Triton

Voyager's Multimedia IO subsystem

Table of Contents

1. Introduction

The multimedia subsystem of Voyager (called Triton) is a plugin-based, extendable, flexible architecture, originally designed for stable playback of any kind of multimedia streams from any kind of source. The system is not yet capable of recording, it's planned for later versions.

The MMIO subsystem has a layered architecture. It means that in order to play back a given multimedia content, the subsystem builds a logical structure of at least five levels. This structure describes the inner structure of the given multimedia content, and makes it possible to divide the processing and playback of multimedia contents into different modular parts, making it easier to find and fix the bugs in plugins. The modular structure also helps so that once support for a new format is added, it will probably work from all the supported sources (like from local file or from an internet stream), and vice versa, a new plugin for a new source will make it possible to play back all the supported formats without any additional code.

The current version of MMIO has the following layers defined:
- Medium
- Channel
- Elementary Stream
- Raw Stream
- Terminator Nodes (Visualization)

These layers and the layered structure are handled internally by the system, and all these are built up from a URL. This effectively means that playing back a given file (or any kind of multimedia content) is as simple as opening a URL with one MMIO API call, and sending it a Play command.

These URLs look this simple:
file://c:/Musics/My_Favourite.mp3
http://local.video.store.com:8080/Horror/Killing_The_Killer.avi
rtsp://radio.com/live.ra

2. Basic overview

2.1. Plugin List, Process Tree, Stream Groups

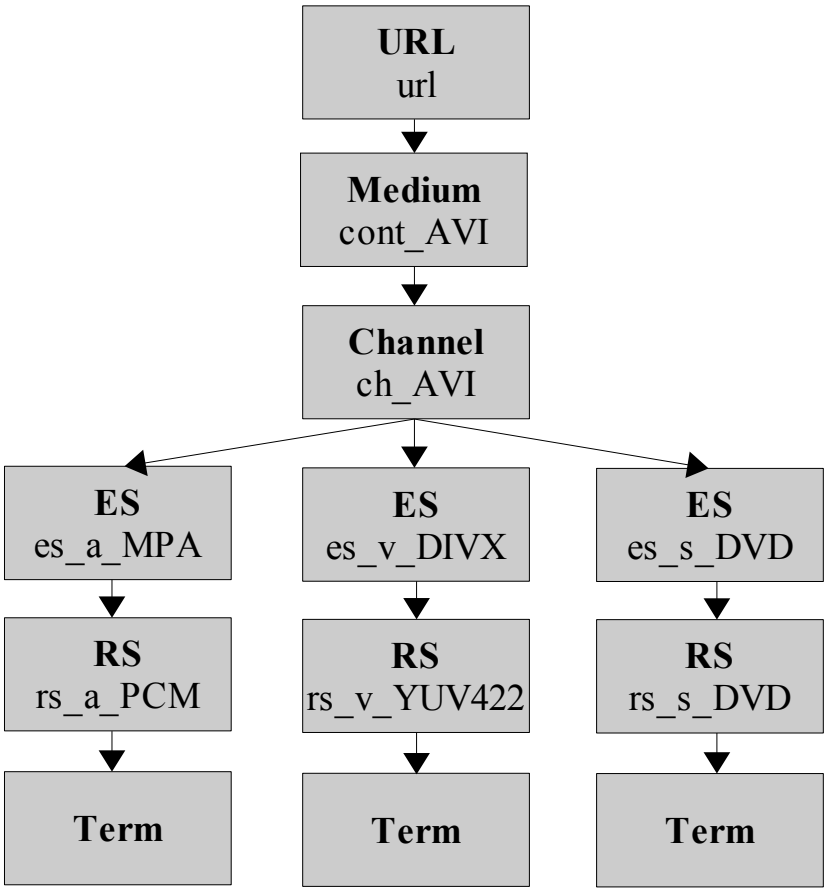
There are three main expressions of MMIO with which one has to be familiar with.

The Plugin List means the list of available MMIO Plugins from where the MMIO subsystem can select plugins to be used for the playback of a given URL. This list can be queried and examined, so the application will get an idea about the capabilities of the system it runs on. Alternatively, it can decide to deregister old plugins and register new ones.

The Process Tree is probably the most important thing of the MMIO subsystem. It is a tree structure, having different kind of nodes connected together, describing the layered architecture of the multimedia content to be played back. This tree is built somewhat automatically by the MMIO subsystem, using the registered plugins of the Plugin List. The top of the tree has a URL node,

containing only the URL string itself, while the bottom of the tree will have Terminator nodes, the "Visualization" nodes for every supported stream of the URL. Here is an example of an imaginery URL and the Process Tree created from it by the MMIO subsystem:

URL: ***http://www.server.com/Films/LOTR_Trailer.avi***



As you can see, this example has an AVI file containing one video stream in DIVX format, one audio stream for the video in MPEG Audio format, and one Subtitle stream for all these, in DVD Subtitle format. All these elementary streams (ES) have a raw stream (RS) descendant, meaning that a decoder has been found for all of them, and all of them has a Terminator (Term) node, meaning that all the decoder outputs can be sent to the screen or speaker, so all of them can be played back. More about node types and their purpose later.

Having three Terminator nodes means that we have three independent streams we can play back. We could send a "Start Playback" command to all the three of these Terminator nodes, which would mean that the playback of all these streams would be started, and we would be able to watch and listen to the movie.

However, usually it's not this simple. Nothing guarantees that starting the playback this way, independently, stream-by-stream, will result in an experience where the video and the audio goes together. There must be "lipsync" as it's usually called, meaning that the audio and the video has to go together in order to have an enjoyable experience.

That's the reason why there is a third expression in the MMIO subsystem to get familiar with: the

Stream Groups.

Stream Groups are logical entities, grouping Terminator nodes together. One can create a Stream Group from the three Terminator nodes of the example above, and send the "Start Playback" command to the Stream Group. The Stream Group itself will take care of the lipsync, and will make sure that all the streams grouped together in it will move and act together.

2.2. Nodes and node types in the process tree

The Process Tree is built from nodes. Every node in the tree has a type, and has a string describing the format which that node can provide. Using the Process Tree example we had before, there is a node which is a **URL node** (its type is URL), and can provide "urls" to any other nodes that are connecting to it.

The MMIO subsystem looked through its list of available plugins, and found a media handler plugin that can handle that URL node. Using that plugin, a new node was created, which is a **Media node**, describing the media contained in the URL. This new Media node was created by the media handler plugin (http handler plugin, in our example), and that plugin also checked what kind of stuff it can provide. It saw that the URL contains a file, which has the extension of ".avi", so it reports that this node can provide an AVI container format to everyone who connects to it. As it's a Media node, everybody will be able to use the node's type-specific functions, like seeking, reading and other Media-specific functions.

The MMIO subsystem tried to "grow" this process tree, so again looked into its plugin list, and found a plugin which can handle AVI container formats (cont_AVI format strings). This is the AVI demuxer plugin. Using that plugin, new nodes of the Process Tree could be built. The first one is a **Channel node**, which describes logical channels in a given container. There is only one channel in an AVI file, but for example the MPEG Transport Stream containers, used by the digital television broadcasting systems have a lot of channels in one container, namely a lot of TV channels, and each of the channels have at least one Video elementary stream and one or more Audio elementary streams. So, the Channel node is to be able to group the available **Elementary Stream Nodes** of a given container.

It's still the AVI demuxer plugin which examined the AVI file format (using the Media node to read and seek in the file) and found out that there are three Elementary Streams (ES) in this AVI file, namely one video stream in DIVX format, one audio stream in MPEG Audio format, and a subtitle stream with DVD-format subtitles. So, the AVI demuxer plugin has created *four* new nodes into the Process Tree.

The ball is at the side of the MMIO subsystem again. It's still trying to grow the process tree, so it's looking for plugins to handle the three new Nodes of the tree. It's lucky again, and based on the plugin list, it finds decoder plugins for all the three formats, namely a DIVX decoder plugin for the es_v_DIVX format, an MPEG Audio decoder plugin for the es_a_MPA format, and a DVD Subtitle decoder/renderer plugin for the es_s_DVD format.

Using these decoder plugins for the corresponding nodes, each of the decoder plugins create one new Node into the tree, a **Raw Stream Node**. The Raw Stream (RS) terminology means decoded elementary streams, so while reading data from Elementary Streams would give encoded/compressed data, reading from Raw Streams gives decoded/uncompressed data, which can

be directly shown on the screen, or sent to the audio card.

The MMIO subsystem does not give up, and still checks if it has plugins to handle the new nodes. It finds a video output plugin, which is capable of displaying YUV422 formatted images on the screen, and connects it to the video RS node. Also finds an audio output plugin which can handle the PCM audio format, and connects it to the audio RS node. The subtitle RS node will be handled by a subtitle visualizer plugin. All these plugins have one thing in common: they "eat" data from the upper layers, but do not provide any data, as they consume them by showing it. That's why they are called **Terminator Nodes**, because they terminate a given branch of the Process Tree.

Now the MMIO subsystem can see that all the leafs of the Process Tree are either Terminator nodes (so there is no task to do with them), or there was no plugin to handle them (none of the nodes are such in our example). So, it gives up growing the tree, the Process Tree has been built successfully.

3. The MMIO API for application programmers

3.1. Forewords

Now that you've reached this chapter, you've either read through the hard part, or you've simply skipped the previous parts. If you've read through the previous parts, you know the basic idea behind the MMIO subsystem, and you'll be able to use it in a more advanced way, customize it for your needs, so the "Advanced usage" chapter will be interesting for you. However, if you haven't read through the previous parts, it's not a problem, the MMIO subsystem can be used very easily without that knowledge too, the "Quick usage" chapter is for you then.

3.2. Quick usage

We tried to create the MMIO API so that playback of a multimedia content can be done quickly, even with sparse knowledge about the MMIO API. So, if somebody does not want to do any special tricks with the multimedia content, just play it back, it's very easy to do with MMIO.

There are only some steps to follow:

- Initialize the MMIO library for the process by calling `MMIOInitialize()`
- Use the `MMIOOpen()` API to open the URL and get a Process Tree for it
- Create a Stream Group containing all the leaf nodes of that Process Tree using the `MMIOCreateEmptyStreamGroup()` and then the `MMIOAddStreamsToGroup()` APIs
- Start the playback of the Stream Group by setting its direction to "play" using the `MMIOSetDirection()` API
- Once the playback is done, destroy the Stream Group using the `MMIODestroyStreamGroup()` API
- Close the opened URL using the `MMIOClose()` API, which will destroy the Process Tree
- Uninitialize the MMIO library using the `MMIOUninitialize()` function

I know, it might seem to be a long list, but belive me, the code itself for it with comments is not

longer either:

```
#include <stdlib.h>
#include “MMIO.h”

int main(int argc, char *argv[])
{
  mmioProcessTreeNode_p pURL;
  mmioStreamGroup_p pStreamGroup;

  /* Initialize MMIO subsystem */
  MMIOInitialize(“.MMIO”);

  /* Open the URL and create the Stream Group from it */
  MMIOOpen(“file://example.avi”, MMIO_NODETYPE_TERMINATOR, NULL, &pURL);
  MMIOCreateEmptyStreamGroup(&pStreamGroup);
  MMIOAddStreamsToGroup(pStreamGroup, pURL);

  /* Start the playback of the contents */
  MMIOSetDirection(pStreamGroup, MMIO_DIRECTION_PLAY);

  /* Now wait for the playback to stop... */
  /* Sleep, or do whatever you want in here */
  /* ... */

  /* Destroy the Stream Group and the URL. */
  /* It will eventually also stop the playback. :) */
  MMIODestroyStreamGroup(pStreamGroup);
  MMIOClose(pURL);

  /* Uninitialize the MMIO subsystem, we don't use it anymore. */
  /* The parameter means that if it detects a memory leak, it should print */
  /* information about it to the screen. */
  MMIOUninitialize(1);

  return 0;
}
```

Of course, this code has no checks for failures anywhere, so I wouldn't call it the most fail-safe program written ever, but you can get the idea.

Also, you may have spotted that you should somehow solve what is not solved here, namely how to wait for the end of the playback. You can put there Sleep() calls, empty loops, an things like that. Sure, it can be done in a more professional way too, MMIO can notify you by sending you events like that, but it's discussed in the next chapter.

3.3. Advanced usage

This chapter will try to describe the possibilities of the MMIO subsystem by going through the steps required to play back a given URL. For greater understanding, it will be described what the given MMIO functions do in the background, what they are created for.

3.3.1. Initializing MMIO

The first step for any application using the MMIO API should be to initialize the MMIO subsystem by calling the `MMIOInitialize()` function.

This function allocates some internal MMIO resources for the calling process, and initializes the memory handling functions provided by the MMIO subsystem. These functions can be reached by including the `MMIOMem.h` header file and by using the `MMIOmalloc()`, `MMIOfree()` and other MMIO-prefixed memory handling functions. It's a simple memory allocation tracking set of functions, so memory leaks of plugins and the MMIO subsystem can be detected and fixed when these are used. Feel free to use these functions if you want to detect memory leaks in your own code too.

There is one optional parameter for this function, which tells where to find the MMIO plugins. If this parameter is not given (it is NULL), then the API will look for the MMIO plugin registry file and the MMIO plugins in the $HOME\.MMIO directory. Otherwise it will use the path given there.

3.3.2. Managing the plugin list

Once the MMIO subsystem is initialized for the given process, it can work on its registry of plugins. The registry itself is a simple text file (at least in the current implementation), containing lines for every registered MMIO plugin. Still, it's not recommended to touch that file by hand, as it may change in the future, so better use the plugin-list handling functions provided by MMIO.

The list of registered MMIO plugins (for the current registry) can be queried by calling the `MMIOQueryRegisteredPluginList()` API, which will return the head of a simple linked list, where each of the elements of the linked list will describe one registered plugin.

As this list is allocated by the MMIO subsystem, there is an API to free this list. Use the `MMIOFreeRegisteredPluginList()` to free the memory allocated for the list.

If you want to register a new plugin or deregister an old one, there are functions for that too. The `MMIORegisterPlugin()` and `MMIODeregisterPlugin()` APIs can be used to modify the plugin-list of the current registry. For these functions, you have to know the filename of the plugin, the internal plugin-name of the plugin, the formats supported by the plugin and the importance factor of the plugin. The developer of the plugin should know these, so when he/she creates an installer for his plugin, he can use the `MMIORegisterPlugin()` function for that.

To make it even more easy to install new plugins, plugins (optionally) can contain information about what is their internal name, what formats they support, and what is their suggested (default) importance factor. If a given plugin contains this information, it can be queried using the `MMIOQueryPluginsOfBinary()` function. It will give back a linked list, containing the list of plugins implemented in the given binary plugin file. Please note that it can happen that one binary file contains more than one plugins, as the plugin internal names can differ, so there is nothing against putting more than one plugins with different internal names into one binary file.

The linked list that is created by the `MMIOQueryPluginsOfBinary()` function contains every

information that is required for the plugin registration, so even unknown plugins can be registered this way. The list can be freed by the `MMIOFreePluginsOfBinary()` function, when it's not needed anymore.

3.3.3. Opening an URL

The most general descriptor for the MMIO subsystem is a URL. The URL is a simple text in the usual form: protocol://resource_name. The MMIO subsystem creates Process Trees from the URLs by creating first a URL node, and then looking for plugins that can handle the leaf nodes until every leaf node is a Terminator node (or no plugin can handle that node).

It should be noted here that every Process Tree has a parent node, because there is a Root node in MMIO. Every URL node must be the child of this Root node. This Root node can be got by calling the `MMIOGetRootNode()` API, and walking through the children of this node one can get the list of currently opened URLs and their Process Trees.

If one is masochist enough, the building of a Process Tree based on the URL can be done by hand, using the low-level functions, like `MMIOLoadPlugin()`, `MMIOUnloadPlugin()`, `MMIOInitializePlugin()`, `MMIOUninitializePlugin()`, `MMIOExamineNodeWithPlugin()`, `MMIOFreeExamineResult()`, `MMIOLinkPluginToNode()` and `MMIOUnlinkNode()`. However, I don't recommend doing this for anything else than debugging a newly created plugin, as this requires the deeper knowledge of how the MMIO plugin system works.

It's much easier to leave all this to the MMIO subsystem. Use the `MMIOOpen()` function to process a given URL and create a Process Tree from it, which is already linked to the Root node. This function uses the registry and all those functions mentioned above, and tries to build the Process Tree based on those informations as much as possible.

It is also possible to mix the two ways, to let the MMIO subsystem build the Process Tree to a given level, for example until it reaches the Terminator nodes, but put there a given Terminator node by hand using those low-level APIs mentioned above. It can be done by giving an `iOpenLevel` parameter smaller then MMIO_NODETYPE_TERMINATOR. This way, `MMIOOpen()` will return when reaches the given level, without further growing the Process Tree.

It's possible that the `MMIOOpen()` function will not know what to do when it reaches a given node. Imagine, if there are more than one plugins registered in the system that can handle a given format. In this case, one has to decide which one to use for the given node. The `MMIOOpen()` API creates a list of possible plugins for these nodes. This list is ordered by the “importance” factor of the plugins, which is a simple number in the registry, attached to the given plugin. The `MMIOOpen()` will try to use the plugins for the problematic node in the order of the list, and the first plugin which accepts the node will be used in there. To make this decision better controllable, the API has a parameter called `pfnSortList`, which (if not NULL) points to a function to be called by `MMIOOpen()` to re-sort the list of plugins to try for a given node. This way, one can set a callback in there to always move its own decoder plugin to the top of the list, no matter what “importance” is set to it in the registry, or one can program some kind of heuristics into his/her own callback to select the proper video renderer plugin for the current hardware configuration.

The very same callback is used when a plugin offers multiple output formats for a given node, so one would have to choose one from them. The callback function can reorder the list, this way influence the Process Tree creation process. Please check the description of the `MMIOOpen()` function for more information about this.

3.3.4. Walking through the process tree

Having built the Process Tree, we can examine the contents of the tree in more ways. The quickest and simplest way is to let MMIO print the structure of the process tree to the screen. The `MMIOShowProcessTree()` API is exactly for this purpose. It comes very handy for debugging purposes, to see how deeply the tree could be build and where it failed further growing.

Of course, one could walk the tree by hand, using the `pParent`, `pFirstChild` and `pNextBrother` fields of the `mmioProcessTreeNode_t` structure. However, there is a much easier method to do this. The `MMIOWalkProcessTree()` API can be used to walk through each of the nodes of the process tree and call a given callback function for them. This way, it's easy to collect information about the process tree, find given nodes in it, without knowing the internal structure of the nodes.

3.3.5. Creating stream groups

Once we have a Process Tree, we possibly have one or more Terminator nodes in that tree. Those nodes can be instructed to do the playback of their streams, to seek (if supported) in the stream, and things like that.

However, starting the playback of an audio stream and a video stream will not mean that they will be played back in perfect sync. Most probably there will come some glitches in the playback, and the streams will simply go out of sync.

To solve this, MMIO has the concept of Stream Groups.

A Stream Group is like an intelligent container, where Terminator nodes can be put. The Stream Group will take care to keep these Terminator nodes in sync by collecting their position information all the time, and reporting them back if any of them are out of sync. The Stream Group always has one Terminator node which is the boss. This is called the Main Stream, and this is the one to which all the other streams will be synchronized.

By default, the Stream Group has a bit of intelligence, and as Streams (Terminator nodes) are added to the given Stream Group, it always selects the Main Stream based on an algorithm, so that if there is at least one audio stream then it will be the Main Stream (because it's always better to synchronize to an audio stream by speeding up or slowing down the video stream, than making hickups in the audio stream to catch up with the video).

So, to be able to do a perfect playback of a multimedia content, one has to create a Stream Group using the `MMIOCreateEmptyStreamGroup()` API, and then add the Terminator nodes of the streams to be played back into this group using the `MMIOAddStreamsToGroup()` function. Please note that this function is designed to be recursive, so it one wants to add every Terminator

node of a given Process Tree to the Stream Group, it's possible to call this function only once, for the root of the Process Tree, and it will walk through the tree to collect all the Terminator nodes.

It's also possible to remove Streams from the Group by using the `MMIORemoveStreamsFromGroup()`, and if somebody is not satisfied with the Main Stream selected by the function, it can be changed by using the `MMIOSetMainStreamOfGroup()` function.

There are currently no functions to get the current Main Stream of a Group, or to check which Streams are part of a given Stream Group, but these informations can be easily got by directly accessing the mmioStreamGroup_p type (the Stream Group handle), and looking into the structure.

Once a Stream Group is not in use anymore, it must be destroyed by using the `MMIODestroyStreamGroup()` function. It will stop the playback if it's ongoing, and remove every Terminator node (every Stream) from the Stream Group, and free all the resources allocated for the Group.

3.3.6. Setting direction and position of Stream Groups

The playback of a given multimedia content can be started by setting the direction of a Stream Group to MMIO_DIRECTION_PLAY, using the `MMIOSetDirection()` API. This will result in setting the direction of every Terminator node of the Group.

The playback direction of a Stream is an integer number, where the value 1000 means the normal speed playback, 2000 the double speed playback and so on. It's also possible (theoretically) to have half speed playback (value 500), and even more, to play backwards (negative values).

However, not every multimedia format and codec supports all the directions. If an unsupported direction is requested, the `MMIOSetDirection()` API will return with the proper error code.

The current playback direction of a Stream Group can be queried by using the `MMIOGetDirection()` API.

The same way, one can jump to a given position of the multimedia content. Some call it seeking, we call it “setting the position of the playback”. The position is in milliseconds, and can be set by the `MMIOSetPosition()` API. The current position can be queried by `MMIOGetPosition()`, and the length (the maximum position) of the current streams can be got by calling the `MMIOGetLength()` API.

It must be noted here again, that not every content can tell a length, or set position of the playback. For example, a streamed video throught the internet might not be able to report its length, and also, it might not be able to seek in the audio or video stream. The functions return the proper result code in these cases.

3.3.7. Events of Stream Groups

The Stream Groups can report events to the owner, like when the direction of the playback changes (e.g. It suddenly stops because of a problem), when the stream reaches its end, when there is an error in the stream, and things like that (please check the MMIO_STREAMGROUP_EVENT_ defines in `MMIO.h` header file).

One can subscribe for such events by using the `MMIOSubscribeEvents()` API, and giving a mask for the events in interest. Then the `MMIOGetEvent()` API can be used to wait for any of the subscribed events to arrive, with a given timeout, of course.

This way it's possible to write code that waits for the end of stream event, or which continuously prints the current position of the playback (by processing the position info event), and so on.

3.3.8. Uninitializing the MMIO subsystem

Once a process is done with using the MMIO subsystem, it can call the MMIOUninitialize() API. This will free all the resources allocated by the MMIO subsystem.

It has one parameter, which tells if the function should report the memory leaks it detects, if it detects any. If the parameter is True, and the function detects a memory leak, it will print the details of the allocated memory chunks to the screen, so one can debug the faulty plugin or application.

3.4. Simple Decoding with the MMIO subsystem

Even though the MMIO subsystem was designed and created for easy playback of media, one can always use it in other ways, thanks for its modular structure.

If all the application wants from MMIO is to decode a given media, it's easy to do.

First of all, the Process Tree has to be built for the URL. However, as we don't want the decoded audio or video frames to be sent to the outputs, it's enough to build the Process Tree until we reach the Raw Stream nodes. In other words, we can use the `MMIOOpen()` API as always, but give it an open level of `MMIO_NODETYPE_RAWSTREAM`. This way, the MMIO subsystem will not try to look for and use any output plugins, it will only build the Process Tree until it reaches the Raw Stream nodes.

To get the required Raw Stream node from the Process Tree, one can walk the tree to search for it using the `MMIOWalkProcessTree()` API, or choose the harder way to walk the tree by hand, using the tree link fields of the `mmioProcessTree_t` structure. No matter which method is choosen, the required Raw Stream node can be found easily.

Once we have it, we can ask the Raw Stream node directly to give us the raw (decoded) frame, just like the Terminator nodes would do it, but instead of being a Terminator node and displaying it, we can save it, or re-encode it into another format, or whatever is the task to do.

To ease this process, another utility function was created. It's called `MMIOGetOneSimpleRawFrame()`, and all it does is that it sets the playback speed in the given

Raw Stream node to `MMIO_DIRECTION_PLAY` (this is needed so that if this API is called more than once, then the stream will give us back different frames, not the very same first frame all the time) and then gets one frame from the Raw Stream into the caller's buffer.

This API works well for decoding images or audio data, but it will most probably fail for the more sophisticated decoder plugins, for example for video decoder plugins that support B-Frames. The reason is that MMIO decoder plugins are allowed to "swallow" the buffers in which they should put the decoded data and ask for another buffer (temporarily), if needed. Now, for supporting B-Frames in video decoders, the most effective method is to use this feature, but this would make this API much more complex, so it wouldn't be called `MMIOGetOneSimpleRawFrame()` anymore.

4. Creating Plugins for the MMIO subsystem

This chapter is for the ones planning to get familiar with the internals of the MMIO subsystem, planning to create a new plugin for it, or just simply are interested in the small but bloody details.

4.1. What is an MMIO Plugin

The MMIO plugins are platform-dependent binary files. It usually means that they take the form of some kind of a dynamically loadable module the given platform supports, for example they are `.DLL` files on OS/2 or Windows, `.so` files on Linux, and so on.

These binary files have to have some well defined entry points. These entry points are queried by the MMIO subsystem when the binary file is loaded into memory, and these entry points are used to call into the plugin. Most of these entry points are mandatory to have, some are optional. More about these will come later in this documentation.

The plugins must be so that they have to be instantiable. It means that one of the mandatory entry points is there to "Initialize" the plugin, which should allocate a memory area big enough to work in, and return that pointer as a new instance handle. This handle (pointer) will be passed on to every call to the plugin from that point, so the plugin can work in that area. Once the plugin is not needed by the system, the "Uninitialize" entrypoint is called, which should clean up every resources used by that instance, and free the memory area pointed by the instance handle.

This way, the plugins are instantiable, and they can be used more than once in the same time, in the same process.

One of the tasks of the plugins is to work on the Process Tree: check it, create new nodes based on the information collected from the existing parts of the tree and based on the capabilities of the plugin, and implement the node-specific functions for the nodes it has created.

To make this a bit more clear, here is a real-live example for that: The MPEG Audio Decoder plugin has to check the existing part of the tree to see if that really contains an MPEG Audio Elementary Stream, and if it does, it should create a new Raw Stream node, and implement the node-specific functions for that node, namely to decode the audio data (to create "Raw Stream" from an "Elementary Stream").

To ease this process, the MMIO subsystem has quite some Plugin Support functions, offered for usage for the plugins. This plugin helper API has its function names starting with `MMIOPs`, more about them later.

As the MMIO Subsystem was made to be portable to different systems, it has its own porting layer, called the Triton Porting Layer (TPL). This porting layer can also be used by the plugins, and it's a good idea to make use of them if one wants to create portable plugins. Of course, it's not always a real possibility, for example the system-specific plugins like audio output or video output plugins will always contain system-specific code, but for most of the other plugins it's a great help.

4.2. The Triton Porting Layer

For the MMIO Subsystem to be portable, it was needed to create a small, lightweight porting layer, which hides the most commonly used platform-specific stuffs in a cross-platform manner. The Triton Porting Layer only contains things which are really used by the MMIO Subsystem, so it's not bloated and it's really small.

The Triton Porting Layer deals with the following topics:

- Dynamic Executable Loader
- Message Queue
- Mutual Exclusive Semaphore
- Simple Event Semaphore
- Counting Event Semaphore
- Threading and Scheduling
- High-Resolution 64bits Timer

All the APIs of the porting layer has the prefix `tpl_`, so they are easy to spot them.

4.2.1. Dynamic Executable Loader

The dynamic executable loader is the thing which can load binary executables on the current platform. This is the one to use to load `.DLL` files on OS/2 or Windows, or to load `.so` files on Linux.

Once a binary executable is loaded, a handle is returned for it. That handle can be used to unload it, or to query the address of its public entry points or exported symbols, by name.

4.2.2. Message Queues

Message queues are one of the most popular things to use for thread-safe messaging. These are basically First In First Out (FIFO) queues, implemented in a thread-safe manner, so they can be used to send notification or command messages from one thread (or more threads) to a given thread.

Once a message queue is created, data can be sent into the queue. Data can be waited for, and once

somebody sends data into the queue, that data will arrive to the waiter for further processing. This makes it a very effective messaging model for small data messages, most notably for command queues and notification queues.

4.2.3. Mutual Exclusive Semaphores

Mutual exclusive semaphores (or mutex semaphores for short) are usually used to protect a given variable or a given area of code from concurrent access.

A mutex semaphore is done so, that it can be requested and then released, but it can be requested only once at one time. This way, if somebody wants to protect a given variable from concurrent access, it should always request a given semaphore before working with the variable, and release it when the tampering with the variable is done. As the mutex semaphore can only be owned once at a time, it's guaranteed that no more than one thread will be able to use that variable at the same time, assuming that the access to that variable is properly protected by the same mutex semaphore in the code.

4.2.4. Simple Event Semaphores

Simple event semaphores are semaphores that can be either in a posted or reset state. The semaphore can be waited for being posted, it can be posted, and it can be reset.

Waiting for an event semaphore to be posted takes no CPU time, so this makes it a very effective tool for waiting for events in the code.

4.2.5. Counting Event Semaphores

The counting event semaphore is an extended version of the simple event semaphore. It has the same features, but instead of being able to reset it, it has a counter of posts, which is increased by every post and decreased by every wait. The wait function blocks the CPU only if the post count is zero.

This tool is usable for cases where it's not just the fact that the event has occurred that is important, but it's also important to know how many times it has happened, and a given code should be executed for every instance of the given event.

4.2.6. Threading, Scheduling

The porting layer has support to start, stop and wait for threads, this way making it possible to create cross-platform multithreaded applications. The priority of the thread can also be set to five predefined priorities (Idle, Low, Regular, High, RealTime).

These threads can also have influence on their scheduling, in the way that they can sleep for the given number of milliseconds, or they can give up their remaining timeslice.

4.2.7. High Resolution Timer

The porting layer implements a high resolution timer. This is an always-running timer with a high resolution. The resolution depends on the current computer on which the code runs, but it can be queried. The current value of the counter can be queried anytime.

The implementation is most probably using the CPU's internal cycle counter and the CPU's frequency, and it returns it as a 64bits value.

High resolution timers are very handy for cases when very precise timing is needed, like in case of playback of multimedia content.

4.3. Public entry points of the plugin

The MMIO plugins must have some well-defined, exported entry points. Some of these public entry points are mandatory to be present, some are optional. Once a dynamically loadable module has all the mandatory entry points exported, it's treated to be a valid MMIO plugin.

These entry points are identified by their names, so it must be taken care to export them by name on systems where there are other ways for exporting functions (like on OS/2, where one can export functions both by name and by ordinal). Making it short, once a binary file can be loaded by TPL, and all the mandatory public entry points can be queried for it, the MMIO subsystem will be able to use that file as a valid plugin.

4.3.1. Optional public entry points

The current version of the MMIO subsystem defines two optional entry points for MMIO plugins. These two of entry points are to make it possible for the binaries to tell MMIO automatically what kind of plugin or plugins they implement. These are the methods that are used by the `MMIOQueryRegisteredPluginList()` and `MMIOFreeRegisteredPluginList()` APIs to implement their behaviour.

These two entry points must have the names `MMIOQueryPluginInfoForRegistry()` and `MMIOFreePluginInfoForRegistry()`. If both of these are implemented and exported, they will be called by the two MMIO functions mentioned above to get a list of plugins implemented in the binary.

The first one of these functions is to return the internal name of the plugin, the list of supported formats by the plugin, and the proposed importance number for the plugin. The second one is to clean up and free the results allocated by the first call.

4.3.2. Mandatory public entry points

The mandatory entry points are the ones which implement the real plugin functionality. They must be present in every plugin, and all of them must be present. Currently, there are seven mandatory entry points for the plugins.

As it's possible to implement more than one plugins in one binary file, these mandatory plugin entry points have a prefix in their name. The exported functions with the same prefix belong to the same plugin. This prefix is the internal name of the plugin, which is set in the registry in plugin registration time. It's either set by hand (to the correct value, of course, by the plugin developer) or automatically using the information returned by the two optional entry points described above.

For example, let's say that one has a binary file implementing a file reader plugin. If the internal name of the file reader plugin is “file”, then the plugin has to export the following functions:

- `file_GetPluginDesc()`
  This function is to return general information about the plugin, like plugin name, version, author, or any kind of a free-form string.

- `file_Initialize()`
  This function is to instantiate the plugin. It should allocate a memory area for the plugin and return the pointer to that area. This pointer is treated as the instance handle for this plugin, and passed to every other plugin functions. All the state variables and other per-instance variables of the plugin must be in this area, making it possible to run more than one instance at the same time in the same process.

- `file_Examine()`
  This function should examine a given Process Tree node, and return the list of supported formats for that node, if any. This is used by MMIO to check if the given node and its parents can be used by this plugin or not.

- `file_FreeExamineResult()`
  This function is usually called right after the previous one. Its task is to free the list allocated by the previous call.

- `file_Link()`
  Once MMIO makes sure that the plugin can handle a given Process Tree node (by checking the result of the previous Examine() call), it selects the plugin for that node, and calls this entry point, telling the plugin to link itself to the node. This function should build additional nodes originated from the node to link to, and fill the node structures with data about the new nodes. Basically, this is the function which makes the Process Trees grow when needed.

- `file_Unlink()`
  When the playback is done, and the plugin is not needed anymore, the Process Tree is being destroyed by MMIO. This is the time when every plugin which created new nodes should destroy its nodes. This is the time when this entry point is called to unlink from the given node. The function should delete all the nodes behind the given node, as those are the nodes that were created by the plugin itself. All the resources allocated for the nodes should be freed here.

- `file_Uninitialize()`
  As we has an Initialize() function to instantiate the plugin, we must have the pair for it. This is

the function which should clean up a given instance. In practice, it means that every resource allocated for that instance should be freed, being it memory, threads, semaphores or any other resource.

4.4. Node types and the type-specific functions

It's the job of the Link() function of the plugin to create the new nodes, according to the information it gathered from the upper level nodes of the Process Tree.

The nodes contain information like the parent, children and brother nodes, information about the owner plugin, the stream group in which it belongs to, and most importantly the type of the node and the type-specific information.

There are quite some node-types, and every node-type has a different set of data (information about the node) and functions. These are the type-specific data and the type-specific functions for the nodes.

It's easy to create and link new nodes and set the common (non-type-specific) informations of the nodes, especially because there are helper functions created for these common tasks. Please see the description of the Plugin Support functions later in this documentation. However, the type-specific are has to be allocated and filled by the given plugins themselves.

4.4.1. URL nodes

The URL nodes are the most top-level nodes. These are always created and destroyed by the MMIO subsystem itself. This node has the URL itself. It has no type-specific function, only a data element containing the URL.

This is the parent node for the Medium nodes. This node is created by the `MMIOOpen()` function itself, and the first plugin (most probably a media handler plugin) gets this node to examine and link to.

4.4.2. Medium nodes

The Medium nodes are created by the media handler plugins. When they examine the URL of the URL node and they are asked to link to it, they create a node of this kind. According to the protocol part of the URL, they set the media-capabilities field of the type-specific area of the node to describe for the other plugins what can be done with this node.

The best Medium nodes (if one can say so) can read, seek and tell the media. However, there might be cases when a medium cannot be seeked, or the current position cannot be told. The correct capabilities are set in these cases.

The function pointers to the Read(), Seek(), Tell() and ReadPacket() functions are also contained in the type-specific area.

You might be interested why we have both a Read() and a ReadPacket() function in here. Now, the first one treats and reads the medium as if it would be a stream. The application asks for some bytes, and it returns them if it can. However, there are some media types where the data arrives in packets, and the packet borders must be preserved. The simple Read() would not be able to report these packet borders for these cases, so a special ReadPacket() is introduced to be used then, which only returns as much bytes as much there was in the given packet, even if more than one packets have already arrived.

4.4.3. Channel nodes

This chapter is to be done.

4.4.4. Elementary Stream nodes

This chapter is to be done.

4.4.5. Raw Stream nodes

This chapter is to be done.

4.4.6. Terminator nodes

This chapter is to be done.

4.5. Plugin Support functions are for your help

This chapter is to be done.

5. Programming guidelines

This chapter is to be done.
Use MMIOmem.h and the Triton Porting Layer, where possible, so the plugins will be portable, memory leaks will be detectable and trackable. Use the Plugin helper functions, they're handy.

6. API Reference

6.1. Triton Porting Layer API Reference

This chapter is to be done

6.2. MMIO Subsystem, Plugin Support API Reference

This chapter is to be done

6.3. MMIO Subsystem, General API Reference

This chapter is to be done