTPUTSET;
  return 0;
}


/*  This method is called by the next filter in the
```

```
    graph when it needs a buffer for the
    given time.  Return the filter playing status
    (MM_STATUS_PLAYING, MM_STATUS_STOP, MM_STATUS_EOF,...)
 */
static int32_t NextBuffer(MmChannel_t *c,MmTime_t t,MmBuffer_t **buffer)
{
  /*
  In the case of the mpegsystem parser filter:
    - if we are not MM_STATUS_PLAYING  return our status;
    - check channel stream id (audio/video )
    - acquire the next buffer with the channel's MediaBufferAllocator
    interface (mba->AcquireBuffer())
    - fill the buffer with data
    - return current playing status
    - the Library handles releasing the buffer
  */
}

/*
    When the next filter in the graph
    is finished with the buffer we gave it in
    NextBuffer(), it calls this function to release
    it back into its pool. In the case of the mpegsystem
    parser filter, call the ReleaseBuffer()
    function provided through the MediaBufferAllocator
    interface.
 */
static int32_t ReleaseBuffer(MmChannel_t *c,MmBuffer_t *b)
{
  return c->user->mba->ReleaseBuffer(c->user->mbac,b);
}

/*
    This function is called by the library when
    an output channel is being released.
    In the case of the mpegsystem parser filter,
    call the FreeBuffer() function
    provided through the MediaBufferAllocator interface.
 */
static int32_t DestroyBuffers(MmChannel_t *c)
{
  if( c->user->mba )
c->user->mba->FreeBuffers(c->user->mbac);
  return 0;
}
```

Decoder

Let's take a closer look at how we would implement this interface:

```
static MediaOutput media_output =
        {
            singleIterateOutputChannels,
            AcquireOutputChannel,
            ReleaseOutputChannel,
            noGetStreamer,
            IterateOutputFormats,
            acceptVerifyOutputFormats,
            SetOutputFormat,
            NextBuffer,
            ReleaseBuffer,
            DestroyBuffers
        };
```

```
// From the mmconvenience library:
static MmChannel_t *singleIterateOutputChannels(const MmFilter_t
*f,int32_t * const cookie);


/*
    This is where the library flags
    our output channel as being in use.
    Acquire the given output channel if its
    available and mark it as acquired.
    Return -1 on error 0 on success.
 */
static int32_t AcquireOutputChannel(MmChannel_t *c)
{
  MmFilter_t *f=c->filter;
  // Make sure the output channel isn't already acquired
  if( c->flags&MM_CHANNEL_ACQUIRED )
    return -1;
  /* Make sure our input channel has already
     been acquired and its input set
     since otherwise we won't know the dimensions,
     etc of our output channel
   */
  if( !(f->ichannels[0].flags&MM_CHANNEL_INPUTSET)
)
    return -1;
  // Flag the output channel as acquired
  c->flags |= MM_CHANNEL_ACQUIRED;
  return 0;
}


/*
    This is where the library flags our output
    channel as being released. Mark the given channel
    as no longer acquired.
 */
static int32_t ReleaseOutputChannel(MmChannel_t *c)
{
  c->flags &= ~(MM_CHANNEL_ACQUIRED|MM_CHANNEL_OUTPUTSET);
  return 0;
}


/*
    This is where a filter can give another filter a
    streamed interface to one of its output channel(s).
    Since our mpegvideo decoder uses buffered output channels
    we just return NULL;
    This function has been implemented for you in the
    mmconvenience library and has the following prototype:
*/
AOIStream_t *noGetStreamer(MmChannel_t *c);

/*
    This is where the mmedia library can query a filter
    for all possible output formats that a given channel
    has available. Our mpegvideo decoder has quite a
    few output formats for the video output channel.
    So we just go through all of them, returning one
    format at a time, and incrementing the iterator.
    Return 0 when we are done.
 */
static int32_t IterateOutputFormats(MmChannel_t *c,MmFormat_t *fmt,int32_t
* const cookie)
{
  // save our iterator
  int32_t cnum = *cookie;
  // if our iterator >= 5  we are done
  if( cnum >= 5 )
  return 0;

  // Initialize our proposed output format with a known value
  memset(fmt,0,sizeof(MmFormat_t));
  memcpy(&fmt->mf,&c->format.mf,sizeof(AODataFormat_t));
  fmt->mf.mtype       = MEDIA_TYPE_VIDEO;
  fmt->min_buffers    = 1;
  // fill out some specific output format value according to this
iteration
  switch( cnum )
  {
    case 0:
      fmt->mf.fourcc  = MmFOURCC('Y','U','Y','2'); // Pg_VIDEO_FORMAT_YUY2
      fmt->mf.u.video.depth =16;
      break;
    case 1:
      fmt->mf.fourcc  = MmFOURCC('R','G','B','2'); // Pg_IMAGE_DIRECT_8888:
      fmt->mf.u.video.depth = 32;
      break;
    case 2:
      fmt->mf.fourcc  = MmFOURCC('R','G','B','4'); // Pg_IMAGE_DIRECT_888:
      fmt->mf.u.video.depth = 24;
      break;
    case 3:
      fmt->mf.fourcc  = MmFOURCC('R','G','B','6'); // Pg_IMAGE_DIRECT_565:
      fmt->mf.u.video.depth = 16;
      break;
    case 4:
      fmt->mf.fourcc  = MmFOURCC('R','G','B','5'); // Pg_IMAGE_DIRECT_555:
      fmt->mf.u.video.depth = 16;
      break;
    default:
      break;
  }
  // Adjust our format min buffer size according to this iteration
  fmt->min_buffersize = fmt->mf.u.video.width
* fmt->mf.u.video.height * ((fmt->mf.u.video.depth+7)>>3);
  // Increment our iterator for next iteration
  (*cookie)++;
  return 100;
}

/*
    This is where the mmedia library gives the filter
    a last chance to accept or reject a negotiated
    output format for a given channel. Return 0 to
    reject the proposed output format, 100 to accept
    it. This method has been implemented for you in
    the mmconvenience library.
    This method has the following prototype:
 */
int32_t acceptVerifyOutputFormats(MmChannel_t *c,const MmFormat_t
*f);


/*
    This is where the library sets the channel
    negotiated output format, and the MediaBufferAllocator
    interface is used to acquire/release buffers.
    Return -1 on error, 0 on success.
 */
static int32_t SetOutputFormat( MmChannel_t *c, const MmFormat_t
*fo,
const MediaBufferAllocator *mba, MmChannel_t *mbac )
{
  if( c->flags&MM_CHANNEL_OUTPUTSET )
    return -1;
  /* Save the MediaBufferAllocator interface our
     output channels will use
     to allocate an output buffer and its
     associated channel.
   */
  c->user->mbac = mbac;
  c->user->mba  = mba;
  // save the negotiated output format
  memcpy(&c->format,fo,sizeof(MmFormat_t));
  // mark the channel's output set
  c->flags     |= M_CHANNEL_OUTPUTSET;
  return 0;
}
```

```
/*
    This function is called by the next filter
    in the graph when it needs a buffer for the
    given time. Return the filter playing status
    (MM_STATUS_PLAYING, MM_STATUS_STOP,MM_STATUS_EOF,...)
 */
static int32_t NextBuffer(MmChannel_t *c,MmTime_t t,MmBuffer_t **buffer)
{
  /*
  In the case of the mpegvideo decoder filter:
  - If we are not MM_STATUS_PLAYING  return  our status
  - Acquire a buffer through the channel's MediaBufferAllocator
  interface (mba->AcquireBuffer())
  - If we're at the EOF, release the buffer; otherwise:
  - Fill the buffer with data
  - Return current playing status
   */
}


/*
    When  the next filter in the graph
    is finished with the buffer we gave it in NextBuffer(),
    it calls this function to release it back into its pool.
    In the case of the mpegvideo decoder filter just call
    the ReleaseBuffer() function provided through the
    MediaBufferAllocator interface.
 */
static int32_t ReleaseBuffer(MmChannel_t *c,MmBuffer_t *b)
{
  return c->user->mba->ReleaseBuffer(c->user->mbac,b);
}

/*
    This function is called by the library when
    an output channel is being released.
    In the case of the mpegvideo decoder filter we call
    the FreeBuffer() function provided through the
    MediaBufferAllocator interface.
 */
static int32_t DestroyBuffers(MmChannel_t *c)
{
  if( c->user->mba )
    c->user->mba->FreeBuffers(c->user->mbac);
  return 0;
}
```

Controlling processing

The **`MediaControl`** interface defines the methods for starting, stopping, pausing, and resuming media filters.

```
static MediaControl media_control =
{
  Start,
  Stop,
  Pause,
  Resume,
  Status
};
```

These methods should do the following:

Start()	The Multimedia library calls this function to start a graph. Filters can use this function to initialize threads, and perform other one-time things they need to do.
Stop()	The Multimedia library calls this method to stop the graph. It should only be called when the library is getting ready to destroy the graph. The filters that receive this call should stop threads, and other one-time things.
Pause()	The Multimedia library calls this method to pause the graph temporarily. This method is called any time a user wants to temporarily stop the graph, or wants to seek into the graph.
Resume()	The Multimedia library calls this method to resume the graph. This is called when the user wants to resume playback after a temporary pause, or after seeking somewhere.
Status()	The Multimedia library calls this method to extract the filter's current status. The filter should return its current playing status.

Parser and Decoder

Both our parser and decoder would implement these methods:

```
        static int32_t Start(MmFilter_t *f,MmTime_t media_time)
        {
            // start the filter
        }

        /       static int32_t Stop(MmFilter_t *f)
        {
            // stop the filter
        }

        static int32_t Pause(MmFilter_t *f)
        {
            // pause the filter
        }

        static int32_t Resume(MmFilter_t *f,MmTime_t media_time,MmTime_t
real_time)
        {
            // resume the filter
        }

        static int32_t Status(MmFilter_t *f)
        {
            // return the playing status
        }
```

Seeking to a location in the stream

The **MediaSeeker** interface defines the seek method, and the filters that need to be informed when the graph is seeking to a new location.

```
static MediaSeeker media_seeker =
{
  Seek
};
```

The Multimedia library calls this function when the graph needs to seek to a certain location in the media. Theoretically, all the filter has to do is empty its buffers, and let the normal **`AOStreamer`** handle the rest.

Parser and Decoder

```
static int32_t Seek(MmFilter_t *f, MmTime_t media_time)
{
  // to do: seek to a new position
}
```

Providing access to a filter's resources

The **`AOResourceAccess`** interface defines the methods that expose any internal resources to the outside world for reading or writing.

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

These methods should do the following:

GetResources()
: The Multimedia library calls this method to query a filter for the availability of a particular resource, for example when an application calls *MmGetResourceValue()* or one of its convenience macros. This method has the following prototype:

```
static const AOResource_t *GetResources(void *handle);
```

SetResource()
: The Multimedia library calls this method to set the resource *res* to the value *data* in a filter. It's called, for example, when an application calls *MmSetResourceValue()* or one of its convenience macros. This method has the following prototype:

```
static int32_t SetResource(void *handle,const char *res,const void
*data);
```

You need to implement this interface only for filters that need to expose their internal resources.

Parser

In this example we implement both these functions.

The Addon Interface library also defines the type **AOResource_t** that's used for internal resource storage and handling data.

```
typedef struct
{
  char *name;     // name of resource
  char *description;  // description of resource
  void *value;    // filled in later with the value
  void *info;     // typing info (ie range, list of items, etc)
  int32_t type;   // AOR_TYPE_* flags
} AOResource_t;
```

A simple AOResourceAccess example

Let's assume that our filter wants to expose the following resources:

- The current position in the media stream:

  ```
  MmTime_t position; // read-only resource
  ```

- The duration of the media stream:

  ```
  MmTime_t duration; // read-only resource
  ```

- A Flag to put the filter in debug mode:

  ```
  uint32_t debug; // read/write resource
  ```

In our filter's internal data structure we have:

```
struct media_filter_user
{
  /*
   other data
  */
  uint32_t debug;      // put the filter in debug mode
  MmTime_t position;   // position in the stream
  MmTime_t duration;   // duration of the stream
  AOResource_t *res;   // data structure needed to store or handle
resources
};
```

We need to define a **AOResource_t** record for each one of these resources:

```
static const MmTime_t timerange[]  = {0,86400000000,1}; // min,
max, default value
static const int32_t  debugrange[] = {0,10,0}; // min, max, default
value

static const AOResource_t resources[] =
{
  {"Duration","Duration",(void*)offsetof(struct
  media_filter_user,duration),&timerange,
    AOR_TYPE_LONGLONG|AOR_TYPE_READABLE },
  {"Position","Position",(void*)offsetof(struct
  media_filter_user,position),&timerange,
    AOR_TYPE_LONGLONG|AOR_TYPE_READABLE },
  {"Debug","Debug Output",(void*)offsetof(struct media_filter_user,debug),&
  debugrange,
AOR_TYPE_POINTER|AOR_TYPE_READABLE|AOR_TYPE_WRITABLE
},
  { 0 }
};
```

In the *Create()* method of the **`AODeConstructor`** interface, we need to allocate some memory and set up our resource pointers:

```
static void *Create(const AOICtrl_t *interfaces)
{
  MmFilter_t *f;
  AOResource_t *res;
  // create the filter object
  if( !(f = (MmFilter_t*) calloc(1,sizeof(MmFilter_t))) )
    return 0;
  // allocate the filter user data
  if( !(f->user = (MmFilterUser_t*) calloc(1,sizeof(MmFilterUser_t)))
)
    return (MmFilter_t*) Destroy(f);
// allocate our resource data structure
  if( !(f->user->res = (AOResource_t*) malloc(sizeof(resources))))
    return (MmFilter_t*) Destroy(f);
// initialize the resource data structure
  memcpy(f->user->res,&resources,sizeof(resources));
  res = f->user->res;
// adjust the resources pointers to the correct offset value
  while( res->name )
  {
    char *p    = (char*) f->user;
    res->value = (void*) (&p[(int32_t)res->value]);
    res++;
  }
  /*
   ....
  */
  return f;
}
```

And finally, the implementation of the *GetResources()* and *SetResource()* functions of the **`AOResourceAccess`** interface:

```
static const AOResource_t *GetResources(void *handle)
{
  MmElement_t *e = (MmElement_t*) handle;
  if( e && e->type==MM_ELEMENT_FILTER
)
  {
    MmFilter_t *f = (MmFilter_t*) handle;
// success
// return a pointer to our resource data structure
    return f->user->res;
}
  return 0;
}

static int32_t SetResource(void *handle,const char *name,const void
*data)
{
  MmElement_t *e = (MmElement_t*) handle;
  if( e && e->type == MM_ELEMENT_FILTER
)
  {
    MmFilter_t   *f   = (MmFilter_t*) handle;
    AOResource_t *res = f->user->res;
    while( res->name )
    {
      if( strcmp(res->name,name) == 0 )
      {
        // found it!
        if(strcmp(name,"Debug") == 0 )
        {
          //fprintf(stderr, "setting debug  to %d\n", (int32_t)data);
          f->user->debug = (int32_t) data;
        }
        return 0;
      }
      res++;
    }
  }
  return -1;
}
```

Making the Multimedia library aware of the filter

The Multimedia library uses the Addon Interface (AOI) library to load multimedia filters and perform multimedia format negotiations at runtime. This means that if you export the set of interfaces discussed above and drop your filter compiled as a DLL into the **`/lib/mmedia/dll`** directory, any multimedia application that uses the multimedia architecture will be able to use the services your filter provides without recompilation.

Parser

Our exported mpegsystem parser interface list:

```
#ifdef VARIANT_dll
 AOInterface_t interfaces[] =
 #else
AOInterface_t mpegs_parser_interfaces[] =
 #endif
 {
   { "Name",1,"mpegs_parser" },
   { "Version",MM_VERSION,0 },
   { "AODeConstructor",AODECONSTRUCTOR_VERSION,&media_filter
},
   { "MediaInput",MM_INPUT_VERSION,&media_input },
   { "MediaOutput",MM_OUTPUT_VERSION,&media_output },
   { "MediaSeeker",MM_SEEKER_VERSION,&media_seeker },
   { "MediaControl",MM_CONTROL_VERSION,&media_control },
   { "AOStreamInspector",AOSTREAMINSPECTOR_VERSION,&stream_inspector
},
   { "AOResourceAccess",AORESOURCEACCESS_VERSION,&resource_access
},
   { 0,0 }
 };
```

Decoder

Our exported mpegvideo decoder interface list:

```
#ifdef VARIANT_dll
 AOInterface_t interfaces[] =
 #else
 AOInterface_t mpegv_decoder_interfaces[] =
 #endif
 {
   { "Name",1,"mpegv_decoder" },
   { "Version",MM_VERSION,0 },
   { "AODeConstructor",AODECONSTRUCTOR_VERSION,&media_filter
},
   { "MediaInput",MM_INPUT_VERSION,&media_input },
   { "MediaOutput",MM_OUTPUT_VERSION,&media_output },
   { "MediaSeeker",MM_SEEKER_VERSION,&media_seeker },
   { "MediaControl",MM_CONTROL_VERSION,&media_control },
{ "AOFormatInspector",AOFORMATINSPECTOR_VERSION,&format_inspector
},
   { 0,0 }
 };
```

Chapter 4

Mm — Multimedia Library

These functions provide application-level multimedia functionality. Using these functions, you can:

- Initialize the Multimedia library.
- Create or destroy multimedia graphs.
- Add multimedia filters to a graph, and connect the filters with channels.
- Control (play, pause, stop, seek) the graph, or individual filters in the graph.
- Get and set graph resources, or resources for individual filters or channels.

MmAcquireInputChannel()

Acquire a multimedia input channel

Synopsis:

```
MmChannel_t *MmAcquireInputChannel(MmFilter_t *filter,
                                   int32_t mtype);
```

Arguments:

filter A pointer to the filter you want to acquire an input channel for.

mtype The media type of the input channel you want to acquire. Can be one of:

- MEDIA_TYPE_IMAGE
- MEDIA_TYPE_VIDEO
- MEDIA_TYPE_AUDIO
- MEDIA_TYPE_TEXT
- MEDIA_TYPE_UNKNOWN

You can OR These flags with MEDIA_TYPE_COMPRESSED if the data is compressed.

Library:

mmedia

Description:

This function acquires an input channel from *filter* of the media type *mtype*. This function returns NULL if it can't acquire an input channel.

Returns:

A pointer to a new channel (an **MmChannel_t**) if successful, NULL if not.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmAcquireOutputChannel(), *MmAttachChannels()*, **`MmChannel_t`**, *MmReleaseChannel()*

MmAcquireOutputChannel()

Acquire a multimedia output channel

Synopsis:

```
MmChannel_t *MmAcquireOutputChannel(MmFilter_t *filter,
                                    int32_t mtype);
```

Arguments:

filter A pointer to the filter you want to acquire an output channel for.

mtype The media type of the output channel you want to acquire. Can be one of:

- MEDIA_TYPE_IMAGE
- MEDIA_TYPE_VIDEO
- MEDIA_TYPE_AUDIO
- MEDIA_TYPE_TEXT
- MEDIA_TYPE_UNKNOWN

You can OR these flags with MEDIA_TYPE_COMPRESSED if the data is compressed.

Library:

mmedia

Description:

This function acquires an output channel from *filter* of the media type *mtype*. This function returns NULL if it can't acquire an output channel.

Returns:

A pointer to a new channel (an **MmChannel_t**) if successful, NULL if not.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmAcquireInputChannel(), *MmAttachChannels()*, **`MmChannel_t`**, *MmReleaseChannel()*

MmAttachChannels()

Attach two multimedia channels

Synopsis:

```
int32_t MmAttachChannels(MmChannel_t *oc,
                         MmChannel_t *ic);
```

Arguments:

oc	A pointer to the output channel you want to connect.
ic	A pointer to the input channel you want to connect.

Library:

mmedia

Description:

This function attaches the output channel *oc* to the input channel *ic*. If the channels are buffered, this function handles negotiating formats between channels.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// assuming channel1 is an output channel acquired from one
// filter, and channel2 is an input channel acquired from
// another filter:

if (MmAttachChannels(channel1,channels)==0)
{
  // we've attached the channels
}
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmAcquireInputChannel(), *MmAcquireOutputChannel()*, **`MmChannel_t`**, *MmReleaseChannel()*

Structure that defines an input or output channel

Synopsis:

```
See below.
```

Description:

This structure defines an instance of an input or output channel returned from a call to an addon's *MediaInput->AcquireInputChannel()* or *MediaOutput->AcquireOutputChannel()* function.

Filters are connected using these channels; an output channel is connected to an input channel. Each channel is stored as a **MmChannel_t** structure, whose first element is a **MmElement_t** structure, which provides easy identification.

This structure has at least the following members:

MmElement_t *element*

The type of media structure this structure represents (in the case of a channel, it's MM_ELEMENT_CHANNEL). The *element* also contains a unique ID.

int32_t *direction*

The channel direction. One of MM_CHANNEL_INPUT or MM_CHANNEL_OUTPUT.

MmFormat_t *flags*

The status of the channel. Can be a combination of:

- MM_CHANNEL_ACQUIRED — the channel is acquired for exclusive use.
- MM_CHANNEL_INPUTSET — the input stream for the channel is set.
- MM_CHANNEL_STREAM — the channel is a streaming (rather than buffered) channel.
- MM_CHANNEL_OUTPUTSET — the output stream for the channel is set.

MmFormat_t *format*

The format of the media element, used to negotiate between filters.

MmFilter_t *filter*

The filter that owns this element.

MmChannel_t *lchannel*

The channel this element is connected to.

Classification:

Neutrino

See also:

MmAcquireInputChannel(), *MmAcquireOutputChannel()*, *MmAttachChannels()*, *MmFindChannelsFilter()*, *MmReleaseChannel()*, **`MediaInput`**, **`MediaOutput`**

The Extending the Multimedia Framework chapter.

MmCreateGraph()

Create a multimedia graph

Synopsis:

```
MmGraph_t *MmCreateGraph(const char *name);
```

Arguments:

name	A string for the graph's name resource. This can't be an empty string.

Library:

mmedia

Description:

This function returns a pointer to a multimedia graph (an opaque **MmGraph_t** structure) and sets its ID resource.

When you're finished with a graph, you should use *MmDestroyGraph()* to destroy the graph and free its resources.

Returns:

A pointer to a **MmGraph_t** structure, or NULL if an error occurred.

Examples:

```
// Create a new graph:

MmGraph_t *graph=MmCreateGraph("My new graph");
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmInitialize(), *MmDestroyGraph()*

Synopsis:

```
int32_t MmDestroyFilter(MmFilter_t *filter);
```

Arguments:

filter	The filter you want to destroy.

Library:

mmedia

Description:

This function destroys the *filter*, and removes it from its graph. Before you call this function, you must make sure that the *filter*:

- is valid
- is stopped — use *MmStop()*
- has no attached channels — use *MmDetachChannel()*.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
  // filter was previously created.

  MmDestroyFilter(filter);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmFindChannelsFilter(), *MmFindFilter()*

MmDestroyGraph()

Destroy a multimedia graph

Synopsis:

```
int32_t MmDestroyGraph(MmGraph_t *graph);
```

Arguments:

graph	A pointer to the multimedia graph to destroy.

Library:

mmedia

Description:

This function destroys a multimedia graph and frees its resources. Calling *MmDestroyGraph()* causes the graph to perform the following steps:

1. Signal all the filters in the graph to stop.
2. Once all filters are stopped, detach and release each filter's channels.
3. Free the filters.
4. Destroy the graph.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// Destroy a previously created graph:

MmDestroyGraph(graph);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmInitialize(), MmCreateGraph()

MmDetachChannel()

Detach a multimedia channel

Synopsis:

```
int32_t MmDetachChannel(MmChannel_t *c);
```

Arguments:

c	A pointer to the channel you want to detach.

Library:

`mmedia`

Description:

This function detaches the channel *c*.

Returns:

0	Success.
-1	An error occurred.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmAcquireInputChannel(), *MmAcquireOutputChannel()*, **`MmChannel_t`**, *MmReleaseChannel()*

Base structure for graphs, filters, and channels

Synopsis:

`See below.`

Description:

This structure defines the base “class” for graphs, filters, and channels.

This structure has at least the following members:

`int32_t` *type*	A flag indicating the element type, which can be one of: • MM_ELEMENT_GRAPH • MM_ELEMENT_FILTER • MM_ELEMENT_CHANNEL.
`char *`*ID*	A unique string identifier.

Classification:

Neutrino

See also:

The Extending the Multimedia Framework chapter.

MmFilter_t

Filter structure

Synopsis:

```
See below.
```

Description:

This structure defines an instance of a filter returned from a call to an Addon's *AODeConstructor->Create()* function. The filter instance is stored as a **MmFilter_t** structure, whose first element is a **MmElement_t** structure, for easy identification.

This structure has at least the following members:

MmElement_t *element*

The element type (MM_ELEMENT_FILTER) and unique ID of the filter.

**MmChannel_t ****ichannels*

The input channels for this filter.

**MmFilterUser_t ****user*

Variables specific to this filter. You define **MmFilterUser_t**.

**MmChannel_t ****ochannels*

The output channels for this filter.

**const AOICtrl_t ****interfaces*

Interfaces for this filter.

**MmGraph_t ****graph*

The graph that contains this filter.

Classification:

Neutrino

See also:

MmAcquireInputChannel(), *MmAcquireOutputChannel()*, **MmChannel_t**, *MmDestroyFilter()*, *MmFindChannelsFilter()*, *MmFindFilter()*, *MmFindMediaReader()*, *MmFindMimetypesFilter()*, *MmPrintGraph()*

AODeConstructor, **MediaClock**, **MediaControl**, **MediaInput**, **MediaOutput**, **MediaReader**, **MediaWriter**.

The Extending the Multimedia Framework chapter.

MmFindChannelsFilter()

Find the best filter for an output channel

Synopsis:

```
MmFilter_t *MmFindChannelsFilter(MmGraph_t *graph,
                                 MmChannel_t *channel);
```

Arguments:

graph	The graph you want to add the found filter to.
channel	A pointer to the channel you want to find the best filter for.

Library:

`mmedia`

Description:

This function creates a context for the best filter it can find in the list of interfaces for the output *channel* in the *graph*, acquires an input channel from it, and attaches the two channels. It returns the newly created filter.

If the output channel is a streamer, the function calls *MmFindStreamsAddon()* to find the best filter for the stream type, and attaches it to. If the output channel uses buffers, the function finds the addon that gives the best rating for each channel using *MmFindFormatsAddon()*.

Returns:

A pointer to a new filter (an **`MmFilter_t`**) if successful, NULL if not, or if no addon with a rating greater than 0 can be found.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmDestroyFilter(), **`MmFilter_t`**, *MmFindFilter()*, *MmFindMediaReader()*, *MmFindMimetypesFilter()*

MmFindFilter()

Find a filter

Synopsis:

```
MmFilter_t *MmFindFilter(MmGraph_t *graph,
                         const char *name);
```

Arguments:

graph	A pointer to the graph you want to find the filter in.
name	The name of the filter you want to find.

Library:

`mmedia`

Description:

This function returns a context for a filter called *name* in the *graph*.

Returns:

A pointer to a filter (**`MmFilter_t`**) if one is found, NULL otherwise.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmDestroyFilter(), **`MmFilter_t`**, *MmFindChannelsFilter()*, *MmFindMediaReader()*, *MmFindMimetypesFilter()*

MmFindMediaReader()

Find a reader filter for a file stream

Synopsis:

```
MmFilter_t *MmFindMediaReader(MmGraph_t *graph,
                              AOIStream_t *stream);
```

Arguments:

graph	A pointer to the graph you want to attach a stream reader filter to.
stream	A pointer to the file stream you want to read.

Library:

mmedia

Description:

This function creates a **stream_reader** filter context, attaches the *stream* to it, and returns it.

Returns:

A pointer to a filter (**MmFilter_t**) if one is found, NULL otherwise.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmDestroyFilter(), **MmFilter_t**, *MmFindChannelsFilter()*, *MmFindFilter()*, *MmFindMimetypesFilter()*

MmFindMimetypesFilter()

Find the best filter for a mimetype

Synopsis:

```
MmFilter_t *MmFindMimetypesFilter(MmGraph_t *graph,
                                  const char *mimetype);
```

Arguments:

graph	A pointer to the graph you want to find the best stream filter for.
mimetype	A pointer to the mimetype to find a filter for.

Library:

`mmedia`

Description:

This function creates a context for the best filter for the *mimetype*, and adds it to the *graph*.

If there's no addon with a rating greater than 0 for the stream, the function returns NULL. Otherwise, the function creates a new filter from the best addon and returns the new filter.

Returns:

A pointer to a filter (**`MmFilter_t`**) if a suitable one is found, or NULL if no filter with a rating greater than 0 is found for the stream.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmDestroyFilter(), **`MmFilter_t`**, *MmFindChannelsFilter()*, *MmFindFilter()*, *MmFindMediaReader()*

Structure that defines a buffered media format

Synopsis:

```
See below.
```

Description:

This structure defines a Multimedia library-specific structure that includes the general `AODataFormat_t` structure, as well as buffer constraints for use when negotiating formats between filters. The Multimedia library uses this structure to negotiate between filters with buffered channels. The negotiation works like this:

1. The library queries the output filter for the formats it will output by calling the *IterateFormats* method of the filter's `MediaOutput` interface.
2. The library then gets the input filter to rate each of the formats using the *RateFormat* method of the filter's `MediaInput` interface.
3. The library then selects the most highly rated format, and sets the format for both channels by calling the *SetFormat()* method of the input filter's `MediaInput` interface and output filter's `MediaOutput` interface.

This structure has at least the following members:

`AODataFormat_t` *mf*

The media type, FOURCC, and format.

`int32_t` *min_buffersize*

The minimum buffer size, in bytes.

`int32_t` *max_buffersize*

The maximum buffer size, in bytes.

☞ The *max_buffersize* member isn't used by any QNX-provided filters.

`int32_t` *align_width*

For video formats, the number of pixels to align the width with. Can be 2, 4, or 8 pixels.

`int32_t` *align_height*

For video formats, the number of pixels to align the height with. Can be 2, 4, or 8 pixels.

`int32_t` *align_start*

The number of bytes to align the start address on, if special alignment is required.

`int32_t` *align_stride*

The number of bytes to align the stride with, if stride alignment is required.

`int32_t` *scatter_gather*

Indicates whether scatter/gather DMA can be used, where 0 is no and 1 is yes.

`int32_t` *min_buffers*

The minimum number of buffers to use.

`int32_t` *max_buffers*

The maximum number of buffers to use.

☞ The *max_buffers* member isn't used by any QNX-provided filters.

`int32_t` *matchflags*

Flags used for format negotiation. These flags indicate how the filter prefers to allocate memory. The flags are:

- **`MM_ALLOCATE_VIDEO`** — prefers to allocate buffers in video memory.
- **`MM_ALLOCATE_SHARED`** — prefers to allocate buffers in shared memory.
- **`MM_ALLOCATE_RAM`** — prefers to allocate buffers in RAM.

`int32_t` *exclflags*

Flags used for format negotiation. These flags indicate whether the filter requires buffers to be allocated, and who should allocate them. If neither filter allocates buffers, the Multimedia library uses the default **`MediaBufferAllocator`**. If either filter sets these flags, it must also implement its own **`MediaBufferAllocator`** interface. The flags are:

- **`MM_ALLOCATE_FILTER`** — set this flag if the filter requires buffers to be allocated. Then OR in one of the other flags.
- **`MM_ALLOCATE_INPUT`** — the input filter must allocate the buffers.
- **`MM_ALLOCATE_OUTPUT`** — the output filter must allocate the buffers.

Classification:

Neutrino

See also:

The Extending the Multimedia Framework chapter.

MmGetDuration()

Gets the duration of a graph

Synopsis:

```
MmTime_t MmGetDuration(MmGraph_t *graph);
```

Arguments:

graph	A pointer to the graph you want to get the duration for.

Library:

mmedia

Description:

This function returns the duration (length) of a graph, in milliseconds.

Returns:

The duration of the graph in milliseconds on success, NULL otherwise.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmGetResourceValue()

MmGetResourceValue()

Get the value of a resource for a graph or filter

Synopsis:

```
const void *MmGetResourceValue(const void *element,
                               const char *resource);
```

Arguments:

element	A pointer to the graph or filter you want to get the resource for.
resource	The name of the resource you want to get the value of.

Library:

mmedia

Description:

This function gets the resource value data for the resource. The returned value is read-only. You must cast the return value to the proper type.

You can get more complete information for a resource (including type, range, etc.) by using *MmGetResource()*. Use *MmSetResourceValue()* to change the value of a resource.

These defined convenience macros make getting specific value types easier:

- *MmGetResourceINT64(element, resource, destination)* - *destination* is returned as an `int_64`.
- *MmGetResourceINT32(element, resource, destination)* - *destination* is returned as an `int_32`.
- *MmGetResourceSTRING(element, resource, destination>)* - *destination* is returned as a `char *`.

Returns:

A pointer to the resource value if successful, NULL otherwise.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmSetResourceValue()

Structure that defines a media graph

Synopsis:

```
See below.
```

Description:

This structure defines a storage format for all the filters required to process multimedia data from source to destination.

This structure has the following as its first member:

`MmElement_t` *element*

The type of structure (MM_ELEMENT_GRAPH) and a unique ID.

Classification:

Neutrino

See also:

The Extending the Multimedia Framework chapter.

MmInitialize()

Initialize the Multimedia library

Synopsis:

```
int32_t MmInitialize(const char *addon_path);
```

Arguments:

addon_path	The path to multimedia DLLs that the library searches on initialization. Set to NULL to use the default **/lib/dll/mmedia**.

Library:

mmedia

Description:

This function initializes the Multimedia library **libmmedia**. When the Multimedia library is initialized, every DLL in the multimedia directory is examined for its **interfaces** symbol. To reduce start-up time for a multimedia application, you should put all multimedia DLLs in the default multimedia directory (**/lib/dll/mmedia**) rather than in **/lib/dll**.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// Initialize the Multimedia library with its default filters:

MmInitialize(NULL);

// Initialize the Multimedia library to load the filters in
// /usr/local/lib/filters/

MmInitialize("/usr/local/lib/filters");
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmCreateGraph()

MmMimeInfo()

Acquire a multimedia channel

Synopsis:

```
const AOMimeInfo_t *MmMimeInfo(const AOICtrl_t *control);
```

Arguments:

control	An opaque variable that you should initialize to 0 before the first call.

Library:

`mmedia`

Description:

@@@ Need more info @@@ : This function returns an array of 0-value terminated **`AOMimeInfo_t`** structures for the given *control*.

`AOMimeInfo_t` contains mime information. It contains at least the following members:

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

Returns:

A pointer a an array of **`AOMimeInfo_t`** on success, NULL otherwise.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmGetResourceValue()

MmPause()

Pause a multimedia graph

Synopsis:

```
int32_t MmPause(MmGraph_t *graph);
```

Arguments:

graph	A pointer to a graph.

Library:

mmedia

Description:

This function temporarily pauses *graph* by calling the **MediaControl->***Pause()* function for each filter in the graph.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
  // Assuming the given graph is created, hooked up,
  // and started, and we want to pause it:

  MmPause(graph);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmResume(), *MmSeek()*

MmPrintGraph()

Print a graph tree to the display

Synopsis:

```
void MmPrintGraph(const MmFilter_t *filter,
                  int32_t level);
```

Arguments:

filter	The filter you want to start printing at.
level	Set this argument to 0.

Library:

`mmedia`

Description:

This function prints the graph tree, starting at the given *filter*, to stdout. You should set the *level* to 0, as the printing is done recursively, with *level* incrementing down the filter graph.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmCreateGraph()

MmReleaseChannel()

Release a multimedia channel

Synopsis:

```
int32_t MmReleaseChannel(MmChannel_t *channel);
```

Arguments:

channel	A pointer to the channel you want to release.

Library:

mmedia

Description:

This function releases the *channel* from its filter.

If *channel* is attached to another filter, you should call *MmDetachChannel()* instead of this function. You should call *MmReleaseChannel()* only if you can't attach a given channel after acquiring it.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// We previously acquired this channel, and now we're done
// with it, so we want to release the channel.

MmReleaseChannel(channel);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmAcquireInputChannel(), *MmAcquireOutputChannel()*, **`MmChannel_t`**, *MmDetachChannel()*

MmResume()

Resume a paused multimedia graph

Synopsis:

```
int32_t MmResume(MmGraph_t *graph);
```

Arguments:

graph	A pointer to a graph.

Library:

mmedia

Description:

This function resumes a paused *graph* by calling the **MediaControl->***Resume()* function for each filter in the graph.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// Assuming the given graph is created, hooked up, and
// started, and we want to continue playback:

MmResume(graph);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmPause(), *MmSeek()*

Seek in a multimedia graph

Synopsis:

```
int32_t MmSeek(MmGraph_t *graph,
               MmTime_t mt);
```

Arguments:

graph	A pointer to a graph.
mt	The media time to seek to, in microseconds.

Library:

`mmedia`

Description:

This function seeks the *graph* to the time *mt* (in microseconds). You don't need to pause and resume the graph; if the graph isn't already paused, *MmSeek()* calls *MmPause()*, does the seek, and then resumes the graph with *MmResume()*.

MmSeek() calls the *MediaControl->Seek()* function for all filters in the graph.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// Assuming the given graph is created, hooked up, started,
// and we want to seek to 10 seconds into the playback,
// and resume playback:

MmPause(graph);
MmSeek(graph,10000000);
MmResume(graph);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmPause(), *MmResume()*

MmSetDefaultClock()

Set a graph's synchronization clock

Synopsis:

```
int32_t MmSetDefaultClock(MmGraph_t *graph);
```

Arguments:

filter	A pointer to the graph you want to set the synchronization clock for.

Library:

mmedia

Description:

This function sets the *graph*'s synchronization clock to the default clock, if it doesn't already have a synchronization clock. The default clock is based on the realtime clock.

Returns:

0	Success.
-1	An error occurred.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmInitialize()

Synopsis:

```
int32_t MmSetResourceValue(const void *element,
                           const char *resource,
                           void *data);
```

Arguments:

element	A pointer to the graph or filter you want to set the resource for.
resource	The name of the resource you want to set the value of.
data	The value you want to set *resource* to.

Library:

mmedia

Description:

This function sets the resource *data* for *element*. The resource string is the same as in *MmGetResource()*. The *data* type depends on the resource type.

After changing a resource's value, you should reload any **AOResource_t** structures that your application currently has cached, because some or all of the resources could have been affected.

Returns:

0	Success.
-1	An error occurred.

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmGetResourceValue()

Synopsis:

```
int32_t MmStart(void *element,
                MmTime_t mt);
```

Arguments:

element	A pointer to a graph or filter.
mt	The current media time, in milliseconds. See below.

Library:

mmedia

Description:

This function starts a graph's filters, or an individual filter if *element* is a filter, starting at the time *mt* (in microseconds). The *element* argument is type `void *` to prevent compiler errors when you pass different structures as this parameter.

If *element* is a graph, the function calls all the graph's filters' *MediaControl->Start()* functions, with the media time *mt*. It then calls all the filters' *MediaControl->Resume()* function, since filters are started in a paused state.

You should call *MmStart()* only once for a graph. If at some point you need to change one of the filters, perform the following steps:

1. *MmPause()* the graph.
2. *MmStop()* the filter you want to remove.
3. Detach the filter.
4. Destroy the filter.
5. Create a new filter.
6. Attach the new filter.
7. *MmStart()* the new filter.
8. *MmResume()* the graph.

If you want to start the graph at a point other than the beginning of the stream (`0`), you should either seek to a position in the graph, and then start at that position, or start the graph at `0` and then seek to the position:

Example 1:

```
MmStart(graph, 0);
MmSeek(graph, t);
```

Example 2:

```
if(!MmSeek(graph, t))
MmStart(graph, t)
```

Returns:

0	Success.
-1	An error occurred.

Examples:

```
  // Assuming the graph is created, and whatever filters
  // were needed are added:

  MmStart(graph,0);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmPause(), *MmResume()*, *MmStop()*

Return the status of a multimedia graph or filter

Synopsis:

```
int32_t MmStatus(const void *element);
```

Arguments:

element	A pointer to a graph or filter.

Library:

mmedia

Description:

This function returns the status of the media *element*. If the status isn't an error, it's a positive number:

- MM_STATUS_PLAYING
- MM_STATUS_PAUSING
- MM_STATUS_PAUSED
- MM_STATUS_EOF
- MM_STATUS_STOPPING
- MM_STATUS_STOPPED
- MM_STATUS_TIMEDOUT

Returns:

>0	Success.
<0	An error occurred.

Examples:

```
// Assuming the given graph is created, and hooked up,
// and we want to know the current status (running, not
// running, paused, playing, etc.)

int32_t status=MmStatus(graph);

//
// Assuming the given filter is created, and hooked up,
// and we want to know the current status (running, not
// running, paused, playing, etc.)

int32_t status=MmStatus(filter);
```

Classification:

Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	No

See also:

MmGetResourceValue()

MmStop()

Stop a multimedia graph

Synopsis:

```
int32_t MmStop(const void *element);
```

Arguments:

element	A pointer to a graph or filter.

Library:

mmedia

Description:

If *element* is a graph, this function signals all of the filters in the graph to stop. If *element* is a filter, the function signals that filter to stop. *MmStop()* calls the *MediaControl->Stop()* for each filter to signal the stop.

Once all signals have been sent, the function waits for the *Status()* function in the **MediaControl** interface instance of each filter to return MM_STATUS_STOPPED.

Call this function for a graph only once, just before you detach (*MmDetach()*) and destroy (*Destroy()*) all the filters in the graph.

Returns:

0	Success.
-1	An error occurred.

Examples:

```
// Assuming the graph is created and previously started:

MmStop(graph);

// Assuming the given filter is created and hooked up,
// and for some reason we want to stop it:

MmStop(filter);
```

Classification:

Neutrino

Safety	
Interrupt handler	No

continued. . .

Safety	
Signal handler	No
Thread	No

See also:

MmPause(), *MmResume()*, *MmStart()*

Chapter 5

Multimedia Filter Reference

In this chapter. . .

This chapter provides reference information about the filters provided with the Multimedia library. The source and binary for each filter is available in the Multimedia TDK, while most binaries are also shipped with QNX Neutrino. There are some exceptions: the Xing MPEG audio decoder, MPEG audio parser, MPEG video parser, and MPEG system parser filter binaries are available only in the TDK.

Note that in some instances, filters use libraries that require special licensing, or are distributed under an open-source license.

Reader filter

The Multimedia library simplifies reading multimedia data by using a generic stream reader filter. It encapsulates three stream readers, a file reader, HTTP reader, and CD reader (listed below).

- **Binary**: `/lib/dll/mmedia/stream_reader.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/readers/stream_reader/`

File reader

A streamer filter that reads a file-based stream.

- **Binary**: `/lib/dll/mmedia/fildes_streamer.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/streamers/fildes_reader/`

HTTP reader

A streamer filter that reads an HTTP-based stream.

- **Binary**: `/lib/dll/mmedia/http_streamer.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/streamers/http_streamer/`

CDDA reader

A streamer filter that reads the audio CDDA format. It implements these resources:

`TracksCount`	`int32_t` value containing the number of tracks a media stream contains. Read-only.
`Tracks`	`int32_t value` containing the number of the current track. Read / write.
`CDDA_MILLISECONDS`	`int32_t value` containing the amount of read-ahead buffer the CDDA reader is using. This value can be between 300 milliseconds and 1 minute. Read / write.
`Error`	`string` value containing an error message. Read-only.

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/cdda_reader.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/streamers/cdda_reader/`

Writer filters

Audio writer

This filter outputs audio to an audio card. The audio writer accepts mono or stereo information (surround sound formats aren't supported), with either 8-bit or 16-bit depth. Data is in PCM format — see the Neutrino *Audio Developer's Guide* for more information on this format.

It implements these resources:

`Position`	`int64_t` value containing the position in the media stream. Read-only.
`Volume`	`int32_t` value containing the volume, from 0 to 100. Read / write.
`Balance`	`int32_t` value containing the balance, from 0 to 100 (50 is "normal"). Read / write.

- **Binary**: `/lib/dll/mmedia/audio_writer.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/writers/audio_writer/`

Raw file writer

A raw file writer filter that reads from a channel stream, and dumps it out to an output stream, most likely a file.

- **Binary**: `/lib/dll/mmedia/rawfile_writer.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/writers/rawfile_writer/`

WAV file writer

A WAV format file writer filter that writes uncompressed PCM audio data to a WAV file.

- **Binary**: `/lib/dll/mmedia/wavfile_writer.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/writers/wavfile_writer/`

Window writer

A window writer filter that outputs video to a Photon window.

It implements these resources:

`Position`	**`int64_t`** value containing the position in the media stream. Read-only.
`PtWidget_t`	a pointer to the widget that the writer sends video to. Read/write.
`Width`	**`int32_t`** value containing the width of the video image. Read-only.
`Height`	**`int32_t`** value containing the height of the video image. Read-only.
`DisplayArea`	a pointer to a **`PhArea_t`** containing the current position and dimension of the video widget. Read/write.
`ScalerEnabled`	**`int32_t`** value indicating whether the hardware video overlay scaler is in use. Read-only.

- **Binary**: **`/lib/dll/mmedia/window_writer.so`**
- **Source**: *tdk_install_dir***`/src/lib/mmedia/filters/writers/window_writer/`**
- **Dependencies:** Photon

Decoder filters

Ogg Vorbis decoder

A decoder filter for the Ogg Vorbis audio format. This filter requires a floating point unit (FPU) on the target platform — if no FPU exists, the Multimedia library uses the Ogg integer decoder instead.

This filter expects a single input channel of streaming compressed audio data in Ogg Vorbis format. The output channel has a fourcc of **`RAWA`**, scale of 1, depth (sample rate) of 2, and at least 1 buffer. The number of channels and duration depends on the source data.

Implemented resources:

`Duration`	**`int64_t`** value containing the length of the media stream. Read-only.
`Position`	**`int64_t`** value containing the position in the media stream. Read-only.

- **Binary**: **`/lib/dll/mmedia/ogg_decoder.so`**

- **Source**:
  tdk_install_dir`/src/lib/mmedia/filters/decoders/ogg_decoder/`
- **Dependencies:** Loads the vorbis and ogg libraries (*tdk_install_dir*`/src/lib/mmedia/codecs/`).

Ogg Vorbis integer decoder

An integer-based decoder filter for the Ogg Vorbis audio format. This filter doesn't require an FPU on the target platform. Otherwise, it is identical to the floating-point version of the decoder.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/oggi_decoder.so`
- **Source**:
  tdk_install_dir`/src/lib/mmedia/filters/decoders/oggi_decoder/`
- **Dependencies:** Loads the Tremor and Ogg libraries

FF MPEG video decoder

An MPEG1 SYSTEM (ISO/IEC 11172-1) stream decoder (video and audio) and MPEG1 VIDEO (ISO/IEC 11172-2) stream decoder (video only).

- **Binary**: `/lib/dll/mmedia/ff_mpegv_decoder.so`
- **Source**:
  tdk_install_dir`/src/lib/mmedia/filters/decoders/ff_mpegv_decoder/`
- **Dependencies:** Loads the LGPL `ffmpeg` open-source project library `libavcodec.so` (see `ffmpeg.sourceforge.net`).

Xing MPEG audio decoder

A Xing MPEG audio decoder filter with support for MPEG1 layer 1, layer 2, and layer 3 audio decoding (MP1, MP2, and MP3). It also provides non-ISO-compliant extension of the MPEG1 audio layer 3 format (MPEG 2-5). It allows very low sampling rates down to 8000 samples per second (8kHz).

☞ This decoder is shipped with the Multimedia TDK only.

- **Binary**: `/lib/dll/mmedia/xing_mpega_decoder.so`

- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/decoders/xing_mpega_decoder/`
- **Dependencies:** Loads the `xing_audio` library.

Parser filters

AIFF parser

A parser filter for Apple's `AIFF` audio format with support for ITU G.711 ulaw, PCM8, and PCM16 compression codes.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/aif_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/aif_parser/`

AU parser

A parser filter for the `AU` audio format with support for these compression codes:

- MULAW_8 — 8 bit mu-law G.711
- ALAW_8 — 8 bit A-law G.711
- LINEAR_8 — 8 bit fixed-point samples
- LINEAR_16 — 16 bit fixed-point samples

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/au_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/au_parser/`

AVI parser

A parser filter for the `AVI` format.

☞ Only the audio part of the stream is decoded/parsed, as this format requires additional licensing.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/avi_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/avi_parser/`

IFF parser

A parser filter for the IFF/8SVX audio Interchange File Format with support for the PCM8_S (8 bits pcm samples) compression code.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/iff_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/iff_parser/`

MIDI parser

A parser filter for the MIDI audio format.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/midi_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/midi_parser/`

MPEG audio parser

☞ This parser is shipped with the Multimedia TDK only.

A parser filter for the MPEG1 audio format with support for the interpretation of MPEG1 layer 1, layer 2, and layer 3 (MP1, MP2, and MP3) streams.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.
`IcyInfo`	a string value containing any icecast embedded information
`ID3`	a pointer to a buffer containing an MPEG ID3 tag header, if available. It's up to the application to parse this header.

- **Binary**: `/lib/dll/mmedia/mpega_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/mpega_parser/`

MPEG system parser

☞ This parser is shipped with the Multimedia TDK only.

A parser filter for the MPEG1 System format (audio/video).

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/mpegs_parser.so`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/mpegs_parser/`

MPEG video parser

☞ This parser is shipped with the Multimedia TDK only.

A parser filter for the MPEG1 video format.

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.

`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/mpegv_parser`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/mpegv_parser/`

WAV parser

A parser filter for the WAV audio format with support for:

- WAVE_FORMAT_PCM — 8 and 16 bits PCM samples
- WAVE_FORMAT_ADPCM — Microsoft adaptive differential pulse code modulation format
- WAVE_ITU_ALAW — ITU G.711 standard A-law compression format
- WAVE_ITU_MULAW — ITU G.711 standard mu-law compression format
- WAVE_IMA_ADPCM — Interactive Multimedia Association ADPCM format
- WAVE_GSM610 — European GSM 06.10 standard format compression codes

Implemented resources:

`Duration`	`int64_t` value containing the length of the media stream. Read-only.
`Position`	`int64_t` value containing the position in the media stream. Read-only.

- **Binary**: `/lib/dll/mmedia/wav_parser`
- **Source**: *tdk_install_dir*`/src/lib/mmedia/filters/parsers/mpegv_parser/`

Chapter 6
Multimedia library Structure Reference

In this chapter...

This chapter provides reference information about the data structures used by the Multimedia library.

AOMimeInfo_t

This structure contains mime information. An addon uses **AOMimeInfo_t** to let an application know what mimetypes it can process.

The structure contains at least the following members:

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

AOResource_t

This is the resource structure your addon has to use to give access to internal resources.

The structure contains at least the following members:

char **name*	The name of the resource.
char **description*	A short description of the resource.
void **value*	A pointer to the actual value of the resource.
void **info*	A pointer to typing information (such as a range, list of items, etc.).
int32_t *type*	The resource type, which is one of:

- AOR_TYPE_LONG — *value* points to a **int32_t**, and *info* points to a three-**int32_t** array containing minimum, maximum, and increment values.
- AOR_TYPE_LONGLONG — *value* points to a **int64_t**, and *info* points to a three-**int64_t** array containing minimum, maximum, and increment values.
- AOR_TYPE_FLOAT — *value* points to a **float**, and *info* points to a three-**float** array containing minimum, maximum, and increment values.
- AOR_TYPE_STRING — *value* points to an allocated string buffer, and *info* points to an **int32_t** that contains the maximum length of the string.

- AOR_TYPE_RADIO — a radio button; *value* points to an **int32_t** indicating the index of the selected button, and *info* points to a structure containing an **int32_t** for the count value, followed by count **char *** pointers to button labels.
- AOR_TYPE_TOGGLE — a toggle button; *value* points to an **int32_t**, with 0 indicating the button isn't selected, and 1 indicating it is. There is no *info* 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. These values are used when automatically generating a GUI for a DDL's resources, for example:

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

AODataFormat_t

This structure defines a description of a media format. Use the *mtype* field to determine the output type (audio or video), as well as its compressed state. This structure provides a quick way to determine if you need to hook up a given channel to a decoder, or to a final output if you're playing back media.

Unless you're writing your own filters, you'll never have to touch *fourcc* or *u*.

This structure has the following members:

uint32_t *mtype*
: A flag indicating media type, which can be one of:

- MEDIA_TYPE_IMAGE
- MEDIA_TYPE_VIDEO
- MEDIA_TYPE_AUDIO

These flags can be ORed with MEDIA_TYPE_COMPRESSED if the data is compressed.

uint32_t *fourcc*
: 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:

- RGB6 — 16-bit RGB.
- RGB5 — 15-bit RGB.
- RGB4 — 24-bit RGB.
- RGB2 — 32-bit RGB.

u	A straight union for the above media formats. The union contains members *image*, *audio*, and *video*, of type **AOImageFormat_t**, **AOAudioFormat_t**, and **AOVideoFormat_t** respectively.

AOImageFormat_t

This structure defines an image format, and is used by **AODataFormat_t**.

This structure has at least the following members:

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

AOAudioFormat_t

This structure defines an audio format, and is used by **AODataFormat_t**.

This structure has at least the following members:

uint32_t *channels*	The number of audio channels per frame. For example, a stereo signal has 2 channels.
uint32_t *depth*	The audio depth (sample rate) in bytes.
int32_t *frame_rate*	The scaled frame rate. This value is divided by *scale* for the actual frame rate.
int32_t *scale*	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 *frame_rate* to 2997, and *scale* to 100.
int32_t *duration*	The duration of the audio, in frames.

AOVideoFormat_t

This structure defines a video format, and is used by **AODataFormat_t**.

This structure has at least the following members:

uint32_t *width*	The width of the video image, in pixels.
uint32_t *height*	The height of the video image, in pixels.
uint32_t *depth*	The color depth (number of bits per pixel).
int32_t *frame_rate*	The scaled frame rate. This value is divided by *scale* for the actual frame rate.
int32_t *scale*	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 *frame_rate* to 2997, and *scale* to 100.
int32_t *duration*	The duration of the video, in frames. Set to 0 if unknown.

Chapter 7

Multimedia Interface Reference

This chapter provides reference information about the interfaces that a filter has to implement to be used by the Multimedia library. Some of the interfaces are defined in the Addon Interface Library (the **`AO*`** interfaces), while others are defined in the Multimedia library (**`Media*`** interfaces). For a description of how to write a filter, see the Extending the Multimedia library chapter.

Interface	**Description**	**Used in**
`AODeConstructor`	Create and destroy interface for a filter	All filters
`AOStreamInspector`	A stream inspection interface	Any filter with a streaming input of raw data
`AOFormatInspector`	A format inspection interface	Decoders and writers working with formatted data
`AOExtInspector`	A file extension interface	Any reader that takes streaming input (not implemented by existing Multimedia filters)
`AOMimetypeInspector`	A mimetype inspection interface	Parsers
`AOResourceAccess`	A filter resource manipulation interface	Any filter that exposes resources
`MediaReader`	An input stream interface	Readers
`MediaWriter`	A stream writer interface	Writers that output to a streamer (such as the raw file or wave file writers)
`MediaInput`	Input channel interface	Filters with one or more input channels
`MediaOutput`	Output channel interface	Filters with one or more output channels
`MediaControl`	Playback control interface	Filters that need to respond to a *Pause()*, for example, frame-based filters that need to start at the beginning of a frame
`MediaSeeker`	Seek interface	Filters that need to respond to a *Seek()*
`MediaClock`	Clock synchronization interface	Writers. The library implements a default **`MediaClock`**, which is used by the existing audio writer filter.

continued. . .

Interface	Description	Used in
`MediaBufferAllocator`	Buffer allocation interface	Any filter that uses custom buffer allocation. The library implements a default **`MediaBufferAllocator`**, which existing Multimedia filters use.

More information about the **`AO*`** interfaces can be found in the *Addon Interfaces Library Reference*.

MediaBufferAllocator

Functions for buffer allocation

Synopsis:

```
static MediaBufferAllocator media_buffer_allocator =
{
    AllocateBuffers,
    FreeBuffers,
    AcquireBuffer,
    ReleaseBuffer
}
```

Description:

The Multimedia library implements a **MediaBufferAllocator** that your filters can use.

This interface defines custom buffer allocation and manipulation. You can implement buffer allocating, locking, and unlocking using this interface.

AllocateBuffers()

```
int32_t (*AllocateBuffers)(MmChannel_t *channel,
                           const MmFormat_t *format);
```

This function should allocate buffers for a given *channel* with the given *format* specifications.

If successful, this function should return 0.

FreeBuffers()

```
int32_t (*FreeBuffers)(MmChannel_t *channel);
```

This function should free the buffers for the *channel*.

If successful, this function should return 0.

AcquireBuffer()

```
MmBuffer_t *(*AcquireBuffer)(MmChannel_t *channel,
                             uint32_t unused);
```

This function should acquire a buffer for a *channel* with a minimum size of the *min_buffersize* field of the **MmFormat_t** structure filled in at format negotiation time, and return a pointer to it, or NULL if it can't acquire a buffer.

ReleaseBuffer()

```
int32_t (*ReleaseBuffer)(MmChannel_t *channel,
                         MmBuffer_t *buffer);
```

This function should release the given *buffer* from the *channel* for subsequent use.

If successful, this function should return 0.

Classification:

Neutrino

See also:

Extending the Multimedia Framework.

MediaClock

Functions for media sychronization

Synopsis:

```
static MediaClock media_clock =
 {
    Initialize,
    Uninitialize,
    Value,
    SetScale
 };
```

Description:

The Multimedia library implements a default **MediaClock** interface that your filters can use. This interface defines the functions needed for media clock synchronization. Synchronization is normally defined in an audio output filter, since it yields the best synchronization. **MediaClock** allows multiple filters to get accurate timestamps for accurate synchronization.

MediaClock defines the following functions:

- *Initialize()*
- *Uninitialize()*
- *Value()*
- *SetScale()*

Initialize()

```
int32_t (*Initialize)(MmFilter_t *filter);
```

This function should initialize the *filter*'s media clock portion.

If successful, this function should return 0.

Uninitialize()

```
int32_t (*Uninitialize)(MmFilter_t *filter);
```

This function should free the *filter*'s media clock data.

If successful, this function should return 0.

Value()

```
MmTime_t (*Value)(MmFilter_t filter);
```

This function should return the *filter*'s current timestamp. The timestamp is independent of your position in any media stream; it's an incrementing value you can use to perform synchronization. This function returns an **MmTime_t**, an **int_64** that represents time in microseconds.

SetScale()

```
int32_t (*SetScale)(MmFilter_t *filter,
                    int32_t scale);
```

This function should set the *scale* for the returned time from the given *filter*. The default time should be in microseconds, and is divided by *scale* when this function is called. For example, if *scale* is set to 1000, then the filter should return time in milliseconds.

Classification:

Neutrino

See also:

Extending the Multimedia Framework.

Synopsis:

```
static MediaControl media_control =
{
  Start,
  Stop,
  Pause,
  Resume,
  Status
};
```

Description:

This interface defines the functions required to control a filter. Filter control is usually necessary only for final output filters, such as audio output or video display, but may also be required if your filter spawns worker threads you'd like to start, stop, pause, or resume.

Start()

```
int32_t (*Start)(MmFilter_t *filter,
                 MmTime_t start_time);
```

This function should start the *filter* in a paused state. Don't think of this as starting the decoding. This function is called when a graph has been constructed and it's ready to go. Once it's started, you can use *Resume()* to begin decoding/playback.

If the *start_time* is nonzero, the filter should seek to the given time, if it can.

If successful, this function should return 0.

Stop()

```
int32_t (*Stop)(MmFilter_t *filter);
```

This function should stop the *filter*. Use *Stop()* only when you're about to remove or delete the filter. You should signal any running threads to stop at the point when you call *Stop()*.

If successful, this function should return 0.

Pause()

```
int32_t (*Pause)(MmFilter_t *filter);
```

This function should pause playback of the *filter* temporarily. If you are going to call *Seek()* on a filter, you should first *Pause()*, then *Seek()*, and then *Resume()*.

If the filter has a running thread, it will most likely set its status to MM_STATUS_PAUSING in the *Pause()* function, which the thread will recognize, and change to MM_STATUS_PAUSED. The filter then enters a paused state.

If successful, this function should return 0.

Resume()

```
int32_t (*Resume)(MmFilter_t *filter,
                  MmTime_t media_time,
                  MmTime_t real_time);
```

This function should resume playback of the *filter*. Each paused filter is given the same *real_time* value and *media_time* value so that they can stay synchronized. "Media time" is the elapsed media file time, while "real time" is the system clock time. Both values are of type `MmTime_t`, an `int_64` that stores time in microseconds.

☞ Don't use *media_time* for seeking; it's meant only for making slight synchronization adjustments.

If successful, this function should return 0.

Status()

```
int32_t (*Status)(MmFilter_t *filter);
```

This function should return the status of the *filter*. This should be one of:

- MM_STATUS_PLAYING — the filter is processing data normally.
- MM_STATUS_PAUSING — the graph is in a paused state, and has requested each filter to pause. The filter is finishing any processing it needs to complete before pausing.
- MM_STATUS_PAUSED — the filter is in a paused state.
- MM_STATUS_STOPPING — the graph is in a stopped state, and has requested each filter to stop. The filter is finishing any processing it needs to complete before stopping.
- MM_STATUS_STOPPED — the filter is in a stopped state.
- MM_STATUS_EOF — the filter is at the end of a file (buffer or stream).
- MM_STATUS_TIMEDOUT — the filter has some data to process, but can't continue processing for some reason (for example, there's not enough input data to fill the output buffer).

☞ None of the current QNX-provided filters set their status to MM_STATUS_TIMEDOUT.

- a negative number signifying an error.

Classification:

Neutrino

See also:

`MediaSeeker`.

Extending the Multimedia Framework.

MediaInput

Functions for a filter with an input channel

Synopsis:

```
static MediaInput media_input =
{
  IterateChannels,
  AcquireChannel,
  ReleaseChannel,
  RateFormat,
  SetFormat,
  SetMediaOutput,
  SetInputStream
};
```

Description:

This interface defines the functionality required by any filter that gets its input data from another filter, either streaming or buffered.

IterateChannels()

```
MmChannel_t *(*IterateChannels)(const MmFilter_t *filter,
                                int32_t * const cookie);
```

This function should iterate through all the input channels in the *filter*. Set the value of *cookie* to 0 for the first call. *IterateChannels()* returns NULL to indicate that there are no more channels. This function doesn't discriminate between acquired and nonacquired channels. To determine whether a channel is acquired, check its *flags* member. See **MmChannel_t** in the Structure Reference chapter for more information.

If your filter has only one channel, you can use *singleIterateInputChannels()* from the convenience library, which returns the input channel the first time it's called, and NULL thereafter.

```
MmChannel_t *singleIterateInputChannels(const MmFilter_t *f,
                                        int32_t * const cookie);
```

AcquireChannel()

```
int32_t (*AcquireChannel)(MmChannel_t *channel);
```

This function should acquire the *channel*, and set its *flags* member to MM_CHANNEL_ACQUIRED. This locks the channel for exclusive use. You can release it with *ReleaseChannel()*.

This function should return 0 if successful, and nonzero if it can't acquire the channel.

If your filter has only one input channel, you can use *singleAcquireInputChannel()* from the convenience library. This function acquires the channel *c* by setting its *flags* member to MM_CHANNEL_ACQUIRED. This locks the channel for exclusive use.

```
int32_t singleAcquireInputChannel(MmChannel_t *c);
```

ReleaseChannel()

```
int32_t (*ReleaseChannel)(MmChannel_t *channel);
```

This function should release the *channel*. It is then available to be acquired later.

This function should return 0 if successful, and nonzero if it can't release the channel.

If your filter has only one input channel, you can use *ReleaseChannel()* from the convenience library. This function releases the channel *c*, which makes it available for use by another filter.

singleReleaseInputChannel()

```
int32_t singleReleaseInputChannel(MmChannel_t *c);
```

RateFormat()

```
int32_t (*RateFormat)(MmChannel_t *channel,
                      MmFormat_t *format,
                      int32_t *const cookie);
```

This function should rate the multimedia *format*, depending on how well the filter is able to process data in that format. If the format isn't specific enough (some members may be 0 to indicate it can take any value), the function can use the *cookie* to go through the permutations of the formats the filter can process. When there are no more permutations, or if the filter can't process the given format at all, the function should return a rating of 0.

The parameter *cookie* must be initialized to 0 for the first call to the function.

Use *RateFormat()* only for buffered channels, not streamed channels.

If your filter doesn't connect to the buffered output of another filter (for example, if it connects to streaming or unparsed data), you can use the placeholder *noRateInputFormat()* from the convenience library. It simply returns 0.

```
int32_t noRateInputFormat(MmChannel_t *c,
                          MmFormat_t *f,
                          int32_t * const cookie);
```

SetFormat()

```
int32_t (*SetFormat)(MmChannel_t *channel,
                     const MmFormat_t *format);
```

This function should set the *channel*'s format to *format*. When negotiation is performed, the format in the two channels that you want to connect should both be set to the same format, so that they can share buffers.

Use *SetFormat()* only for buffered channels, not streamed channels.

If successful, this function should return 0.

If your filter doesn't need to connect an input channel to the buffered output channel of another filter (for example, if it connects to streaming or unparsed data), it can use the placeholder *noSetInputFormat()* from the convenience library. It simply returns -1.

```
int32_t noSetInputFormat(MmChannel_t *c,
                         const MmFormat_t *f);
```

SetMediaOutput()

```
int32_t (*SetMediaOutput)(MmChannel_t *channel,
                          const MediaOutput *mo);
```

This function should give the *channel* the MediaOutput interface, *mo*, that it can use to retrieve buffer data from its connected output channel. It should also set the *channel*'s *flags* member to MM_CHANNEL_OUTPUTSET.

Use *SetMediaOutput()* only for buffered channels, not streamed channels.

If successful, this function should return 0.

If your filter doesn't need to connect an input channel to the buffered output channel of another filter (for example, if it connects to streaming or unparsed data), you can use the placeholder *noSetMediaOutput()* method in the convenience library. It simply returns -1.

```
int32_t noSetMediaOutput(MmChannel_t *c,
                         const MediaOutput *m);
```

SetInputStream()

```
int32_t (*SetInputStream)(MmChannel_t *channel,
                          const AOStream_t *stream);
```

This function should set the *channel*'s input to *stream*. It should also set the *channel*'s *flags* member to MM_CHANNEL_INPUTSET.

Use *SetInputStream()* only for buffered channels, not streamed channels.

A filter that doesn't connect its input channel(s) to streamed data (for example, if it connects to a buffered channel) can use the placeholder *noSetInputStream()* method from the convenience library. It simply returns -1.

```
int32_t noSetInputStream(MmChannel_t *c,
                         AOIStream_t *sobj);
```

Classification:

Neutrino

See also:

MediaOutput

Extending the Multimedia Framework.

MediaOutput

Functions for filter with an output channel

Synopsis:

```
static MediaOutput media_output =
{
    IterateChannels,
    AcquireChannel,
    ReleaseChannel,
    GetStreamer,
    IterateFormats,
    VerifyFormat,
    SetFormat,
    NextBuffer,
    ReleaseBuffer,
    DestroyBuffers
};
```

Description:

This interface defines the functionality required by any filter that allows a **MediaInput** to get its input data from the filter, either streaming or buffered.

IterateChannels()

```
MmChannel_t *(*IterateChannels)(const MmFilter_t *filter,
                                int32_t * const cookie);
```

This function should iterate through all the output channels in the *filter*. Set the value of *cookie* to 0 for the first call. *IterateChannels()* should return NULL when there are no more channels. This function doesn't discriminate between acquired and nonacquired channels. To determine whether a channel is acquired, check its *flags* member. See **MmChannel_t** in the Multimedia Library Structure Reference chapter for more information.

If your filter has only one output channel you can use *singleIterateOutputChannels()* from the convenience library. This function returns the output channel on the first call, and returns NULL on subsequent calls.

```
MmChannel_t *singleIterateOutputChannels(const MmFilter_t *f,
                                         int32_t * const cookie);
```

AcquireChannel()

```
int32_t (*AcquireChannel)(MmChannel_t *channel);
```

This function should acquire the *channel*. This locks the channel for exclusive use. You can release it with *ReleaseChannel()*.

If successful, this function should return 0.

ReleaseChannel()

```
int32_t (*ReleaseChannel)(MmChannel_t *channel);
```

This function should release the *channel*.

If successful, this function should return 0.

GetStreamer()

```
AOIStream_t (*GetStreamer)(MmChannel_t *channel);
```

This function should return the streamer for the *channel*, if it's a streaming channel.

If your filter doesn't provide streaming output, it can use the placeholder *noGetStreamer()* from the convenience library. It returns NULL.

```
AOIStream_t *noGetStreamer(MmChannel_t *c);
```

IterateFormats()

```
int32_t (*IterateFormats)(MmChannel_t *channel,
                          MmFormat_t *mf,
                          int32_t * const cookie);
```

This function should iterate through the possible output formats (*mf*) for this *channel*. The *cookie* argument is an opaque variable, and should be initialized to 0 for the first call. *IterateFormats()* should return 0 until there are no more formats to iterate through.

You don't need to implement this method for output channels that are streamers. In that case, you can use the placeholder method from the multimedia convienience library: *noIterateFormats()*

```
int32_t noIterateFormats(MmChannel_t *channel,
                         MmFormat_t *fmt,
                         int32_t * const cookie);
```

VerifyFormat()

```
int32_t (*VerifyFormat)(MmChannel_t *channel,
                        const MmFormat_t *format);
```

This function should return a rating for the *format*. This is the last chance for the filter to reject or rate a given format. The best-rated format is chosen.

If your filter doesn't need to have a last-chance verification of a format, you can use *acceptVerifyOutputFormats()* from the convenience library. This method simply accepts the proposed format.

```
int32_t acceptVerifyOutputFormats(MmChannel_t *c,
                                  const MmFormat_t *fl);
```

SetFormat()

```
int32_t (*SetFormat)(MmChannel_t *channel,
                     const MmFormat_t *format,
                     const MediaBufferAllocator *mba,
                     MmChannel_t *mbac);
```

This function should set the *channel*'s data *format* and buffer allocator *mba*. The buffer allocator is the interface/channel that originally allocated the buffer data. It's used to acquire/lock and release/unlock buffers.

If you don't need to implement this method, you can use the placehoder *noSetOutputFormat()* from the convenience library (include **mmconvenience.h**):

```
int32_t noSetOutputFormat(MmChannel_t *c,
                          const MmFormat_t *fo,
                          const MediaBufferAllocator *mba,
                          MmChannel_t *mbac);
```

NextBuffer()

```
int32_t (*NextBuffer)(MmChannel_t *channel,
                      MmTime_t media_time,
                      MmBuffer_t **buffer);
```

This function should return the next buffer. If *media_time* is nonzero and buffers are retrievable based on the timestamp, the function returns the appropriate timestamp's buffer.

Usually, you ask a previous link for its buffer with the given timestamp. The value returned is one of MM_STATUS_*: MM_STATUS_PLAYING if a buffer was returned, and MM_STATUS_PAUSED, MM_STATUS_EOF, etc., if for whatever reason a buffer wasn't returned. If the buffer is timed out (optional), you can return MM_STATUS_TIMEDOUT.

If you don't need to implement this method, you can use the placehoder *noNextBuffer()* from the convenience library (include **mmconvenience.h**):

```
int32_t noNextBuffer(MmChannel_t *c,
                     MmTime_t t,
                     MmBuffer_t **buffer);
```

ReleaseBuffer()

```
int32_t (*ReleaseBuffer)(MmChannel_t *channel,
                         MmBuffer_t *buffer);
```

This function should release the *buffer* so that it can be reused in a future *NextBuffer()* call.

If successful, this function should return 0.

If you don't need to implement this method, you can use the placeholder *noReleaseBuffer()* from the convenience library:

```
int32_t noReleaseBuffer(MmChannel_t *c,
                        MmBuffer_t *b);
```

DestroyBuffers()

This function should free any buffer resources.

If you don't need to implement this method, you can use a pointer to the placeholder *noDestroyBuffers()* from the convenience library:

```
int32_t noDestroyBuffers(MmChannel_t *c);
```

Classification:

Neutrino

See also:

MediaInput.

Extending the Multimedia Framework.

MediaReader

Functions for a reader filter

Synopsis:

```
static MediaReader media_reader =
 {
    SetInputStream
 };
```

Description:

This interface defines the functionality required by any filter that is a graph input point. Typically this is a filter that has no input channels, but does have a media stream that it gets its data from. **MediaReader** defines the following function:

SetInputStream()

```
int32_t (*SetInputStream)(MmFilter_t *filter,
                          const AOStream_t *stream);
```

This function should set the *filter*'s input stream to *stream*.

If successful, this function should return 0.

Classification:

Neutrino

See also:

MediaWriter

Extending the Multimedia Framework.

Synopsis:

```
static MediaSeeker media_seeker =
{
  Seek
};
```

Description:

This interface defines the functionality for seeking a graph to a different timestamp. It defines the following function:

Seek()

```
int32_t (*Seek)(MmFilter_t *filter,
                MmTime_t media_time);
```

This function should seek the *filter* to the timestamp *media_time*. “Media time” is the elapsed media file time, and is of type **MmTime_t**, an **int_64** that stores time in microseconds.

If successful, this function should return 0.

Classification:

Neutrino

See also:

MediaControl

Extending the Multimedia Framework.

MediaWriter

Functions for a writer filter

Synopsis:

```
static MediaWriter media_writer =
 {
    SetOutputStream
 };
```

Description:

This interface defines the functionality required by any filter that is a graph output point. Typically this is a filter that has no output channels, but does have a stream that it writes its data to (such as video or audio output). **MediaWriter** defines the following function:

SetOutputStream()

```
int32_t (*SetOutputStream)(MmFilter_t *filter,
                           const AOStream_t *stream);
```

This function should set the *filter*'s output stream to *stream*.

If successful, this function returns 0.

Classification:

Neutrino

See also:

MediaReader

Extending the Multimedia Framework.

AODeConstructor

Functions to create or destroy a filter

Synopsis:

```
static AODeConstructor media_filter =
{
        Create,
        Destroy
};
```

Description:

This interface defines the functions required to create and destroy a handle (**MmFilter_t** in the case of multimedia filters) for use with other interfaces in the filter.

Create()

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

This function should create and return a handle for the filter. It should initialize the filter, including setting up most of the filter's variables, as well as its channels and their variables. This typically involves allocating and initializing a **MmFilterUser_t** structure, and assigning it to the *user* member of the **MmFilter_t** filter handle.

The *interfaces* parameter is typically not required. It's included in case you need access to other interfaces in your filter.

Destroy()

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

This function should destroy the filter *handle*, freeing any memory allocated for it.

If successful, this function should return 0.

Classification:

Neutrino

See also:

AoResourceAccess

Extending the Multimedia Framework.

AOExtInspector

Inspect a file extension

Synopsis:

```
static AOExtInspector media_filter =
{
    RateExtension
};
```

Description:

This interface defines the functionality required to determine if an inspector's filter can process data from or to a file stream with the given extension. It defines a single function:

RateExtension()

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

This function should return a rating, from 0 to 100, of how sure you are that the filter can process data in a file with the *extension*, where 100 is the best rating. You can use this function before you create a handle for a filter.

☞ As a guideline, multimedia rating functions in existing addons return 80 when they can process data.

Classification:

Neutrino

See also:

AoFormatInspector, **AoMimetypeInspector**, **AoStreamInspector**

Extending the Multimedia Framework.

Synopsis:

```
static AOFormatInspector media_filter =
{
    RateFormat
};
```

Description:

This interface defines the functions required to determine if a given format can be processed with this inspector's addon. It defines a single function:

RateFormat()

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

This function should return a rating, from 0 to 100, of how sure the addon is that it can process data of the given *format*, where 100 is the best rating. You can use this function before you create a handle for a filter.

☞ As a guideline, multimedia rating functions in existing addons return 80 when they can process data.

Classification:

Neutrino

See also:

AoExtInspector, **AoMimetypeInspector**, **AoStreamInspector**

Extending the Multimedia Framework.

AOMimetypeInspector

Rate a MIME type

Synopsis:

```
static AOMimetypeInspector media_filter =
{
    RateMimetype
};
```

Description:

This interface defines the functions required to determine if an filter can process data from a stream with the given mimetype. It defines a single function:

RateMimetype()

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

This function should return a rating, from 0 to 100, of how sure you are that your addon can process data with the given *mimetype*. You can use *RateMimetype()* before you create a handle for a filter.

☞ As a guideline, multimedia rating functions in existing addons return 80 when they can process data.

Classification:

Neutrino

See also:

AoExtInspector, **AoFormatInspector**, **AoStreamInspector**

Extending the Multimedia Framework.

Synopsis:

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

Description:

This interface defines the functions required to access (read and write) resources for a filter.

GetResources()

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

This function should return the 0-value terminated list of resources for the given addon *handle*. Before extracting the values, cast the **void*** pointer to the appropriate type.

SetResource()

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

This function should set the *resource* for the given addon (*handle*) to the *data* specified. You must cast the data pointer to the appropriate type to set the resource values.

If successful, this function should return 0.

Classification:

Neutrino

See also:

AoDeConstructor

Extending the Multimedia Framework.

AOStreamInspector

Rate a stream

Synopsis:

```
static AOStreamInspector media_filter =
{
    RateStream
};
```

Description:

Use this interface to determine if a given stream can be processed with this inspector's addon.

RateStream()

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

This function should return a rating, from 0 to 100, of how sure the filter is that it can process the given *stream*, where 100 is the best rating.

☞ As a guideline, multimedia rating functions in existing filters return 80 when they can process data.

You should use only the **AOStreamer**'s *Sniff()* function in this function. You should never call *Read()* from this function. You can use *RateStream()* to create a handle for the filter with the highest rating.

Classification:

Neutrino

See also:

AoExtInspector, **AoFormatInspector**, **AoMimetypeInspector**

Extending the Multimedia Framework.

Appendix A

The MIDI Configuration File

In this appendix. . .

The MIDI parser, **`midi_parser.so`**, requires a configuration file (**`midi.cfg`**) that describes its runtime environment, including the paths to sound patches and instrument configurations.

The file **`midi.cfg`** describes the runtime environments of the MIDI media filter,including the paths of sound patches, instruments configurations, and so on. Configuration files define the mapping of MIDI programs to instrument files. Multiple files may be specified, and statements in later ones will override earlier ones.

The parser looks for **`midi.cfg`** in the following order:

- In the file named by the **MMEDIA_MIDI_CFG** environment variable.
- In **`$HOME/.media/config/midi.cfg`**
- In **`/etc/config/media/midi.cfg`**

Configuration file format

The **`midi.cfg`** configuration file can have the following statements:

`# this is a comment`

Comment lines start with the **`#`** character.

`dir` *directory*
: The *directory* becomes the current working directory. For example:
`dir /usr/share/media/midi`

`source` *file*
: Reads another configuration file, then continues processing the current one. For example:
`source midi.cfg`
Here, the media filter will load the configuration file
`/usr/share/media/midi/midi.cfg`

`bank` *number*
: Selects the tone bank to modify. Patch mappings that follow will affect this tone bank.

`drumset` *number*
: Selects the drum set to modify. Patch mappings that follow will affect this drum set.

`number` *file [options]*

Specify that the MIDI program number in the current tone bank or drum set should be played using the patch file *file*. Options may be any of the following:

`amp`=*a*
: Amplify the instrument's volume by *a* percent. If no value is specified, it's automatically determined whenever the instrument is loaded.

note=*n*	Specifies a fixed MIDI note to use when playing the instrument. If *n* is 0, the instrument is played at whatever note the Note On event triggering it has. For percussion instruments, if no value is specified in the configuration file, the default in the patch file is used.
keep={**loop**|**env**}	By default, percussion instruments have their loop and envelope information stripped. Strangely shaped envelopes are removed automatically from melodic instruments as well. Use **keep** to prevent envelope or loop data from being stripped. For example, the Short and Long Whistle percussion instruments (General MIDI numbers **71** and **72**) need to have **keep=loop keep=env** specified in the configuration file.

Example

This is a short example **midi.cfg** file:

```
#this is a comment
dir /usr/share/media/midi
source default.cfg
```

Index