Isomet Modular Synthesiser (iMS) API

v1.4.2

Generated by Doxygen 1.8.10

Wed Nov 1 2017 15:36:36

Contents

Chapter 1

iMS Library and API Documentation

1.1 Contents

1.2 Overview

The iMS (Isomet Modular Synthesiser) System represents an expansive range of hardware devices and software interfaces designed to permit the rapid development and integration of Acousto-Optic (AO) technology into end-user systems.

By modularising the hardware components and supplying a well defined application interface, the systems integrator only needs to specify the required iMS hardware configuration, select an RF amplifier and AO device, and begin writing application software at a high level of abstraction. All the fundamental details of underlying host communications protocols and I/O data formats are handled internally by the software leaving you to concentrate on the elements of the design that matter to your application.

Application software communicates with the iMS System through the Application Programmer's Interface (API) which is supplied by Isomet along with compiled library objects for a number of different platforms, documentation and examples as part of the iMS Software Development Kit (SDK). The API can also be used without accompanying iMS hardware to develop applications that create iMS compatible data such as Image Files and Compensation Tables.

In addition to the API and accompanying C++ library, the SDK also includes software utilities and script wrappers that allow you to get up and running quickly without having to write any software at all, or to set up complex tasks using a few simple scripting commands.

This documentation covers all of the files, classes and other constructs made available to the application programmer through the API. It also provides some background detail on the key concepts and software architecture of the library to help facilitate understanding.

There are numerous code examples given throughout the documentation to explain how to perform particular operations. You are allowed and encouraged to copy these examples as a basis for developing your own applications.

1.3 What's Included

The core of the Software Development Kit is the C++ iMS library and API. All interaction with iMS hardware ultimately passes through this API. However we have also provided a number of other software utilities and wrappers that allow you to use the iMS System at a higher level of abstraction.

Included in the SDK are:

- The core iMSLibrary binaries for a number of different platforms and toolsets.
- Accompanying C++ header files for application interface
- `iMSNET` An experimental .NET assembly written in C# that wraps the core library and permits user application development in any .NET language targetting the .NET Framework
- ims_hw_server is a command line daemon type process that can handle all communication with an iMS system, decoupling it from user application business logic. A gRPC streaming interface connects the server to application software, either on the same host or across a network.
- iMS Studio is a full featured GUI front end application that can be used to create Images, Tone Buffers and Compensation Functions and play them on an iMS system. This is often a good starting point for users wishing to explore the capabilities of an iMS before starting development of custom software.

1.3.1 Application Programmer's Interface

For full control of an iMS System, we encourage users to develop their software applications in C++ using the iMS defined API and linked against the supplied library files. The libraries are extensively used and tested and provide access to every available feature on the hardware.

Every class and function in the API is documented within this documentation set and Isomet are happy to assist your development through examples, walkthroughs and design assistance or consultancy.

1.3.2 .NET Wrapper

A .NET wrapper (iMSNET.dll) is supplied that encapsulates the C++ binaries and provides access to nearly all functionality in a convenient format for development of .NET graphical applications on Microsoft Windows.

Most classes and functions exposed through the .NET wrapper have the same or similar naming and functionality to those presented by the C++ library .dll although there are some differences to both.

At present, support for iMS application development using the .NET wrapper library is good but experimental and documentation is limited. See, however .NET Wrapper

1.4 Platform

The iMS software library and API has been written purely in native ANSI-C++ with some use of features introduced in C++11 (ISO/IEC 14882:2011), including the C++ Standard Library. There is no use of features associated with the updated C++14 or later specifications.

There are no dependencies on external dynamic libraries other than those supplied as part of a normal OS distribution.

The compiled library code is currently supplied as a Windows-only .dll dynamic library. It is sufficient in your application development to reference the accompanying .lib file in your linker script and ensure that the .dll can be found by the executable at run-time either by placing it in the same location or at a location discoverable in the %Path% environment variable. A walkthrough of this process is given in the SDK tutorials.

The library has been compiled and released using Microsoft Visual Studio 2013 (v120), 2015 (v140) and 2017 (v150) for Windows 7 Professional 32-bit and 64-bit and also using Microsoft Visual Studio 2015 (v140) and 2017 (v150) for Windows 10 32-bit and 64-bit. We do not recommend using earlier versions of Visual Studio as we cannot guarantee their usage and we explicitly do not support earlier versions of Microsoft Windows (Vista/XP/2000 and earlier). You may use alternative IDE development tools at your own risk although if you contact us we may be able to assist with any issues discovered. Windows 8/8.1 support is believed to work but is unverified.

	Visual Studio 2013 (v120)		Visual Studio 2015 (v140)		Visual Studio 2017 (v141)		QCC 4.7.3	
	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	X86	armle-v7
Microsoft Windows 2000/XP/Vista or earlier	✗	✗	✗	✗	✗	✗		
Microsoft Windows 7 Professional	✓	✓	✓	✓	✓	✓		
Microsoft Windows 8/8.1	⚠	⚠	⚠	⚠	⚠	⚠		
Microsoft Windows 10 Professional	✗	✗	✓	✓	✓	✓		
QNX Neutrino 6.6							✓	✓

Table 1 Toolset Version Compatibility Table

Cross-platform support for other OSes is underway. Support for the QNX Neutrino 6.6 RTOS is available.

Included in the SDK are .dll's for 32-bit and 64-bit applications along with standard (release mode) libraries for your application deployment and debug libraries (suffix _dbg.dll) for your own development.

Chapter 2

Software Library Architecture

2.1 Software Library Architecture

Figure 1 below provides a conceptual view of the architecture of the iMS Software Library. It does not represent a structured class diagram of the internal library detail but should give an initial feeling for how the library is constructed.

iMS Software Library Architecture.png

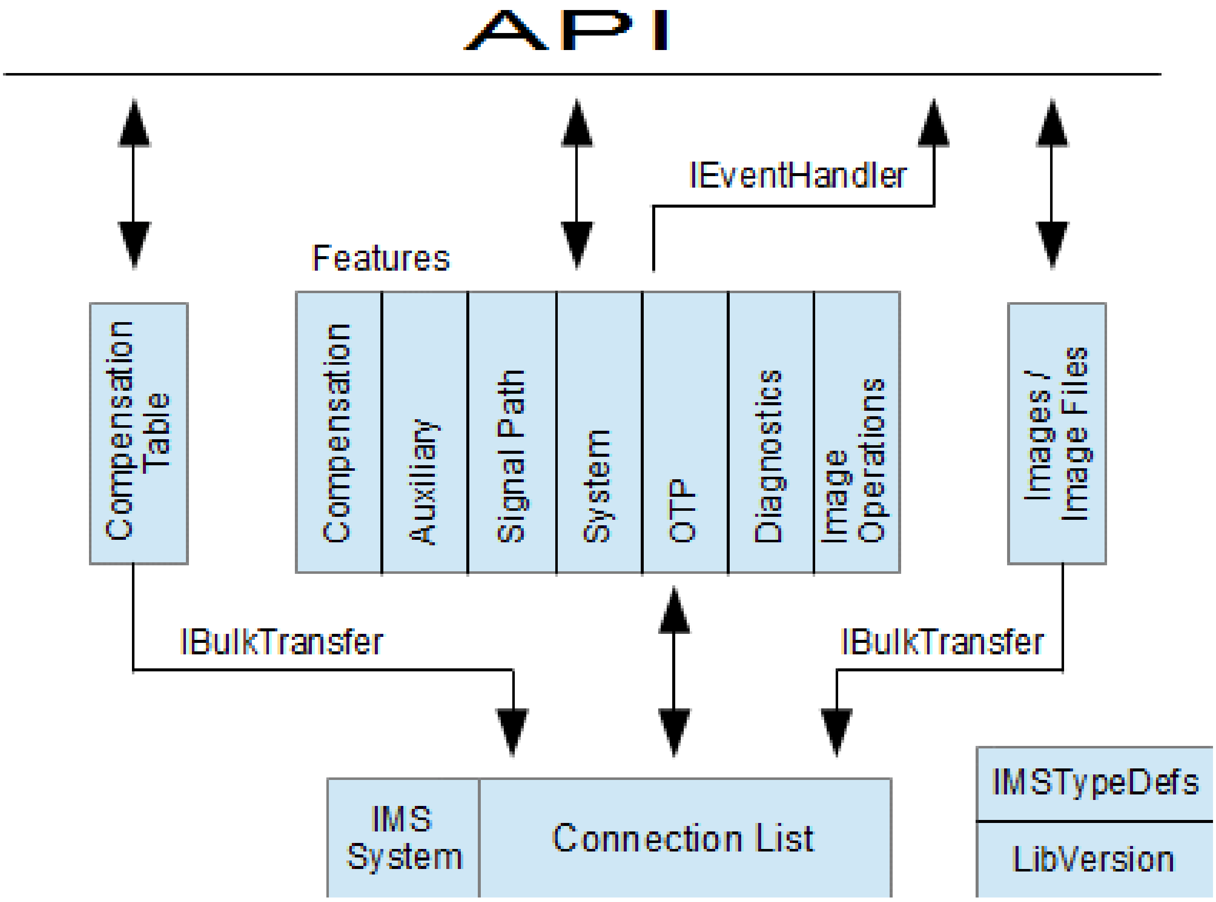


Figure 2.1: (Figure 1) Library Architecture Overview

2.1.1 Connection List and IMS System

These two modules represent the lowest layer of communications with the iMS hardware present in the library. ConnectionList maintains an internal knowledge of how to interact with the hardware across all supported connection types. This is abstracted so that other parts of the library can communicate with the hardware irrespective of the connected interface.

IMSSystem maintains a working knowledge of the capabilities, configuration and detail of the connected system so that both the application and internal library elements can determine what it can and cannot do and how to reinterpret data in a format that is suited to the hardware.

2.1.2 Features

The features are a suite of functional blocks that are a bit like the library books of the library. Each knows how to perform a particular function that is associated with the overall iMS functionality.

- `Compensation` : functions for storing and downloading amplitude, phase and synchronous data tables
- `Auxiliary` : functions for additional features that are not associated with the core RF synthesis and output capabilities, e.g. GPIO
- `Signal Path` : functions for controlling the RF signal path, including fixed tone calibration modes
- `System Functions` : functions for enabling and controlling system and communications features
- `FileSystem` : functions for writing and reading non-volatile memory
- `Diagnostics` : functions for monitoring system status, e.g. temperature, VSWR and current draw
- `Image Operations` : functions for downloading images, image sequences and configuring, starting and stopping playback

2.1.3 Compensation Tables

iMS Synthesisers contain a set of frequency-addressed look-up tables for applying various compensations to the RF signal output. The tables are indexed by the nearest programmed frequency to the current frequency being output by that RF channel. Table entries are linearly spaced in frequency starting from the lowest frequency supported by that Synthesiser up to the highest frequency supported. The number of entries in the table is hardware specific.

Compensations available include:

- Amplitude: a value between 0 and 100% for modifying the output amplitude according to frequency. Used for compensating for AOD efficiency as well as filter attenuation and DDS roll-off.
- Phase: 0 - 360 degrees. Represents the per-channel phase difference applied to enable beam steered applications. Value represents the amount of additional phase applied from the previous channel to this one (channel 1 is unmodified).
- Sync Analog: A value between 0.0 and 1.0 that can be output on one of the synchronous DAC outputs (updated in step with the RF image point data).
- Sync Digital: A binary value that can be output on the synchronous digital outputs (updated in step with the RF image point data).

2.1.4 Images / Image Files

This set of classes are used for creating the core RF image data that is at the heart of the Isomet iMS concept. An Image File can contain one or more iMS images plus a sequence table that defines the default order for playing back images, the conditions for triggering images, numbers of repeats, image point rate, delays and other vital data. An image itself is composed of image points (from 1 up to many millions), each point consisting of up to 4 RF channels with independent specification of frequency, amplitude and phase and optional synchronous output data.

2.1.5 Utilities

Additional Utility functions exist for supplying the version information for the API (LibVersion) and useful type definitions for frequency (in kHz and MHz), amplitude (as a percentage) and phase (degrees from 0 to 360) amonst others.

Chapter 3

Cross Language Support and Scripting Wrappers

We recognise that not all users prefer to develop their applications in C++ and for convenience we also supply a number of wrappers and support libraries for writing software in alternative languages including scripting languages.

3.1 .NET Wrapper

The .NET library iMSNET.dll wraps around the C++ library, exposing many of the same functions or providing a thin layer to translate C++ concepts such as std::list into .NET compatible frameworks such as IEnumerable. Most classes and functions are recognisable from the C++ API documentation - one noticeable exception is the Image class which is renamed iMSImage to avoid confusion with the .NET entity System.Image. Just as with the C++ library, all classes and functions are contained within the same namespace iMS.

3.1.1 Initialisation

One very important point of note when using the .NET library is that is must be explicitly initialised before first using the library within your application. So within the application's startup routine, you must call the function:

```
iMSNET.Init();
```

3.1.2 Concepts

Most concepts familiar to .NET programmers can be applied to code written against the iMSNET library. For example, to iterate through all the ImagePoint's in an Image, in C++ one might write the code

```
iMS::Image img(500, iMS::ImagePoint());
for (iMS::Image.iterator it = img.first(); it != img.end(); ++it) {
 // ...
}
```

In C#, the iMSImage class implements IEnumerable, so one could instead write:

```
iMS.iMSImage img = new iMS.iMSImage(500, new iMS.ImagePoint());
foreach (var pt in img) {
 // ...
}
```

3.1.3 WPF and INotifyPropertyChanged

Classes ImagePoint, CompensationPoint and TBEntry all implement the INotifyPropertyChanged interface which is a key concept in WPF (Windows Presentation Foundation) and UWP (Universal Windows Platform) applications. As a result, they may all be used directly within ViewModels where the MVVM design pattern is being used.

3.1.4 More Information

For further information, we recommend creating a new .NET project in Visual Studio, adding the iMSNET.dll library as an assembly reference and using the Object Browser to examine all of the available classes and functions, comparing them with the C++ documentation.

3.2 Python Wrapper

We are planning to develop and release a wrapper for the C++ library in the Python scripting language to allow users to create their own iMS compatible .py scripts. At present, this is not yet complete but we welcome feedback from users on the usefulness of this feature or whether other scripting languages (e.g. Perl, TCL) would be handy.

Chapter 4

Utilities: iMS Hardware Server

4.1 iMS Hardware Server

Although the core of the SDK is the C++ library and API upon which all applications can be built, we also supply a number of software utilities to assist you in your usage of an iMS System.

The first of these is a command line application called 'ims_hw_server' which runs as a background process in a command window.

ims_hw_server is built upon the C++ API hardware layer and abstracts away some of the requirements of the C++ library to offer a simplified model for performing routine tasks with iMS System hardware. It operates as a client-server model exposing a set of services to the client over a TCP/IP socket using Google's Remote Procedure Call protocol: `gRPC`

Source code that implements the server API for developing client applications is provided as part of the SDK and full documentation for the server can be found here:

`iMS HW Server Documentation`

Note

The Server listens on TCP port 28241. You may be required to enable access on this port through Windows Firewall or other Firewall software or hardware

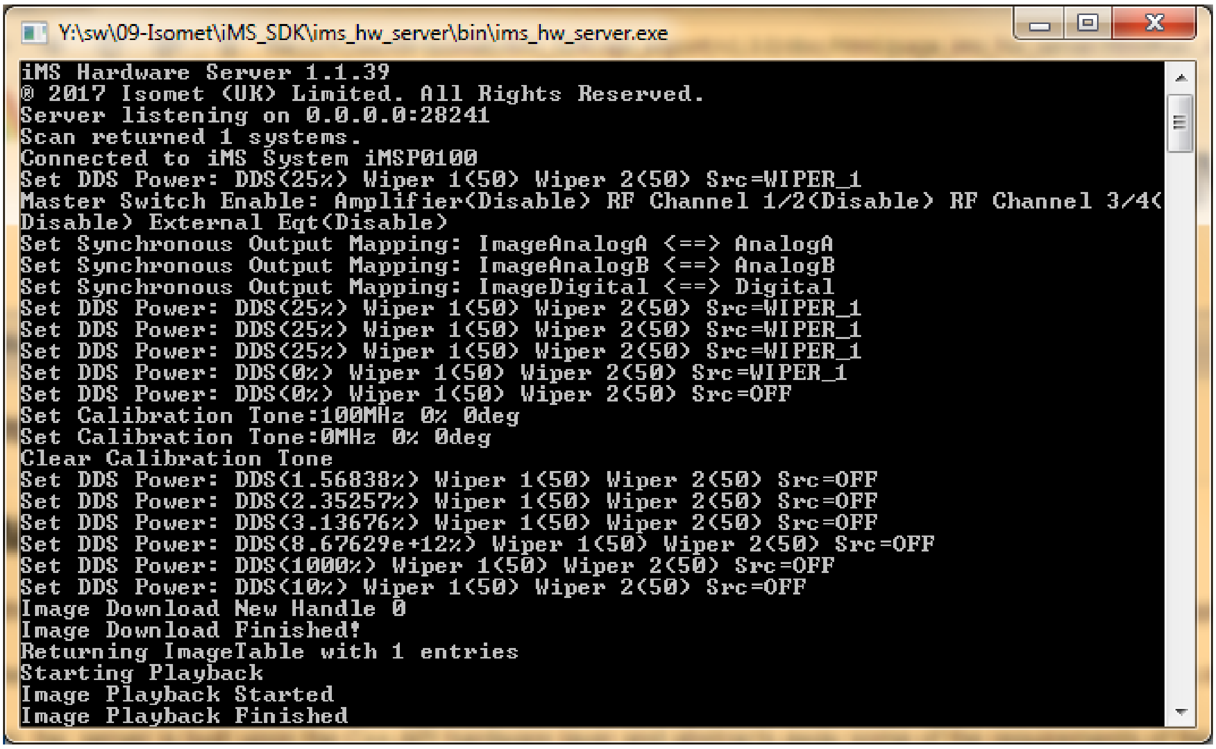


Figure 4.1: Example ims_hw_server Window

Chapter 5

Utilities: iMS Studio

5.1 iMS Studio

iMS Studio is a Graphical IDE built upon the iMSNET .Net wrapper and linking to the ims_hw_server application for access to iMS hardware. It serves as a fully featured application for creating Images, Compensation Tables and Tone Buffers, for managing ImageProjects and ImageGroups, for testing iMS Systems and as a learning tool for understanding how the iMS System is designed.

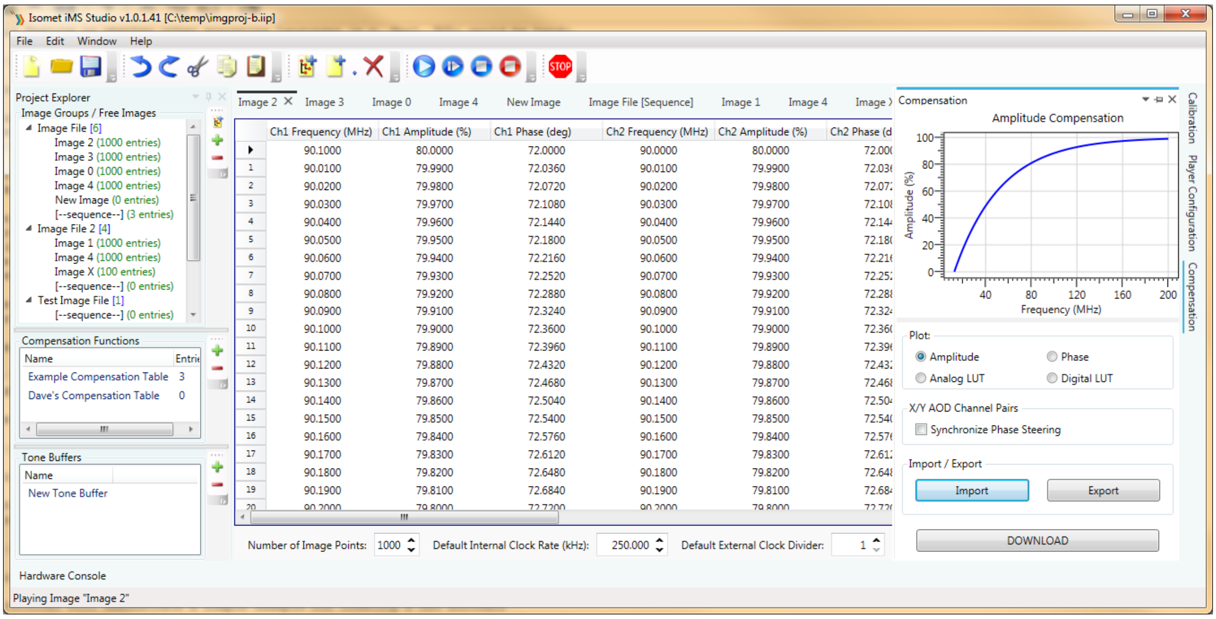


Figure 5.1: Example iMS Studio Window

iMS Studio is built around a Docking Manager style GUI. The centre of the screen is the document pane where the user can place Images, ImageSequences, CompensationTables and ToneBuffers. Around the edge of the window the user can place tabs that perform a variety of different functions. These tabs can be pinned into place using the drawing pin icon at the top right of the tab, removed with the little cross icon, auto-hidden to the side of the window, or floated away from the main window completely. The application starts with a default layout, but do play around with the tabs to arrange them in a style that suits your way of working.

In the default layout, the following tabs are visible:

- **Project Explorer.** This displays all of the ImageGroups, Free Images, Compensation Functions and Tone Buffers in an ImageProject and allows you to add and remove them. You can also drag Images between

ImageGroups to move or copy them.

- **Hardware Console.** The app launches the hardware server in the background and displays console output in this window. If another server is found to be running on the system, the application will connect to that instead and display a message in this window indicating the Process to which it is attached.
- **Compensation.** This tab plots a graph of the Compensation Table that will be programmed into an iMS System. The Compensation Table can be imported and exported to a file on the disk. Where a 4 channel synthesiser is used for 2 channel pairs in a beam steered X/Y configuration, the 2 pairs can be synchronised from here.
- **Signal Path.** Allows the user to control the power setting of the RF signal outputs, as well as enabling / disabling a connected amplifier and setting the Synchronous Data routing.
- **Calibration.** This is used to enable the single tone mode used for AOD calibration. All 4 channels output the same signal and no look-up compensation is applied.
- **Player Configuration.** For playback of single Images, this tab controls the various options that are used during the playback setup, for instance clock and trigger source, number of Image Repeats and whether Compensation should be active or bypassed.

Note

At the release date of SDK v1.3.0, ImageSequence playback has not yet been completed and has been disabled. Also Compensation Tables can only be downloaded by importing from a disk Compensation file as Compensation Table generation from a Compensation Function has not yet been completed.

Chapter 6

Tutorial 1(a): Setting up a project and connecting to an iMS

6.1 Tutorial 1(a): Setting up a project and connecting to an iMS

This tutorial will demonstrate a simple example for creating a new software project in Visual Studio 2013/2015, how to reference the API and write a simple application that connects to the iMS. In part (b) we extend the example to play back an image that can be observed on a spectrum analyzer or oscilloscope.

6.1.1 Prerequisites

You will need to have available:

- an iMS system with a synthesiser (e.g. iMS4) and a controller (e.g. iMSL)
- a Windows PC with Visual Studio 2013 installed (Community edition is free of charge)
- a copy of the iMS SDK

6.1.2 Step 1

Connect the iMS system to your PC's USB port. Apply power to the iMS.

Start up Visual Studio 2013. From the "Community 2013" window, press Start -> New Project... Expand the Installed Templates to see under "Visual C++" -> "Win32" and select "Win32 Console Application"

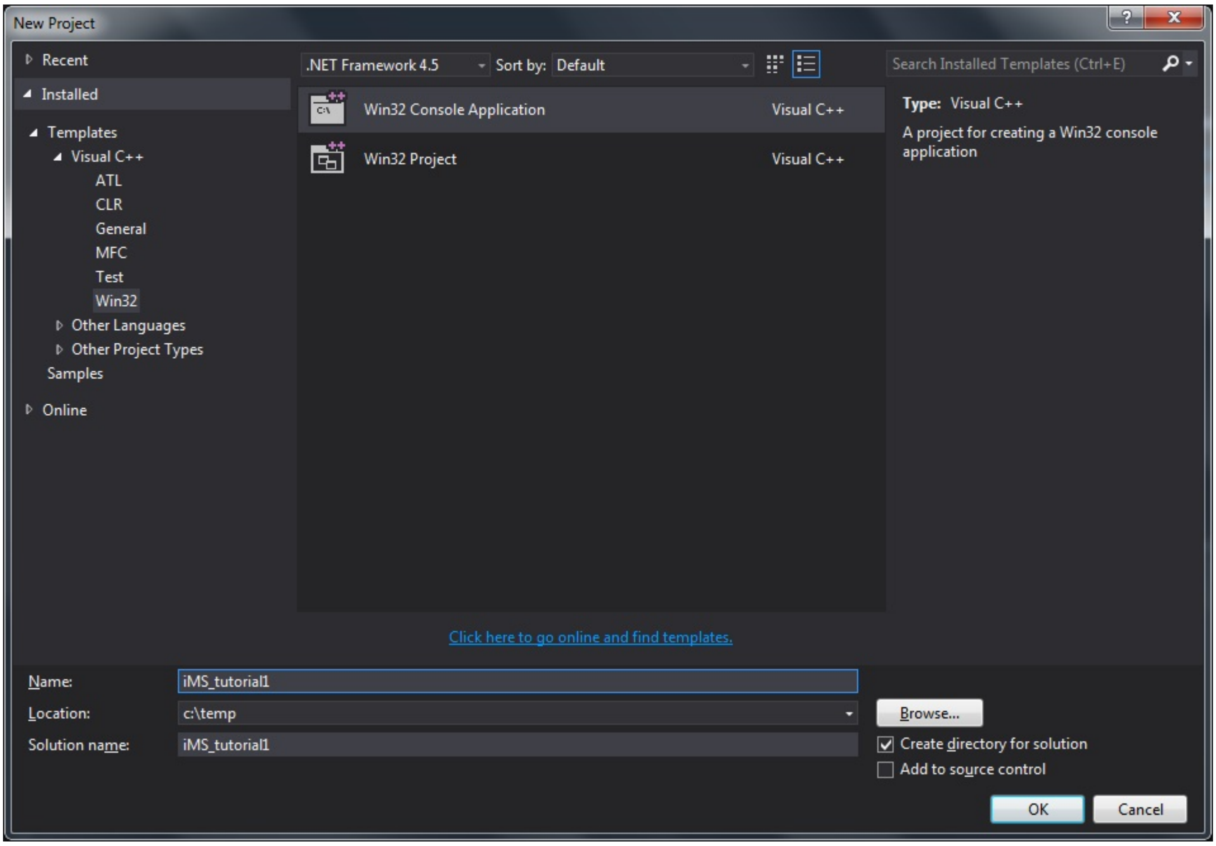


Figure 6.1: New Project Wizard

Choose a location to suit you and give the project a name such as "iMS_tutorial1". Make sure the "Create a directory for solution" box is checked.

6.1.3 Step 2

In the Application Wizard that comes up, select "Next" (not "Finish") then ensure the box next to "Empty Project" is checked. Click Finish.

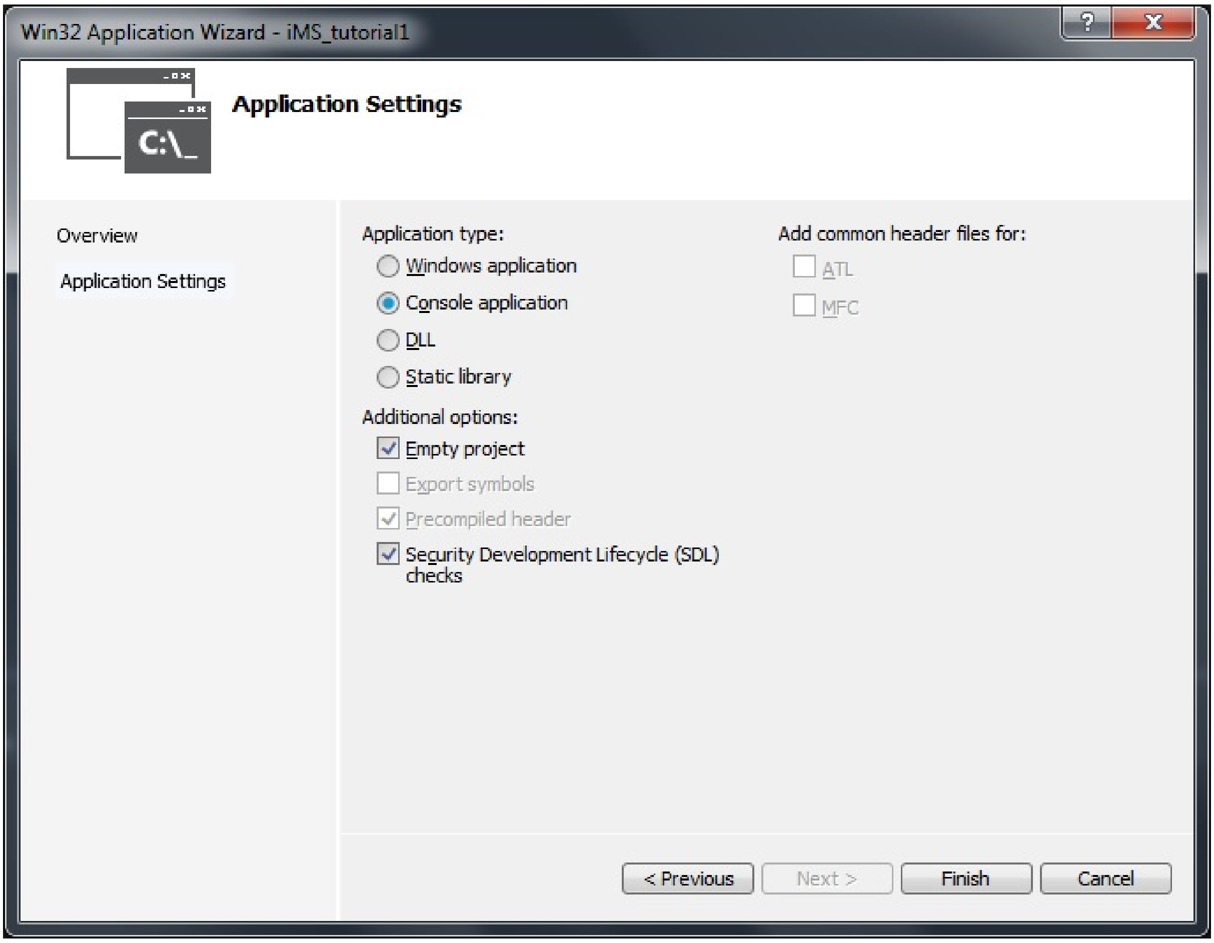


Figure 6.2: Create Empty Project

6.1.4 Step 3

Right click on "Header Files", select Add -> Existing Item...

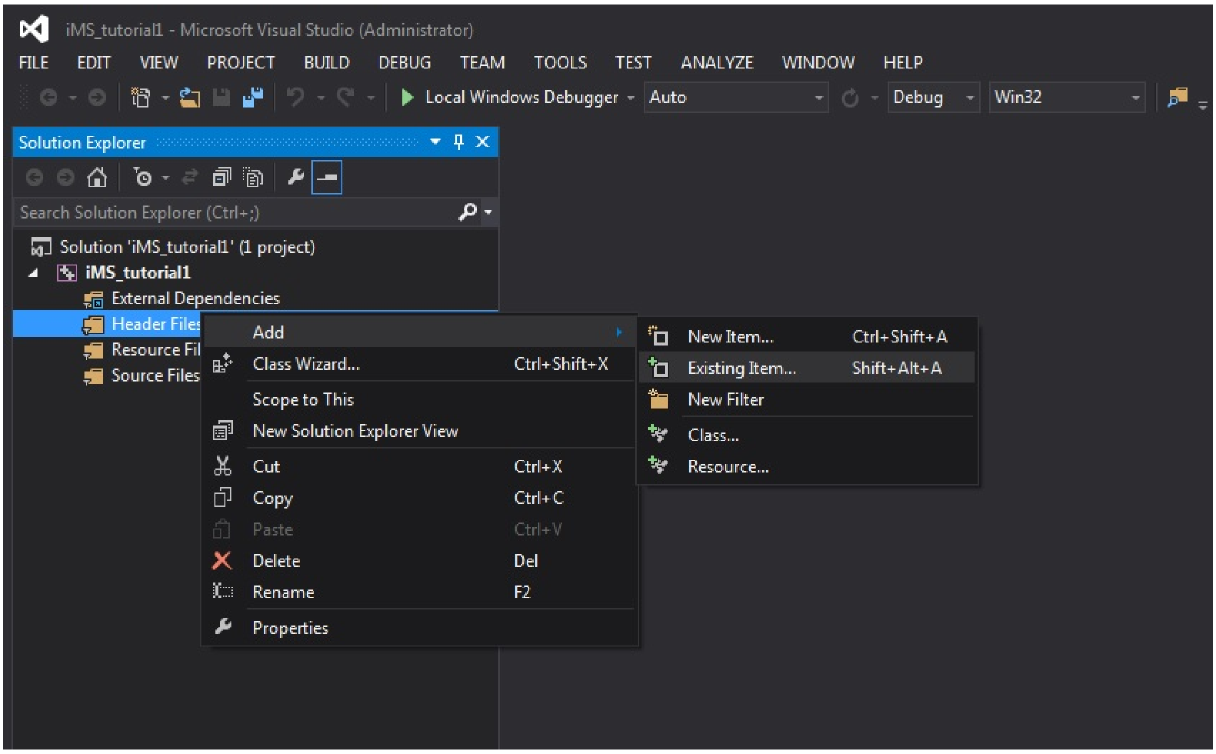


Figure 6.3: Add Existing Item

then navigate to where you have downloaded the Isomet iMS SDK, in the /include subdirectory and select all of the header (∗.h) files. Click "Add".

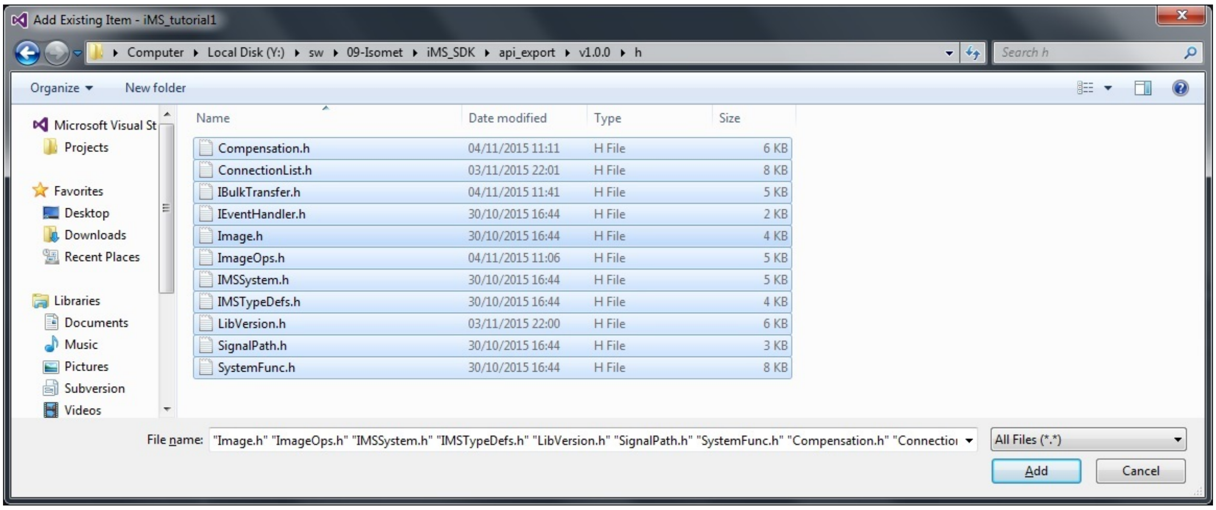


Figure 6.4: Add API Header files

6.1.5 Step 4

Right Click on "Source Files", select Add -> New Item...

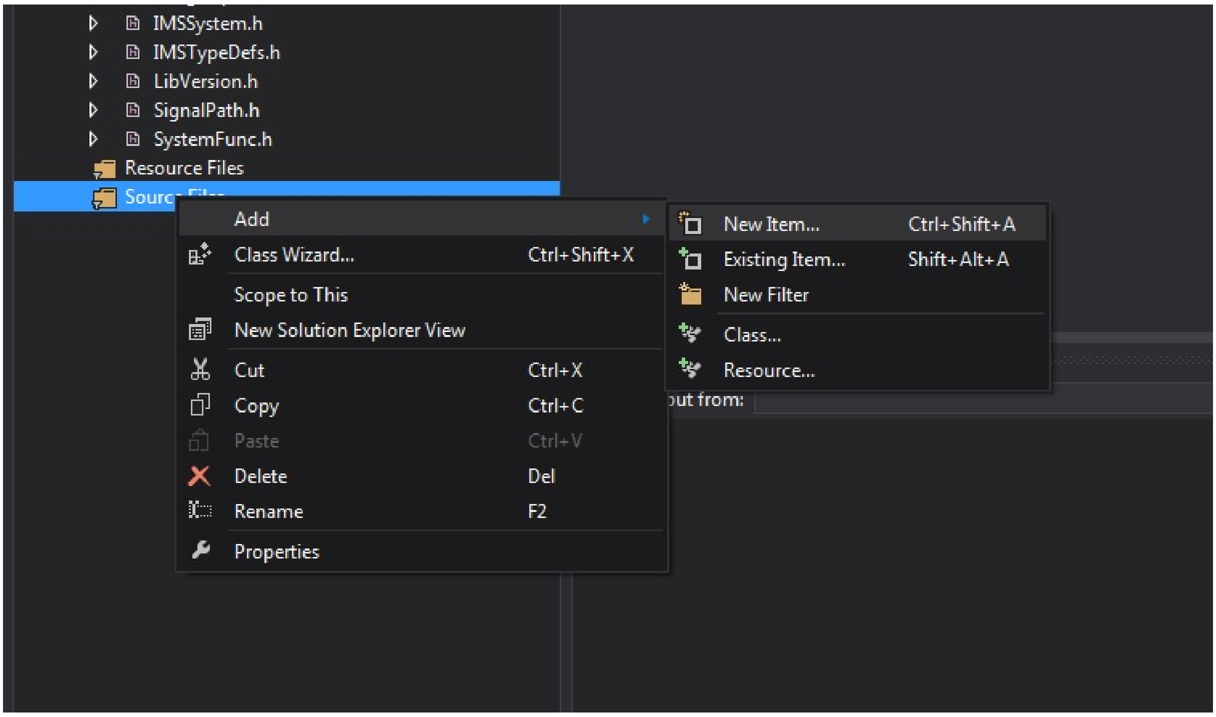


Figure 6.5: Add New Source File

In the Add new item dialog box, select "C++ File (.cpp)" and click Add. You can call the source file anything you like, but the default "Source.cpp" is fine.

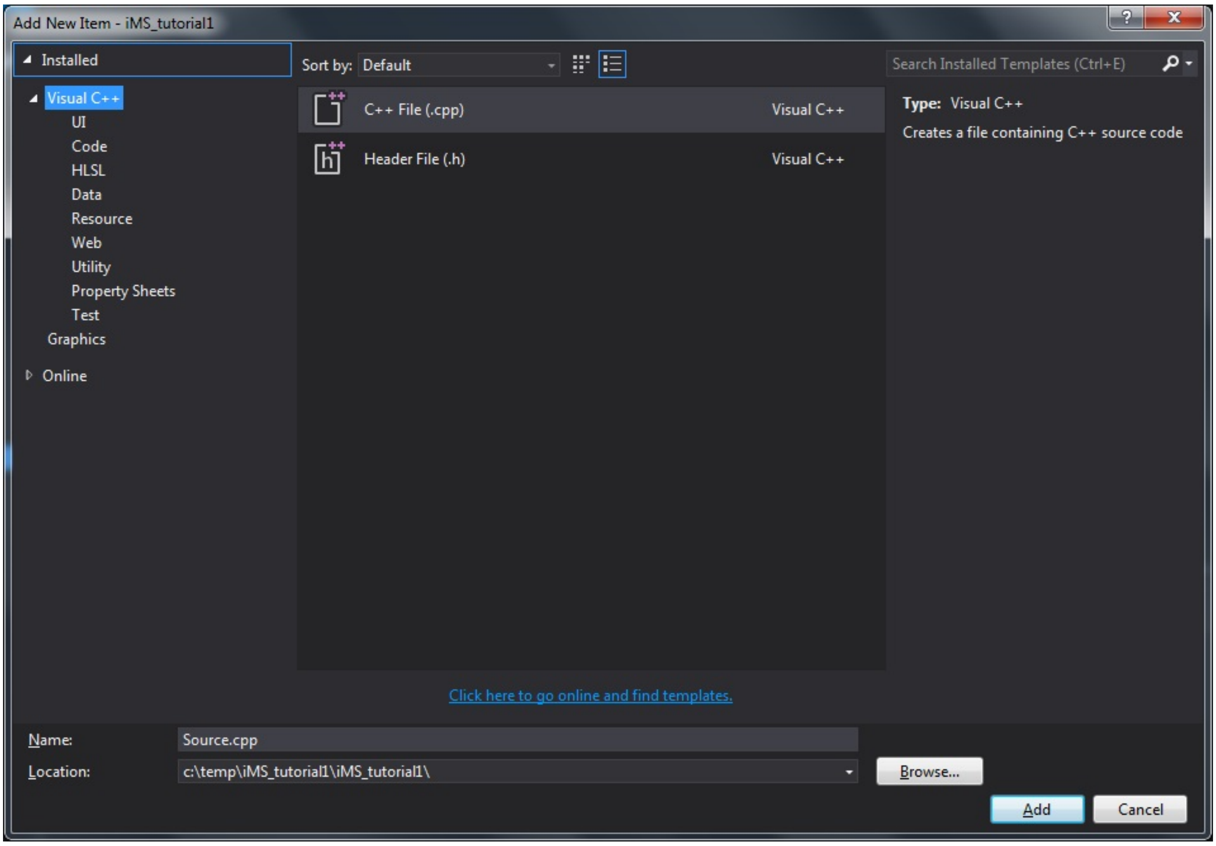


Figure 6.6: Add New C++ File

6.1.6 Step 5

The new source file will open in the main editor window in Visual Studio. Add the following code to get started:

```
// These are the API header files we will need in this tutorial
#include "ConnectionList.h"
#include "IMSSystem.h"
#include "SystemFunc.h"
#include "ImageOps.h"
#include "Compensation.h"

// These are the C++ standard library headers we will need
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <thread>
#include <vector>

// All API classes are defined in the iMS namespace.  For convenience, we can declare this here
using namespace iMS;

int main(int argc, char* argv)
{
  // End application with success code
   std::cout << "Press ENTER to finish";
 std::cin.get();
  return 0;
}
```

6.1.7 Step 6

We now have a blank source file to work in, but we need to configure Visual Studio to know how to compile against the software library. Right click against the project name (iMS_tutorial1 - the line BELOW "Solution 'iMS_tutorial1'") and select properties.

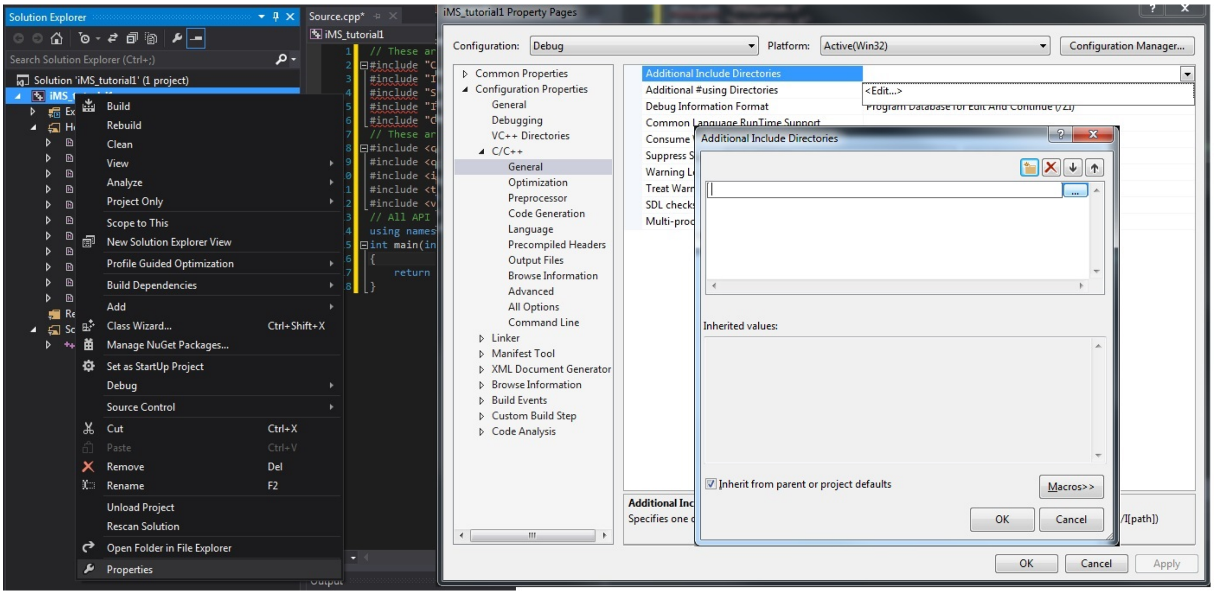


Figure 6.7: Additional Include Directories

Select "Configuration Properties -> C/C++ -> General" and next to "Additional Include Directories" click the arrow to the right of the drop down box and "<Edit...>". Click the icon for "New Line" and the "..." button on the right to bring up a "Select Directory" dialog box.

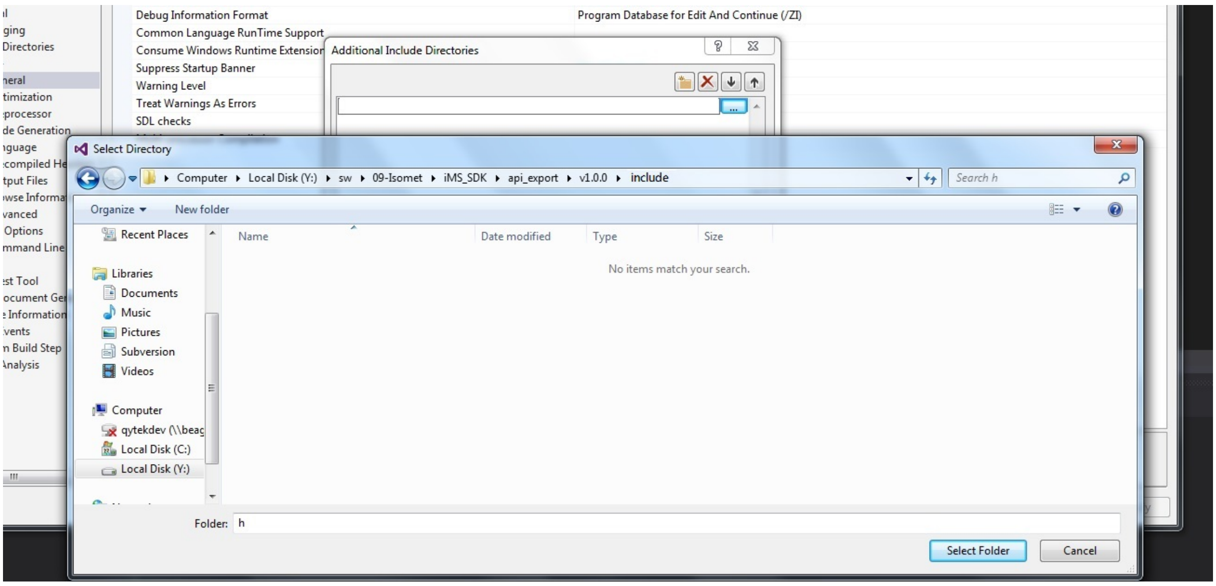


Figure 6.8: Select Directory

Browse to the folder where the SDK header files were included (/include) and click Select Folder then "OK".

6.1.8 Step 7

From the Property Pages, now expand "Linker -> General" and select "Additional Library Directories" and go through the same process to add a reference to the SDK folder containing the 32-bit DLL (/lib/i386)

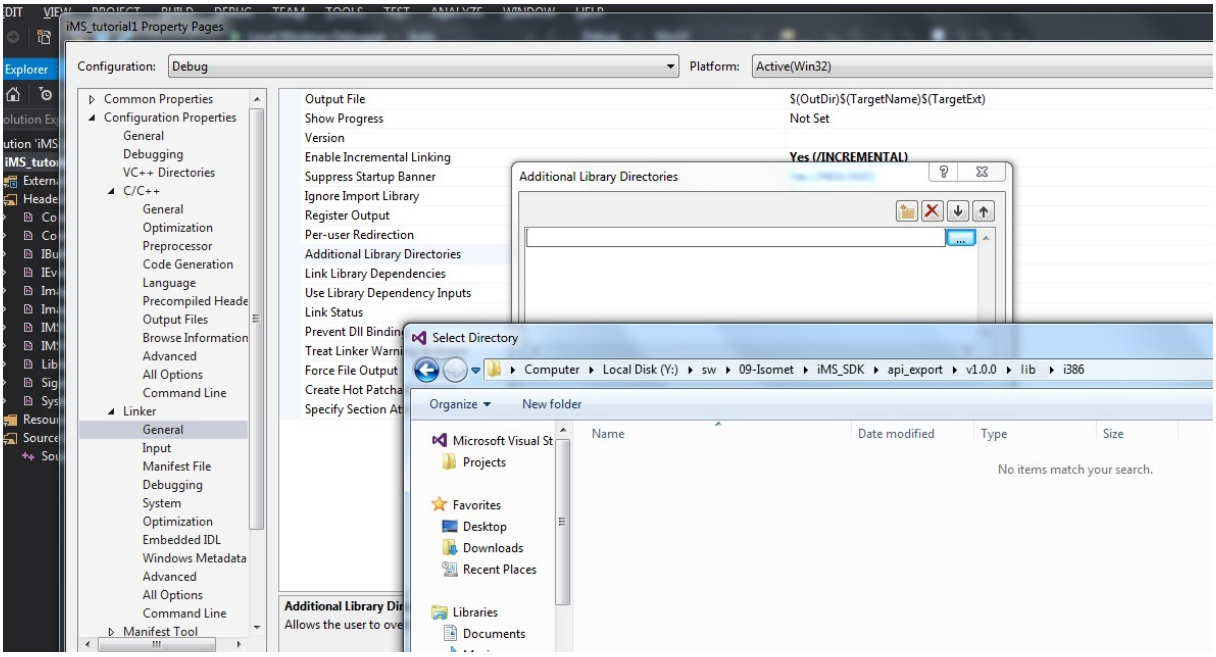


Figure 6.9: Select DLL Directory

In the Property Pages, navigate to "Linker -> Input" and click on the arrow next to "Additional Dependencies" followed by "Edit...". In the text box type in "iMSLibrary_dbg.lib" This is a reference to the 32-bit debug mode DLL which you can use for application development. Click OK then OK again to close the property pages.

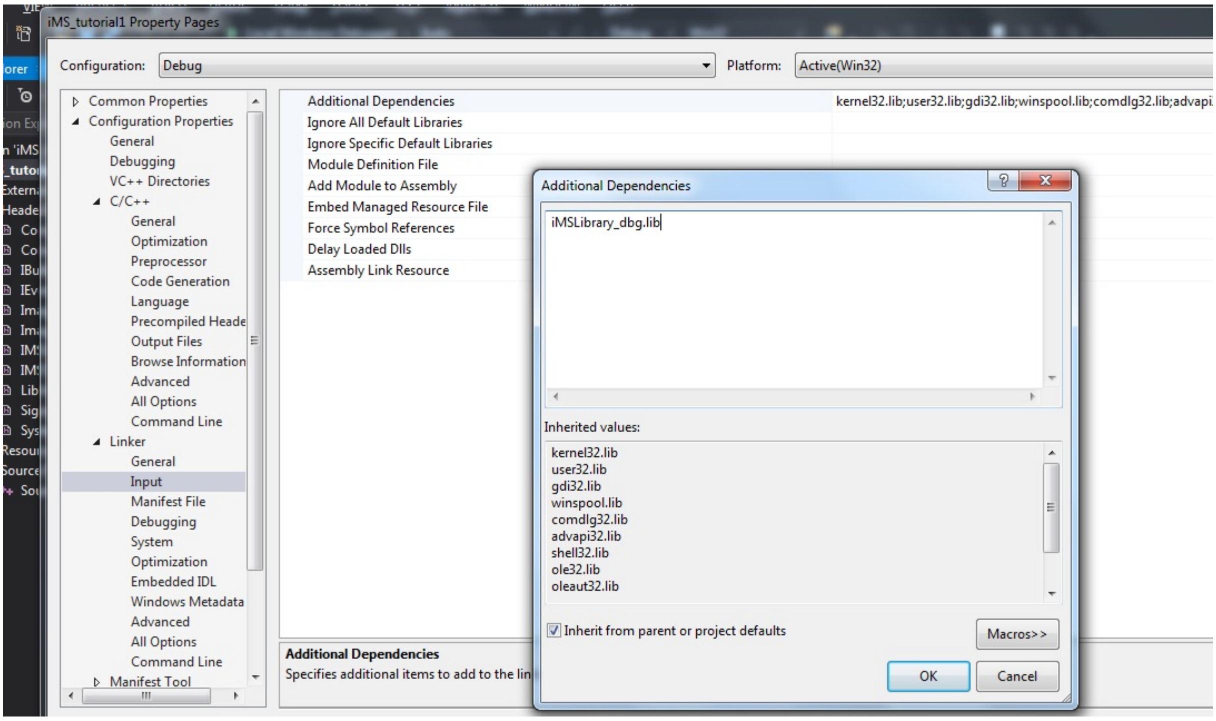


Figure 6.10: Additional Dependencies

6.1.9 Step 8

The last step in setting up the project is in Windows Explorer to copy the .dll iMSLibrary_dbg.dll from /lib/i386 to the project folder you created: (e.g. C:\temp\iMStutorial1\iMStutorial1). It should be in the same folder as the source file Source.cpp.

6.1.10 Step 9

Now we will add some code to search for an iMS System and try to connect to the first one that we find

In the main function, copy the following code and insert before the "return 0;" line:

```
    // These two lines initialise the iMS Connection and scan the host for all connected iMS's
    ConnectionList * connList = new ConnectionList();
    std::vector<IMSSystem> fulliMSList = connList->scan();
    IMSSystem myiMS;
    if (fulliMSList.size() > 0) {
        // Get the first iMS that we find
        myiMS = fulliMSList.front();
        // and connect to it.
        myiMS.Connect();
        std::cout << "Connecting to IMS System on port: " << myiMS.ConnPort() << " ... ";
        if (!(myiMS.Synth().IsValid()) || !(myiMS.Ctlr().IsValid())) {
            // There was a problem trying to initialise the iMS.  We didn't find a valid system.
            std::cout << "FAILED!" << std::endl;
            // Tidy up and return with a failure code
            delete connList;
            std::cout << "Press ENTER to finish";
            std::cin.get();
            return -1;
        }
        else
        {
            // Everything OK.
            std::cout << "SUCCESS!" << std::endl;

            // All USER CODE goes here.

            std::cout << "Press ENTER to finish";
            std::cin.get();
        }
    }
    else {
        // There was a problem trying to discover an iMS. Check the USB connection and power.
        std::cout << "No iMS Found." << std::endl;
        // Tidy up and return with a failure code
        delete connList;
        std::cout << "Press ENTER to finish";
        std::cin.get();
        return -1;
    }

    // All done for now.  Disconnect from the iMS and tidy up
    myiMS.Disconnect();
    delete connList;
    return 0;
}
```

This code will create a connection list (see ConnectionList.h) that knows how to communicate with an iMS system on all supported connection types. We then ask the list to scan these connections to discover any iMS's connected to the host. Any that it finds are returned in an array complete with all the information that we were able to find out about them (configuration, model numbers, serial numbers etc). If none were found, the application aborts and exits.

In this example, the first iMS in the array is connected to, but you could be more specific by interrogating the iMS's serial numbers or other data - see the documentation for IMSSystem.h for more detail.

The application displays the message "Connecting to IMS System on port: " with a string descriptor of the connection port, and tests that it has a valid connection with both a synthesiser and a controller. If not, it will abort and warn the user.

You can run this application (press F5) and it should show the "Connecting" message then exit successfully. Your first iMS application!

Continue to Tutorial 1(b): Programming and Playing an Image

Chapter 7

Tutorial 1(b): Programming and Playing an Image

7.1 Tutorial 1(b): Programming and Playing an Image

In Tutorial 1(a): Setting up a project and connecting to an iMS you learnt how to set up an iMS project in Visual Studio 2013 with the correct references to the iMS API and library, and wrote a simple program that connected to an iMS system. If you completed that tutorial successfully, you can now continue to create an example image and watch it playing on the iMS RF output.

You will need:

- to have successfully completed Tutorial 1(a): Setting up a project and connecting to an iMS
- a spectrum analyser, oscilloscope or some other test equipment with a bandwidth of 100MHz or greater

This tutorial will illustrate two of the key concepts required to understand the operation of an iMS System: Compensation Tables and Image Output.

7.1.1 Step 1: Creating & Downloading a Compensation Table

Compensation Tables provide the system designer with a method for adjusting RF drive signal output according to frequency dependent effects in the output signal chain. They can also be used to generate analog and digital outputs for system requirements that are linked to frequency output.

Each table consists of a look-up function of 2^N entries (N can be read from iMS::IMSSynthesiser::Capabilities::LUTDepth). Each entry is linearly spaced from the lowest Frequency supported by the Synthesiser to the highest.

There are four tables: one for amplitude compensation, one for phase steering, one for analog system output and one for digital system output.

For this example, we will create an amplitude table that steps down every 10MHz from 50MHz to 100MHz. This is a contrived example that can act as a template for your own requirements.

In the source code that you created in part 1(a), insert the following code after the comment line "`All USER CODE goes here...`"

```
// Check for the existence of a file containing LUT contents in the current working directory
CompensationTable table(myiMS);
std::ifstream f("tutorial1.lut");
if (f.good()) {
    f.close();
    // Create a compensation table from the pre-existing file
    CompensationTable new_table(myiMS, "tutorial1.lut");
    table = new_table;
}
else {
    f.close();
    // Create a new compensation table with the amplitude initialised to 100% throughout
    CompensationTable new_table(myiMS, CompensationPoint(Percent(100.0)));
```

```
            // For loop iterates through every frequency point in the look-up table, halving the
 amplitude
            // at each 10MHz step between 50 and 100MHz.
            unsigned int index = 0;
            for (CompensationTable::iterator pt = new_table.begin(); pt != new_table.end(); ++pt, index
++)
            {
                if ((new_table.FrequencyAt(index)) > 90.0) {
                    pt->Amplitude(Percent(100.0 / 16.0));
                }
                else if ((new_table.FrequencyAt(index)) > 80.0) {
                    pt->Amplitude(Percent(100.0 / 8.0));
                }
                else if ((new_table.FrequencyAt(index)) > 70.0) {
                    pt->Amplitude(Percent(100.0 / 4.0));
                }
                else if ((new_table.FrequencyAt(index)) > 60.0) {
                    pt->Amplitude(Percent(100.0 / 2.0));
                }
            }
            // Save table to disk so we don't have to recreate it next time
            new_table.Save("tutorial1.lut");
            table = new_table;
        }

        CompensationTableDownload tdl(myiMS, table);
        tdl.StartDownload();
```

Here, the first line of code creates a new blank compensation table called 'table'. We then attempt to open a file called `tutorial1.lut` which contains a previously generated copy of the look-up table. If the application finds it, it will load the contents of the file instead of regenerating them.

If not, a second new table is created this time with the amplitude initialised to a default value of 100%. A for-loop uses CompensationTable::iterator to address each of the points, determining their frequency and adjusting the amplitude to create a 'stairstep' effect at 10MHz intervals as the frequency rises from 50MHz to 100MHz

Once the table is complete, it is saved to disk to enable it to be recalled on the next program run.

The final two lines call the CompensationTable Downloader class, initialising it with the newly created table, then start it downloading the table to the hardware.

7.1.2 Step 2: Creating & Downloading an Image

A primary feature of the iMS System is its ability to store and playback one or more RF images. An RF image can be downloaded to an iMS Controller and then played back under the control of an internal or externally supplied clock, initiated by software or some external trigger signal. Each image contains a sequence of 'image points', from just a few up to many millions. Each image point contains information for Frequency, Amplitude and Phase (known as an 'FAP triad') for up to 4 RF channels plus some synchronous data that can be output externally to drive other hardware in the system. Some Controllers support multiple images that can be arranged into complex sequences - these are called 'image files.'

For this tutorial, we will generate a simple single 4096-point image which linearly ramps up from 50MHz to 100MHz then loops around and repeats indefinitely. We will then play this back at a slow 1kHZ internal clock rate so that the output ramp time is 4.1sec which can be observed on test equipment. It is a contrived example, barely useful in AO equipment, but which serves to explain the fundamental principles.

Copy and paste the following source code after the code you added in part 1:

```
    // Create the initial conditions:
    // a default FAP triad, an empty image and the upper and lower frequency bounds
        FAP fap(MHz(50.0), Percent(100.0), Degrees(0.0));
        Image img;
        MHz lf(50.0);
        MHz uf(100.0);

    // Loop through appending 4096 points increasing linearly in frequency
        for (int i = 0; i <= 4095; i++)
        {
            // linear ramp
            fap.freq = lf + (uf - lf)   * ((double)i / 4096.0);
            img.AddPoint(ImagePoint(fap));
        }
    // Set Internal Clock rate for 4.1sec ramp time
        img.ClockRate(kHz(1.0));
```

```
        // Clear any leftover image playback to permit download
        ImagePlayer ForceStop(myiMS, img);
        ForceStop.Stop(ImagePlayer::StopStyle::IMMEDIATELY);

    // Create download object and initiate
        ImageDownload idl(myiMS, img);
        idl.StartDownload();

    // Create image player object with post-delay and repeating continuously
        ImagePlayer player(myiMS, img, ImagePlayer::PlayConfiguration(ImagePlayer::Repeats::FOREVER));
        player.SetPostDelay(std::chrono::milliseconds(500));

        std::cout << "Press ENTER to play" << std::endl;
        std::cin.get();
        player.Play();
```

In the first few lines, an empty image is created along with a single FAP triad initialised to 50MHz, 100% amplitude with no phase offset. the upper and lower bounds of the sweep are defined as 'lf' and 'uf' MHz objects.

A for-loop is used to create the 4,096 image points that will make up the new image. First the frequency is set according to its linear position in the ramp sweep. Then, a new image point is created from the FAP triad (the triad is replicated across all 4 channels of the image point), and it is appended to the Image.

Finally, the image's default internal clock rate is declared to be 1kHz.

That's it - all that is required to create a simple image!

To use the image, it must be downloaded to the Controller then instructed to play back. Some Controllers do not support simultaneous playback and download, so the next two lines create an ImagePlayer object that is just used to send a Stop command to the hardware. The stop command is issued with a StopStyle::IMMEDIATELY property to terminate any ongoing playback straightaway. If this wasn't used, the default behaviour would be to stop the playback only after the final point in the image/

Following on from this, an ImageDownload object is created and initialised with the image we have just generated. This uses the same large binary object mechanism (see BulkTransfer.h) as the Compensation Table downloader to send the image to memory in the Controller.

Next, we create an ImagePlayer with its configuration initialised to repeat the image forever, until stopped on request by software. We also want to add in a pause of half a second after each repeat so it is clear to the observer that the image has completed.

The download and playback configuration done, we just await confirmation from the user before instructing the iMS and its Image Player to begin playing.

7.1.3 Step 3: Observing the Output

Connect a spectrum analyser or oscilloscope to any of the RF outputs on the iMS System. You should observe the following pattern:

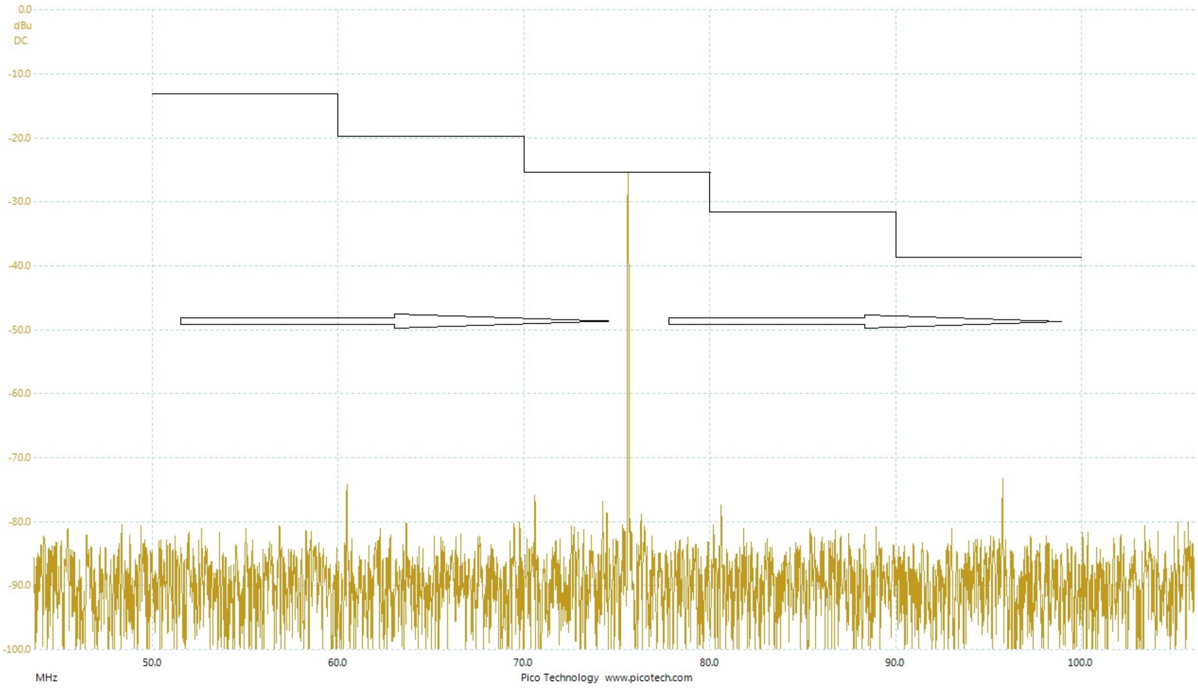


Figure 7.1: Image Output observed on Spectrum Analyzer

Note

You may wonder whether the request for the user to press ENTER before playback is necessary. In fact, try removing the `std::cin.get()` line so that the example runs smoothly from image download into playback without user input and you will find that playback may not start on some hardware configurations.

This is a consequence of the inability of some hardware to perform both image download and image playback at the same time. The function call `idl.StartDownload()` asks the Image Downloader to begin the download process. In fact, the function spawns a thread in the background that performs the bulk download of the image data, and returns immediately to user code. The API can trigger callbacks to the user application that indicate when a download has started, finished and whether it was successful or failed. It can also perform a post-download verify to check that the image contents were downloaded correctly. We will cover these in a later tutorial when we look at the Message Handling system.

Because the background process is still downloading when the `player.Play()` function is called, the ImagePlayer realises it cannot continue, and returns without beginning the image playback.

The `Play()` function returns `true` on a successful play attempt, so the simplest solution is to replace the code line `player.Play();` with `while`(!player.Play()); This will repeat the function call until it succeeds. For a more elegant solution that doesn't block user code, see the future tutorial on Message Handling.

7.1.4 Full Tutorial 1 Code Listing

```
// These are the API header files we will need in this tutorial
#include "ConnectionList.h"
#include "IMSSystem.h"
#include "SystemFunc.h"
#include "ImageOps.h"
#include "Compensation.h"

// These are the C++ standard library headers we will need
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <thread>
#include <vector>

// All API classes are defined in the iMS namespace.  For convenience, we can declare this here
using namespace iMS;
```

```
int main(int argc, char* argv)
{
    // These two lines initialise the iMS Connection and scan the host for all connected iMS's
    ConnectionList * connList = new ConnectionList();
    std::vector<IMSSystem> fulliMSList = connList->scan();
    IMSSystem myiMS;

    if (fulliMSList.size() > 0) {
        // Get the first iMS that we find
        myiMS = fulliMSList.front();
        // and connect to it.
        myiMS.Connect();
        std::cout << "Connecting to IMS System on port: " << myiMS.ConnPort() << " ... ";
        if (!(myiMS.Synth().IsValid()) || !(myiMS.Ctlr().IsValid())) {
            // There was a problem trying to initialise the iMS.  We didn't find a valid system.
            std::cout << "FAILED!" << std::endl;
            // Tidy up and return with a failure code
            delete connList;
            std::cout << "Press ENTER to finish";
            std::cin.get();
            return -1;
        }
        else
        {
            // Everything OK.
            std::cout << "SUCCESS!" << std::endl;

            // Check for the existence of a file containing LUT contents in the current working directory
            CompensationTable table(myiMS);
            std::ifstream f("tutorial1.lut");
            if (f.good()) {
                f.close();
                // Create a compensation table from the pre-existing file
                CompensationTable new_table(myiMS, "tutorial1.lut");
                table = new_table;
            }
            else {
                f.close();
                // Create a new compensation table with the amplitude initialised to 100% throughout
                CompensationTable new_table(myiMS,
      CompensationPoint(Percent(100.0)));
                // For loop iterates through every frequency point in the look-up table, halving the
       amplitude
                // at each 10MHz step between 50 and 100MHz.
                unsigned int index = 0;
                for (CompensationTable::iterator pt = new_table.begin(); pt !=
      new_table.end(); ++pt, index++)
                {
                    if ((new_table.FrequencyAt(index)) > 90.0) {
                        pt->Amplitude(Percent(100.0 / 16.0));
                    }
                    else if ((new_table.FrequencyAt(index)) > 80.0) {
                        pt->Amplitude(Percent(100.0 / 8.0));
                    }
                    else if ((new_table.FrequencyAt(index)) > 70.0) {
                        pt->Amplitude(Percent(100.0 / 4.0));
                    }
                    else if ((new_table.FrequencyAt(index)) > 60.0) {
                        pt->Amplitude(Percent(100.0 / 2.0));
                    }
                }
                // Save table to disk so we don't have to recreate it next time
                new_table.Save("tutorial1.lut");
                table = new_table;
            }

            CompensationTableDownload tdl(myiMS, table);
            tdl.StartDownload();

            // Create the initial conditions:
            // a default FAP triad, an empty image and the upper and lower frequency bounds
            FAP fap(MHz(50.0), Percent(100.0), Degrees(0.0));
            Image img;
            MHz lf(50.0);
            MHz uf(100.0);

            // Loop through appending 4096 points increasing linearly in frequency
            for (int i = 0; i <= 4095; i++)
            {
                // linear ramp
                fap.freq = lf + (uf - lf)   * ((double)i / 4096.0);
                img.AddPoint(ImagePoint(fap));
            }
            // Set Internal Clock rate for 4.1sec ramp time
            img.ClockRate(kHz(1.0));

            // Clear any leftover image playback to permit download
```

```
            ImagePlayer ForceStop(myiMS, img);
            ForceStop.Stop(ImagePlayer::StopStyle::IMMEDIATELY);

            // Create download object and initiate
            ImageDownload idl(myiMS, img);
            idl.StartDownload();

            // Create image player object with post-delay and repeating continuously
            ImagePlayer player(myiMS, img, ImagePlayer::PlayConfiguration(
      ImagePlayer::Repeats::FOREVER));
            player.SetPostDelay(std::chrono::milliseconds(500));

            std::cout << "Press ENTER to play" << std::endl;
            std::cin.get();
            player.Play();

            std::cout << "Press ENTER to finish";
            std::cin.get();
        }
    }
    else {
        // There was a problem trying to discover an iMS. Check the USB connection and power.
        std::cout << "No iMS Found." << std::endl;
        // Tidy up and return with a failure code
        delete connList;
        std::cout << "Press ENTER to finish";
        std::cin.get();
        return -1;
    }

    // All done for now.  Disconnect from the iMS and tidy up
    myiMS.Disconnect();
    delete connList;
    return 0;
}
```

Chapter 8

Tutorial 2: Using the API Message Handling System

8.1 Tutorial 2: Using the API Message Handling System

To Be Continued...

Chapter 9

Glossary

9.1 Glossary

	Description
FAP	A compound object containing a Frequency variable, an Amplitude variable and a Phase variable
Image Point	Contains a description of the instantaneous output of an iMS Synthesiser. It has 4 channels each represented by a Frequency, Amplitude and Phase triad.
Image	An Image stores a sequence of Image Points, from a few to many million. An image can be stored in Controller memory, uploaded and downloaded from the host system. AO scan patterns are created by playing back the image's point sequence under the influence of an internal oscillator, or an externally provided clock
Image Group	An Image group combines multiple Images and a Sequence Table. It can be stored in Controller memory (not supported by all Controllers), uploaded and downloaded from the host system. Complex sequences of AO scan patterns can be created by indexing multiple Images using the Sequence Table and/or host software interaction
Image Project	An Image Project stores multiple Image Groups, Compensation Functions, Tone Buffers and Free Images along with other useful metadata and is used for saving iMS data to host filesystems. Image Projects cannot be downloaded to Controller hardware
Sequence Table	A sequence table is associated with an Image File and programs the Controller with the desired sequence in which to play back multiple Images. The Sequence Table can also program Image repeats, Image delays and override default Image oscillator frequency

Compensation Table	Compensation Tables are downloaded to the Synthesiser for the purpose of providing Amplitude Compensation (to counteract diffraction efficiency effects and other RF signal path frequency responses), Phase Steering (for Beam Steered AODs) and Custom-mapped frequency-dependent Synchronous Data outputs
Compensation Function	A Compensation Function is an abstraction of a Compensation Table that allows full specification of Compensation Table data from just a few Frequency points. A Compensation Table can be derived from an interpolation of data contained within a Compensation Function
Triad	Another name for a FAP

Chapter 10

Release Notes

10.1 v1.4.2

(1st November 2017)

- **(ADD)** ConnectionList.h: Added config() and modules() methods to ConnectionList class to configure the discovery process and limit its scope to improve scan time
- **(ADD)** SystemFunc.h: Added missing field XYCompEnable to StartupConfiguration struct
- **(REMOVE)** ConnectionList.h: Removed ConnectionTypesList typedef and associated connList private class member plus iterators
- **(BUGFIX)** ImageProject.h: Fixed - v1.4.0 introduced a bug that would save MHz values as Hz values in .iip and .xml Image Project save files.

10.2 v1.4.1

(6th October 2017)

- **(ADD)** SystemFunc.h: Added Master Reference Clock mode support
- **(ADD)** SystemFunc.h: Support for Master Reference clock programming into StartupConfiguration class
- **(ADD)** Auxiliary.h: LED setting for PLL Lock Status
- **(CHANGE)** iMS4b Synthesiser frequency range now extends to 200.8MHz (previously limited to 196.6MHz). Requires firmware build $>=$ 2.1.64
- **(BUGFIX)** ConnectionList.h: Corrected thread timeout bug in RS422 connection module that caused excessive delay on calling ConnectionList::scan()

10.3 v1.4.0

(4th August 2017)

- **(ADD)** Auxiliary.h: Auxiliary::LED_SOURCE::OVERTEMP added to enable overtemperature LED configuration
- **(ADD)** ConnectionList.h: Support for CM_ENET (Ethernet Interface) and CM_RS422 (Serial Interface)
- **(ADD)** IMSSystem.h: Database updated with support for latest iMS4 Synthesiser H/W Revision

- **(ADD)** SystemFunc.h: Readback of iMS4 System Temperature
- **(ADD)** Message.h: Added MEMORY_TRANSFER_ERROR event
- **(ADD)** SignalPath.h: Support for Dual Optical Encoder inputs, tracking filter configuration and Velocity-Frequency Compensation
- **(ADD)** SignalPath.h: Synchronous Digital Output Data Configuration for delay time and pulse length
- **(ADD)** IConnectionManager.h: Suffix to ConnString result: serial number now followed by ":" then string indicating connection type
- **(CHANGE)** ConnectionList.h: Added a 1sec timeout to USB message transfer
- **(CHANGE)** Compensation.h: CompensationTable can now be loaded from a file that was generated for a Synthesiser with a different operating frequency range. The table is interpolated using the nearest neighbour algorithm.

10.4 v1.3.0

(16th February 2017)

- **(ADD)** Compensation.h: CompensationPointSpecification and CompensationFunction classes, removing the unimplemented Compensation class in the process
- **(ADD)** Compensation.h: Explicit constructors for CompensationTable to allow it to be created without being connected to an iMS System
- **(ADD)** IMSSystem.h: Open() function to report on the status of a connection to an iMS System
- **(ADD)** Image.h: Defined a new class ImageGroup to hold multiple Images and a single ImageSequence
- **(ADD)** ImageProject.h: New class ImageProject to contain ImageGroup's, CompensationFunction's, ToneBuffer's and Free Image's as well as Load from / Save to disk
- **(ADD)** Containers.h: Most classes requiring a container now inherit from either ListBase or DequeBase
- **(ADD)** ImageOps.h: New Event IMAGE_DOWNLOAD_NEW_HANDLE reports the new Image Index handle
- **(ADD)** SignalPath.h: Added the EnableImagePathCompensation and EnableXYPhaseCompensation functions.
- **(ADD)** SignalPath.h: UpdateLocalToneBuffer now supports PhaseCompensation enable
- **(ADD)** ToneBuffer.h: Added optional name string field to ToneBuffer class
- **(BUGFIX)** CompensationTableDownload: Regular pause added in download process to allow buffers to flush and ensure no data is lost due to buffer overruns

10.5 v1.2.6

(12th August 2016)

- **(BUGFIX)** Corrected a buffer overrun that sometimes occurred during the Image Download memory transfer. The bug was most noticeable for small Images but in fact could randomly occur for any Image where the byte size is not a multiple of 1024 (and particularly where the byte size modulo 1024 is small).
- **(BUGFIX)** Corrected the initialisation of a member variable in the ImagePlayer class that would cause a phantom ImagePlayerEvents::POINT_PROGRESS Event to be triggered at the end of Image playback
- **(BUGFIX)** Const Correctness applied to GetSyncA(), SetSyncA(), GetSyncD() and SetSyncD().
- **(BUGFIX)** Added defaults for StartupConfiguration::SyncDigitalSource, StartupConfiguration::SyncAnalogASource and StartupConfiguration::SyncAnalogBSource

10.6 v1.2.5

(12th July 2016)

- **(ADD)** const overloads of begin() and end() members to enable range based for-loops on const objects for CompensationTable, Image, ImageSequence, ImageTable and ToneBuffer classes
- **(BUGFIX)** added mutex lock around a member variable requiring thread synchronisation in the IMSSystem↩ ::Disconnect() method affecting iMSP Controllers
- **(BUGFIX)** Modified ConnectionList::scan() method to terminate Serial Number string when encountering any non-ASCII value. Overcomes a bug in which some devices were incorrectly programmed without a null terminator on the serial number stored in EEPROM.

10.7 v1.2.4

(18th May 2016)

- **(CHANGE)** ImageDownload support for Internal oscillator mode with Prescaler Disabled allowing clock resolution down to 10ns (previously 1us).
- **(ADD)** Added SequenceDownload class to program newly created sequences into the Controller, at the back of the Controller Sequence queue, SequenceManager class for triggering playback or modifying sequences already in the queue and SequenceEvents for enabling interrupts from the Controller Sequence Engine to user application software.
- **(ADD)** Added ImageSequenceEntry struct for user creation of entries to load into an ImageSequence, SequenceTermAction for specification of the process to be performed at the end of a sequence and class ImageSequence for creating sequences to be downloaded to a Controller.
- **(ADD)** Optional Name field to Image class. The first 16 characters of the name string are downloaded to the Image Index Table so that Images resident in memory on the Controller may be referenced by a descriptive name.
- **(ADD)** Interrupt handling code to respond to interrupt requests from the Controller, interrupts individually enabled when user code subscribes to events that are interrupt driven. All interrupts disabled on SD↩ K Disconnect() or application termination.
- **(BUGFIX)** Added logic to ensure the FileSystemManager::Delete() function will not attempt to remove files from an index that lies outside the FileSystemTable scope.
- **(ADD)** Added Interrupt capability through EventHandler mechanism so iMS hardware can efficiently notify user application code when state changes occur in the hardware subsystems
- **(ADD)** IEventHandler::EventAction virtual methods for returning 2 integer values and a byte vector to user application code

10.8 v1.2.3

(Internal Release Only)

10.9 v1.2.2

(Internal Release Only)

10.10 v1.2.1

(Internal Release Only)

10.11 v1.2.0

(7th April 2016)

- **(ADD)** Added ImagePlayer capability to delay playback start until triggered from external signal
- **(ADD)** Extended ImagePlayer capability to play Images directly from the ImageIndexTable
- **(ADD)** Added ImageTableEntry struct for user readback of Controller Image Memory contents using ImageTableViewer class.
- **(CHANGE)** Image UUID's now returned as fixed size 16-byte std::arrays instead of variable std::vectors
- **(ADD)** Added Analog Sync (x2) and Digital Sync data fields to ImagePoint class
- **(CHANGE)** Embedded iMS hardware database in .dll so separate imshw.db file no longer required
- **(ADD)** Support for Image Index Table (IIT) for multiple images stored in Controller large capacity memory. IIT readback on SDK Connect() and stored for reference in the IMSSystem object.
- **(ADD)** Extended IConnectionManager interface to add MemoryUpload() MemoryDownload() and MemoryProgress() methods to trigger and monitor fast transfer large capacity direct memory I/O to supported Controllers
- **(CHANGE)** Auxiliary::GetAnalogData and Diagnostics::GetDiagnosticsData now return measurement data as Percent() instead of double

10.12 v1.1.0

(22nd March 2016)

- **(BUGFIX)** SetCalibrationTone amplitude now programmes correctly (was erroneously 1/4 of the required amplitude due to bitwidth mismatch)
- **(ADD)** Support for iMSP Controller through USB3.0 Interface
- **(ADD)** IEventHandler::EventAction virtual method for returning floating point data to user application code
- **(ADD)** Diagnostics support for software readback of Logged Hours, Temperature and RF Power Amplifier Forward / Reflected Power (VSWR) and DC Current measurements
- **(ADD)** Monitoring of Pixel Checksum Error Count between Controller and Synthesiser
- **(ADD)** External Equipment Enable through on-board high power Optoisolator
- **(ADD)** External Update Signal for synchronisation of DDS IC to external system clock (SystemFunc::UpdateClockSource)
- **(ADD)** Startup Configuration for programming Synthesiser with non-volatile configuration to be retrieved at power-up or reset
- **(ADD)** Assignment of Synchronous Output data (12-bit digital 'SDOR', 2 analog outputs SDAC A/B) to various data sources (SignalPath::SYNC_SRC)
- **(ADD)** Tone Buffer for storage of 256 unique FAP on the Synthesiser device with control through software selection or external signal input

- **(ADD)** Synthesiser Filesystem for permanent storage of Compensation Table, Tone Buffer, DDS Scripts and User File Data
- **(ADD)** DDS Scripting for manual programming of DDS Synthesiser IC
- **(ADD)** Auxiliary Classes including External Analog I/O, DDS Profile Control, LED Control

10.13 v1.0.1

(18th December 2015)

- **(BUGFIX)** Connection Manager cleans up object before disposal to avoid an exception being raised when the application returns without first disconnecting from the iMS.
- **(CHANGE)** Internal Oscillator or External Clock Divider ratios now programmed into the Controller by the ImagePlayer, not the Image.
- **(CHANGE)** ImagePlayer class must now be provided with the Image that is to be played (which must have already been downloaded to the hardware). It will not play an Image that does not match the memory on the hardware.
- **(ADD)** External Clock Divider Ratio to Image class.
- **(ADD)** UUID (Universally Unique Identification) to Image class to permit comparison of Image objects both with each other and with Images resident in memory on the hardware.

10.14 v1.0.0

(15th November 2015)

Chapter 11

Bug List

Member iMS::FileSystemManager::Delete (FileSystemIndex index)

Prior to v1.2.4 it was possible to attempt to delete an entry >= MAX_FST_ENTRIES. Doing so would have generated an exception. The condition is now checked for and the function will fail (return false) if attempted.

Member iMS::SignalPath::SetCalibrationTone (const FAP &fap)

In v1.0 SDK calibration tone amplitude would be 25% of value provided in fap. Corrected in 1.1.0.

Chapter 12

Namespace Index

12.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 13

Hierarchical Index

13.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 14

Class Index

14.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 15

File Index

15.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 16

Namespace Documentation

16.1 iMS Namespace Reference

The entire API is encapsulated by the iMS namespace.

Classes

- class Auxiliary

  Provides auxiliary additional functions not directly related to Synthesiser operation.
- class AuxiliaryEvents

  All the different types of events that can be triggered by the Auxiliary class.
- class CompensationEvents

  All the different types of events that can be triggered by the Compensation and CompensationTableDownload classes.
- class CompensationFunction

  Class for performing Compensation related functions with the Synthesiser.
- class CompensationFunctionList

  A List of CompensationFunction's used as a container by ImageProject.
- class CompensationPoint

  Stores 4 data fields containing amplitude, phase, sync analogue and sync digital compensation data.
- class CompensationPointSpecification

  Completely specifies the desired compensation at a spot frequency.
- class CompensationTable

  A table of CompensationPoints storing look-up data that can be transferred to memory in the Synthesiser.
- class CompensationTableDownload

  Provides a mechanism for downloading and verifying Compensation Tables to a Synthesiser's Look-Up memory.
- class ConnectionList

  Creates iMS Connection Interfaces and scans them to discover available iMS Systems.
- class DDSScriptDownload

  Provides a mechanism for transferring DDS Scripts into Filesystem memory.
- class DDSScriptRegister

  Create a register write to send to the DDS IC.
- class Degrees

  Type Definition for all operations that require an angle specification in degrees.
- class DequeBase

  Template Class encapsulating a deque object and acting as a base deque class for other classes in the library to inherit from.
- class Diagnostics

Provides a mechanism for retrieving diagnostics data about the attached iMS System.
- class DiagnosticsEvents

  All the different types of events that can be triggered by the Diagnostics class.
- struct FAP

  FAP (Frequency/Amplitude/Phase) triad stores the instantaneous definition of a single RF output.
- class FileSystemManager

  Provides user management operations for working with Synthesiser FileSystems.
- struct FileSystemTableEntry

  Contains all the parameters that uniquely locate a File within the Synthesiser FileSystem.
- class FileSystemTableViewer

  Provides a mechanism for viewing the FileSystemTable associated with an iMS System.
- class Frequency

  Type Definition for all operations that require a frequency specification.
- struct FWVersion

  Stores the version number of firmware running on iMS hardware.
- class IBulkTransfer

  Interface Specification class for sending large binary data objects to the iMS.
- class IEventHandler

  Interface Class for an Event Handler to be defined in User Code and subscribed to library events.
- class Image

  A sequence of ImagePoints played out sequentially by the Controller and driven by the Synthesiser.
- class ImageDownload

  Provides a mechanism for downloading and verifying Images to a Controller's memory.
- class ImageDownloadEvents

  All the different types of events that can be triggered by the ImageDownload class.
- class ImageGroup

  An ImageGroup collects together multiple associated images and a single ImageSequence for controlling Image playback order.
- class ImageGroupList

  A List of ImageGroup's used as a container by ImageProject.
- class ImagePlayer

  Once an Image has been downloaded to Controller memory, ImagePlayer can be used to configure and begin playback.
- class ImagePlayerEvents

  All the different types of events that can be triggered by the ImagePlayer class.
- class ImagePoint

  Stores 4 FAP Triads containing frequency, amplitude and phase data for 4 RF channels.
- class ImageProject

  An ImageProject allows the user to organise their data and store it on the host computer.
- class ImageSequence

  An ImageSequence object completely defines a sequence to be played back on an iMS Controller in terms by containing a list of ImageSequenceEntry 's plus a terminating action and optional value.
- struct ImageSequenceEntry

  An ImageSequenceEntry object can be created by application software to specify the parameters by which an Image is played back during an ImageSequence.
- struct ImageTableEntry

  An ImageTableEntry is created by the SDK on connecting to an iMS System, one for each Image that is stored in Controller memory and allocated in the Image Index Table. Further ImageTableEntries are added to the table each time an Image is downloaded to the Controller.
- class ImageTableViewer

  Provides a mechanism for viewing the ImageTable associated with an iMS System.
- class IMSController

Stores Capabilities, Description, Model & Version Number of an iMS Controller.

- class IMSOption

  An iMS Synthesiser can support one iMS Option, which adds an additional hardware function to the capabilities of the Synthesiser.
- class IMSSynthesiser

  Stores Capabilities, Description, Model & Version Number of an iMS Synthesiser.
- class IMSSystem

  An object representing the overall configuration of an attached iMS System and permits applications to connect to it.
- class kHz

  Type Definition for all operations that require a frequency specification in kiloHertz.
- class LibVersion

  Access the version information for the API.
- class ListBase

  Template Class encapsulating a list object and acting as a base list class for other classes in the library to inherit from.
- class MHz

  Type Definition for all operations that require a frequency specification in MegaHertz.
- class Percent

  Type Definition for all operations that require a percentage specification.
- class RFChannel

  Type that represents the integer values 1, 2, 3 and 4, one each for the RF Channels of an iMS Synthesiser.
- class SequenceDownload

  This class is a worker for transmitting an ImageSequence to an iMS Controller and joining it to the back of the sequence queue.
- class SequenceEvents

  All the different types of events that can be triggered by the SequenceManager class.
- class SequenceManager
- class SignalPath

  Controls Signal routing and other parameters related to the RF output signals.
- class SignalPathEvents

  All the different types of events that can be triggered by the SignalPath class.
- struct StartupConfiguration

  The Synthesiser stores in its non-volatile memory a set of configuration values that are preloaded on startup.
- class SystemFunc

  Provides System Management functions not directly related to RF signal generation or signal path control.
- class SystemFuncEvents

  All the different types of events that can be triggered by the SystemFunc class.
- class ToneBuffer

  An array of 4-channel FAP Tones stored in memory on the Synthesiser.
- class ToneBufferDownload

  Provides a mechanism for downloading ToneBuffer's to a Synthesiser's LTB memory.
- class ToneBufferEvents

  All the different types of events that can be triggered by the ToneBuffer and ToneBufferDownload classes.
- class ToneBufferList

  A List of ToneBuffer's used as a container by ImageProject.
- class UserFileReader

  Provides a mechanism for retrieving User File data from the Synthesiser FileSystem.
- class UserFileWriter

  Provides a mechanism for committing User File data to the Synthesiser FileSystem.
- struct VelocityConfiguration

  Sets the parameters required to control the operation of the Encoder Input / Velocity Compensation function.

Typedefs

- using DDSScript = std::vector< DDSScriptRegister >

  `DDSScript` stores the sequence of register writes to be loaded onto the Synthesiser. Can be manipulated using the normal container operations provided by std::vector
- using FileSystemIndex = int

  FileSystemIndex represents the entry number for a particular file in the FileSystemTable.
- using ImageIndex = int

  Each ImageIndex is an offset into the Image Index Table that uniquely refers to an Image stored in Controller Memory.
- typedef ImageGroup ImageFile

  For backwards compatibility with code written against SDK 1.2.6 or earlier.
- using TBEntry = ImagePoint

  TBEntry is synonymous with ImagePoint An entry in the Tone Buffer contains four FAPs, one per output channel and is therefore comparable to a single ImagePoint making up one entry in an Image.

Enumerations

- enum FileSystemTypes : std::uint8_t {
  FileSystemTypes::NO_FILE = 0, FileSystemTypes::COMPENSATION_TABLE = 1, FileSystemTypes::TON↩
  E_BUFFER = 2, FileSystemTypes::DDS_SCRIPT = 3,
  FileSystemTypes::USER_DATA = 15 }

  All of the different (up to 15) types of file available to the filesystem.
- enum FileDefault : bool { FileDefault::DEFAULT = true, FileDefault::NON_DEFAULT = false }

  Default flag tags a file entry for execution at startup (only one per filetype)
- enum ImageRepeats { ImageRepeats::NONE, ImageRepeats::PROGRAM, ImageRepeats::FOREVER }

  Each Image can be repeated, either a programmable number of times, or indefinitely.
- enum SequenceTermAction : std::uint8_t {
  SequenceTermAction::DISCARD = 0, SequenceTermAction::RECYCLE = 1, SequenceTermAction::STOP↩
  _DISCARD = 2, SequenceTermAction::STOP_RECYCLE = 3,
  SequenceTermAction::REPEAT = 4, SequenceTermAction::REPEAT_FROM = 5 }

  Operation to perform on the completion of the last repeat of the last entry in a Sequence.

Variables

- const unsigned int MAX_FST_ENTRIES = 33

  Maximum number of entries that may be stored in the FileSystem.

16.1.1 Detailed Description

The entire API is encapsulated by the iMS namespace.

In User application code, either add the line 'using namespace iMS;' at the start of your application, or prefix all classes, functions etc with 'iMS::'

Author

Dave Cowan

Since

1.0

16.1.2 Typedef Documentation

16.1.2.1 using iMS::TBEntry = typedef ImagePoint

TBEntry is synonymous with ImagePoint An entry in the Tone Buffer contains four FAPs, one per output channel and is therefore comparable to a single ImagePoint making up one entry in an Image.

Since

1.1

16.1.3 Enumeration Type Documentation

16.1.3.1 enum iMS::FileDefault : bool [strong]

Default flag tags a file entry for execution at startup (only one per filetype)

Since

1.1

Enumerator

DEFAULT Default indicates the Synthesiser should attempt to execute that file during its startup procedure.

NON_DEFAULT Non-default is the normal state for most files.

16.1.3.2 enum iMS::FileSystemTypes : std::uint8_t [strong]

All of the different (up to 15) types of file available to the filesystem.

Since

1.1

Enumerator

NO_FILE No file stored at this FileSystemTable entry.

COMPENSATION_TABLE File contains Compensation table data.

TONE_BUFFER File contains ToneBuffer data.

DDS_SCRIPT File contains a DDS Script for manual programming of the DDS.

USER_DATA File contains user data for application use.

16.1.3.3 enum iMS::ImageRepeats [strong]

Each Image can be repeated, either a programmable number of times, or indefinitely.

Since

1.2.1

Enumerator

NONE The Image is played back only once.

PROGRAM The Image is played back a programmable number of times according to the value set in the PlayConfiguration table.

FOREVER The Image is played back repeatedly until stopped by the application.

16.1.3.4 enum iMS::SequenceTermAction : std::uint8_t [strong]

Operation to perform on the completion of the last repeat of the last entry in a Sequence.

Since

1.2.4

Enumerator

DISCARD Delete the ImageSequence from the Sequence Queue and move on to the next Sequence, if it exists, otherwise Stop.

RECYCLE Move the ImageSequence to the end of Sequence Queue and move on to the next Sequence, if it exists, otherwise repeat this ImageSequence.

STOP_DISCARD Delete the ImageSequence from the Sequence Queue and stop playback.

STOP_RECYCLE Move the ImageSequence to the end of Sequence Queue and stop playback.

REPEAT No effect on the Sequence Queue. Repeat the current Sequence.

REPEAT_FROM No effect on the Sequence Queue. Repeat the current Sequence starting from the ImageSequenceEntry index specified in the Termination Value.

Chapter 17

Class Documentation

17.1 iMS::Auxiliary Class Reference

Provides auxiliary additional functions not directly related to Synthesiser operation.

```
#include <include\Auxiliary.h>
```

Public Types

- enum LED_SOURCE : std::uint16_t {
  LED_SOURCE::OFF = 0, LED_SOURCE::ON = 1, LED_SOURCE::PULS = 2, LED_SOURCE::NPULS = 3,
  LED_SOURCE::PIXEL_ACT = 4, LED_SOURCE::CTRL_ACT = 5, LED_SOURCE::COMMS_HEALTHY = 6,
  LED_SOURCE::COMMS_UNHEALTHY = 7,
  LED_SOURCE::RF_GATE = 8, LED_SOURCE::INTERLOCK = 9, LED_SOURCE::LASER = 10, LED_SO↩URCE::CHECKSUM = 11,
  LED_SOURCE::OVERTEMP = 12, LED_SOURCE::PLL_LOCK = 13 }

  Selects the function to be assigned to an LED.
- enum LED_SINK { LED_SINK::GREEN, LED_SINK::YELLOW, LED_SINK::RED }

  Which LED to assign function to.
- enum DDS_PROFILE : std::uint16_t { DDS_PROFILE::OFF = 0, DDS_PROFILE::EXTERNAL = 16, DDS_↩PROFILE::HOST = 32 }

  Control Source for Profile input to DDS Synthesiser IC.
- enum EXT_ANLG_INPUT { EXT_ANLG_INPUT::A, EXT_ANLG_INPUT::B }

  Reference enum for addressing both analog inputs.

Public Member Functions

Constructor & Destructor

- Auxiliary (const IMSSystem &ims)

  Constructor for Auxiliary Object.
- ∼Auxiliary ()

  Destructor for Auxiliary Object.

LEDs

- bool AssignLED (const LED_SINK &sink, const LED_SOURCE &src) const

  Assignment function for LEDs.

DDS Profile Control

- bool SetDDSProfile (const DDS_PROFILE &prfl) const

  Control the DDS Profile feature.
- bool SetDDSProfile (const DDS_PROFILE &prfl, const std::uint16_t &select) const

External Analog I/O

- bool UpdateAnalogIn ()

  Instructs the synthesiser to capture the current value of both the external analog inputs.
- const std::map$<$ EXT_ANLG_INPUT, Percent $>$ & GetAnalogData () const

  Returns the analog measurements read by the conversion triggered by a call to UpdateAnalogIn()
- bool UpdateAnalogOut (Percent &pct) const

  Instructs the synthesiser to update the analog output value provided externally.

Event Notifications

- void AuxiliaryEventSubscribe (const int message, IEventHandler $*$handler)

  Subscribe a callback function handler to a given AuxiliaryEvents event.
- void AuxiliaryEventUnsubscribe (const int message, const IEventHandler $*$handler)

  Unsubscribe a callback function handler from a given AuxiliaryEvents event.

17.1.1 Detailed Description

Provides auxiliary additional functions not directly related to Synthesiser operation.

Author

Dave Cowan

Date

2016-02-18

Since

1.1

17.1.2 Member Enumeration Documentation

17.1.2.1 enum iMS::Auxiliary::DDS_PROFILE : std::uint16_t [strong]

Control Source for Profile input to DDS Synthesiser IC.

Since

1.1

Enumerator

OFF Profile Selection disabled (default)

EXTERNAL Profile can be controlled from external signal pin inputs.

HOST Profile can be controlled from user application software.

17.1.2.2 enum iMS::Auxiliary::EXT_ANLG_INPUT [strong]

Reference enum for addressing both analog inputs.

Since

1.1

Enumerator

A Refer to analog input A.

B Refer to analog input B.

17.1.2.3 enum iMS::Auxiliary::LED_SINK [strong]

Which LED to assign function to.

Since

1.1

Enumerator

GREEN Synthesiser Green LED.

YELLOW Synthesiser Yellow LED.

RED Synthesiser Red LED.

17.1.2.4 enum iMS::Auxiliary::LED_SOURCE : std::uint16_t [strong]

Selects the function to be assigned to an LED.

Since

1.1

Enumerator

OFF LED turned off.

ON LED turned on.

PULS LED slowly pulses.

NPULS LED slowly pulses with opposite phase to PULS.

PIXEL_ACT Illuminates whenever there is activity on the Pixel Interface between Controller and Synthesiser.

CTRL_ACT Illuminates whenever serial communications activity is detected.

COMMS_HEALTHY Illuminates when communications is in a normal condition.

COMMS_UNHEALTHY Illuminates when Communications Healthy state has detected a timeout (no message received within healthy comms window)

RF_GATE Illuminates when RF Gate to power amplifier is enabled and interlock is not set.

INTERLOCK Illuminates when interlock is active (overtemperature, user disabled or no connection to amplifier/acoust-optic device)

LASER Illuminates when external equipment is turned on by user.

CHECKSUM Illuminates when a checksum error is detected on the pixel interface between Controller and Synthesiser (remains on until cleared in software)

OVERTEMP Illuminates when iMS system is overtemperature or a fan has failed.

PLL_LOCK Illuminates when master clock circuit PLL is locked (either to internal TCXO or externally supplied reference)

17.1.3 Constructor & Destructor Documentation

17.1.3.1 iMS::Auxiliary::Auxiliary (const IMSSystem & *ims*)

Constructor for Auxiliary Object.

An IMSSystem object, representing the configuration of an iMS target must be passed by const reference to the Auxiliary constructor.

The IMSSystem object must exist before the Auxiliary object, and must remain valid (not destroyed) until the Auxiliary object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System

Since

1.1

17.1.4 Member Function Documentation

17.1.4.1 bool iMS::Auxiliary::AssignLED (const LED_SINK & *sink*, const LED_SOURCE & *src*) const

Assignment function for LEDs.

Provide two inputs indicating which LED to target and what function to assign to it.

Parameters

in	*sink*	Which LED to target
in	*src*	the function that the LED should now perform

Returns

true if the assignment request was sent successfully

Since

1.1

17.1.4.2 void iMS::Auxiliary::AuxiliaryEventSubscribe (const int *message*, IEventHandler ∗ *handler*)

Subscribe a callback function handler to a given AuxiliaryEvents event.

Auxiliary can callback user application code when an event occurs that affects the signal path. Supported events are listed under AuxiliaryEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to a AuxiliaryEvents event. For the period that a callback is subscribed, each time an event in Auxiliary occurs that would trigger the subscribed AuxiliaryEvents event, the user function callback will be executed.

Parameters

in	*message*	Use the AuxiliaryEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.1

17.1.4.3 void iMS::Auxiliary::AuxiliaryEventUnsubscribe (const int *message*, const IEventHandler ∗ *handler*)

Unsubscribe a callback function handler from a given AuxiliaryEvents event.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the Auxiliary object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the AuxiliaryEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.1

17.1.4.4 const std::map<EXT_ANLG_INPUT, Percent>& iMS::Auxiliary::GetAnalogData ( ) const

Returns the analog measurements read by the conversion triggered by a call to UpdateAnalogIn()

Returns

a std::map containing one entry for each analog input to the Synthesiser. The value associated with each entry in the map is returned as a percentage object where 100% represents the full scale analog voltage (typically 10.0V)

Since

1.1

17.1.4.5 bool iMS::Auxiliary::SetDDSProfile (const DDS_PROFILE & *prfl*) const

Control the DDS Profile feature.

The DDS IC used at the heart of the Synthesiser has a 4-wide signal input that can be used for modulation (FSK, PSK, ASK), to start/stop the sweep accumulators or used to ramp up/ramp down the output amplitude. By default, the feature is disabled but this function can be used to set the control source for the profile signal either to external for hardware selection or to host for software selection

Parameters

in	*prfl*	select the profile pin control source

Returns

true if the profile control soruce request was sent successfully

17.1.4.6 bool iMS::Auxiliary::SetDDSProfile (const DDS_PROFILE & *prfl*, const std::uint16_t & *select*) const

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

in	*select*	chooses the profile signal value to provide when driven from software
in	*prfl*	select the profile pin control source

17.1.4.7 bool iMS::Auxiliary::UpdateAnalogIn ( )

Instructs the synthesiser to capture the current value of both the external analog inputs.

There are 2 external analog input sources which can provide an auxiliary measurement of external signal data to user software for example to monitor environmental data.

Call this function to initiate a measurement conversion. Once completed, the results will be returned to user code by a callback with Event EXT_ANLG_UPDATE_AVAILABLE. The callback handler can then read the conversion results from the GetAnalogData() function.

Returns

true if the conversion request was sent successfully.

Since

1.1

17.1.4.8 bool iMS::Auxiliary::UpdateAnalogOut (Percent & *pct*) const

Instructs the synthesiser to update the analog output value provided externally.

There is a single channel of analog output data which may provide an auxiliary analog signal to the external signal for example to indicate some internal system parameter state.

Parameters

in	*pct*	The percentage value to output where 100% represents full scale analog voltage (typ. 10.0V)

Returns

true if the update request was sent successfully

The documentation for this class was generated from the following file:

- Auxiliary.h

17.2 iMS::AuxiliaryEvents Class Reference

All the different types of events that can be triggered by the Auxiliary class.

`#include <include\Auxiliary.h>`

Public Types

- enum Events { EXT_ANLG_UPDATE_AVAILABLE, EXT_ANLG_READ_FAILED, **Count** }

  List of Events raised by the Auxiliary module.

17.2.1 Detailed Description

All the different types of events that can be triggered by the Auxiliary class.

Some events contain floating point parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2016-02-11

Since

1.1

17.2.2 Member Enumeration Documentation

17.2.2.1 enum iMS::AuxiliaryEvents::Events

List of Events raised by the Auxiliary module.

Enumerator

EXT_ANLG_UPDATE_AVAILABLE Previous Analog Input Update request completed; data available to be read.

EXT_ANLG_READ_FAILED Previous Analog Input Update request completed; request failed.

The documentation for this class was generated from the following file:

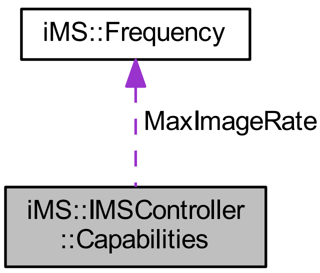
- Auxiliary.h

17.3 iMS::IMSController::Capabilities Struct Reference

Returns information about the capabilities of the Controller hardware.

`#include <include/IMSSystem.h>`

Collaboration diagram for iMS::IMSController::Capabilities:



Public Attributes

- int nSynthInterfaces { 1 }

  A Controller can have multiple Synthesiser interfaces. This field reports how many there are (NOT necessarily how many Synthesisers are connected)
- bool FastImageTransfer { false }

  Some Controllers support a mechanism for transferring bulk Image data much faster than through the standard protocol.
- int MaxImageSize { 4096 }

  The maximum number of points that can be stored in a single Image downloaded to the Controller.
- bool SimultaneousPlayback { false }

  Indicates whether the Controller supports Image downloading and Image playback simultaneously.
- Frequency MaxImageRate { 250.0 }

  The maximum clock rate supported during Image playback.

17.3.1 Detailed Description

Returns information about the capabilities of the Controller hardware.

This struct is initialised during the Connection Scan process

Author

Dave Cowan

Date

2015-11-03

Since

1.0

The documentation for this struct was generated from the following file:

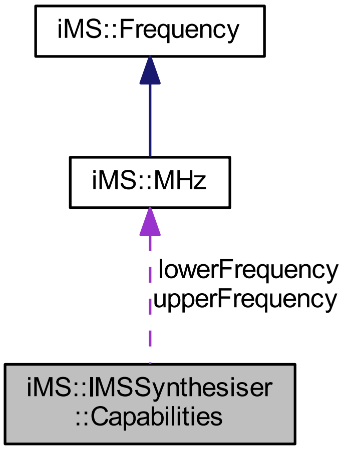
- IMSSystem.h

17.4 iMS::IMSSynthesiser::Capabilities Struct Reference

Returns information about the capabilities of the Synthesiser hardware.

```
#include <include/IMSSystem.h>
```

Collaboration diagram for iMS::IMSSynthesiser::Capabilities:



Public Attributes

- MHz lowerFrequency { 0.0 }

  the Lowest RF output frequency that can be reproduced by the Synthesiser
- MHz upperFrequency { 250.0 }

  the Highest RF output frequency that can be reproduced by the Synthesiser
- int freqBits { 16 }

  the internal bit representation of RF frequency data
- int amplBits { 10 }

  the internal bit representation of RF amplitude data
- int phaseBits { 12 }

  the internal bit representation of RF phase data
- int LUTDepth { 12 }

  the power-of-2 length of Compensation Tables (number of frequency bits used to address the table)
- int LUTAmplBits { 12 }

  the field width of amplitude data stored in the Compensation Tables
- int LUTPhaseBits { 14 }

  the field width of phase data stored in the Compensation Tables
- int LUTSyncABits { 12 }

  the field width of analogue synchronous data stored in the Compensation Tables
- int LUTSyncDBits { 12 }

  the field width of digital synchronous data stored in the Compensation Tables

17.4.1 Detailed Description

Returns information about the capabilities of the Synthesiser hardware.

This struct is initialised during the Connection Scan process

Author

Dave Cowan

Date

2015-11-03

Since

1.0

The documentation for this struct was generated from the following file:

- IMSSystem.h

17.5 iMS::CompensationEvents Class Reference

All the different types of events that can be triggered by the Compensation and CompensationTableDownload classes.

```
#include <include\Compensation.h>
```

Public Types

- enum Events {
  RX_DDS_POWER, DOWNLOAD_FINISHED, DOWNLOAD_ERROR, VERIFY_SUCCESS,
  VERIFY_FAIL, **Count** }

  List of Events raised by the Compensation Class and Compensation Table Downloader.

17.5.1 Detailed Description

All the different types of events that can be triggered by the Compensation and CompensationTableDownload classes.

Some events contain integer parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.5.2 Member Enumeration Documentation

17.5.2.1 enum **iMS::CompensationEvents::Events**

List of Events raised by the Compensation Class and Compensation Table Downloader.

Enumerator

RX_DDS_POWER Not used.

DOWNLOAD_FINISHED Event raised when CompensationTableDownload has confirmed that the iMS Controller received all of the Compensation Table data.

DOWNLOAD_ERROR Event raised each time the CompensationTableDownload class registers an error in the download process.

VERIFY_SUCCESS Event raised on completion of a download verify, if the download was successfully verified.

VERIFY_FAIL Event raised on completion of a download verify, if the download failed. `param` contains the number of failures recorded.

The documentation for this class was generated from the following file:

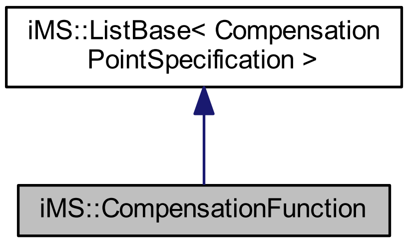
- Compensation.h

17.6 iMS::CompensationFunction Class Reference

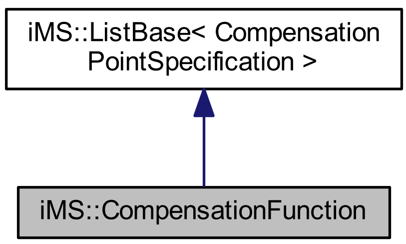
Class for performing Compensation related functions with the Synthesiser.

`#include <include/Compensation.h>`

Inheritance diagram for iMS::CompensationFunction:



Collaboration diagram for iMS::CompensationFunction:



Public Member Functions

Constructor & Destructor

- CompensationFunction ()
  Constructor for Compensation Object.
- ∼CompensationFunction ()
  Destructor for Compensation Object.
- CompensationFunction (const CompensationFunction &)
  Copy Constructor.
- CompensationFunction & operator= (const CompensationFunction &)
  Assignment Constructor.

Additional Inherited Members

17.6.1 Detailed Description

Class for performing Compensation related functions with the Synthesiser.

The purpose of this class is to perform compensation tasks such as measuring the diffraction efficiency of an AO device across a range of frequencies. Such data can then be used to build Compensation tables.

It is not used for storing Compensation Table data or for downloading Compensation Tables. See the CompensationTable and CompensationTableDownload classes for these requirements.

Author

Dave Cowan

Date

2016-11-03

Since

1.3

17.6.2 Constructor & Destructor Documentation

17.6.2.1 iMS::CompensationFunction::CompensationFunction ( )

Constructor for Compensation Object.

Since

1.3

The documentation for this class was generated from the following file:

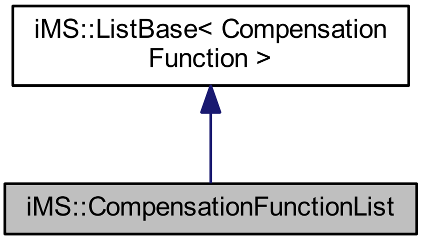
- Compensation.h

17.7 iMS::CompensationFunctionList Class Reference

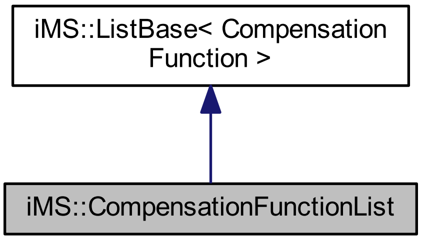
A List of CompensationFunction's used as a container by ImageProject.

```
#include <include/Image.h>
```

Inheritance diagram for iMS::CompensationFunctionList:



Collaboration diagram for iMS::CompensationFunctionList:



Additional Inherited Members

17.7.1 Detailed Description

A List of CompensationFunction's used as a container by ImageProject.

Date

2016-11-09

Since

1.3

The documentation for this class was generated from the following file:

- ImageProject.h

17.8 iMS::CompensationPoint Class Reference

Stores 4 data fields containing amplitude, phase, sync analogue and sync digital compensation data.

```
#include <include/Compensation.h>
```

Public Member Functions

- CompensationPoint (const CompensationPoint &)
  Copy Constructor.
- CompensationPoint & operator= (const CompensationPoint &)
  Assignment Constructor.
- bool operator== (CompensationPoint const &rhs) const
  Equality Operator.

Constructors & Destructor

- CompensationPoint (Percent ampl=0.0, Degrees phase=0.0, unsigned int sync_dig=0, double sync_anlg=0.0)
- ∼**CompensationPoint** ()

Get/Set field data for the CompensationPoint

- void Amplitude (const Percent &ampl)
  Setter for Amplitude field.
- const Percent & Amplitude () const
  Getter for Amplitude field.
- void Phase (const Degrees &phase)
  Setter for Phase field.
- const Degrees & Phase () const
  Getter for Phase field.
- void SyncDig (const unsigned int &sync)
  Setter for Digital Sync Data field.
- const std::uint32_t & SyncDig () const
  Getter for Digital Sync Data field.
- void SyncAnlg (const double &sync)
  Setter for Analogue Sync Data field.
- const double & SyncAnlg () const
  Getter for Analogue Sync Data field.

17.8.1 Detailed Description

Stores 4 data fields containing amplitude, phase, sync analogue and sync digital compensation data.

A CompensationPoint represents one entry in the CompensationTable and is defined for a fixed frequency that is linearly spaced within the frequency range reproducible by the Synthesiser.

Each point has 4 fields, one each for amplitude compensation, phase steering, synchronous analogue and digital data.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.8.2 Constructor & Destructor Documentation

17.8.2.1 iMS::CompensationPoint::CompensationPoint (Percent *ampl* = 0.0, Degrees *phase* = 0.0, unsigned int *sync_dig* = 0, double *sync_anlg* = 0.0)

brief Compensation Point Constructor

Parameters

in	*ampl*	The initial Amplitude Compensation value
in	*phase*	The initial Phase Steering value
in	*sync_dig*	The initial Synchronous Digital Data value
in	*sync_anlg*	The initial Synchronous Analogue Data value

Since

1.0

17.8.3 Member Function Documentation

17.8.3.1 void iMS::CompensationPoint::Amplitude (const Percent & *ampl*)

Setter for Amplitude field.

Amplitude, specified as a percentage figure from 0 - 100%, is applied to the signal amplitude passing from the Controller to the Synthesiser, resulting in a combined amplitude signal that is compensated for any variation in frequency response of the RF signal chain.

Parameters

in	*ampl*	The Amplitude value to set the Compensation field to

17.8.3.2 const Percent& iMS::CompensationPoint::Amplitude ( ) const

Getter for Amplitude field.

Returns

the CompensationPoint's Amplitude value

17.8.3.3 bool iMS::CompensationPoint::operator== (CompensationPoint const & *rhs*) const

Equality Operator.

Since

1.3

17.8.3.4 void iMS::CompensationPoint::Phase (const Degrees & *phase*)

Setter for Phase field.

Phase, specified in Degrees from 0 - 360, defines an additional phase offset applied to RF Channel 2 compared with RF Channel 1. The same phase offset is added cumulatively to subsequent output channels so that RF Channel 4 has an offset of 3 times the table phase value when compared with RF Channel 1.

Parameters

in	*phase*	The Phase value to set the Compensation field to

17.8.3.5 const Degrees& iMS::CompensationPoint::Phase ( ) const

Getter for Phase field.

Returns

the CompensationPoint's Phase value

17.8.3.6 void iMS::CompensationPoint::SyncAnlg (const double & *sync*)

Setter for Analogue Sync Data field.

Analogue Sync data can be routed to the SDAC signals output externally from the Synthesiser. They can be used for custom-scaled analogue frequency signals or any other purpose that requires a frequency-dependent analogue signal. The analogue value is specified in the range 0.0 to +1.0 which is converted to an unsigned bit representation stored in the CompensationTable. Any values outside the range will be clamped. The number of bits used is hardware dependent and can be read from the IMSSynthesiser::Capabilities struct.

Parameters

`in`	*sync*	The Analogue Sync value to set the Compensation field to

17.8.3.7 const double& iMS::CompensationPoint::SyncAnlg ( ) const

Getter for Analogue Sync Data field.

Returns

the CompensationPoint's Analogue Sync Data field

17.8.3.8 void iMS::CompensationPoint::SyncDig (const unsigned int & *sync*)

Setter for Digital Sync Data field.

Digital Sync data can be routed to the SDIO signals output externally from the Synthesiser. They can be used for triggering external hardware, for test purposes, or anything else that requires a frequency-dependent logic signal. The number of bits available is dependent on the hardware and can be read from the IMSSynthesiser::Capabilities struct. The least significant bit of the unsigned int always maps to SDIO[0]

Parameters

`in`	*sync*	The Digital Sync value to set the Compensation field to

17.8.3.9 const std::uint32_t& iMS::CompensationPoint::SyncDig ( ) const

Getter for Digital Sync Data field.

Returns

the CompensationPoint's Digital Sync Data field

The documentation for this class was generated from the following file:

- Compensation.h

17.9 iMS::CompensationPointSpecification Class Reference

Completely specifies the desired compensation at a spot frequency.

`#include <include/Compensation.h>`

Constructor & Destructor

- CompensationPointSpecification (CompensationPoint pt=CompensationPoint(), MHz f=50.0)

  Constructor for CompensationPointSpecification Object.
- ∼CompensationPointSpecification ()

  Destructor for CompensationPointSpecification Object.
- CompensationPointSpecification (const CompensationPointSpecification &)

  Copy Constructor.
- CompensationPointSpecification & operator= (const CompensationPointSpecification &)

  Assignment Constructor.
- bool operator== (CompensationPointSpecification const &rhs) const

  Equality Operator.
- void Freq (const MHz &f)

  Sets the frequency (in MHz) at which the CompensationPointSpecification is valid.
- const MHz & Freq ()

  Gets the CompensationPointSpecification frequency.
- void Spec (const CompensationPoint &pt)

  Sets the specification data for this CompensationPointSpecification frequency point.
- const CompensationPoint & Spec ()

  Gets the specification data for this CompensationPointSpecification frequency point.

17.9.1 Detailed Description

Completely specifies the desired compensation at a spot frequency.

A CompensationPointSpecification object is the basic unit of a Compensation Function. It is required to know the Frequency at which the specification is made and this frequency must fall within the frequency range of the Synthesiser on which the resulting CompensationTable will be programmed else the specification will be disregarded in the CompensationFunction calculation.

The calling software can program any of the Compensation parameters (amplitude, phase, synchronous analog or digital) and the programmed value will be used to generate CompensationTable data by the CompensationFunction calculation.

Author

Dave Cowan

Date

2016-11-03

Since

1.3

17.9.2 Constructor & Destructor Documentation

17.9.2.1 iMS::CompensationPointSpecification::CompensationPointSpecification (**CompensationPoint** *pt =* **CompensationPoint**`()`, **MHz** *f =* `50.0`)

Constructor for CompensationPointSpecification Object.

Since

1.3

17.9.3 Member Function Documentation

17.9.3.1 bool iMS::CompensationPointSpecification::operator== (CompensationPointSpecification const & *rhs*) const

Equality Operator.

Since

1.3

The documentation for this class was generated from the following file:

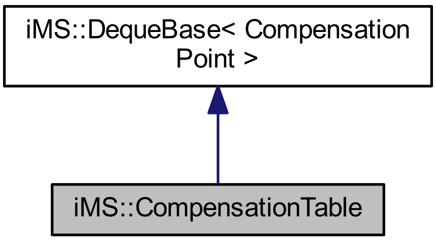
- Compensation.h

17.10 iMS::CompensationTable Class Reference

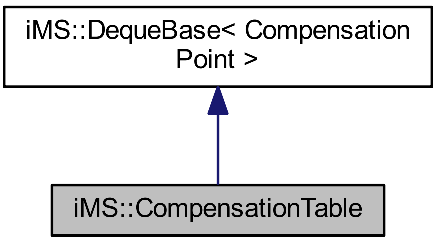
A table of CompensationPoints storing look-up data that can be transferred to memory in the Synthesiser.

`#include <include/Compensation.h>`

Inheritance diagram for iMS::CompensationTable:



Collaboration diagram for iMS::CompensationTable:



Public Member Functions

- const bool Save (const std::string &fileName) const
  Save Table contents to file using latest protocol version.

Constructors & Destructors

- CompensationTable ()
  Default Constructor.
- CompensationTable (const IMSSystem &iMS)
  Empty Constructor.
- CompensationTable (int LUTDepth, const MHz &lower_freq, const MHz &upper_freq)
- CompensationTable (const IMSSystem &iMS, const CompensationPoint &pt)
  Fill Constructor.
- CompensationTable (int LUTDepth, const MHz &lower_freq, const MHz &upper_freq, const CompensationPoint &pt)
- CompensationTable (const IMSSystem &iMS, const std::string &fileName)
  File Read Constructor.
- CompensationTable (int LUTDepth, const MHz &lower_freq, const MHz &upper_freq, const std::string &fileName)
- CompensationTable (const IMSSystem &iMS, const int entry)
  Non-volatile Memory Constructor.
- ∼CompensationTable ()
  Destructor.
- CompensationTable (const CompensationTable &)
  Copy Constructor.
- CompensationTable & operator= (const CompensationTable &)
  Assignment Constructor.

Helper Functions

- const std::size_t Size () const
  Returns the Number of Entries in the CompensationTable.
- const MHz FrequencyAt (const unsigned int index) const
  Returns the frequency represented by a given entry in the CompensationTable.

Additional Inherited Members

17.10.1 Detailed Description

A table of CompensationPoints storing look-up data that can be transferred to memory in the Synthesiser.

A CompensationTable always contains a list of CompensationPoints whose length is defined by the available memory depth in an iMS Synthesiser to which the CompensationTable is targetted.

For this reason, a valid IMSSystem object is required to be passed as a const reference to the Constructor because the table will be initialised to the length of the Synthesiser's look-up memory (read from IMSSynthesiser::Capabilities::LUTDepth). Note that a dummy IMSSystem object could also be created with this field set to the LUT depth (in bits, i.e. 12 => 4096 deep LUT). Once the CompensationTable has been constructed, the IMSSystem object is no longer required and may be destroyed.

The length of the CompensationTable cannot be altered after construction.

The CompensationTable can be constructed with all entries initialised to zero, or to a default value. Subsequently, random access is possible for both reading and modifying CompensationPoints, although a faster method for accessing contents is to use the iterators.

Each entry of a CompensationTable has a unique frequency associated with it. Although not part of the table contents itself, it can be readily calculated from the upper and lower frequency bounds of the Synthesiser. A helper function is available to do this calculation.

A CompensationTable may be saved to disk in a '.lut' file. A Constructor also exists to read back from a previously saved .lut file, creating a CompensationTable from the contents of the file.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.10.2 Constructor & Destructor Documentation

17.10.2.1 iMS::CompensationTable::CompensationTable ( )

Default Constructor.

The default constructor should not normally be used by application code since the length of the table will be left undefined. However it is a required constructor to complete the ImageProject class. If using, the new object should then be assigned from another CompensationTable to ensure that it does not contain dangling pointers

Since

1.3

17.10.2.2 iMS::CompensationTable::CompensationTable (const IMSSystem & *iMS*)

Empty Constructor.

An IMSSystem object must be passed by const reference to the CompensationTable constructor in order to determine the correct depth of the LUT memory.

Parameters

in	*iMS*	the IMSSystem object representing the system the CompensationTable will be constructed for

Since

1.0

17.10.2.3 iMS::CompensationTable::CompensationTable (int *LUTDepth*, const MHz & *lower_freq*, const MHz & *upper_freq*)

This Explicit Empty Constructor makes it possible to create Compensation Tables without being physically connected to an iMS System.

Parameters

in	*LUTDepth*	the number of entries in the Compensation Look-Up Table
in	*lower_freq*	the Lowest Frequency reproducible by the Synthesiser
in	*upper_freq*	the Highest Frequency reproducible by the Synthesiser

Since

1.3

17.10.2.4 iMS::CompensationTable::CompensationTable (const IMSSystem & *iMS*, const CompensationPoint & *pt*)

Fill Constructor.

Use this constructor to preload the CompensationTable with identical values of `CompensationPoint`

Parameters

in	*iMS*	the IMSSystem object representing the system the CompensationTable will be constructed for
in	*pt*	The CompensationPoint that will fill each of the new elements of the CompensationTable

Since

1.0

17.10.2.5 iMS::CompensationTable::CompensationTable (int *LUTDepth*, const MHz & *lower_freq*, const MHz & *upper_freq*, const CompensationPoint & *pt*)

This Explicit Fill Constructor makes it possible to create Compensation Tables without being physically connected to an iMS System.

Parameters

in	*LUTDepth*	the number of entries in the Compensation Look-Up Table
in	*lower_freq*	the Lowest Frequency reproducible by the Synthesiser
in	*upper_freq*	the Highest Frequency reproducible by the Synthesiser
in	*pt*	The CompensationPoint that will fill each of the new elements of the CompensationTable

Since

1.3

17.10.2.6 iMS::CompensationTable::CompensationTable (const IMSSystem & *iMS*, const std::string & *fileName*)

File Read Constructor.

Use this constructor to preload the CompensationTable with data read in from a file on disk

Parameters

in	*iMS*	the IMSSystem object representing the system the CompensationTable will be constructed for
in	*fileName*	A string pointing to a '*.lut' file on the filesystem containing preexisting CompensationTable data

Since

1.0

17.10.2.7 iMS::CompensationTable::CompensationTable (int *LUTDepth*, const MHz & *lower_freq*, const MHz & *upper_freq*, const std::string & *fileName*)

This Explicit File Read Constructor makes it possible to create Compensation Tables without being physically connected to an iMS System.

Parameters

in	*LUTDepth*	the number of entries in the Compensation Look-Up Table
in	*lower_freq*	the Lowest Frequency reproducible by the Synthesiser
in	*upper_freq*	the Highest Frequency reproducible by the Synthesiser
in	*fileName*	A string pointing to a '*.lut' file on the filesystem containing preexisting

Since

1.3

17.10.2.8 iMS::CompensationTable::CompensationTable (const IMSSystem & *iMS*, const int *entry*)

Non-volatile Memory Constructor.

Use this constructor to preload the CompensationTable with data recalled from an entry in the Synthesiser File↵ System.

Parameters

in	*iMS*	the IMSSystem object representing the system the CompensationTable will be constructed for
in	*entry*	the entry in the FileSystem Table from which to recall a Compensation Table CompensationTable data

Since

1.1

17.10.3 Member Function Documentation

17.10.3.1 const MHz iMS::CompensationTable::FrequencyAt (const unsigned int *index*) const

Returns the frequency represented by a given entry in the CompensationTable.

Each entry in the CompensationTable has an implied frequency at which it will become active.

Parameters

in	*index*	The CompensationTable entry to retrieve the associated Frequency for

Returns

The Frequency value which the CompensationTable entry represents

Since

1.0

17.10.3.2 const bool iMS::CompensationTable::Save (const std::string & *fileName*) const

Save Table contents to file using latest protocol version.

The contents of this CompensationTable can be saved to disk for retrieval at a later time. Calling this function will write out the contents of the table to a file which is opened at the filesystem location given by the string fileName.

Warning

If the file already exists, it is overwritten, without warning.

If the function cannot create the file, it will not save the table, and return false.

fileName can be any valid filesystem location and any name, but we recommend the use of the file extension '.lut'

Parameters

in	*fileName*	the name and location of the file to write CompensationTable data to

Returns

true if the save operation completed successfully

Since

1.0

17.10.3.3 const std::size_t iMS::CompensationTable::Size ( ) const

Returns the Number of Entries in the CompensationTable.

Returns

std::size_t representing the number of CompensationTable entries (which is defined in the Constructor)

Since

1.0

The documentation for this class was generated from the following file:

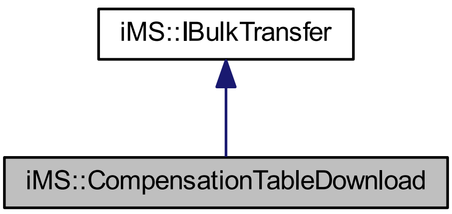
- Compensation.h

17.11 iMS::CompensationTableDownload Class Reference

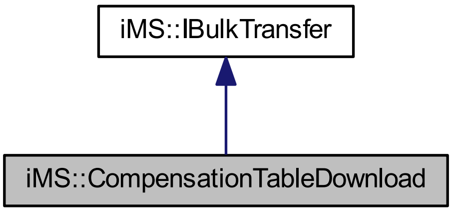
Provides a mechanism for downloading and verifying Compensation Tables to a Synthesiser's Look-Up memory.

`#include <include\Compensation.h>`

Inheritance diagram for iMS::CompensationTableDownload:



Collaboration diagram for iMS::CompensationTableDownload:



Public Member Functions

Constructor & Destructor

- CompensationTableDownload (IMSSystem &ims, const CompensationTable &tbl)
  Constructor for CompensationTableDownload Object.
- ∼CompensationTableDownload ()
  Destructor for CompensationTableDownload Object.

Bulk Transfer Initiation

- bool StartDownload ()
  Initiates a Bulk Transfer download.
- bool StartVerify ()
  Initiates a Bulk Transfer verify.

Retrieve Error Information

- int GetVerifyError ()
  Returns the address of the next verify error or -1 if none.

Event Notifications

- void CompensationTableDownloadEventSubscribe (const int message, IEventHandler ∗handler)
  Subscribe a callback function handler to a given CompensationEvents entry.
- void CompensationTableDownloadEventUnsubscribe (const int message, const IEventHandler ∗handler)
  Unsubscribe a callback function handler from a given CompensationEvents entry.

Store in Synthesiser Non-Volatile Memory

- const FileSystemIndex Store (FileDefault def, const std::string &FileName) const
  Store Table contents to non-volatile memory on the synthesiser.

17.11.1 Detailed Description

Provides a mechanism for downloading and verifying Compensation Tables to a Synthesiser's Look-Up memory.

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.11.2 Constructor & Destructor Documentation

17.11.2.1 iMS::CompensationTableDownload::CompensationTableDownload (IMSSystem & *ims*, const CompensationTable & *tbl*)

Constructor for CompensationTableDownload Object.

The pre-requisites for an CompensationTableDownload object to be created are: (1) - an IMSSystem object, representing the configuration of an iMS target to which the CompensationTable is to be downloaded. (2) - a complete CompensationTable object to download to the iMS target.

CompensationTableDownload stores const references to both. This means that both must exist before the CompensationTableDownload object, and both must remain valid (not destroyed) until the CompensationTableDownload object itself is destroyed. Because they are stored as references, the IMSSystem and CompensationTable objects themselves may be modified after the construction of the CompensationTableDownload object.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System which is the target for downloading the Image
in	*tbl*	A const reference to the CompensationTable which shall be downloaded to the target

Since

1.0

17.11.3 Member Function Documentation

17.11.3.1 void iMS::CompensationTableDownload::CompensationTableDownloadEventSubscribe (const int *message*, IEventHandler ∗ *handler*)

Subscribe a callback function handler to a given CompensationEvents entry.

CompensationTableDownload can callback user application code when an event occurs in the download process. Supported events are listed under CompensationEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to an CompensationEvents entry. For the period that a callback is subscribed, each time an event in CompensationTableDownload occurs that would trigger the subscribed CompensationEvents entry, the user function callback will be executed.

Parameters

in	*message*	Use the CompensationEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.0

17.11.3.2 void iMS::CompensationTableDownload::CompensationTableDownloadEventUnsubscribe (const int *message*, const IEventHandler ∗ *handler*)

Unsubscribe a callback function handler from a given CompensationEvents entry.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the CompensationTableDownload object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the CompensationEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.0

17.11.3.3 int iMS::CompensationTableDownload::GetVerifyError ( ) [virtual]

Returns the address of the next verify error or -1 if none.

After the application has been notified of a failed verify, it can probe the BulkTransfer derived object to obtain the approximate address at which the BulkTransfer failed. The address is provided as a byte offset from the start of the BulkTransfer binary object.

Due to the way in which the BulkTransfer mechanism splits the transfer into individual messages, there will be one error recorded for each message that results in a verify fail. Therefore, the address will only be approximate, to the nearest message size boundary and if there are multiple byte fails within the scope of a single message, only one error will be recorded.

Calling this function repeatedly will result in returning the next recorded verify error. If there are no errors left, or the transfer was successful (i.e. there were no verify failures recorded) the function will return -1.

Returns

byte address of transfer failure or -1 if none.

Since

1.0

Implements iMS::IBulkTransfer.

17.11.3.4 bool iMS::CompensationTableDownload::StartDownload ( ) [virtual]

Initiates a Bulk Transfer download.

If the user has subscribed to the relevant event notifications, the BulkTransfer derived object will issue a completion event at the end of the download process and will also warn the user anytime a download messaging error occurs.

Returns

Boolean indicating whether Download has started successfully

Since

1.0

Implements iMS::IBulkTransfer.

17.11.3.5 bool iMS::CompensationTableDownload::StartVerify ( ) [virtual]

Initiates a Bulk Transfer verify.

If the user has subscribed to the relevant event notifications, the BulkTransfer derived object will raise an event to the application at the end of the verify process to indicate whether the verification was successful or not.

Returns

Boolean indicating whether Verify has started successfully

Since

1.0

Implements iMS::IBulkTransfer.

17.11.3.6 const FileSystemIndex iMS::CompensationTableDownload::Store (FileDefault *def*, const std::string & *FileName*) const

Store Table contents to non-volatile memory on the synthesiser.

The contents of this CompensationTable can be stored to an area of non-volatile memory on the Synthesiser for retrieval at a future time, including after subsequent power cycles. The data stored can be used to select between alternative CompensationTables (e.g. for different AOD crystal materials) without needing to recalculate or download from Software.

The table can be flagged to be used as a default at startup in which case the Synthesiser will use the contents as a default LUT program allowing the Synthesiser to be used with no connection to a host system.

Parameters

in	*def*	mark the entry as a default and the Synthesiser will attempt to program the data to the Local Tone Buffer on power up.
in	*FileName*	a string to tag the download with in the File System Table (limited to 8 chars)

Returns

the index in the File System Table where the data was stored or -1 if the operation failed

Since

1.1

The documentation for this class was generated from the following file:

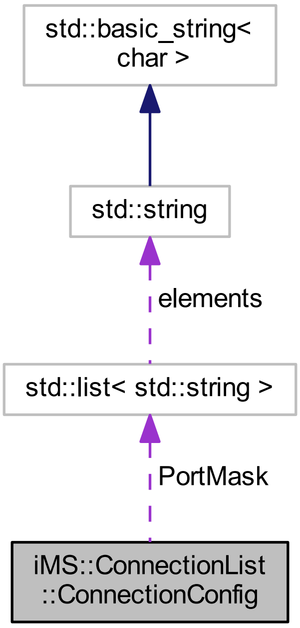
- Compensation.h

17.12 iMS::ConnectionList::ConnectionConfig Struct Reference

Controls the behaviour of a Connection Module during its discovery process.

`#include <include\ConnectionList.h>`

Collaboration diagram for iMS::ConnectionList::ConnectionConfig:



Public Member Functions

- ConnectionConfig (bool inc=true, std::list< std::string > mask=std::list< std::string >())

Public Attributes

- bool IncludeInScan
- std::list< std::string > PortMask

17.12.1 Detailed Description

Controls the behaviour of a Connection Module during its discovery process.

The ConnectionList class maintains an internal map of ConnectionConfig configuration structs, one per module included in the ConnectionList.

Each Connection Module has a discovery mechanism which is invoked when the ConnectionList performs a scan. Before calling the discovery function, the ConnectionList first checks the ConnectionConfigMap for details about how the discovery function for that module should be configured. Firstly, it checks to see if the module should be included in the scan, and only calls the discovery function if this is set to true. Secondly, a user supplied list of strings is passed to the discovery functions which, if non-empty, acts as a mask, only permitting discovery on interface ports that can be matched to an entry in the list. If the list is empty, all interface ports are queried.

Since

1.4.2

17.12.2 Constructor & Destructor Documentation

17.12.2.1 iMS::ConnectionList::ConnectionConfig::ConnectionConfig (bool *inc* = `true`, std::list< std::string > *mask* = `std::list< std::string >()`)

Constructor for ConnectionConfig

Default Constructor enables scan on all available interface ports

17.12.3 Member Data Documentation

17.12.3.1 bool iMS::ConnectionList::ConnectionConfig::IncludeInScan

If true, the Connection Module associated with the ConnectionConfig is enabled for iMS discovery

17.12.3.2 std::list<std::string> iMS::ConnectionList::ConnectionConfig::PortMask

A list of interfaces (ports) that may be queried. For example, an Ethernet Connection Module might include a reference to a host static IP address that is known to reside on a network containing iMS devices (e.g. "192.168.↩ 1.100"). An application might know that it is expecting to find an iMS connected to Windows serial port COM8 so it would add "COM8" to the PortMask. If the PortMask is empty, the module will iterate through every interface port that is available to it.

The documentation for this struct was generated from the following file:

- ConnectionList.h

17.13 iMS::ConnectionList Class Reference

Creates iMS Connection Interfaces and scans them to discover available iMS Systems.

`#include <include/ConnectionList.h>`

Classes

- struct ConnectionConfig

  Controls the behaviour of a Connection Module during its discovery process.

Public Types

- typedef std::map< std::string, ConnectionConfig > ConnectionConfigMap

  Type of the internal object that links Connection Modules to their Configuration structs.

Public Member Functions

- ConnectionList ()

  Constructor initialises the list with the connection types.
- ∼ConnectionList ()

  Destructor.

System Discovery

- ConnectionConfigMap & config ()

Configure the Connection process to each supported Connection interface.
- const std::list< std::string > & modules () const

  Returns a list of string identifiers for each of the Connection Modules.
- std::vector< IMSSystem > scan ()

  Probe each of the known connection types for attached iMS Systems.

17.13.1 Detailed Description

Creates iMS Connection Interfaces and scans them to discover available iMS Systems.

For software to interact with an iMS system, it must first discover it on one of the supported connection types, then open a link to it.

The ConnectionList maintains a private list of modules for all the known supported connection types (USB, Ethernet, etc.). Each connection module is stored as a pointer to an object within this list and implements a common interface so that other code within the library can communicate with the iMS using the module and with no knowledge required about what type of connection is used.

The list of supported module types enabled by the ConnectionList is dependent on which platform the host application is operating on. To see which module types are supported, browse the list of modules returned by the call to `modules()`:

```
auto& modules = connList->modules();
for (auto&& mname : connList->modules())
{
    std::cout << "Module: " << mname << std::endl;
}
```

By default, all modules are enabled to scan for iMS systems and the function call to `scan()` will attempt to open a connection on every available port to the module. In this context, a "port" is a term used generically to refer to a unique point of access on which an iMS or multiple iMS's may be discovered. A module may have multiple ports. For example, the CM_ETH connection module will have one port for each network interface on the system with each interface port being recognised by its host IP address.

Application software can choose to limit the range of the ConnectionList::scan mechanism by only enabling the modules on which the application is expecting to find iMS Systems. Within a single module, the scan can be restricted further by adding a PortMask to the connection configuration. If a PortMask is defined, a module will only scan the ports that are present within it. Limiting the scope of the ConnectionList::scan in either of these ways can dramatically improve application startup time.

Once instantiated, ConnectionList can perform a `scan`, which starts a discovery algorithm on each of the available enabled modules in turn. When complete, it will return an array (std::vector) of IMSSystem objects, each fully populated with the iMS configuration, model, serial number etc.

Attention

It is important to understand that all communications to/from iMS hardware happens through the connection module held in this list, therefore the ConnectionList object once created must be maintained within scope until the software no longer needs to communicate with the iMS. Do not delete this object after the scan has completed, unless you don't intend on communicating with the iMS!

```
#include "ConnectionList.h"
#include "IMSSystem.h"

using namespace iMS;

int main(int argc, char* argv)
{
    // Create List of Connection Modules
    ConnectionList* connList = new ConnectionList();

    // Get Connection List Configuration and list of modules supported on this platform
    auto& conncfg = connList->config();
    auto& modules = connList->modules();

    // Disable scan on serial port connection module
    if (std::find(modules.begin(), modules.end(), "CM_SERIAL") !=
```

```
      modules.end()) conncfg["CM_SERIAL"].IncludeInScan = false;
    // Limit Ethernet Connection Module to only scan on host NIC with IP address 192.168.2.128
    if (std::find(modules.begin(), modules.end(), "CM_ETH") !=
      modules.end()) conncfg["CM_ETH"].PortMask.push_back("192.168.2.128");

    // Scan all enabled connection types for iMS systems and return an array of results
    std::vector<IMSSystem> iMSList = connList->scan();

    // Our iMS object
    IMSSystem myiMS;
    for (std::vector<IMSSystem>::const_iterator iter = iMSList.begin(); iter != iMSList.end(); ++iter)
    {
        myiMS = (*iter);
        // Look for the first iMS system that contains an iMSL type Controller
        if (myiMS.Ctlr().IsValid() && (myiMS.Ctlr().Model == "iMSL"))
        {
            break;
        }
        // None found
        if (iter == iMSList.end())
        {
            std::cout << "No iMS found." << std::endl;
            // Don't forget to free the ConnectionList
            delete connList;
            return -1;
        }
    }

    // Open the connection
    myiMS.Connect();
    std::cout << "Connecting to IMS System on port: " << myiMS.ConnPort() << std::endl;

    // .... Do something with the iMS

    // All done.  Disconnect.
    myiMS.Disconnect();

    // Always free the ConnectionList memory after all iMS functions are complete and connections closed
    delete connList;

    return 0;
}
```

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.13.2 Member Function Documentation

17.13.2.1 ConnectionConfigMap& iMS::ConnectionList::config ( )

Configure the Connection process to each supported Connection interface.

Returns a reference to the internal configuration map. When ConnectionList is constructed, it loads the map with Keys, one for each Connection Module supported by the platform. The Value associated with each key is a ConnectionConfig struct which by default enables iMS discovery on all available interface ports.

It is up to the user's application to restrict the scope of the scan by modifying the configuration as desired. Applications should not add Keys to or remove Keys from the configuration map.

Returns

Returns a reference to the internal Connection Configuration Map

Since

1.4.2

17.13.2.2 const std::list<std::string>& iMS::ConnectionList::modules ( ) const

Returns a list of string identifiers for each of the Connection Modules.

Each Connection Module has a unique string identifier. The string identifier is used as the "Key" in the Connection↵ ConfigMap. This function returns a list of all the Connection Modules supported by this platform.

Returns

Returns a const reference to a list of all supported Connection Modules

Since

1.4.2

17.13.2.3 std::vector<IMSSystem> iMS::ConnectionList::scan ( )

Probe each of the known connection types for attached iMS Systems.

The `scan()` function iterates through the list of connection types, opening a port on each in an implementation defined manner. On a successful open, it will send a sequence of query messages to identify if an iMS Controller and/or an iMS Synthesiser(s) is present. If any of the query messages results in a valid response without timing out, the function creates an IMSSystem object and begins populating the object with information it can find out about it(them) either by sending further query messages to the device or by cross-referencing a hardware database built into the library.

Once all connection types have been probed and all iMS Systems discovered, the IMSSystem objects are loaded into a vector which is returned for application processing.

Returns

Returns an array of discovered iMS Systems

Since

1.0

The documentation for this class was generated from the following file:

- ConnectionList.h

17.14 iMS::DDSScriptDownload Class Reference

Provides a mechanism for transferring DDS Scripts into Filesystem memory.

```
#include <include\Auxiliary.h>
```

Public Member Functions

- DDSScriptDownload (IMSSystem &ims, const DDSScript &script)

  Construct the DDSScriptDownload object from a reference to the iMS device and a const reference to the DDS Script to download.

- ∼DDSScriptDownload ()

  DDSScriptDownload destructur.
- const FileSystemIndex Program (const std::string &FileName, FileDefault def=FileDefault::NON_DEFAULT) const

  Causes the DDS Script object to be programmed into the filesystem.

17.14.1 Detailed Description

Provides a mechanism for transferring DDS Scripts into Filesystem memory.

Use this class to program newly created DDS Scripts to the Synthesiser. The class will automatically find and allocate space in the filesystem and update the filesystem table with the newly created entry.

Setting the FileDefault flag to DEFAULT will cause the downloaded script to be executed at every subsequent powerup.

Use the FileSystemManager class for any additional actions as required, such as setting/clearing default flags, executing scripts and deleting unwanted scripts.

Author

Dave Cowan

Date

2016-03-01

Since

1.1

17.14.2 Constructor & Destructor Documentation

17.14.2.1 iMS::DDSScriptDownload::DDSScriptDownload (IMSSystem & *ims,* const DDSScript & *script*)

Construct the DDSScriptDownload object from a reference to the iMS device and a const reference to the DDS Script to download.

Parameters

in	*ims*	the iMS target System
in	*script*	the DDSScript to download

17.14.3 Member Function Documentation

17.14.3.1 const FileSystemIndex iMS::DDSScriptDownload::Program (const std::string & *FileName,* FileDefault *def =* FileDefault::NON_DEFAULT) const

Causes the DDS Script object to be programmed into the filesystem.

Calculates the amount of storage space required, finds a space large enough and transfers the script byte data to be stored at the selected location in Synthesiser non-volatile memory. The new entry is logged in the Filesystem table (FST) along with the default flag, if set.

Parameters

in	*FileName*	a max 8 char string to use to refer to the DDS Script (stored in the FST)
in	*def*	Optional parameter indicating whether to set the default flag for future startup execution

Returns

the index of the script in the FST (or -1 if programming failed, e.g. insufficient space or no free FST entries)

Since

1.1

The documentation for this class was generated from the following file:

- Auxiliary.h

17.15 iMS::DDSScriptRegister Class Reference

Create a register write to send to the DDS IC.

`#include <include\Auxiliary.h>`

Public Types

- enum Name : std::uint8_t {
  Name::CSR = 0, Name::FR1 = 1, Name::FR2 = 2, Name::CFR = 3,
  Name::CFTW0 = 4, Name::CPOW0 = 5, Name::ACR = 6, Name::LSRR = 7,
  Name::RDW = 8, Name::FDW = 9, Name::CW1 = 10, Name::CW2 = 11,
  Name::CW3 = 12, Name::CW4 = 13, Name::CW5 = 14, Name::CW6 = 15,
  Name::CW7 = 16, Name::CW8 = 17, Name::CW9 = 18, Name::CW10 = 19,
  Name::CW11 = 20, Name::CW12 = 21, Name::CW13 = 22, Name::CW14 = 23,
  Name::CW15 = 24, Name::UPDATE = 64 }

  the abbreviated register name for each register accessible in the DDS IC

Public Member Functions

- int append (const std::uint8_t &)

  Add an additional byte to the end of the data array.
- std::vector< std::uint8_t > bytes () const

  Get the full byte array for programming to the FileSystem Shouldn't be called in user code.

Constructor & Destructor

- DDSScriptRegister (Name name)

  Constructor for DDSScriptRegister Object.
- DDSScriptRegister (Name name, const std::initializer_list< std::uint8_t > &data)
- DDSScriptRegister (const DDSScriptRegister &)

  Copy Constructor.
- DDSScriptRegister & operator= (const DDSScriptRegister &)

  Assignment Constructor.
- ~DDSScriptRegister ()

  Destructor.

17.15.1 Detailed Description

Create a register write to send to the DDS IC.

The DDS IC that generates RF signals on the Synthesiser can be manually programmed to access advanced features that wouldn't normally be available through the iMS API. To do this requires a knowledge and understanding of the Analog Devices AD9959 Frequency Synthesiser IC and its register map.

If it is decided that it is necessary to manually program the AD9959, a sequence of register writes can be generated (called a DDS Script) and stored in the Synthesiser Filesystem. The application software may then recall the script from the filesystem and execute it to commit the register writes to the AD9959.

Each individual register write is an invocation of the DDSScriptRegister class. The class object consists of a key-value pair where the key is the name of the register to access (corresponding to the register abbreviation in the datasheet) and the value is a list of bytes to transfer to the AD9959 following the register command.

There must be the exact number of data bytes sent after the register command as specified in the datasheet. The class knows internally what this number is and enforces it, so that any extra bytes are truncated and any missing are zero-filled.

Note that the bottom four bits of data sent to the CSR register cannot be overwritten since they define the hardware interface to the register access port.

Some of the register writes do not take effect until a signal line called Update Clock is asserted to the AD9959. This can be triggered by creating a DDSScriptRegister object with the Name property set to UPDATE. It takes no byte data as input.

Author

Dave Cowan

Date

2016-03-01

Since

1.1

17.15.2 Member Enumeration Documentation

17.15.2.1 enum iMS::DDSScriptRegister::Name : std::uint8_t [strong]

the abbreviated register name for each register accessible in the DDS IC

Since

1.1

Enumerator

CSR Channel Select Register.

FR1 Function Register 1.

FR2 Function Register 2.

CFR Channel Function Register.

CFTW0 Channel Frequency Tuning Word.

CPOW0 Channel Phase Offset Word.

ACR Amplitude Control Register.

LSRR Linear Sweep Ramp Rate.

RDW LSR Rising Delta Word.

FDW LSR Falling Delta Word.

CW1 Channel Word 1.

CW2 Channel Word 2.

CW3 Channel Word 3.

CW4 Channel Word 4.

CW5 Channel Word 5.

CW6 Channel Word 6.

CW7 Channel Word 7.

CW8 Channel Word 8.

CW9 Channel Word 9.

CW10 Channel Word 10.

CW11 Channel Word 11.

CW12 Channel Word 12.

CW13 Channel Word 13.

CW14 Channel Word 14.

CW15 Channel Word 15.

UPDATE Issue Update Clock (pseudo register write)

17.15.3 Constructor & Destructor Documentation

17.15.3.1 iMS::DDSScriptRegister::DDSScriptRegister (Name *name*)

Constructor for DDSScriptRegister Object.

Example:

```
DDSScriptRegister reg5(DDSScriptRegister::Name::UPDATE);
```

Parameters

in	*name*	Create the register object accessing the specified Register

Since

1.1

17.15.3.2 iMS::DDSScriptRegister::DDSScriptRegister (Name *name*, const std::initializer_list< std::uint8_t > & *data*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Example:

```
DDSScriptRegister reg2(DDSScriptRegister::Name::CFTW0, {
      0x33, 0x33, 0x33, 0x33 });
```

Parameters

in	*name*	Create the register object accessing the specified Register
in	*data*	initialise the data byte array from this input field

17.15.4 Member Function Documentation

17.15.4.1 int iMS::DDSScriptRegister::append (const std::uint8_t &)

Add an additional byte to the end of the data array.

Returns

the new number of bytes in the array

Since

1.1

The documentation for this class was generated from the following file:

- Auxiliary.h

17.16 iMS::Degrees Class Reference

Type Definition for all operations that require an angle specification in degrees.

`#include <include/IMSTypeDefs.h>`

Public Member Functions

- Degrees (double arg)

  Construct a Degrees object from a double argument and check its value is within the range 0.0 <= arg < 360.0. If not, the object is still constructed, but the value is wrapped around to fit within the range.
- Degrees & operator= (double arg)

  Assignment of a double argument in degrees to an existing Degrees object.
- operator double () const

  Return a double representing the Degrees object's value.

Static Public Member Functions

- static unsigned int RenderAsImagePoint (const IMSSystem &, const Degrees)

  Used internally by the library to convert a Degrees object into a hardware-dependent integer representation used by the Image for RF Output phase.
- static unsigned int RenderAsCompensationPoint (const IMSSystem &, const Degrees)

  Used internally by the library to convert a Degrees object into a hardware-dependent integer representation used by the Compensation Table for channel phase increment.
- static unsigned int RenderAsCalibrationTone (const IMSSystem &, const Degrees)

  Used internally by the library to convert a Degrees object into a hardware-dependent integer representation used by the Calibration Tone for channel phase increment.

17.16.1 Detailed Description

Type Definition for all operations that require an angle specification in degrees.

Internally, the Degrees value is stored as a double precision variable and is limited to sit within the range 0.0 <= Percent < 360.0.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.16.2 Constructor & Destructor Documentation

17.16.2.1 iMS::Degrees::Degrees (double *arg*) [inline]

Construct a Degrees object from a double argument and check its value is within the range 0.0 <= arg < 360.0. If not, the object is still constructed, but the value is wrapped around to fit within the range.

Parameters

in	*arg*	The percentage value

Since

1.0

17.16.3 Member Function Documentation

17.16.3.1 iMS::Degrees::operator double ( ) const [inline]

Return a double representing the Degrees object's value.

Since

1.0

17.16.3.2 Degrees& iMS::Degrees::operator= (double *arg*) [inline]

Assignment of a double argument in degrees to an existing Degrees object.

The double argument of the assigner must be within the range 0.0 <= arg < 360.0 else it will be wrapped around to fit within the range.

```
// needed for PI
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>

Degrees phase = atan2(1.0, -1.0) * (360.0 / 2 * M_PI);
std::cout << "The arctangent for [x=-1, y=1] is " << phase << " degrees" << std::endl;
```

prints:

```
The arctangent for [x=-1, y=1] is 135.000000 degrees
```

Since

1.0

17.16.3.3 static unsigned int iMS::Degrees::RenderAsCalibrationTone (const IMSSystem & , const Degrees) [static]

Used internally by the library to convert a Degrees object into a hardware-dependent integer representation used by the Calibration Tone for channel phase increment.

Not intended for use in application code

Since

1.1.0

17.16.3.4 static unsigned int iMS::Degrees::RenderAsCompensationPoint (const IMSSystem & , const Degrees) [static]

Used internally by the library to convert a Degrees object into a hardware-dependent integer representation used by the Compensation Table for channel phase increment.

Not intended for use in application code

17.16.3.5 static unsigned int iMS::Degrees::RenderAsImagePoint (const IMSSystem & , const Degrees) [static]

Used internally by the library to convert a Degrees object into a hardware-dependent integer representation used by the Image for RF Output phase.

Not intended for use in application code

The documentation for this class was generated from the following file:

- IMSTypeDefs.h

17.17 iMS::DequeBase< T > Class Template Reference

Template Class encapsulating a deque object and acting as a base deque class for other classes in the library to inherit from.

`#include <include/Containers.h>`

Public Member Functions

- void clear ()

    clears the contents
- iterator insert (iterator pos, const T &value)

    Inserts a single new element into the DequeBase.
- iterator insert (const_iterator pos, size_t count, const T &value)

    Inserts multiple copies of an element into the DequeBase.
- iterator insert (iterator pos, const_iterator first, const_iterator last)

    Inserts a range of elements into the DequeBase.
- void push_back (const T &value)

    Appends the given element value to the end of the container.

- void pop_back ()

  Removes the last element of the container.
- void push_front (const T &value)

  Prepends the given element value to the beginning of the container.
- void pop_front ()

  Removes the first element of the container.
- iterator erase (iterator pos)

  Removes the element at pos.
- iterator erase (iterator first, iterator last)

  Removes the elements in the range [first; last].
- std::size_t size () const

  Returns the number of elements in the container.

Constructors & Destructor

- DequeBase (const std::string &Name="[no name]", const std::time_t &modified_time=std::time(nullptr))

  Create a default empty List with optional name.
- ~DequeBase ()

  Destructor.
- DequeBase (size_t, const T &, const std::string &Name="[no name]", const std::time_t &modified_↩ time=std::time(nullptr))

  Fill Constructor.
- DequeBase (const_iterator first, const_iterator last, const std::string &Name="[no name]", const std::time↩ _t &modified_time=std::time(nullptr))

  Range Constructor.
- DequeBase (const DequeBase &)

  Copy Constructor.
- DequeBase & operator= (const DequeBase &)

  Assignment Constructor.

Element Access

- T & operator[ ] (int idx)

  Random Write Access to an element in the Deque.
- const T & operator[ ] (int idx) const

  Random Access to an element in the Deque.

DequeBase Unique Identifier

- bool operator== (DequeBase const &rhs) const

  Equality Operator checks Deque object UUID's for equivalence.
- const std::array< std::uint8_t, 16 > GetUUID () const

  Returns a vector representing the Unique Identifier assigned to the DequeBase object.

Timestamping

- const std::time_t & ModifiedTime () const

  Returns Time at which the Container was last modified.
- std::string ModifiedTimeFormat () const

  Returns Human-readable string for the time at which the Container was last modified.

Container Description

- const std::string & Name () const

  A string stored with the Container to aid human users in identifying its purpose.
- std::string & **Name** ()

Iterator Specification

Use these iterators when you want to iteratively read through or update the entries stored within a DequeBase. Iterators can be used to access elements at an arbitrary offset position relative to the element they point to.

Two types of iterators are supported; both are random access iterators. Dereferencing const_iterator yields a reference to a constant entry in the DequeBase(const DequeBase&).

- typedef std::deque< T >::iterator iterator

  Iterator defined for user manipulation of DequeBase.
- typedef std::deque< T >::const_iterator const_iterator

  Const Iterator defined for user readback of DequeBase.
- iterator begin ()

  Returns an iterator pointing to the first element in the DequeBase container.
- iterator end ()

  Returns an iterator referring to the past-the-end element in the DequeBase container.
- const_iterator begin () const

  Returns a const_iterator pointing to the first element in the DequeBase container.
- const_iterator end () const

  Returns a const_iterator referring to the past-the-end element in the DequeBase container.
- const_iterator cbegin () const

  Returns a const_iterator pointing to the first element in the DequeBase container.
- const_iterator cend () const

  Returns a const_iterator referring to the past-the-end element in the DequeBase container.

17.17.1 Detailed Description

template<typename T>class iMS::DequeBase< T >

Template Class encapsulating a deque object and acting as a base deque class for other classes in the library to inherit from.

Date

2016-11-09

Since

1.3

17.17.2 Member Function Documentation

17.17.2.1 template<typename T> iterator iMS::DequeBase< T >::begin ( )

Returns an iterator pointing to the first element in the DequeBase container.

Returns

An iterator to the beginning of the DequeBase container.

17.17.2.2 template<typename T> const_iterator iMS::DequeBase< T >::begin ( ) const

Returns a const_iterator pointing to the first element in the DequeBase container.

Returns

A DequeBase to the beginning of the DequeBase container.

Since

1.2.5

17.17.2.3 template<typename T> const_iterator iMS::DequeBase< T >::cbegin ( ) const

Returns a const_iterator pointing to the first element in the DequeBase container.

Returns

A const_iterator to the beginning of the DequeBase container.

17.17.2.4 template<typename T> const_iterator iMS::DequeBase< T >::cend ( ) const

Returns a const_iterator referring to the past-the-end element in the DequeBase container.

Returns

A const_iterator to the element past the end of the DequeBase.

17.17.2.5 template<typename T> void iMS::DequeBase< T >::clear ( )

clears the contents

Since

1.3

17.17.2.6 template<typename T> iterator iMS::DequeBase< T >::end ( )

Returns an iterator referring to the past-the-end element in the DequeBase container.

The past-the-end element is the theoretical element that would follow the last element in the DequeBase container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function can be used in combination with DequeBase::begin to specify a range including all the elements in the container.

Returns

An iterator to the element past the end of the DequeBase

17.17.2.7 template<typename T> const_iterator iMS::DequeBase< T >::end ( ) const

Returns a const_iterator referring to the past-the-end element in the DequeBase container.

Returns

A const_iterator to the element past the end of the DequeBase.

Since

1.2.5

17.17.2.8 template<typename T> const std::array<std::uint8_t, 16> iMS::DequeBase< T >::GetUUID ( ) const

Returns a vector representing the Unique Identifier assigned to the DequeBase object.

Returns

UUID as an array of uint8_t's

17.17.2.9 template<typename T> iterator iMS::DequeBase< T >::insert (iterator *pos*, const T & *value*)

Inserts a single new element into the DequeBase.

Since

1.3

17.17.2.10 template<typename T> const std::time_t& iMS::DequeBase< T >::ModifiedTime ( ) const

Returns Time at which the Container was last modified.

Any time the container is modified (added to, deleted from, elements updated), the system time is recorded. This happens coincident with the UUID if the container also being updated. This function returns to the user that timestamp.

Returns

a reference to a std::time_t representing the time at which the container was last modified

Since

1.3

17.17.2.11 template<typename T> std::string iMS::DequeBase< T >::ModifiedTimeFormat ( ) const

Returns Human-readable string for the time at which the Container was last modified.

Since

1.3

17.17.2.12 template<typename T> const std::string& iMS::DequeBase< T >::Name ( ) const

A string stored with the Container to aid human users in identifying its purpose.

Updating the Container Name does not cause the Container UUID to change.

17.17.2.13 template<typename T> bool iMS::DequeBase< T >::operator== (DequeBase< T > const & *rhs*) const

Equality Operator checks Deque object UUID's for equivalence.

Each Deque object created in software has its own UUID (Universally Unique ID) assigned. In order to confirm whether two deque objects are identical, their UUIDs are compared. Deque objects can also be compared with Deques residing on iMS Controller hardware, since the UUID of a deque is stored in memory on the hardware.

Parameters

in	*rhs*	A Deque object to perform the comparison with

Returns

True if the supplied Deque is identical to this one.

Since

1.0.1

17.17.2.14 template<typename T> T& iMS::DequeBase< T >::operator[ ] (int *idx*)

Random Write Access to an element in the Deque.

Parameters

in	*idx*	Integer offset into the image with respect to the first element in the sequence (DequeBase::begin())

Returns

A reference to an element.

Since

1.3

17.17.2.15 template<typename T> const T& iMS::DequeBase< T >::operator[ ] (int *idx*) const

Random Access to an element in the Deque.

The fastest and preferred method for reading back elements from a Dequeis to use const_iterator to retrieve elements in sequence. In some circumstances however this is not suitable, and so the array subscript operator is defined to permit applications to access an ImagePoint at any arbitrary position for readback.

Parameters

in	*idx*	Integer offset into the image with respect to the first element in the sequence (DequeBase::cbegin())

Returns

A const reference to an element.

Since

1.0

The documentation for this class was generated from the following file:

- Containers.h

17.18 iMS::Diagnostics Class Reference

Provides a mechanism for retrieving diagnostics data about the attached iMS System.

`#include <include\Diagnostics.h>`

Public Types

- enum TARGET { TARGET::SYNTH, TARGET::AO_DEVICE, TARGET::RF_AMPLIFIER }

    Sets which iMS device to request diagnostics data for.
- enum MEASURE {
  MEASURE::FORWARD_POWER_CH1, MEASURE::FORWARD_POWER_CH2, MEASURE::FORWARD_POWER_CH3, MEASURE::FORWARD_POWER_CH4,
  MEASURE::REFLECTED_POWER_CH1, MEASURE::REFLECTED_POWER_CH2, MEASURE::REFLECTED_POWER_CH3, MEASURE::REFLECTED_POWER_CH4,
  MEASURE::DC_CURRENT_CH1, MEASURE::DC_CURRENT_CH2, MEASURE::DC_CURRENT_CH3,
  MEASURE::DC_CURRENT_CH4 }

    Selects which diagnostics measurement to access.

Public Member Functions

Constructor & Destructor

- Diagnostics (const IMSSystem &ims)

    Constructor for Diagnostics Object.
- ∼Diagnostics ()

    Destructor for Diagnostics Object.

Event Notifications

- void DiagnosticsEventSubscribe (const int message, IEventHandler ∗handler)

    Subscribe a callback function handler to a given DiagnosticsEvents event.
- void DiagnosticsEventUnsubscribe (const int message, const IEventHandler ∗handler)

    Unsubscribe a callback function handler from a given DiagnosticsEvents event.

Read Temperatures

- bool GetTemperature (const TARGET &tgt) const

    Triggers a temperature reading from the target device.

Read Hours

- bool GetLoggedHours (const TARGET &tgt) const

Triggers a logged hours reading from the target device.

Get Diagnostics Information

- bool UpdateDiagnostics ()

  Triggers a Diagnostics Conversion to read measurement data from the RF Power Amplifier.
- const std::map< MEASURE, Percent > & GetDiagnosticsData () const

  Returns a reference to the map of diagnostics data values currently stored by the Diagnostics class.

17.18.1 Detailed Description

Provides a mechanism for retrieving diagnostics data about the attached iMS System.

Author

Dave Cowan

Date

2016-03-08

Since

1.1

17.18.2 Member Enumeration Documentation

17.18.2.1 enum iMS::Diagnostics::MEASURE [strong]

Selects which diagnostics measurement to access.

Since

1.1

Enumerator

FORWARD_POWER_CH1 Forward Measured Power for Channel 1.

FORWARD_POWER_CH2 Forward Measured Power for Channel 2.

FORWARD_POWER_CH3 Forward Measured Power for Channel 3.

FORWARD_POWER_CH4 Forward Measured Power for Channel 4.

REFLECTED_POWER_CH1 Reflected Measured Power for Channel 1.

REFLECTED_POWER_CH2 Reflected Measured Power for Channel 2.

REFLECTED_POWER_CH3 Reflected Measured Power for Channel 3.

REFLECTED_POWER_CH4 Reflected Measured Power for Channel 4.

DC_CURRENT_CH1 Measured DC Current for Channel 1.

DC_CURRENT_CH2 Measured DC Current for Channel 2.

DC_CURRENT_CH3 Measured DC Current for Channel 3.

DC_CURRENT_CH4 Measured DC Current for Channel 4.

17.18.2.2 enum iMS::Diagnostics::TARGET [strong]

Sets which iMS device to request diagnostics data for.

Since

1.1

Enumerator

SYNTH Access the Synthesiser Diagnostics (Hours only)

AO_DEVICE Access the AO Device Diagnostics.

RF_AMPLIFIER Access the RF Amplifier Diagnostics.

17.18.3 Constructor & Destructor Documentation

17.18.3.1 iMS::Diagnostics::Diagnostics (const IMSSystem & *ims*)

Constructor for Diagnostics Object.

An IMSSystem object, representing the configuration of an iMS target must be passed by const reference to the Diagnostics constructor.

The IMSSystem object must exist before the Diagnostics object, and must remain valid (not destroyed) until the Diagnostics object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System

Since

1.1

17.18.4 Member Function Documentation

17.18.4.1 void iMS::Diagnostics::DiagnosticsEventSubscribe (const int *message,* IEventHandler ∗ *handler*)

Subscribe a callback function handler to a given DiagnosticsEvents event.

Diagnostics can callback user application code when an event occurs that affects the signal path. Supported events are listed under DiagnosticsEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to a DiagnosticsEvents event. For the period that a callback is subscribed, each time an event in Diagnostics occurs that would trigger the subscribed Diagnostics↩ Events event, the user function callback will be executed.

Parameters

in	*message*	Use the DiagnosticsEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.1

17.18.4.2 void iMS::Diagnostics::DiagnosticsEventUnsubscribe (const int *message*, const **IEventHandler** ∗ *handler*)

Unsubscribe a callback function handler from a given DiagnosticsEvents event.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the Diagnostics object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the DiagnosticsEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.1

17.18.4.3 const std::map<**MEASURE, Percent**>& iMS::Diagnostics::GetDiagnosticsData ( ) const

Returns a reference to the map of diagnostics data values currently stored by the Diagnostics class.

The map contains a set of key-value pairs representing the diagnostics data, one value per entry in the MEASURE enum. Each value is represented as a percentage where 100% represents the full scale analog measured value.

Call UpdateDiagnostics() first to retrieve the latest measurements from the system.

The map of values will be updated after the UpdateDiagnostics() function call and before the DIAGNOSTICS_UP↩ DATE_AVAILABLE event is fired so design the application to avoid accessing the map between these two timings to prevent a potential race condition.

Returns

a reference to the diagnostics measurement map

Since

1.1

17.18.4.4 bool iMS::Diagnostics::GetLoggedHours (const **TARGET** & *tgt*) const

Triggers a logged hours reading from the target device.

Calling this function will read back the current logged hours count from the timing circuit built into the Synthesiser, RF Power Amplifier or Acoust-Optic Device.

The function returns as soon as the request has been sent and a DiagnosticsEvents LoggedHours event will fire as soon as the result returns so ensure that the user code has subscribed to the appropriate event first.

Parameters

in	*tgt*	Which of the connected devices to read logged hours data from

Returns

true if the logged hours request was sent successfully.

Since

1.1

17.18.4.5 bool iMS::Diagnostics::GetTemperature (const TARGET & *tgt*) const

Triggers a temperature reading from the target device.

Calling this function will cause the Synthesiser to initiate a temperature conversion on either the RF Power Amplifier or the Acousto-Optic Device. There is no sensor built into the Synthesiser itself.

The function returns as soon as the conversion has been initiated and the result will become available in the background, causing a DiagnosticsEvents TempUpdate event to fire so ensure that the user code has subscribed to the appropriate event first.

Parameters

in	*tgt*	Which of the connected devices to read temperature data from

Returns

true if the temperature conversion was initiated successfully.

Since

1.1

17.18.4.6 bool iMS::Diagnostics::UpdateDiagnostics ( )

Triggers a Diagnostics Conversion to read measurement data from the RF Power Amplifier.

Calling this function will result in a new analog-to-digital conversion sequence being triggered in the diagnostics circuit built into the RF Power Amplifier. This will result in updated values being made available for Forward power, Reflected Power and DC Current across all 4 RF signal channels.

The function returns as soon as the request has been sent and a DiagnosticsEvents UpdateAvailable event will fire as soon as the result returns so ensure that the user code has subscribed to the appropriate event first. If for any reason the conversion was not able to be completed, a ReadFailed event will instead be returned

Returns

true if the update was initiated successfully.

Since

1.1

The documentation for this class was generated from the following file:

- Diagnostics.h

17.19 iMS::DiagnosticsEvents Class Reference

All the different types of events that can be triggered by the Diagnostics class.

```
#include <include\Diagnostics.h>
```

Public Types

- enum Events {
AOD_TEMP_UPDATE, RFA_TEMP_UPDATE, SYN_LOGGED_HOURS, AOD_LOGGED_HOURS, RFA_LOGGED_HOURS, DIAGNOSTICS_UPDATE_AVAILABLE, DIAG_READ_FAILED, **Count** }

  List of Events raised by the Diagnostics module.

17.19.1 Detailed Description

All the different types of events that can be triggered by the Diagnostics class.

Some events contain floating point parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2016-03-08

Since

1.1

17.19.2 Member Enumeration Documentation

17.19.2.1 enum iMS::DiagnosticsEvents::Events

List of Events raised by the Diagnostics module.

Enumerator

AOD_TEMP_UPDATE Received a temperature update from the Acousto-Optic device.

RFA_TEMP_UPDATE Received a temperature update from the RF Power Amplifier.

SYN_LOGGED_HOURS Returns the number of hours logged by the Synthesiser while powered up.

AOD_LOGGED_HOURS Returns the number of hours logged by the Acousto-Optic Device while powered up.

RFA_LOGGED_HOURS Returns the number of hours logged by the RF Power Amplifier while powered up.

DIAGNOSTICS_UPDATE_AVAILABLE Indicates to the application that an update of diagnostics data is available to be read.

DIAG_READ_FAILED Indicates that the update that was requested has failed to respond with updated results.

The documentation for this class was generated from the following file:

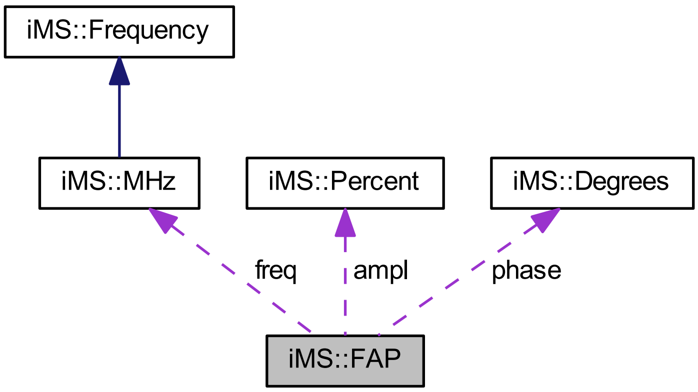
- Diagnostics.h

17.20 iMS::FAP Struct Reference

FAP (Frequency/Amplitude/Phase) triad stores the instantaneous definition of a single RF output.

`#include <include/IMSTypeDefs.h>`

Collaboration diagram for iMS::FAP:



Public Member Functions

- FAP ()

  Default construct a FAP object with zero data.
- FAP (double f, double a, double p)

  Construct a FAP object from raw double precision input data.
- FAP (MHz f, Percent a, Degrees p)

  Construct a FAP object from pre-existing MHz, Percent and Degrees objects.

Equality Operators

- bool operator== (const FAP &other) const

  Equality operators compare FAPs against each other.
- bool **operator!=** (const FAP &other) const

Public Attributes

- MHz freq

  The RF Channel Output Frequency.
- Percent ampl

  The RF Channel Output Amplitude.
- Degrees phase

  The RF Channel Output Phase.

17.20.1 Detailed Description

FAP (Frequency/Amplitude/Phase) triad stores the instantaneous definition of a single RF output.

The FAP struct, also known as a triad, stores one frequency (in MHz), one amplitude (Percent) and one phase (Degrees) value which uniquely specifies the instantaneous output of any one RF channel output.

4 FAP's make up a single ImagePoint, one per RF channel, and sequences of ImagePoints then make up an Image.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.20.2 Member Function Documentation

17.20.2.1 bool iMS::FAP::operator== (const FAP & *other*) const

Equality operators compare FAPs against each other.

Since

1.1.0

The documentation for this struct was generated from the following file:

- IMSTypeDefs.h

17.21 iMS::FileSystemManager Class Reference

Provides user management operations for working with Synthesiser FileSystems.

`#include <include\FileSystem.h>`

Public Member Functions

Constructor & Destructor

- FileSystemManager (IMSSystem &ims)
  Constructor for FileSystemManager Object.
- ∼FileSystemManager ()
  Destructor for FileSystemManager object.

File System Operations

- bool Delete (FileSystemIndex index)
  Removes the Entry indicated by the provided index from the FileSystemTable.
- bool Delete (const std::string &FileName)
- bool SetDefault (FileSystemIndex index)
  Tags a File for execution at Synthesiser startup.
- bool SetDefault (const std::string &FileName)
- bool ClearDefault (FileSystemIndex index)
  Removes the Default Flag assigned to a FileSystemTableEntry.
- bool ClearDefault (const std::string &FileName)
- bool Sanitize ()
  Reorganises the FileSystemTable and ensures it contains valid contents.

Miscellaneous Functions

- bool FindSpace (std::uint32_t &addr, const std::vector< std::uint8_t > &data) const
  Locates an area in the FileSystem memory large enough to store the provided contents.
- bool Execute (FileSystemIndex index)
  Causes the Synthesiser to access the FileSystem data represented by the index and execute it.
- bool Execute (const std::string &FileName)

17.21.1 Detailed Description

Provides user management operations for working with Synthesiser FileSystems.

Author

Dave Cowan

Date

2016-01-21

Since

1.1

17.21.2 Constructor & Destructor Documentation

17.21.2.1 iMS::FileSystemManager::FileSystemManager (IMSSystem & *ims*)

Constructor for FileSystemManager Object.

The FileSystemManager object requires an IMSSystem object, which will have had its FileSystemTable read back during initialisation. It must therefore exist before the FileSystemManager object, and must remain valid (not destroyed) until the FileSystemManager object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System whose FileSystemTable is to be operated upon.

Since

1.1

17.21.3 Member Function Documentation

17.21.3.1 bool iMS::FileSystemManager::ClearDefault (FileSystemIndex *index*)

Removes the Default Flag assigned to a FileSystemTableEntry.

Parameters

in	*index*	the Entry in the FST to unset as default (from 0 to MAX_FST_ENTRIES-1).

Returns

true if the default flag was unset successfully

Since

1.1

17.21.3.2 bool iMS::FileSystemManager::ClearDefault (const std::string & *FileName*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Removes the tag indicating a file should be executed at startup referencing it by its allocated filename

Parameters

in	*FileName*	a string representing the name of the file to unset as default

Returns

true if the filename was recognised and the default flag was unset successfully

Since

1.1

17.21.3.3 bool iMS::FileSystemManager::Delete (FileSystemIndex *index*)

Removes the Entry indicated by the provided index from the FileSystemTable.

The Entry is removed from the FST. The file data itself is not overwritten but once the entry has been deleted, it is impractival to recover the FileSystem data subsequently. The space 'freed up' by the deletion will become available for future file downloads and the release FST entry may be reused.

Bug Prior to v1.2.4 it was possible to attempt to delete an entry >= MAX_FST_ENTRIES. Doing so would have generated an exception. The condition is now checked for and the function will fail (return false) if attempted.

Parameters

in	*index*	the Entry in the FST to delete (from 0 to MAX_FST_ENTRIES-1).

Returns

true if the deletion process was carried out successfully

Since

1.1

17.21.3.4 bool iMS::FileSystemManager::Delete (const std::string & *FileName*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Deletes a file from the FileSystemTable referencing it by its allocated filename

Parameters

in	*FileName*	a string representing the name of the file to delete

Returns

true if the filename was recognised and the deletion process was carried out successfully

Since

1.1

17.21.3.5 bool iMS::FileSystemManager::Execute (FileSystemIndex *index*)

Causes the Synthesiser to access the FileSystem data represented by the index and execute it.

The execution of the FileSystem contents is defined in a FileSystemTypes specific way:

- COMPENSATION_TABLE data is loaded into the Compensation Look-Up Table
- TONE_BUFFER data is loaded into the Local Tone Buffer memory
- DDS_SCRIPT data is written register at a time to the DDS IC (the User must ensure that no Image Data is currently being played back to prevent unexpected behaviour)
- USER_DATA no action is performed

Parameters

in	*index*	the Entry in the FileSystemTable to operate on

Returns

if the Execution was started successfully.

Since

1.1

17.21.3.6 bool iMS::FileSystemManager::Execute (const std::string & *FileName*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

in	*FileName*	a string representing the name of the file to operate on

Since

1.1

17.21.3.7 bool iMS::FileSystemManager::FindSpace (std::uint32_t & *addr,* const std::vector< std::uint8_t > & *data*) const

Locates an area in the FileSystem memory large enough to store the provided contents.

Given a const reference to a byte array containing the data which the caller wants to place in FileSystem memory, this function operates an algorithm that will search through the FileSystemTable iteratively searching for the lowest possible address in memory that will fit the data in a contiguous block (since the FileSystem does not support distributed storage).

Parameters

out	*addr*	The location in memory where the data may be safely stored
in	*data*	a reference to a byte array representing the data which is to be stored

Returns

true if the algorithm was successful, false if no space could be found

Since

1.1

17.21.3.8 bool iMS::FileSystemManager::Sanitize ( )

Reorganises the FileSystemTable and ensures it contains valid contents.

The Sanitize process will do the following:

- ensure only one default flag is set per filetype, clearing the flag set on any subsequent entries
- check that valid filesystem contents is present for each entry
- look for any filesystem contents that may overlap, removing entries that are aliased
- Reorders the FST according to FileSystemTypes with any default marked entries placed at the front

  Returns

  true if the process completed successfully

  Since

  1.1

17.21.3.9 bool iMS::FileSystemManager::SetDefault (FileSystemIndex *index*)

Tags a File for execution at Synthesiser startup.

A single file of each file type may be marked as being the 'default' of its type. If tagged as such, the Synthesiser will attempt to execute the file during its initialisation process. All file types except USER_DATA may have a default entry.

If multiple files are marked as default, the entry with the lowest index number will take precedence. Any subsequent files marked as default will have their flags cleared during initialisation.

Parameters

in	*index*	the Entry in the FST to mark as default (from 0 to MAX_FST_ENTRIES-1).

Returns

true if the default flag was set successfully

Since

1.1

17.21.3.10 bool iMS::FileSystemManager::SetDefault (const std::string & *FileName*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. Tags a File for execution at Synthesiser startup referencing it by its allocated filename

Parameters

in	*FileName*	a string representing the name of the file to mark default

Returns

true if the filename was recognised and the default flag was set successfully

Since

1.1

The documentation for this class was generated from the following file:

- FileSystem.h

17.22 iMS::FileSystemTableEntry Struct Reference

Contains all the parameters that uniquely locate a File within the Synthesiser FileSystem.

`#include <include\FileSystem.h>`

Public Member Functions

Constructor & Destructor

- FileSystemTableEntry ()
  Empty Constructor for FileSystemTableEntry Object.
- FileSystemTableEntry (FileSystemTypes type, std::uint32_t addr, std::uint32_t length, FileDefault def)
  Constructor for FileSystemTableEntry Object with no FileName specified.
- FileSystemTableEntry (FileSystemTypes type, std::uint32_t addr, std::uint32_t length, FileDefault def, std::string name)
  Full Constructor for FileSystemTableEntry Object with FileName.
- ∼FileSystemTableEntry ()
  Destructor for FileSystemTableEntry.
- FileSystemTableEntry (const FileSystemTableEntry &)
  Copy Constructor.
- FileSystemTableEntry & operator= (const FileSystemTableEntry &)
  Assignment Constructor.

FileSystemTable entry parameter readback

- const FileSystemTypes Type () const
- const std::uint32_t Address () const
- const std::uint32_t Length () const
- const bool IsDefault () const
- const std::string Name () const

17.22.1 Detailed Description

Contains all the parameters that uniquely locate a File within the Synthesiser FileSystem.

A FileSystemTableEntry object stores the length, address, file type, file name and default flag status of any file stored within the Synthesiser FileSystem.

It is not normally necessary for the user application to create a FileSystemTableEntry object since this will be handled by the individual File Writing method (e.g. CompensationTableDownload::Store()), by the FileSystemManager or during IMSSytem initialisation. However the struct is useful for reading parameter data about a file entry in the table using the various const methods.

Author

Dave Cowan

Date

2016-01-20

Since

1.1

17.22.2 Constructor & Destructor Documentation

17.22.2.1 iMS::FileSystemTableEntry::FileSystemTableEntry ( )

Empty Constructor for FileSystemTableEntry Object.

Since

1.1

17.22.2.2 iMS::FileSystemTableEntry::FileSystemTableEntry (FileSystemTypes *type*, std::uint32_t *addr*, std::uint32_t *length*, FileDefault *def*)

Constructor for FileSystemTableEntry Object with no FileName specified.

Parameters

in	*type*	File Type of table entry
in	*addr*	Address in FileSystem where table entry is stored
in	*length*	number of bytes occupied by file in FileSystem
in	*def*	Flag indicating whether File should be executed at startup

Since

1.1

17.22.2.3 iMS::FileSystemTableEntry::FileSystemTableEntry (FileSystemTypes *type*, std::uint32_t *addr*, std::uint32_t *length*, FileDefault *def*, std::string *name*)

Full Constructor for FileSystemTableEntry Object with FileName.

Parameters

in	*type*	File Type of table entry
in	*addr*	Address in FileSystem where table entry is stored
in	*length*	number of bytes occupied by file in FileSystem
in	*def*	Flag indicating whether File should be executed at startup
in	*name*	8-character string given to table entry describing the contents of the file

Since

1.1

17.22.3 Member Function Documentation

17.22.3.1 const std::uint32_t iMS::FileSystemTableEntry::Address ( ) const

Returns

Address in FileSystem memory of table entry

17.22.3.2 const bool iMS::FileSystemTableEntry::IsDefault ( ) const

Returns

true if entry is marked for execution at startup

17.22.3.3 const std::uint32_t iMS::FileSystemTableEntry::Length ( ) const

Returns

Length in bytes occupied in memory of table entry

17.22.3.4 const std::string iMS::FileSystemTableEntry::Name ( ) const

Returns

string representing descriptive file name given to table entry

17.22.3.5 const FileSystemTypes iMS::FileSystemTableEntry::Type ( ) const

Returns

File Type of table entry

The documentation for this struct was generated from the following file:

- FileSystem.h

17.23 iMS::FileSystemTableViewer Class Reference

Provides a mechanism for viewing the FileSystemTable associated with an iMS System.

`#include <include\FileSystem.h>`

Public Member Functions

Constructor

- FileSystemTableViewer (const IMSSystem &ims)
  Constructor for FileSystemTableViewer Object.

FileSystem Table Information

- const bool IsValid () const
  Indicates whether FileSystemTable object is valid.
- const int Entries () const

Array operator for random access to FileSystemTableEntry s

- const FileSystemTableEntry operator[ ] (const std::size_t idx) const
  The FileSystemTable consists of a container of FileSysteTableEntry objects. Each object may be accessed by calling the viewer object through an array subscript.

Friends

- LIBSPEC std::ostream & operator<< (std::ostream &stream, const FileSystemTableViewer &)
  Stream operator overload to simplify debugging.

17.23.1 Detailed Description

Provides a mechanism for viewing the FileSystemTable associated with an iMS System.

Author

Dave Cowan

Date

2016-01-21

Since

1.1

17.23.2 Constructor & Destructor Documentation

17.23.2.1 iMS::FileSystemTableViewer::FileSystemTableViewer (const IMSSystem & *ims*) [inline]

Constructor for FileSystemTableViewer Object.

The FileSystemTableViewer object requires an IMSSystem object, which will have had its FileSystemTable read back during initialisation. It must therefore exist before the FileSystemTableViewer object, and must remain valid (not destroyed) until the FileSystemTableViewer object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System whose FileSystemTable is to be viewed.

Since

1.1

17.23.3 Member Function Documentation

17.23.3.1 const int iMS::FileSystemTableViewer::Entries ( ) const

Returns

The current number of file entries stored in the FileSystemTable

Since

1.1

17.23.3.2 const bool iMS::FileSystemTableViewer::IsValid ( ) const

Indicates whether FileSystemTable object is valid.

For a FileSystemTable stored on the Synthesiser to be considered valid, certain parameters need to be met. If the initialisation process is unable to establish validity of a FileSystemTable it will mark it as void and the user will not be able to work with it until a new FileSystem has been created and downloaded.

User code should therefore check that the FileSystemTable is valid before working with it.

Returns

true if the FileSystemTable is considered valid.

Since

1.1

17.23.3.3 const FileSystemTableEntry iMS::FileSystemTableViewer::operator[] (const std::size_t *idx*) const

The FileSystemTable consists of a container of FileSysteTableEntry objects. Each object may be accessed by calling the viewer object through an array subscript.

For example:

```
FileSystemTableViewer fstv(myiMS);
if (fstv.IsValid()) {
    int length = 0;
    for (int i=0; i<fstv.Entries(); i++) {
        length += fstv[i].Length();
    }
    std::cout << "Used space in filesystem: " << length << " bytes" << std::endl;
}
```

Since

1.1

17.23.4 Friends And Related Function Documentation

17.23.4.1 LIBSPEC std::ostream& operator<< (std::ostream & *stream*, const FileSystemTableViewer &) [friend]

Stream operator overload to simplify debugging.

Example usage:

```
FileSystemTableViewer fstv(myiMS);
if (!fstv.IsValid()) {
    std::cout << "No Filesystem found" << std::endl;
}
else {
    std::cout << fstv;
}
```

might produce the result:

```
FST[00]* : Type  1 Addr :  8708 Len : 16386 Name : CompTbl1
FST[01]  : Type  1 Addr : 38924 Len : 16386 Name : CompTbl2
FST[02]* : Type  2 Addr : 1024  Len : 6146 Name : ToneUp
FST[03]  : Type  2 Addr : 25094 Len : 6146 Name : ToneDown
FST[04]  : Type 15 Addr : 55310 Len : 1538 Name : User5
FST[05]  : Type 15 Addr : 56848 Len : 1538 Name : User5
FST[06]  : Type  3 Addr : 7170  Len : 17 Name : DDS100M
```

where The index into the FileSystemTable (FST) is given followed by an asterisk if the entry is marked as Default (Execute on startup). Then the File Type is given (refer to FileSystemTypes), followed by the starting address in memory then the number of bytes occupied and finally the allocated filename.

The documentation for this class was generated from the following file:

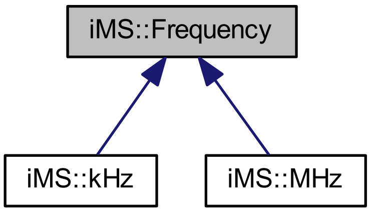
- FileSystem.h

17.24 iMS::Frequency Class Reference

Type Definition for all operations that require a frequency specification.

`#include <include/IMSTypeDefs.h>`

Inheritance diagram for iMS::Frequency:



Public Member Functions

- Frequency (double arg=0.0)

    Construct a Frequency object from a double argument representing Hertz.
- Frequency & operator= (double arg)

    Assignment of a double argument in Hertz to an existing Frequency object.
- operator double () const

    Return a double representing the Frequency value in Hertz.

Static Public Member Functions

- static unsigned int RenderAsPointRate (const IMSSystem &, const Frequency, const bool Prescaler↩ Disable=false)

    Used internally by the library to convert a Frequency object into an hardware-dependent integer representation used by the Image for Internal Oscillator frequency.

17.24.1 Detailed Description

Type Definition for all operations that require a frequency specification.

Internally, the Frequency value is stored as a double precision variable specified in Hertz

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.24.2 Constructor & Destructor Documentation

17.24.2.1 iMS::Frequency::Frequency (double *arg* = 0.0) [inline]

Construct a Frequency object from a double argument representing Hertz.

Parameters

in	*arg*	Frequency in Hertz

Since

1.0

17.24.3 Member Function Documentation

17.24.3.1 iMS::Frequency::operator double ( ) const [inline]

Return a double representing the Frequency value in Hertz.

```
kHz f1(1.2);
Frequency f2 = f1();
std::cout << "f2's Frequency is: " << f2() << "Hz" << std::endl;
```

prints:

```
f2's Frequency is 1200.0Hz
```

Since

1.0

17.24.3.2 Frequency& iMS::Frequency::operator= (double *arg*) [inline]

Assignment of a double argument in Hertz to an existing Frequency object.

```
Frequency f;
f = 1000.0;
// f contains 1000Hz
```

Since

1.0

17.24.3.3 static unsigned int iMS::Frequency::RenderAsPointRate (const IMSSystem & , const Frequency , const bool *PrescalerDisable* = false) [static]

Used internally by the library to convert a Frequency object into an hardware-dependent integer representation used by the Image for Internal Oscillator frequency.

Not intended for use in application code

The documentation for this class was generated from the following file:

- IMSTypeDefs.h

17.25 iMS::FWVersion Struct Reference

Stores the version number of firmware running on iMS hardware.

`#include <include/IMSSystem.h>`

Public Attributes

- int major { -1 }

  returns the Major firmware version number (or -1 if uninitialised)
- int minor { 0 }

  returns the Minor firmware version number
- int revision { 0 }

  returns the firmware revision number
- struct std::tm build_date

  returns a struct indicating the date on which the firmware was created

Friends

- LIBSPEC std::ostream & operator<< (std::ostream &stream, const FWVersion &)

  Use this operator overload to output to a console the firmware version in human-readable format.

17.25.1 Detailed Description

Stores the version number of firmware running on iMS hardware.

Firmware version is always defined as 'M.m.r' where: M = Major Version m = Minor Version r = Revision

Revision increments continuously for each build of firmware that is created. Major and Minor tags are only updated to mark an important release.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.25.2 Friends And Related Function Documentation

17.25.2.1 LIBSPEC std::ostream& operator<< (std::ostream & *stream*, const FWVersion &) [friend]

Use this operator overload to output to a console the firmware version in human-readable format.

For example:

```
std::cout << " FW Version: " << myiMS.Ctlr().GetVersion() << std::endl;
```

might print:

```
FW Version: 1.0.23 Wed 23 September 2015 11:08 GMT
```

The documentation for this struct was generated from the following file:

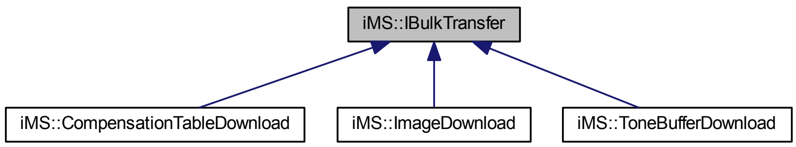
- IMSSystem.h

17.26 iMS::IBulkTransfer Class Reference

Interface Specification class for sending large binary data objects to the iMS.

`#include <include/IBulkTransfer.h>`

Inheritance diagram for iMS::IBulkTransfer:



Public Member Functions

- virtual bool StartDownload ()=0

  Initiates a Bulk Transfer download.
- virtual bool StartVerify ()=0

  Initiates a Bulk Transfer verify.
- virtual int GetVerifyError ()=0

  Returns the address of the next verify error or -1 if none.

17.26.1 Detailed Description

Interface Specification class for sending large binary data objects to the iMS.

There are several instances in which large binary data must be transferred either from the host to the iMS or in the other direction, e.g. download of image data, compensation tables etc. This is known as Bulk Transfer and it implements a background process that supervises the splitting up of large data objects into individual messages compatible with the communications module, queuing them for transfer, verifying the success or failure of the transfer and reporting to the application software when the transfer is complete.

This interface class defines the methods which application software may use to control the Bulk Transfer process. It is inherited by API classes that require the use of a Bulk Transfer, and which implement the Bulk Transfer mechanism.

Completion and success or failure of a Bulk Transfer are indicated by the IEventHandler mechanism, which must be implemented by the derivative class

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.26.2 Member Function Documentation

17.26.2.1 virtual int iMS::IBulkTransfer::GetVerifyError ( ) [pure virtual]

Returns the address of the next verify error or -1 if none.

After the application has been notified of a failed verify, it can probe the BulkTransfer derived object to obtain the approximate address at which the BulkTransfer failed. The address is provided as a byte offset from the start of the BulkTransfer binary object.

Due to the way in which the BulkTransfer mechanism splits the transfer into individual messages, there will be one error recorded for each message that results in a verify fail. Therefore, the address will only be approximate, to the nearest message size boundary and if there are multiple byte fails within the scope of a single message, only one error will be recorded.

Calling this function repeatedly will result in returning the next recorded verify error. If there are no errors left, or the transfer was successful (i.e. there were no verify failures recorded) the function will return -1.

Returns

byte address of transfer failure or -1 if none.

Since

1.0

Implemented in iMS::CompensationTableDownload, iMS::ToneBufferDownload, and iMS::ImageDownload.

17.26.2.2 virtual bool iMS::IBulkTransfer::StartDownload ( ) [pure virtual]

Initiates a Bulk Transfer download.

If the user has subscribed to the relevant event notifications, the BulkTransfer derived object will issue a completion event at the end of the download process and will also warn the user anytime a download messaging error occurs.

Returns

Boolean indicating whether Download has started successfully

Since

1.0

Implemented in iMS::CompensationTableDownload, iMS::ToneBufferDownload, and iMS::ImageDownload.

17.26.2.3 virtual bool iMS::IBulkTransfer::StartVerify ( ) [pure virtual]

Initiates a Bulk Transfer verify.

If the user has subscribed to the relevant event notifications, the BulkTransfer derived object will raise an event to the application at the end of the verify process to indicate whether the verification was successful or not.

Returns

Boolean indicating whether Verify has started successfully

Since

1.0

Implemented in iMS::CompensationTableDownload, iMS::ToneBufferDownload, and iMS::ImageDownload.

The documentation for this class was generated from the following file:

- IBulkTransfer.h

17.27 iMS::IEventHandler Class Reference

Interface Class for an Event Handler to be defined in User Code and subscribed to library events.

`#include <include/IEventHandler.h>`

Public Member Functions

- IEventHandler ()
  Default Constructor.
- virtual ∼IEventHandler ()
  Virtual Destructor.
- bool operator== (const IEventHandler e)
  Used internally to identify Functions subscribed to Events. Not intended for Application usage.

Overrideable User Action on Event

- virtual void EventAction (void ∗sender, const int message, const int param=0)
  This Method must be overriden by a User derived callback class.
- virtual void EventAction (void ∗sender, const int message, const int param, const int param2)
  This Method must be overriden by a User derived callback class.
- virtual void EventAction (void ∗sender, const int message, const double param)
  This Method must be overriden by a User derived callback class.
- virtual void EventAction (void ∗sender, const int message, const int param, const std::vector< std::uint8_t > data)
  This Method must be overriden by a User derived callback class.

17.27.1 Detailed Description

Interface Class for an Event Handler to be defined in User Code and subscribed to library events.

Note that it is not possible to subscribe a single derived class to multiple events from different source objects in the library (as in a system-wide message handler) because the message enum integer values overlap each other. This is a conscious design choice to encourage encapsulation. A class may still be subscribed to multiple events as long as they are triggered from the same source object.

Example:

```
class ImageVerifySupervisor : public IEventHandler
{
private:
    bool m_verifying{ true };
public:
    void EventAction(void* sender, const int message, const int param)
```

```
    {
        switch (message)
        {
        case (ImageDownloadEvents::VERIFY_SUCCESS) : std::cout << "Image
       Verify Successful!" << std::endl; m_verifying = false; break;
        case (ImageDownloadEvents::VERIFY_FAIL) : std::cout << "Image
       Verify FAILED!" << std::endl; m_verifying = false; break;
        }
    }
    bool Busy() const { return m_verifying; };
};
```

The above code snippet defines a class "ImageVerifySupervisor" that inherits from the IEventHandler base class. This is used during the download of an Image to the Controller to determine whether the verification of the download was successful or not.

The class contains a private boolean variable which is initialised to true. It overrides the EventAction interface class method to do something when the VERIFY_SUCCESS and VERIFY_FAIL events are raised. User code can read the Busy() function to determine whether the downloader is still in the process of verifying the download or whether it has finished (which is assumed to be the case once either of the 2 events are received).

To use the class, the application code creates an ImageVerifySupervisor object at the same time as starting a verify on an ImageDownload. It then links the object to the ImageDownload by calling the Subscribe() method for both ImageDownloadEvents::VERIFY_SUCCESS and ImageDownloadEvents::VERIFY_FAIL, passing to the method the address of the ImageVerifySupervisor object as a function pointer.

```
  ImageDownload * dl = new ImageDownload(ims, img);
  ImageVerifySupervisor vs;
  dl->ImageDownloadEventSubscribe(ImageDownloadEvents::VERIFY_SUCCESS, &
      vs);
  dl->ImageDownloadEventSubscribe(ImageDownloadEvents::VERIFY_FAIL, &vs);

  dl->StartVerify();

  while (vs.Busy()) {
      std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }

  dl->ImageDownloadEventUnsubscribe(ImageDownloadEvents::VERIFY_SUCCESS,
       &vs);
  dl->ImageDownloadEventUnsubscribe(ImageDownloadEvents::VERIFY_FAIL, &vs);
delete dl;
```

When the verify completes, it will trigger either the VERIFY_SUCCESS or VERIFY_FAIL events which, through the library's event handling mechanism, will identify the subscribed function and call the EventAction method.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.27.2 Member Function Documentation

17.27.2.1 virtual void iMS::IEventHandler::EventAction (void * *sender*, const int *message*, const int *param* = 0) [virtual]

This Method must be overriden by a User derived callback class.

When a user class derived from IEventHandler is subscribed to receive event notifications from the iMS Library, it is this function that is always called when the event is raised. Therefore it is essential to override this method to process the event and to do something with it.

This overloaded callback function provides integer parameter data to user code.

Parameters

in	*sender*	A pointer to the class that triggers the event callback. Can be used to obtain additional information.
in	*message*	an integer that maps to an enum in the Events class associated with the callback subscription
in	*param*	an optional integer parameter that provides additional information on the callback event.

Since

1.0

17.27.2.2 virtual void iMS::IEventHandler::EventAction (void ∗ *sender,* const int *message,* const int *param,* const int *param2*) [virtual]

This Method must be overriden by a User derived callback class.

When a user class derived from IEventHandler is subscribed to receive event notifications from the iMS Library, it is this function that is always called when the event is raised. Therefore it is essential to override this method to process the event and to do something with it.This overloaded callback function provides integer parameter data to user code.

Parameters

in	*sender*	A pointer to the class that triggers the event callback. Can be used to obtain additional information.
in	*message*	an integer that maps to an enum in the Events class associated with the callback subscription
in	*param*	an integer parameter that provides additional information on the callback event.
in	*param2*	an optional integer parameter that provides further additional information on the callback event.

Since

1.2

17.27.2.3 virtual void iMS::IEventHandler::EventAction (void ∗ *sender,* const int *message,* const double *param*) [virtual]

This Method must be overriden by a User derived callback class.

When a user class derived from IEventHandler is subscribed to receive event notifications from the iMS Library, it is this function that is always called when the event is raised. Therefore it is essential to override this method to process the event and to do something with it.

This overloaded callback function provides floating point parameter data to user code.

Parameters

in	*sender*	A pointer to the class that triggers the event callback. Can be used to obtain additional information.
in	*message*	an integer that maps to an enum in the Events class associated with the callback subscription

in	*param*	an floating point parameter that provides additional information on the callback event.

Since

1.1

17.27.2.4 virtual void iMS::IEventHandler::EventAction (void * *sender,* const int *message,* const int *param,* const std::vector< std::uint8_t > *data*) [virtual]

This Method must be overriden by a User derived callback class.

When a user class derived from IEventHandler is subscribed to receive event notifications from the iMS Library, it is this function that is always called when the event is raised. Therefore it is essential to override this method to process the event and to do something with it.

This overloaded callback function provides a vector of byte data to user code.

Parameters

in	*sender*	A pointer to the class that triggers the event callback. Can be used to obtain additional information.
in	*message*	an integer that maps to an enum in the Events class associated with the call-back subscription
in	*param*	an integer parameter that provides information on the callback event.
in	*data*	a byte vector parameter that provides additional information on the callback event.

Since

1.2

17.27.2.5 bool iMS::IEventHandler::operator== (const IEventHandler *e*)

Used internally to identify Functions subscribed to Events. Not intended for Application usage.

Since

1.0

The documentation for this class was generated from the following file:

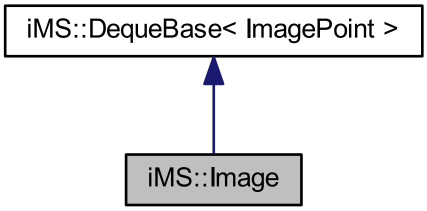
- IEventHandler.h

17.28 iMS::Image Class Reference

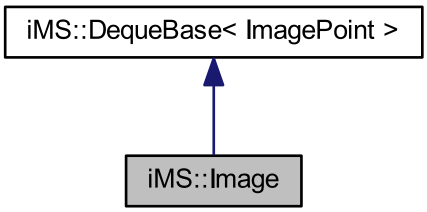
A sequence of ImagePoints played out sequentially by the Controller and driven by the Synthesiser.

```
#include <include/Image.h>
```

Inheritance diagram for iMS::Image:



Collaboration diagram for iMS::Image:



Public Member Functions

Constructors & Destructors

- Image (const std::string &name="")
  Empty Constructor.
- Image (size_t nPts, const ImagePoint &pt, const std::string &name="")
  Fill Constructor.
- Image (size_t nPts, const ImagePoint &pt, const Frequency &f, const std::string &name="")
  Fill Constructor with Internal Clock Initialisation.
- Image (size_t nPts, const ImagePoint &pt, const int div, const std::string &name="")
  Fill Constructor with External Clock Divider Initialisation.
- Image (const_iterator first, const_iterator last, const std::string &name="")
  Range Constructor.
- Image (const_iterator first, const_iterator last, const Frequency &f, const std::string &name="")
  Range Constructor with Internal Clock Initialisation.
- Image (const_iterator first, const_iterator last, const int div, const std::string &name="")
  Range Constructor with External Clock Initialisation.
- Image (const Image &)
  Copy Constructor.
- Image & operator= (const Image &)
  Assignment Constructor.
- ∼Image ()

Destructor.

Insert/Add ImagePoints

- void AddPoint (const ImagePoint &pt)

  Add a single new ImagePoint at the end of the Image.
- iterator InsertPoint (iterator it, const ImagePoint &pt)

  Inserts a single new element into the PointList.
- void InsertPoint (iterator it, size_t nPts, const ImagePoint &pt)

  Inserts multiple copies of an element into the PointList.
- void InsertPoint (iterator it, const_iterator first, const_iterator last)

  Inserts a range of ImagePoints into the PointList.

Remove/Clear ImagePoints

- iterator RemovePoint (iterator it)

  Removes a single ImagePoint from the PointList.
- iterator RemovePoint (iterator first, iterator last)

  Removes a range of ImagePoints from the PointList.
- void Clear ()

  Remove all ImagePoints from the Image.

Image Size

- int Size () const

  Returns the number of ImagePoints in the PointList.

Default Internal Clock Rate

- void ClockRate (const Frequency &f)

  Sets the Internal Clock Rate that shall be the default playback frequency for the Image.
- const Frequency & ClockRate () const

  Returns the default Internal Clock Rate associated with the Image.

Default External Clock Divider

- void ExtClockDivide (const int div)

  Sets the External Clock Divider ratio.
- const int ExtClockDivide () const

  Returns the default External Clock Divider Ratio associated with the Image.

Image Description

- std::string & Description ()

  A string stored with the Image to aid human users in identifying the purpose of an image.
- const std::string & **Description** () const

Additional Inherited Members

17.28.1 Detailed Description

A sequence of ImagePoints played out sequentially by the Controller and driven by the Synthesiser.

An Image contains a list of ImagePoints and a default Clock Rate. Its length is limited only by the available memory of the Controller onto which it is downloaded.

It can be created, copied, modified, merged and more in software on the host running the SDK, stored to disk inside an ImageProject (from SDK rev1.3) and transferred to/from the iMS Controller using the ImageDownload mechanism.

Once in memory on a Controller, the Image can be played back. This can be triggered by software, or by an external trigger input signal applied to the Controller. At the start of playback, the first ImagePoint is programmed by the Controller into the Synthesiser which updates the RF output of all 4 channels.

The Controller then progresses through the Image sequence ImagePoint by ImagePoint, updating the Synthesiser's RF output as it goes. The Image progression can either propagate using an internal clock or under the control of an external signal applied to the Controller. If using the internal clock, the clock is programmed at the point of downloading the Image with the default value for Clock Rate which is stored alongside the PointList data in the Image object.

If using an Image alongside other Images in an ImageGroup, for Controllers that support it, the internal ClockRate may be overriden by the value programmed into the SequenceTable that is a part of the ImageGroup object.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.28.2 Constructor & Destructor Documentation

17.28.2.1 iMS::Image::Image (const std::string & *name* = "")

Empty Constructor.

Parameters

in	*name*	The optional descriptive name to apply to the image

17.28.2.2 iMS::Image::Image (size_t *nPts*, const **ImagePoint** & *pt*, const std::string & *name* = "")

Fill Constructor.

Use this constructor to generate an Image with `nPts` number of points, each one initialised to the value of `pt`

Parameters

in	*nPts*	The size of the Image PointList after construction
in	*pt*	The ImagePoint that will fill each of the new elements of the PointList
in	*name*	The optional descriptive name to apply to the image

Since

1.0

17.28.2.3 iMS::Image::Image (size_t *nPts*, const **ImagePoint** & *pt*, const **Frequency** & *f*, const std::string & *name* = "")

Fill Constructor with Internal Clock Initialisation.

Use this constructor to generate an Image with `nPts` number of points, each one initialised to the value of `pt` and with the Internal default Clock Rate initialised to `f`

Parameters

in	*nPts*	The size of the Image PointList after construction
in	*pt*	The ImagePoint that will fill each of the new elements of the PointList
in	*f*	The default Clock Rate that the Image will playback when using the Internal Clock mode
in	*name*	The optional descriptive name to apply to the image

Since

1.0

17.28.2.4 iMS::Image::Image (size_t *nPts*, const ImagePoint & *pt*, const int *div*, const std::string & *name* = "")

Fill Constructor with External Clock Divider Initialisation.

Use this constructor to generate an Image with `nPts` number of points, each one initialised to the value of `pt` and with the External default Clock Divider Ratio initialised to `div`

Parameters

in	*nPts*	The size of the Image PointList after construction
in	*pt*	The ImagePoint that will fill each of the new elements of the PointList
in	*div*	The default Clock Divider Ratio that the Image will apply to the External Clock when using the External Clock mode
in	*name*	The optional descriptive name to apply to the image

Since

1.0

17.28.2.5 iMS::Image::Image (const_iterator *first*, const_iterator *last*, const std::string & *name* = "")

Range Constructor.

Use this constructor to copy a range of ImagePoints from another Image For example,

```
// Create an image with 1,024 points initialized to 70MHz, 100%
Image img1 (1024, ImagePoint(FAP(70.0,100.0,0.0)));
// Copy the first 500 points into a second image
Image img2 (img1.begin(), img1.begin()+500);
```

Parameters

in	*first*	An iterator that points to the first ImagePoint of a range to construct the new Image from
in	*last*	An iterator that points to the element after the last ImagePoint of a range to construct the new Image from
in	*name*	The optional descriptive name to apply to the image

Since

1.0

17.28.2.6 iMS::Image::Image (const_iterator *first*, const_iterator *last*, const Frequency & *f*, const std::string & *name* = "")

Range Constructor with Internal Clock Initialisation.

Use this constructor to copy a range of ImagePoints from another Image and set the internal default clock frequency

Parameters

in	*first*	An iterator that points to the first ImagePoint of a range to construct the new Image from
in	*last*	An iterator that points to the element after the last ImagePoint of a range to construct the new Image from
in	*f*	The default Clock Rate that the Image will playback when using the Internal Clock mode
in	*name*	The optional descriptive name to apply to the image

Since

1.0

17.28.2.7 iMS::Image::Image (const_iterator *first*, const_iterator *last*, const int *div*, const std::string & *name* = "")

Range Constructor with External Clock Initialisation.

Use this constructor to copy a range of ImagePoints from another Image and set the external default clock divider ratio

Parameters

in	*first*	An iterator that points to the first ImagePoint of a range to construct the new Image from
in	*last*	An iterator that points to the element after the last ImagePoint of a range to construct the new Image from
in	*div*	The default Clock Divider Ratio that the Image will apply to the external clock signal when using the External Clock mode
in	*name*	The optional descriptive name to apply to the image

Since

1.0

17.28.3 Member Function Documentation

17.28.3.1 void iMS::Image::AddPoint (const ImagePoint & *pt*)

Add a single new ImagePoint at the end of the Image.

Extends the length of the Image by one ImagePoint and copies to it the data supplied in the const reference `pt`

Equivalent to

```
img.InsertPoint(img.end(), 1, pt);
```

Parameters

in	*pt*	The ImagePoint to append to the end of the Image

Since

1.0

17.28.3.2 void iMS::Image::Clear ( )

Remove all ImagePoints from the Image.

The PointList is cleared, all ImagePoints are removed from it and destroyed. The new size of the Image will be zero and `Image::begin() == Image::end()`

Since

1.0

17.28.3.3 void iMS::Image::ClockRate (const Frequency & *f*)

Sets the Internal Clock Rate that shall be the default playback frequency for the Image.

An Image shall have associated with it a default Clock Rate. This is the frequency at which the iMS Controller playback will propagate from one ImagePoint to the next when it is operated in Internal Clock Mode (see Image↩Player::PointClock).

If the Controller supports multiple images and playback from ImageGroup's, the ImageGroup will contain a sequence table and in that case, the Clock Rate for playing back the Image as part of a sequence may be overriden by the Clock Rate field specified in the Sequence Table.

Parameters

in	*f*	A Frequency variable to set the Image default internal Clock Rate from.

Since

1.0

17.28.3.4 const Frequency& iMS::Image::ClockRate ( ) const

Returns the default Internal Clock Rate associated with the Image.

The Image contains a default Clock Rate which shall be used as the frequency for playing out an Image when the Controller is configured for Internal Clock mode and is not overriden by the Clock Rate specified in a sequence table.

Returns

A Frequency value that is the default Internal Clock Rate associated with an Image

Since

1.0

17.28.3.5 std::string& iMS::Image::Description ( )

A string stored with the Image to aid human users in identifying the purpose of an image.

A descriptive string can be set alongside the Image to allow users to identify and differentiate between images without having to browse through the point data. The description is optional, and if, not used, the description will simply default to "image".

Updating the Image Description does not cause the Image UUID to change.

17.28.3.6 void iMS::Image::ExtClockDivide (const int *div*)

Sets the External Clock Divider ratio.

An Image shall have associated with it a default external Clock Divider Ratio. This is the ratio of the externally supplied clock signal to the Image playback rate. For example, set this to 100 and a 1MHz external clock signal will result in a 10kHz playback rate.

If the Controller supports multiple images and playback from ImageGroup's, the ImageGroup will contain a sequence table and in that case, the Clock Divider Ratio for playing back the Image as part of a sequence may be overriden by the Clock Divider Ratio field specified in the Sequence Table.

Parameters

in	*div*	An integer variable to set the Image default external Clock Divider ratio from.

Since

1.0.1

17.28.3.7 const int iMS::Image::ExtClockDivide ( ) const

Returns the default External Clock Divider Ratio associated with the Image.

The Image contains a default External Clock Divider Ratio which shall be used as the frequency ratio between the external clock signal and the Image playback frequency when the Controller is configured for External Clock mode and is not overriden by the Clock Divider Ratio specified in a sequence table.

Returns

An integer value representing the default External Clock Divier Ratio associated with an Image

Since

1.0.1

17.28.3.8 iterator iMS::Image::InsertPoint (iterator *it*, const ImagePoint & *pt*)

Inserts a single new element into the PointList.

The ImagePoint pt is inserted before the element pointed to by the iterator it.

Parameters

in	*it*	An ImagePoint will be inserted before the element pointed to by this iterator.
in	*pt*	The ImagePoint to insert into the Image

Since

1.0

17.28.3.9 void iMS::Image::InsertPoint (iterator *it*, size_t *nPts*, const ImagePoint & *pt*)

Inserts multiple copies of an element into the PointList.

`nPts` copies of the ImagePoint `pt` are inserted into the PointList before the element pointed to be iterator `it`

Parameters

in	*it*	Multiple ImagePoints will be inserted before the element pointed to by this iterator.
in	*nPts*	The number of copies of `pt` to insert
in	*pt*	The ImagePoint to insert multiple copies of into the Image

Since

1.0

17.28.3.10 void iMS::Image::InsertPoint (iterator *it*, const_iterator *first*, const_iterator *last*)

Inserts a range of ImagePoints into the PointList.

All of the ImagePoints located between first and last are copied in order into the PointList starting before the element pointed to by iterator it.

For example,

```
// Create an image with 4 points initialised to 70MHz
Image img1( 4, ImagePoint(FAP(70.0,100.0,0.0)));
// Create an image with 3 points initialised to 100MHz
Image img2( 3, ImagePoint(FAP(100.0,100.0,0.0)));
// Insert all of img2 in the middle of img1
img.InsertPoint(img1.begin()+2, img2.begin(), img2.end());
```

img2 contains [70, 70, 100, 100, 100, 70, 70]

Parameters

in	*it*	A range of ImagePoints will be inserted before the element pointed to by this iterator.
in	*first*	An iterator pointing to the first in a range of ImagePoints to be inserted
in	*last*	An iterator pointing to the ImagePoint after the last ImagePoint to be inserted

Since

1.0

17.28.3.11 iterator iMS::Image::RemovePoint (iterator *it*)

Removes a single ImagePoint from the PointList.

Erases a single ImagePoint, reducing the size of the Image by one. The removed ImagePoint is destroyed.

Parameters

in	*it*	Iterator pointing to a single ImagePoint to be removed from the PointList

Returns

An iterator pointing to the new location of the ImagePoint that followed the element erased by the function call. If the operation erased the last ImagePoint in the PointList, this will be equal to `Image::end()`.

Since

1.0

17.28.3.12 iterator iMS::Image::RemovePoint (iterator *first*, iterator *last*)

Removes a range of ImagePoints from the PointList.

Erases a range of ImagePoints from the Image, reducing the size of the Image by the number of ImagePoints removed, which are destroyed.

Parameters

in	*first*	An iterator pointing to the first in a range of ImagePoints to be removed
in	*last*	An iterator pointing to the ImagePoint after the last ImagePoint to be removed

Returns

An iterator pointing to the new location of the ImagePoint that followed the last element erased by the function call. If the operation erased the last ImagePoint in the PointList, this will be equal to `Image::end()`.

Since

1.0

17.28.3.13 int iMS::Image::Size ( ) const

Returns the number of ImagePoints in the PointList.

Returns

The number of ImagePoints in the PointList

Since

1.0

The documentation for this class was generated from the following file:

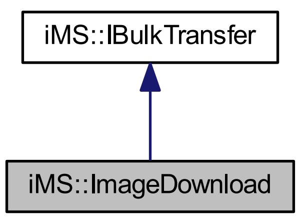
- Image.h

17.29 iMS::ImageDownload Class Reference

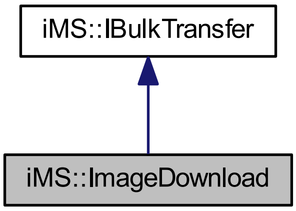
Provides a mechanism for downloading and verifying Images to a Controller's memory.

`#include <include\ImageOps.h>`

Inheritance diagram for iMS::ImageDownload:



Collaboration diagram for iMS::ImageDownload:



Public Member Functions

Constructor & Destructor

- ImageDownload (IMSSystem &ims, const Image &img)
  Constructor for ImageDownload Object.
- ∼ImageDownload ()
  Destructor for ImageDownload Object.

Bulk Transfer Initiation

- bool StartDownload ()
  Initiates a Bulk Transfer download.
- bool StartVerify ()
  Initiates a Bulk Transfer verify.

Retrieve Error Information

- int GetVerifyError ()
  Returns the address of the next verify error or -1 if none.

Event Notifications

- void ImageDownloadEventSubscribe (const int message, IEventHandler ∗handler)
  Subscribe a callback function handler to a given ImageDownloadEvents entry.
- void ImageDownloadEventUnsubscribe (const int message, const IEventHandler ∗handler)
  Unsubscribe a callback function handler from a given ImageDownloadEvent.

17.29.1 Detailed Description

Provides a mechanism for downloading and verifying Images to a Controller's memory.

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.29.2 Constructor & Destructor Documentation

17.29.2.1 iMS::ImageDownload::ImageDownload (IMSSystem & *ims*, const Image & *img*)

Constructor for ImageDownload Object.

The pre-requisites for an ImageDownload object to be created are: (1) - an IMSSystem object, representing the configuration of an iMS target to which the Image is to be downloaded. (2) - a complete Image object to download to the iMS target.

ImageDownload stores const references to both. This means that both must exist before the ImageDownload object, and both must remain valid (not destroyed) until the ImageDownload object itself is destroyed. Because they are stored as references, the IMSSystem and Image objects themselves may be modified after the construction of the ImageDownload object.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A reference to the iMS System which is the target for downloading the Image
in	*img*	A const reference to the Image which shall be downloaded to the target

Since

1.0

17.29.3 Member Function Documentation

17.29.3.1 int iMS::ImageDownload::GetVerifyError ( ) [virtual]

Returns the address of the next verify error or -1 if none.

After the application has been notified of a failed verify, it can probe the BulkTransfer derived object to obtain the approximate address at which the BulkTransfer failed. The address is provided as a byte offset from the start of the BulkTransfer binary object.

Due to the way in which the BulkTransfer mechanism splits the transfer into individual messages, there will be one error recorded for each message that results in a verify fail. Therefore, the address will only be approximate, to the nearest message size boundary and if there are multiple byte fails within the scope of a single message, only one error will be recorded.

Calling this function repeatedly will result in returning the next recorded verify error. If there are no errors left, or the transfer was successful (i.e. there were no verify failures recorded) the function will return -1.

Returns

byte address of transfer failure or -1 if none.

Since

1.0

Implements iMS::IBulkTransfer.

17.29.3.2 void iMS::ImageDownload::ImageDownloadEventSubscribe (const int *message*, IEventHandler * *handler*)

Subscribe a callback function handler to a given ImageDownloadEvents entry.

ImageDownload can callback user application code when an event occurs in the download process. Supported events are listed under ImageDownloadEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to an ImageDownloadEvents entry. For the period that a callback is subscribed, each time an event in ImageDownload occurs that would trigger the subscribed ImageDownloadEvents entry, the user function callback will be executed.

Parameters

in	*message*	Use the ImageDownloadEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.0

17.29.3.3 void iMS::ImageDownload::ImageDownloadEventUnsubscribe (const int *message*, const IEventHandler ∗ *handler*)

Unsubscribe a callback function handler from a given ImageDownloadEvent.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the ImageDownload object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the ImageDownloadEvent::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.0

17.29.3.4 bool iMS::ImageDownload::StartDownload ( ) [virtual]

Initiates a Bulk Transfer download.

If the user has subscribed to the relevant event notifications, the BulkTransfer derived object will issue a completion event at the end of the download process and will also warn the user anytime a download messaging error occurs.

Returns

Boolean indicating whether Download has started successfully

Since

1.0

Implements iMS::IBulkTransfer.

17.29.3.5 bool iMS::ImageDownload::StartVerify ( ) [virtual]

Initiates a Bulk Transfer verify.

If the user has subscribed to the relevant event notifications, the BulkTransfer derived object will raise an event to the application at the end of the verify process to indicate whether the verification was successful or not.

Returns

Boolean indicating whether Verify has started successfully

Since

1.0

Implements iMS::IBulkTransfer.

The documentation for this class was generated from the following file:

- ImageOps.h

17.30 iMS::ImageDownloadEvents Class Reference

All the different types of events that can be triggered by the ImageDownload class.

```
#include <include\ImageOps.h>
```

Public Types

- enum Events {
  DOWNLOAD_FINISHED, DOWNLOAD_ERROR, VERIFY_SUCCESS, VERIFY_FAIL,
  DOWNLOAD_FAIL_MEMORY_FULL, DOWNLOAD_FAIL_TRANSFER_ABORT, IMAGE_DOWNLOAD_↵
  NEW_HANDLE, **Count** }

  List of Events raised by the Image Downloader.

17.30.1 Detailed Description

All the different types of events that can be triggered by the ImageDownload class.

Some events contain integer parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.30.2 Member Enumeration Documentation

17.30.2.1 enum iMS::ImageDownloadEvents::Events

List of Events raised by the Image Downloader.

Enumerator

DOWNLOAD_FINISHED Event raised when ImageDownload has confirmed that the iMS Controller received all of the Image data.

DOWNLOAD_ERROR Event raised each time the ImageDownload class registers an error in the download process.

VERIFY_SUCCESS Event raised on completion of a download verify, if the download was successfully verified.

VERIFY_FAIL Event raised on completion of a download verify, if the download failed. `param` contains the number of failures recorded.

DOWNLOAD_FAIL_MEMORY_FULL Event raise when unable to begin a fast transfer of image data to memory, e.g. Image memory is full.

DOWNLOAD_FAIL_TRANSFER_ABORT Event raise when unable to transfer any data through DMA mechanism.

IMAGE_DOWNLOAD_NEW_HANDLE Event raised when a new download has been accepted prior to memory transfer commencing, reporting the new image index handle.

The documentation for this class was generated from the following file:

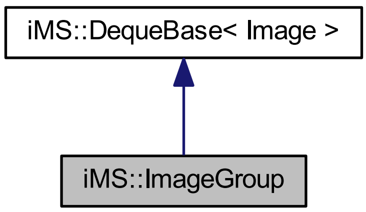
- ImageOps.h

17.31 iMS::ImageGroup Class Reference

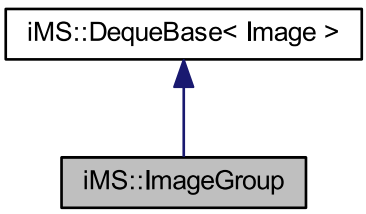
An ImageGroup collects together multiple associated images and a single ImageSequence for controlling Image playback order.

`#include <include/Image.h>`

Inheritance diagram for iMS::ImageGroup:



Collaboration diagram for iMS::ImageGroup:



Public Member Functions

Constructors & Destructor

- ImageGroup (const std::string &name="", const std::time_t &create_time=std::time(nullptr), const std↩ ::time_t &modified_time=std::time(nullptr))
  Create a default empty ImageGroup.
- ImageGroup (const ImageGroup &)
  Copy Constructor.
- ImageGroup & operator= (const ImageGroup &)
  Assignment Constructor.
- ∼ImageGroup ()
  Destructor.

ImageGroup collection modifiers

- void AddImage (const Image &img)
  Adds a new Image to the back of the Image Queue.
- iterator InsertImage (iterator it, const Image &img)
  Inserts a new Image before the specified element in the Image Queue.
- iterator RemoveImage (iterator it)
  Removes an Image at the specified element in the Image Queue.
- iterator RemoveImage (iterator first, iterator last)
  Removes a range of Image's from the specified range of elements in the Image Queue.
- void Clear ()
  Clear ImageGroup.
- int Size () const
  Returns the number of Images in the ImageGroup.

Timestamping

- const std::time_t & CreatedTime () const
  Returns Time at which the Container was created.
- std::string CreatedTimeFormat () const
  Returns Human-readable string for the time at which the ImageGroup was created.

User MetaData

- std::string & Author ()
  Author Set Accessor.
- const std::string & Author () const
  Author Get Accessor.
- std::string & Company ()
  Company Set Accessor.
- const std::string & Company () const
  Company Get Accessor.
- std::string & Revision ()
  Revision Set Accessor.
- const std::string & Revision () const
  Revision Get Accessor.
- std::string & Description ()
  Description Set Accessor.
- const std::string & Description () const
  Description Get Accessor.

ImageGroup Sequence

- ImageSequence & Sequence ()
  ImageSequence Set Accessor.
- const ImageSequence & Sequence () const
  ImageSequence Get Accesor.

Additional Inherited Members

17.31.1 Detailed Description

An ImageGroup collects together multiple associated images and a single ImageSequence for controlling Image playback order.

Individual Image's may be played back on an iMS System freely but to specify more complex behaviour, typically an ImageGroup is used. An ImageGroup can contain one or many images and always has exactly one sequence which may be used to define an order in which those Images are played back on the iMS Controller.

Additionally, user information may be supplied in the form of metadata (name, author, company, revision, description) to assist in identifying the purpose of an ImageGroup.

Date

2016-11-09

Since

1.3

17.31.2 Constructor & Destructor Documentation

17.31.2.1 iMS::ImageGroup::ImageGroup (const std::string & *name* = "", const std::time_t & *create_time* = `std::time(nullptr)`, const std::time_t & *modified_time* = `std::time(nullptr)`)

Create a default empty ImageGroup.

The ImageGroup is created with zero Images in its list and a default ImageSequence with zero entries. Call as `ImageGroup()` or `ImageGroup`("My \c Group"). do not use the create_time or modified_time parameters.

Parameters

name	Optionally, the caller may specify a Name for the ImageGroup. If not specified, it defaults to an empty string.
create_time	Specify the creation time for the ImageGroup. This is intended for use only when loading ImageGroup's from an ImageProject disk file and should not be used.
modified_time	Specify the last modified time for the ImageGroup. This is intended for use only when loading ImageGroup's from an ImageProject disk file and should not be used.

17.31.3 Member Function Documentation

17.31.3.1 void iMS::ImageGroup::AddImage (const Image & *img*)

Adds a new Image to the back of the Image Queue.

Parameters

img	a const reference to the Image to be added

17.31.3.2 std::string& iMS::ImageGroup::Author ( )

Author Set Accessor.

Sets the Author's name for the ImageGroup

Since

1.3

17.31.3.3 const std::string& iMS::ImageGroup::Author ( ) const

Author Get Accessor.

Gets the Author's name for the ImageGroup

Since

1.3

17.31.3.4 void iMS::ImageGroup::Clear ( )

Clear ImageGroup.

Remove all Images from the ImageGroup and all entries from the ImageSequence

17.31.3.5 std::string& iMS::ImageGroup::Company ( )

Company Set Accessor.

Sets the Company name for the ImageGroup

17.31.3.6 const std::string& iMS::ImageGroup::Company ( ) const

Company Get Accessor.

Gets the Company name for the ImageGroup

17.31.3.7 const std::time_t& iMS::ImageGroup::CreatedTime ( ) const

Returns Time at which the Container was created.

At the time the ImageGroup is first created, the system time is recorded. If a ImageGroup is copied or assigned to another object, the new object inherits the Creation time of the parent so the timestamp always refers to the time at which an ImageGroup was initially created.

Returns

a reference to a std::time_t representing the time at which the ImageGroup was created

Since

1.3

17.31.3.8 std::string iMS::ImageGroup::CreatedTimeFormat ( ) const

Returns Human-readable string for the time at which the ImageGroup was created.

Since

1.3

17.31.3.9 std::string& iMS::ImageGroup::Description ( )

Description Set Accessor.

Sets a Description field for the ImageGroup

17.31.3.10 const std::string& iMS::ImageGroup::Description ( ) const

Description Get Accessor.

Gets the Description field for the ImageGroup

17.31.3.11 iterator iMS::ImageGroup::InsertImage (iterator *it*, const Image & *img*)

Inserts a new Image before the specified element in the Image Queue.

Parameters

it	The queue element to insert the image before
img	a const reference to the Image to be inserted

Returns

an iterator to the newly inserted Image

17.31.3.12 iterator iMS::ImageGroup::RemoveImage (iterator *it*)

Removes an Image at the specified element in the Image Queue.

Parameters

it	the queue element to remove

Returns

an iterator to the element following the element removed from the Image Queue

17.31.3.13 iterator iMS::ImageGroup::RemoveImage (iterator *first*, iterator *last*)

Removes a range of Image's from the specified range of elements in the Image Queue.

Parameters

first	the initial queue element in the range to remove
last	the final queue element in the range to remove

Returns

an iterator to the element following the last element removed from the Image Queue

17.31.3.14 std::string& iMS::ImageGroup::Revision ( )

Revision Set Accessor.

Sets the Revision number for the ImageGroup

Please note that this field is not handled internally by the ImageGroup class. It is left to the user application to modify, update or increment the Revision number as well as specifying a numbering scheme as best fits the application.

17.31.3.15 const std::string& iMS::ImageGroup::Revision ( ) const

Revision Get Accessor.

Gets the Revision number for the ImageGroup

17.31.3.16 ImageSequence& iMS::ImageGroup::Sequence ( )

ImageSequence Set Accessor.

Returns a reference to the ImageGroup sequence to allow user application code to modify the Sequence Table.

17.31.3.17 const ImageSequence& iMS::ImageGroup::Sequence ( ) const

ImageSequence Get Accesor.

Returns a const reference to the ImageGroup sequence to allow user application code to view the Sequence Table.

17.31.3.18 int iMS::ImageGroup::Size ( ) const

Returns the number of Images in the ImageGroup.

Returns the number of Images in the ImageGroup

The documentation for this class was generated from the following file:

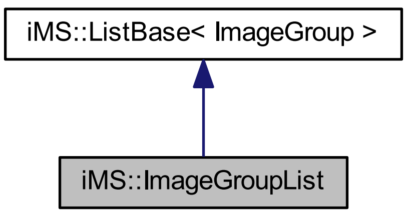
- Image.h

17.32 iMS::ImageGroupList Class Reference

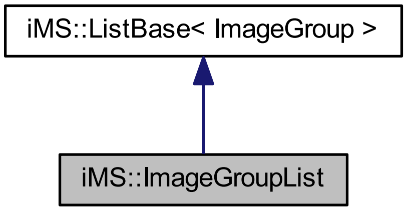
A List of ImageGroup's used as a container by ImageProject.

`#include <include/Image.h>`

Inheritance diagram for iMS::ImageGroupList:



Collaboration diagram for iMS::ImageGroupList:



Additional Inherited Members

17.32.1 Detailed Description

A List of ImageGroup's used as a container by ImageProject.

Date

2016-11-09

Since

1.3

The documentation for this class was generated from the following file:

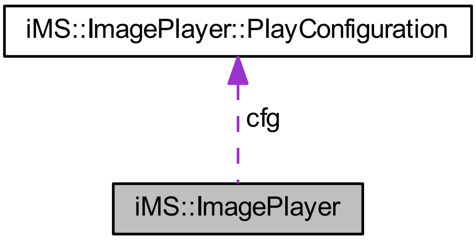
- ImageProject.h

17.33 iMS::ImagePlayer Class Reference

Once an Image has been downloaded to Controller memory, ImagePlayer can be used to configure and begin playback.

`#include <include\ImageOps.h>`

Collaboration diagram for iMS::ImagePlayer:



Classes

- struct PlayConfiguration

  This struct sets the attributes for the ImagePlayer to use when initiating an Image Playback.

Public Types

- enum PointClock { PointClock::INTERNAL, PointClock::EXTERNAL }

  Determines whether Image Progression is under the control of an internal or external clock.
- enum ImageTrigger { ImageTrigger::POST_DELAY, ImageTrigger::EXTERNAL, ImageTrigger::HOS←→T, ImageTrigger::CONTINUOUS }

  At the end of each Image, the next Image in the sequence (or the next Repeat of the same image) will begin after the ImageTrigger condition is satisfied.
- enum Polarity { Polarity::NORMAL, Polarity::INVERSE }

  The external signal connections can be configured to be active on the rising edge or the falling edge (CLK, TRIG), high or low (ENABLE)
- enum StopStyle { StopStyle::GRACEFULLY, StopStyle::IMMEDIATELY }

  The ImagePlayer can end the Image Playback either at the end of the Image or Repeat, or immediately.
- using Repeats = ImageRepeats

  Each Image can be repeated, either a programmable number of times, or indefinitely.

Public Member Functions

Constructor & Destructor

- ImagePlayer (const IMSSystem &ims, const Image &img)

  Constructor for ImagePlayer Object.
- ImagePlayer (const IMSSystem &ims, const Image &img, const PlayConfiguration &cfg)

  Constructor for ImagePlayer Object with User Configuration.
- **ImagePlayer** (const IMSSystem &ims, const ImageTableEntry &ite, const kHz InternalClock)
- **ImagePlayer** (const IMSSystem &ims, const ImageTableEntry &ite, const int ExtClockDivide)
- **ImagePlayer** (const IMSSystem &ims, const ImageTableEntry &ite, const PlayConfiguration &cfg, const kHz InternalClock)
- **ImagePlayer** (const IMSSystem &ims, const ImageTableEntry &ite, const PlayConfiguration &cfg, const int ExtClockDivide)
- ∼ImagePlayer ()

  Destructor for ImagePlayer Object.

Play Control Functions

- bool Play (ImageTrigger start_trig=ImageTrigger::CONTINUOUS)
  Starts Image Playback.
- bool GetProgress ()
  Requests current Point Progress.
- bool Stop (StopStyle stop)
  Halts the Image Playback.
- bool Stop ()
  Halts the Image Playback After Last Point in Image or Repeat.

Post Delay helper function

- void SetPostDelay (const std::chrono::duration< double > &dly)
  Helper function that sets the Post Delay configuration attribute from any compatible std::chrono class (e.g. std::chrono::milliseconds(100.0))

Event Notifications

- void ImagePlayerEventSubscribe (const int message, IEventHandler ∗handler)
  Subscribe a callback function handler to a given ImagePlayerEvent.
- void ImagePlayerEventUnsubscribe (const int message, const IEventHandler ∗handler)
  Unsubscribe a callback function handler from a given ImageDownloadEvent.

Public Attributes

- struct LIBSPEC iMS::ImagePlayer::PlayConfiguration cfg
  Defines the configuration for Image Playback.

17.33.1 Detailed Description

Once an Image has been downloaded to Controller memory, ImagePlayer can be used to configure and begin playback.

ImagePlayer contains a Configuration Structure which holds all of the different attributes that may be used to modify the behaviour of the playback, including internal oscillator or external clock, next-image triggering and image repeating. It does not define the internal oscillator clock rate for ImagePoint playback frequency when not using an external clock; this information is stored in the Image class.

Once constructed, the ImagePlayer.Play() function will begin playback, ImagePlayer.Stop() will end playback (immediately or at the end of an image) and ImagePlayer.GetProgress() will raise an event to the user application indicating the current ImagePoint that has been reached in playback.

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.33.2 Member Typedef Documentation

17.33.2.1 using iMS::ImagePlayer::Repeats = ImageRepeats

Each Image can be repeated, either a programmable number of times, or indefinitely.

Repeats

Since

1.0

17.33.3 Member Enumeration Documentation

17.33.3.1 enum iMS::ImagePlayer::ImageTrigger [strong]

At the end of each Image, the next Image in the sequence (or the next Repeat of the same image) will begin after the ImageTrigger condition is satisfied.

Since

1.0

Enumerator

POST_DELAY A programmable timer is started at the end of the image. The next image is triggered after the timer times out.

EXTERNAL The next image is triggered when an edge is detected on the TRIG signal connected to the Controller.

HOST The next image is triggered when application software sends a 'User Trigger' request.

CONTINUOUS The next image is triggered immediately.

17.33.3.2 enum iMS::ImagePlayer::PointClock [strong]

Determines whether Image Progression is under the control of an internal or external clock.

Since

1.0

Enumerator

INTERNAL ImagePoint progression through the Image at a rate determined by the programming of the internal NCO (Numerically Controlled Oscillator)

EXTERNAL ImagePoint progression through the Image one point per edge detected on the CLK signal connected to the Controller.

17.33.3.3 enum iMS::ImagePlayer::Polarity [strong]

The external signal connections can be configured to be active on the rising edge or the falling edge (CLK, TRIG), high or low (ENABLE)

Since

1.0

Enumerator

NORMAL CLK / TRIG are active on the rising edge. ENABLE is active high.

INVERSE CLK / TRIG are active on the falling edge. ENABLE is active low.

17.33.3.4 enum iMS::ImagePlayer::StopStyle [strong]

The ImagePlayer can end the Image Playback either at the end of the Image or Repeat, or immediately.

Since

1.0

Enumerator

GRACEFULLY The default method for stopping the Image is to action the Stop request at the end of the current Image, or Image Repeat.

IMMEDIATELY Use this to end the Image Playback as soon as the command is processed by the Controller.

17.33.4 Constructor & Destructor Documentation

17.33.4.1 iMS::ImagePlayer::ImagePlayer (const IMSSystem & *ims*, const Image & *img*)

Constructor for ImagePlayer Object.

An IMSSystem object, representing the configuration of an iMS target on which an Image has already been downloaded, must be passed by const reference to the ImagePlayer constructor.

The IMSSystem object must exist before the ImagePlayer object, and must remain valid (not destroyed) until the ImagePlayer object itself is destroyed.

The Image to be played back must also be passed by reference. ImagePlayer will check the unique ID (UUID) of an Image against the value that is in memory on the hardware to ensure that it is playing the same Image that has been downloaded.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System which is the target on which to playback the Image
in	*img*	A const reference to the Image that has been downloaded to the target

Since

1.0

17.33.4.2 iMS::ImagePlayer::ImagePlayer (const IMSSystem & *ims*, const Image & *img*, const PlayConfiguration & *cfg*)

Constructor for ImagePlayer Object with User Configuration.

As per the default constructor, but also receives a const reference to a PlayConfiguration struct which will have already been modified by the application to change the playback behaviour. The attributes of the struct are copied to the internal configuration struct.

An alternative is to use the default constructor and modify the configuration manually after construction.

Parameters

in	*ims*	A const reference to the iMS System which is the target for downloading the Image

in	*img*	A const reference to the Image that has been downloaded to the target
in	*cfg*	A const reference to a PlayConfiguration playback configuration structure

Since

1.0

17.33.5 Member Function Documentation

17.33.5.1 bool iMS::ImagePlayer::GetProgress ( )

Requests current Point Progress.

This function call will request from the Controller the current ImagePoint position within the playback of the current Image. If an Image playback is not in progress, this function call will return false.

Once the Controller has responded with the ImagePoint position, an ImagePlayerEvent::POINT_PROGRESS event will be triggered containing the point position which the application can register to receive.

Returns

true if the Progress request was successfully sent to the Controller

Since

1.0

17.33.5.2 void iMS::ImagePlayer::ImagePlayerEventSubscribe (const int *message*, IEventHandler * *handler*)

Subscribe a callback function handler to a given ImagePlayerEvent.

ImagePlayer can callback user application code when an event occurs during playback. Supported events are listed under ImagePlayerEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to an ImagePlayerEvent. For the period that a callback is subscribed, each time an event in ImagePlayer occurs that would trigger the subscribed ImagePlayerEvent, the user function callback will be executed.

Parameters

in	*message*	Use the ImagePlayerEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.0

17.33.5.3 void iMS::ImagePlayer::ImagePlayerEventUnsubscribe (const int *message*, const IEventHandler * *handler*)

Unsubscribe a callback function handler from a given ImageDownloadEvent.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the ImageDownload object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the ImageDownloadEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.0

17.33.5.4 bool iMS::ImagePlayer::Play (ImageTrigger *start_trig* = ImageTrigger::CONTINUOUS)

Starts Image Playback.

This function will begin the playback of an Image resident in Controller memory immediately on receipt of the message by the Controller. If an Image is already playing, the function call will fail and return false. Likewise, if an Image Download is in progress, the function call will fail and return false. If no Image has been downloaded to the Controller, this function will run successfully, but nothing will happen on the Controller.

Once the Controller has responded indicating that the Image Playback has started, an ImagePlayerEvent::IMAGE_STARTED event will be raised which the application can register to receive

Returns

true if the Play Image request was sent to the Controller successfully.

Since

1.0

17.33.5.5 void iMS::ImagePlayer::SetPostDelay (const std::chrono::duration< double > & *dly*)

Helper function that sets the Post Delay configuration attribute from any compatible std::chrono class (e.g. std::chrono::milliseconds(100.0))

Parameters

in	*dly*	Use one of the derived std::chrono classes to set an appropriate post-image delay

Since

1.0

17.33.5.6 bool iMS::ImagePlayer::Stop (StopStyle *stop*)

Halts the Image Playback.

This function call will end the playback of an Image that is currently taking place on the Controller. There are two methods for stopping Image playback: (1) StopStyle::GRACEFULLY : Ends the Image playback after the last point of the current Image. If the Image is being repeated, either indefinitely or programmatically, playback will halt at the last point of the current Repeat, irrespective of whether there are more repeats programmed to happen. (2) StopStyle::IMMEDIATELY : Ends the Image playback as soon as the message is received by the Controller.

Parameters

in	*stop*	Defines which Image Stop method to use

Returns

true if the Stop message was successfully sent to the Controller.

Since

1.0

17.33.5.7 bool iMS::ImagePlayer::Stop ( ) [inline]

Halts the Image Playback After Last Point in Image or Repeat.

Default Stop function. Identical to `Stop(StopStyle::GRACEFULLY);`

Since

1.0

The documentation for this class was generated from the following file:

- ImageOps.h

17.34 iMS::ImagePlayerEvents Class Reference

All the different types of events that can be triggered by the ImagePlayer class.

`#include <include\ImageOps.h>`

Public Types

- enum Events { POINT_PROGRESS, IMAGE_STARTED, IMAGE_FINISHED, **Count** }

  List of Events raised by the Image Player.

17.34.1 Detailed Description

All the different types of events that can be triggered by the ImagePlayer class.

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.34.2 Member Enumeration Documentation

17.34.2.1 enum iMS::ImagePlayerEvents::Events

List of Events raised by the Image Player.

Enumerator

POINT_PROGRESS Event raised in response to ImagePlayer::GetProgress(). Indicates the number of points into an Image playback.

IMAGE_STARTED Event raised when an Image in the Controller begins playback.

IMAGE_FINISHED Event raised when an Image in the Controller completes playback.

The documentation for this class was generated from the following file:

- ImageOps.h

17.35 iMS::ImagePoint Class Reference

Stores 4 FAP Triads containing frequency, amplitude and phase data for 4 RF channels.

```
#include <include/Image.h>
```

Public Member Functions

- ImagePoint ()

  Default Constructor.
- ImagePoint (FAP fap)

  Constructor with Uniform Channel Data.
- ImagePoint (FAP ch1, FAP ch2, FAP ch3, FAP ch4)

  Constructor with Independent Channel Data.
- ImagePoint (FAP fap, float synca, unsigned int syncd)

  Constructor with Uniform Channel Data and Synchronous Data.
- ImagePoint (FAP ch1, FAP ch2, FAP ch3, FAP ch4, float synca_1, float synca_2, unsigned int syncd)

  Constructor with Independent Channel Data and Synchronous Data.
- bool operator== (ImagePoint const &rhs) const

  Equality Operator checks ImagePoint object for equivalence.

Get/Set FAP data for the image point

- const FAP & GetFAP (const RFChannel) const

  Retrieves Frequency, Amplitude (Percent) and Phase (Degrees) data for one RF channel.
- void SetFAP (const RFChannel, const FAP &)

  Assigns Frequency, Amplitude (Percent) and Phase (Degrees) data for one RF channel.
- FAP & SetFAP (const RFChannel)

  Assigns Frequency, Amplitude (Percent) and Phase (Degrees) data for one RF channel.
- void SetAll (const FAP &)

  Assigns Frequency, Amplitude (Percent) and Phase (Degrees) data for all RF channels.

Get/Set Synchronous data for the image point

- const float & GetSyncA (int index) const
  Retrieve Analogue Synchronous Data.
- void SetSyncA (int index, const float &value)
  Assign Analogue Synchronous Data.
- const unsigned int & GetSyncD () const
  Retrieve Digital Synchronous Data.
- void SetSyncD (const unsigned int &value)
  Assign Digital Synchronous Data.

17.35.1 Detailed Description

Stores 4 FAP Triads containing frequency, amplitude and phase data for 4 RF channels.

An ImagePoint uniquely defines the required output drive setting for each of the 4 RF Channels output by the iMS Synthesiser. Each channel (from 1 to 4) is given its own FAP member variable, which is a combination of Frequency, Amplitude (Percent) and Phase (Degrees) data.

At any instantaneous moment, the status of the 4 iMS RF driver outputs is representable by a single ImagePoint

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.35.2 Constructor & Destructor Documentation

17.35.2.1 iMS::ImagePoint::ImagePoint (FAP *fap*)

Constructor with Uniform Channel Data.

Construct ImagePoint with identical FAP for each channel

Since

1.0

17.35.2.2 iMS::ImagePoint::ImagePoint (FAP *ch1,* FAP *ch2,* FAP *ch3,* FAP *ch4*)

Constructor with Independent Channel Data.

Construct ImagePoint with full specification of FAP for each channel

Since

1.0

17.35.2.3 iMS::ImagePoint::ImagePoint (FAP *fap*, float *synca*, unsigned int *syncd*)

Constructor with Uniform Channel Data and Synchronous Data.

Construct ImagePoint with identical FAP for each channel

Since

1.2

17.35.2.4 iMS::ImagePoint::ImagePoint (FAP *ch1*, FAP *ch2*, FAP *ch3*, FAP *ch4*, float *synca_1*, float *synca_2*, unsigned int *syncd*)

Constructor with Independent Channel Data and Synchronous Data.

Construct ImagePoint with full specification of FAP for each channel

Since

1.2

17.35.3 Member Function Documentation

17.35.3.1 const FAP& iMS::ImagePoint::GetFAP (const RFChannel) const

Retrieves Frequency, Amplitude (Percent) and Phase (Degrees) data for one RF channel.

Returns

FAP triad for the specified RF channel

Since

1.0

17.35.3.2 const float& iMS::ImagePoint::GetSyncA (int *index*) const

Retrieve Analogue Synchronous Data.

Parameters

in	*index*	0 or 1 references 2 independent synchronous data variables

Returns

a floating point value between 0 and 1

Since

1.2

17.35.3.3 const unsigned int& iMS::ImagePoint::GetSyncD ( ) const

Retrieve Digital Synchronous Data.

Returns

an unsigned integer value representing the synchronous data

Since

1.2

17.35.3.4 bool iMS::ImagePoint::operator== (ImagePoint const & *rhs*) const

Equality Operator checks ImagePoint object for equivalence.

Parameters

in	*rhs*	An ImagePoint object to perform the comparison with

Returns

True if the supplied ImagePoint is identical to this one.

Since

1.1

17.35.3.5 void iMS::ImagePoint::SetAll (const FAP &)

Assigns Frequency, Amplitude (Percent) and Phase (Degrees) data for all RF channels.

Since

1.0

17.35.3.6 void iMS::ImagePoint::SetFAP (const RFChannel , const FAP &)

Assigns Frequency, Amplitude (Percent) and Phase (Degrees) data for one RF channel.

Since

1.0

17.35.3.7 FAP& iMS::ImagePoint::SetFAP (const RFChannel)

Assigns Frequency, Amplitude (Percent) and Phase (Degrees) data for one RF channel.

Since

1.2

17.35.3.8 void iMS::ImagePoint::SetSyncA (int *index,* const float & *value*)

Assign Analogue Synchronous Data.

Parameters

in	*index*	0 or 1 references 2 independent synchronous data variables
in	*value*	The floating point value to assign, will be clamped within the range 0 <= value <= 1

Since

1.2

17.35.3.9 void iMS::ImagePoint::SetSyncD (const unsigned int & *value*)

Assign Digital Synchronous Data.

Parameters

in	*value*	The unsigned integer value to assign

Since

1.2

The documentation for this class was generated from the following file:

- Image.h

17.36 iMS::ImageProject Class Reference

An ImageProject allows the user to organise their data and store it on the host computer.

`#include <include/Image.h>`

Public Member Functions

- void Clear ()
  Reset ImageProject to empty state.

Constructors & Destructor

- ImageProject ()
  Default Constructor.
- ImageProject (const std::string &fileName)
  Implicit Load From File Constructor.

Image Group Container

- ImageGroupList & ImageGroupContainer ()
  Set Accessor for the Image Group Container.
- const ImageGroupList & ImageGroupContainer () const
  Get Accessor for the Image Group Container.

Compensation Function Container

- CompensationFunctionList & CompensationFunctionContainer ()
  Set Accessor for the Compensation Function Container.

- const CompensationFunctionList & CompensationFunctionContainer () const
  Get Accessor for the Compensation Function Container.

Tone Buffer Container

- ToneBufferList & ToneBufferContainer ()
  Set Accessor for the Tone Buffer Container.
- const ToneBufferList & ToneBufferContainer () const
  Get Accessor for the Tone Buffer Container.

Free Image Container

- ImageGroup & FreeImageContainer ()
  Set Accessor for the Free Image Container.
- const ImageGroup & FreeImageContainer () const
  Get Accessor for the Tone Buffer Container.

FileSystem Functions

- bool Save (const std::string &fileName)
  Save ImageProject to host disk.
- bool Load (const std::string &fileName)
  Load ImageProject from host disk.

17.36.1 Detailed Description

An ImageProject allows the user to organise their data and store it on the host computer.

An ImageProject permits the user to organise and contain all of their data relating to a specific use case for the iMS. Any number of ImageGroup's, CompensationFunction's, ToneBuffer's and/or Free Images (individual Images not associated in an ImageGroup and with no ImageSequence) may be help in an ImageProject. The data is stored on disk in an efficient custom file format that may be unzipped by the user and inspected in an XML file viewer, if wished.

A simple use case for the ImageProject is to contain a single Image, kept in the FreeImageContainer, and to load/save that Image to disk. A more complex use case might be to hold different ImageGroup's for different purposes that are used as the raw data for a processing task that requires some other software to select between Image's and ImageSequence's according to some system parameter.

Date

2016-11-09

Since

1.3

17.36.2 Constructor & Destructor Documentation

17.36.2.1 iMS::ImageProject::ImageProject ( )

Default Constructor.

Creates an empty ImageProject.

17.36.2.2 iMS::ImageProject::ImageProject (const std::string & *fileName*)

Implicit Load From File Constructor.

Calls the default constructor, then ImageProject::Load() to initialise the object from the filesystem param[in] fileName String pointing to a valid .iip or .xml file on the host filesystem to load into the ImageProject

17.36.3 Member Function Documentation

17.36.3.1 void iMS::ImageProject::Clear ( )

Reset ImageProject to empty state.

Removes all entries from all of the containers

17.36.3.2 CompensationFunctionList& iMS::ImageProject::CompensationFunctionContainer ( )

Set Accessor for the Compensation Function Container.

Use this function to modify, add and remove CompensationFunction's from the ImageProject.

17.36.3.3 ImageGroup& iMS::ImageProject::FreeImageContainer ( )

Set Accessor for the Free Image Container.

Use this function to modify, add and remove individual Image's from the ImageProject. The Free Image Container is configured as an ImageGroup to allow user software to access its Image's using the same function calls as when working with an ImageGroup object. In this regard, the FreeImageContainer may be regarded as a "superset" of the ImageGroup that permits save/load to disk.

17.36.3.4 ImageGroupList& iMS::ImageProject::ImageGroupContainer ( )

Set Accessor for the Image Group Container.

Use this function to modify, add and remove ImageGroup's from the ImageProject.

17.36.3.5 bool iMS::ImageProject::Load (const std::string & *fileName*)

Load ImageProject from host disk.

First clears the ImageProject of any existing content. Then it will try to read in data from the provided file, work out whether it is compressed or uncompressed, and populate the ImageProject containers from the file data. The function is fully backwards compatible with all previous versions of the SDK and should therefore be capable of reading in any file ever generated by the SDK. It is also compatible with ImageProject files generated by the previous generation of Isomet Image software, the iHHS ImageFile Generator

Parameters

in	*fileName*	String representing the full path of a file to load fata from.

17.36.3.6 bool iMS::ImageProject::Save (const std::string & *fileName*)

Save ImageProject to host disk.

Stores all data in the ImageProject containers to a custom file format on disk. The preferred file extension is ".↩ iip" for Isomet Image Project. If the passed file name ends in .xml however, the function will save the data to an uncompressed XML type file which can be read in any XML file viewer.

Warning

XML files saved this way can be very large!

If the passed file name ends in neither .xml nor .iip, the function will save the data in compressed format as if it ended in .iip but respecting the user's choice of file name.

Parameters

in	*fileName*	String representing the full path of a file to save data to, ending in .xml (uncompressed) or .iip (compressed)

17.36.3.7 ToneBufferList& iMS::ImageProject::ToneBufferContainer ( )

Set Accessor for the Tone Buffer Container.

Use this function to modify, add and remove ToneBuffer's from the ImageProject.

The documentation for this class was generated from the following file:

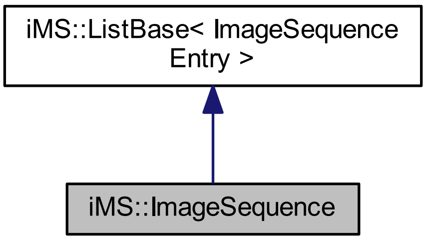
- ImageProject.h

17.37 iMS::ImageSequence Class Reference

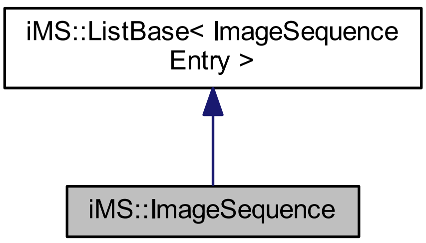
An ImageSequence object completely defines a sequence to be played back on an iMS Controller in terms by containing a list of ImageSequenceEntry 's plus a terminating action and optional value.

`#include <include/Image.h>`

Inheritance diagram for iMS::ImageSequence:



Collaboration diagram for iMS::ImageSequence:



Public Member Functions

Constructors & Destructor

- ImageSequence ()
  Create a default empty Image Sequence.
- ImageSequence (SequenceTermAction action, int val=0)
  Create a default empty Image Sequence with Termination Action specifier.
- ∼ImageSequence ()
  Destructor.
- ImageSequence (const ImageSequence &)
  Copy Constructor.
- ImageSequence & operator= (const ImageSequence &)
  Assignment Constructor.

Control Sequence Terminating Actions

- void OnTermination (SequenceTermAction act, int val=0)
  Update Termination Action.
- const SequenceTermAction & TermAction () const
  return a reference to the currently assign Termination Action
- const int & TermValue () const
  return a reference to the currently assign Termination Action Parameter

Additional Inherited Members

17.37.1 Detailed Description

An ImageSequence object completely defines a sequence to be played back on an iMS Controller in terms by containing a list of ImageSequenceEntry 's plus a terminating action and optional value.

Each ImageSequenceEntry defines the Image to be played back at that point in the sequence, together with relevant parameters such as clock frequency, divider and number of repeats. The ImageSequenceEntry 's are played back in the order in which they appear in the ImageSequence list.

The ImageSequence is a container for the list of ImageSequenceEntry 's. User application code can create the entries and add them / remove them from the front or back of the list, insert them or erase them from anywhere in the list, or assign multiple copies of the entry to the list.

As with Images, ImageSequences have a Unique ID (UUID) associated with them which are used to uniquely refer to sequences when communicating with the iMS Controller through the SequenceManager.

Date

2016-04-24

Since

1.2.4

17.37.2 Constructor & Destructor Documentation

17.37.2.1 iMS::ImageSequence::ImageSequence (SequenceTermAction *action,* int *val* = 0)

Create a default empty Image Sequence with Termination Action specifier.

Parameters

action	The operation to perform once the Sequence has completed playback
val	Optional parameter to the Termination Action

17.37.3 Member Function Documentation

17.37.3.1 void iMS::ImageSequence::OnTermination (SequenceTermAction *act*, int *val* = 0)

Update Termination Action.

Parameters

in	*act*	Assign an operation to perform when the Sequence completes
in	*val*	Optional Parameter to use with some Termination Actions

17.37.3.2 const SequenceTermAction& iMS::ImageSequence::TermAction ( ) const

return a reference to the currently assign Termination Action

Returns

a reference to the currently assign Termination Action

17.37.3.3 const int& iMS::ImageSequence::TermValue ( ) const

return a reference to the currently assign Termination Action Parameter

Returns

a reference to the currently assign Termination Action Parameter

The documentation for this class was generated from the following file:

- Image.h

17.38 iMS::ImageSequenceEntry Struct Reference

An ImageSequenceEntry object can be created by application software to specify the parameters by which an Image is played back during an ImageSequence.

```
#include <include/Image.h>
```

Public Member Functions

- bool operator== (ImageSequenceEntry const &rhs) const

  Equality Operator checks ImageSequenceEntry object for equivalence.

Constructors & Destructor

- ImageSequenceEntry ()

  Default Constructor.
- ImageSequenceEntry (const Image &img, const ImageRepeats &Rpt=ImageRepeats::NONE, const int rpts=0)

Construct ImageSequenceEntry object from Image object resident in application software.
- ImageSequenceEntry (const ImageTableEntry &ite, const kHz &InternalClock=kHz(1.0), const ImageRepeats &Rpt=ImageRepeats::NONE, const int rpts=0)
  Construct ImageSequenceEntry object from an Image resident in Controller memory referenced by its index table entry.
- ImageSequenceEntry (const ImageTableEntry &ite, const int ExtClockDivide=1, const ImageRepeats &Rpt=ImageRepeats::NONE, const int rpts=0)
  Construct ImageSequenceEntry object from an Image resident in Controller memory referenced by its index table entry.
- ∼ImageSequenceEntry ()
  Destructor.
- ImageSequenceEntry (const ImageSequenceEntry &)
  Copy Constructor.
- ImageSequenceEntry & operator= (const ImageSequenceEntry &)
  Assignment Constructor.

Delay Settings

- std::chrono::duration< double > & PostImgDelay ()
  Setter for post Image delay.
- const std::chrono::duration< double > & PostImgDelay () const
  Getter for post Image delay.
- std::chrono::duration< double > & SyncOutDelay ()
  Setter for Synchronous Digital Output signal Delay.
- const std::chrono::duration< double > & SyncOutDelay () const
  Getter for Synchronous Digital Output signal delay.

Sequence Entry Parameters

- const std::array< std::uint8_t, 16 > & UUID () const
  Image Unique Identifier can be used to synchronise Sequence Entries with host software Image objects.
- const int & ExtDiv () const
  returns the programmed External Clock Divider ratio
- const Frequency & IntOsc () const
  returns the programmed Internal Oscillator Frequency
- const ImageRepeats & RptType () const
  returns the configured Repeat style
- const int & NumRpts () const
  returns the number of times to repeat an Image before moving to the next entry in the Sequence

17.38.1 Detailed Description

An ImageSequenceEntry object can be created by application software to specify the parameters by which an Image is played back during an ImageSequence.

An ImageSequence contains a list of ImageSequenceEntry s each of which is programmed with one Image (or ImageTableEntry) specifying the ImagePoint data that will be output during playback. Additional parameters that can be specified include

- Internal Clock Frequency (implicitly defined when programmed from an Image object)
- External Clock Divider (implicitly defined when programmed from an Image object)
- Number of Repeats (to a maximum of 255)
- Amount of delay to be added after the end of Image playback (if programmed for Post-Image Delay mode)
- Amount of delay to apply to the Synchronous Digital Output signals

Author

Dave Cowan

Date

2016-04-24

Since

1.2.4

17.38.2 Constructor & Destructor Documentation

17.38.2.1 iMS::ImageSequenceEntry::ImageSequenceEntry (const Image & *img*, const ImageRepeats & *Rpt* = ImageRepeats::NONE, const int *rpts* = 0)

Construct ImageSequenceEntry object from Image object resident in application software.

If using this construction method, the Internal Clock Rate or External Clock Divider, if required, should first be set using the Image::ClockRate() and Image::ExtClockDivide() functions.

The user can optionally specify the number of times to repeat the Image before moving on to the next entry in the sequence. The default is no repeats, and these parameters may then be ommitted.

Parameters

img	A reference to the Image object which is to be played in the Sequence (must have been downloaded to Controller memory before playback)
Rpt	An optional parameter specifying whether repeats are required (ImageRepeats::PROGRAM) or not (ImageRepeats::NONE)
rpts	An optional integer specifying the number of repeats to perform (max 255).

17.38.2.2 iMS::ImageSequenceEntry::ImageSequenceEntry (const ImageTableEntry & *ite*, const kHz & *InternalClock* = kHz(1.0), const ImageRepeats & *Rpt* = ImageRepeats::NONE, const int *rpts* = 0)

Construct ImageSequenceEntry object from an Image resident in Controller memory referenced by its index table entry.

This is the preferred method for constructing an ImageSequenceEntry object when the Image has already have been downloaded to the Controller. However, the Index Table in the Controller does not store default clock frequency or clock divider information, so this must be specified manually.

The user can optionally specify the number of times to repeat the Image before moving on to the next entry in the sequence. The default is no repeats, and these parameters may then be ommitted.

Parameters

ite	A reference to the Image object from ite ImageTableEntry (can be retrieved from the IMS↵ System object through an ImageTableViewer)
InternalClock	Specifies the clock rate with which to program the Internal NCO oscillator (optional, defaults to 1kHz)
Rpt	An optional parameter specifying whether repeats are required (ImageRepeats::PROGRAM) or not (ImageRepeats::NONE)
rpts	An optional integer specifying the number of repeats to perform (max 255).

17.38.2.3 iMS::ImageSequenceEntry::ImageSequenceEntry (const ImageTableEntry & *ite*, const int *ExtClockDivide* = 1, const ImageRepeats & *Rpt* = ImageRepeats::NONE, const int *rpts* = 0)

Construct ImageSequenceEntry object from an Image resident in Controller memory referenced by its index table entry.

This is the preferred method for constructing an ImageSequenceEntry object when the Image has already have been downloaded to the Controller. However, the Index Table in the Controller does not store default clock frequency or clock divider information, so this must be specified manually.

The user can optionally specify the number of times to repeat the Image before moving on to the next entry in the sequence. The default is no repeats, and these parameters may then be ommitted.

Parameters

ite	A reference to the Image object from ite ImageTableEntry (can be retrieved from the IMS↩ System object through an ImageTableViewer)
ExtClockDivide	divides down the externally supplied clock signal by an integer ratio, e.g. 3 => update every 3rd clock edge (optional, default to 1, i.e. off)
Rpt	An optional parameter specifying whether repeats are required (ImageRepeats::PROGRAM) or not (ImageRepeats::NONE)
rpts	An optional integer specifying the number of repeats to perform (max 255).

17.38.3 Member Function Documentation

17.38.3.1 const int& iMS::ImageSequenceEntry::ExtDiv ( ) const

returns the programmed External Clock Divider ratio

Returns

the programmed External Clock Divider ratio

17.38.3.2 const Frequency& iMS::ImageSequenceEntry::IntOsc ( ) const

returns the programmed Internal Oscillator Frequency

Returns

the programmed Internal Oscillator Frequency

17.38.3.3 const int& iMS::ImageSequenceEntry::NumRpts ( ) const

returns the number of times to repeat an Image before moving to the next entry in the Sequence

Returns

the number of times to repeat an Image before moving to the next entry in the Sequence

17.38.3.4 bool iMS::ImageSequenceEntry::operator== (ImageSequenceEntry const & *rhs*) const

Equality Operator checks ImageSequenceEntry object for equivalence.

Parameters

in	*rhs*	An ImageSequenceEntry object to perform the comparison with

Returns

True if the supplied ImageSequenceEntry is identical to this one.

17.38.3.5 std::chrono::duration<double>& iMS::ImageSequenceEntry::PostImgDelay ( )

Setter for post Image delay.

If the ImageSequence is configured to create a 'pause' at the end of playback for each Image in the sequence, the pause time can be programmed on a per entry basis using this function. Set SequenceManager::Seq←Configuration::trig to ImageTrigger::POST_DELAY to use this feature.

The Pause time may be specified as any std::chrono value using duration_cast but hardware limitations restrict the real delay time to a resolution of 0.1ms and a maximum of 6.5535s.

Returns

a lvalue reference to the Post Delay time

17.38.3.6 const std::chrono::duration<double>& iMS::ImageSequenceEntry::PostImgDelay ( ) const

Getter for post Image delay.

Returns

a const reference (rvalue) for reading the Post Delay time

17.38.3.7 const ImageRepeats& iMS::ImageSequenceEntry::RptType ( ) const

returns the configured Repeat style

Returns

the configured Repeat style

17.38.3.8 std::chrono::duration<double>& iMS::ImageSequenceEntry::SyncOutDelay ( )

Setter for Synchronous Digital Output signal Delay.

The Synchronous Digital Output signals of the Synthesiser can be used to output data from either the FAP Synchronous digital field or from entries in the Compensation Look Up Table. The data updates at the same time as the Image data updated the RF output. Using the SyncOutDelay field, the data can be shifted in time to compensate for latency in the system or to, for example, delay a trigger pulse to the middle of an RF Image Point.

The Delay time may be specified as any std::chrono value using duration_cast but hardware limitations restrict the real delay time to a minimum of 0.01us and a maximum of 655.35us.

Returns

a lvalue reference to the Synchronous Digital Output delay time

17.38.3.9 const std::chrono::duration<double>& iMS::ImageSequenceEntry::SyncOutDelay ( ) const

Getter for Synchronous Digital Output signal delay.

Returns

a const reference (rvalue) for reading the Synchronous Digital Output signal delay time

17.38.3.10 const std::array< std::uint8_t, 16 >& iMS::ImageSequenceEntry::UUID ( ) const

Image Unique Identifier can be used to synchronise Sequence Entries with host software Image objects.

Each Image created in application software is automatically assigned a Unique ID (UUID) which is updated anytime the Image is modified. Sequences are internally specified using the UUID of an Image to ensure absolute consistency with the Image stored in Controller memory and referenced in the Index table. The User can check whether an Image object matches the Image referenced in a ImageSequenceEntry by comparing its UUID.

Returns

a 16 byte array containing the Image UUID.

The documentation for this struct was generated from the following file:

- Image.h

17.39 iMS::ImageTableEntry Struct Reference

An ImageTableEntry is created by the SDK on connecting to an iMS System, one for each Image that is stored in Controller memory and allocated in the Image Index Table. Further ImageTableEntries are added to the table each time an Image is downloaded to the Controller.

`#include <include/Image.h>`

Public Member Functions

Constructors & Destructor

- ImageTableEntry ()
  Default Constructor.
- ImageTableEntry (ImageIndex handle, std::uint32_t address, int n_pts, int size, std::uint32_t fmt, std↩ ::array< std::uint8_t, 16 > uuid, std::string name)
  Full Specification Constructor.
- ImageTableEntry (ImageIndex handle, const std::vector< std::uint8_t > &)
  Construct object from byte array in binary format specific to Controller communications. Used internally to build ImageTableEntries.
- ∼ImageTableEntry ()
  Destructor.
- ImageTableEntry (const ImageTableEntry &)
  Copy Constructor.
- ImageTableEntry & operator= (const ImageTableEntry &)
  Assignment Constructor.

Image Details

- const ImageIndex & Handle () const
  Unique Image Handle within Index Table.
- const std::uint32_t & Address () const
  Byte Address of Start of Image data stored within the Controller's Memory.
- const int & NPts () const
  the number of points in the Image
- const int & Size () const
  the size of the Image in bytes
- const std::uint32_t & Format () const

A Format Specifier relates the byte structure of the Image in Controller Memory to Image Physical Data.

- const std::array< std::uint8_t, 16 > & UUID () const

  Image Unique Identifier can be used to synchronise Image Entries with host software Image objects.

- const std::string & Name () const

  Descriptive Name assigned to an Image to aid User Recognition.

17.39.1 Detailed Description

An ImageTableEntry is created by the SDK on connecting to an iMS System, one for each Image that is stored in Controller memory and allocated in the Image Index Table. Further ImageTableEntries are added to the table each time an Image is downloaded to the Controller.

An ImageTableEntry should not be created by user software since it cannot be used to download Images to an iMS Controller and will not bear any relation to an existing Image on the Controller. Instead, an ImageDownload operation should be performed on an Image object to send the Image data to Controller memory which will automatically create the index data in the Image Index Table.

An ImageTableEntry can then be returned from an ImageTableViewer::operator [ ] function call into the IMSSystem object.

This will result in being able to access relevant information about Images currently on the Controller, including Image Memory Size, number of Image points, address in memory, Name etc.

The returned ImageTableEntry object may also be passed to either an ImagePlayer or ImageSequenceEntry object to permit playback of Images on the Controller.

Author

Dave Cowan

Date

2016-04-03

Since

1.2.1

17.39.2 Constructor & Destructor Documentation

17.39.2.1 iMS::ImageTableEntry::ImageTableEntry ( )

Default Constructor.

It should not be necessary to construct an ImageTableEntry object since this will be done automatically by the SDK on connection to an iMS System or after ImageDownload completes. Entries may then be referenced through the ImageTableViewer class

17.39.3 Member Function Documentation

17.39.3.1 const std::uint32_t& iMS::ImageTableEntry::Address ( ) const

Byte Address of Start of Image data stored within the Controller's Memory.

This is usually for information only as the ImageDownload class in conjunction with Controller firmware will select a memory location with sufficient free capacity. User software is never responsible for memory management and does not require the address for Image operations.

Returns

an unsigned integer representing the absolute address of the Image in the Controller memory address space

17.39.3.2 const std::uint32_t& iMS::ImageTableEntry::Format ( ) const

A Format Specifier relates the byte structure of the Image in Controller Memory to Image Physical Data.

An Image as created in application software consists of physical information such as "frequency of channel 1 at Image Point 1000". This must be translated into a byte format that is understood by the hardware to create the RF signal. There are a number of optimisations that can be performed to trade off between flexibility and update speed, the mapping between real and physical Image data is described by the Format value.

Returns

an unsigned integer representing the Image Format

17.39.3.3 const ImageIndex& iMS::ImageTableEntry::Handle ( ) const

Unique Image Handle within Index Table.

Returns

An Image handle referencing the location of the Image Entry within the Image Table

17.39.3.4 const std::string& iMS::ImageTableEntry::Name ( ) const

Descriptive Name assigned to an Image to aid User Recognition.

Each Image can be assigned a descriptive name to help identify its purpose. The first 16 bytes are transferred to the Controller during Image Download. The Name is optional and will return an empty string if not used. Be aware that due to the 16 byte limitation, the Name returned from the ImageTableEntry may differ from the name assigned to the Image in application software (whose length is unlimited).

Returns

a string object representing the description assigned to the Image

17.39.3.5 const int& iMS::ImageTableEntry::NPts ( ) const

the number of points in the Image

Returns

the number of points in the Image

17.39.3.6 const int& iMS::ImageTableEntry::Size ( ) const

the size of the Image in bytes

Returns

the size of the Image in bytes

17.39.3.7 const std::array<std::uint8_t, 16>& iMS::ImageTableEntry::UUID ( ) const

Image Unique Identifier can be used to synchronise Image Entries with host software Image objects.

Each Image created in application software is automatically assigned a Unique ID (UUID) which is updated anytime the Image is modified. The UUID is downloaded to the Image Table along with the Image and can be used to establish whether an Image resident in memory is identical to an Image present in application software, without having to upload the Image data.

The UUID is also the mechanism that allows Sequences to be created from individual Images, either directly from the Image object, or from Images in Controller memory via the ImageTableEntry.

Returns

a 16 byte array containing the Image UUID.

The documentation for this struct was generated from the following file:

- Image.h

17.40 iMS::ImageTableViewer Class Reference

Provides a mechanism for viewing the ImageTable associated with an iMS System.

`#include <include\ImageOps.h>`

Public Member Functions

Constructor

- ImageTableViewer (const IMSSystem &ims)
  Constructor for ImageTableViewer Object.

Image Table Information

- const int Entries () const

Array operator for random access to ImageTableEntry s

- const ImageTableEntry operator[ ] (const std::size_t idx) const
  The ImageTable consists of a container of ImageTableEntry objects. Each object may be accessed by calling the viewer object through an array subscript.

Friends

- LIBSPEC std::ostream & operator<< (std::ostream &stream, const ImageTableViewer &)
  Stream operator overload to simplify debugging.

17.40.1 Detailed Description

Provides a mechanism for viewing the ImageTable associated with an iMS System.

Author

Dave Cowan

Date

2016-01-21

Since

1.1

17.40.2 Constructor & Destructor Documentation

17.40.2.1 iMS::ImageTableViewer::ImageTableViewer (const IMSSystem & *ims*) [inline]

Constructor for ImageTableViewer Object.

The ImageTableViewer object requires an IMSSystem object, which will have had its ImageTable read back during initialisation. It must therefore exist before the ImageTableViewer object, and must remain valid (not destroyed) until the ImageTableViewer object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System whose ImageTable is to be viewed.

Since

1.2

17.40.3 Member Function Documentation

17.40.3.1 const int iMS::ImageTableViewer::Entries ( ) const

Returns

The current number of entries stored in the ImageTable

Since

1.2

17.40.3.2 const ImageTableEntry iMS::ImageTableViewer::operator[ ] (const std::size_t *idx*) const

The ImageTable consists of a container of ImageTableEntry objects. Each object may be accessed by calling the viewer object through an array subscript.

For example:

```
ImageTableViewer itv(myiMS);
int length = 0;
for (int i=0; i<itv.Entries(); i++) {
    length += itv[i].Size();
}
std::cout << "Used space in Image Memory: " << length << " bytes" << std::endl;
```

Since

1.1

17.40.4 Friends And Related Function Documentation

17.40.4.1 LIBSPEC std::ostream& operator<< (std::ostream & *stream*, const ImageTableViewer &) [friend]

Stream operator overload to simplify debugging.

Example usage:

```
ImageTableViewer itv(myiMS);
if (itv.Entries() > 0) std::cout << itv;
```

might produce the result:

```
Image[0] id : 0 Addr : 0x00400000 Points : 10001 ByteLength : 440044 Format Code : 0 UUID : b31bdf48 - 0902
       - 4277 - 86e1 - a6f0756a6acb
Image[1] id : 1 Addr : 0x0046b6f0 Points : 08501 ByteLength : 374044 Format Code : 0 UUID : 5e03d558 - 46e8
       - 49c4 - 80cf - d32fb51d8628
Image[2] id : 2 Addr : 0x004c6c10 Points : 12461 ByteLength : 548284 Format Code : 0 UUID : 7358b86c - 0e90
       - 4664 - 8b2b - ee0ba24542da
```

The documentation for this class was generated from the following file:

- ImageOps.h

17.41 iMS::IMSController Class Reference

Stores Capabilities, Description, Model & Version Number of an iMS Controller.

`#include <include/IMSSystem.h>`

Classes

- struct Capabilities

  Returns information about the capabilities of the Controller hardware.

Public Member Functions

- const Capabilities GetCap () const

  Returns the Capabilities structure for the Controller.
- const std::string & Description () const

  Returns a descriptive string for the Controller.
- const std::string & Model () const

  Returns the short model number for the Controller.
- const FWVersion & GetVersion () const

  Returns the firmware version for the Controller.
- const ImageTable & ImgTable () const

  Returns the Image Index Table for the Controller.
- const bool IsValid () const

  Returns true if the system scan successfully identified the Controller and initialised this Class.

17.41.1 Detailed Description

Stores Capabilities, Description, Model & Version Number of an iMS Controller.

An IMSController class is a member of the IMSSystem class and contains valid information about an iMS Controller if the ConnectionList::scan() function was able to successfully identify it.

The fields that can be read back to describe the controller can be used in Application code to select between Controllers, display information about them or determine capabilities. The information is also used by internal library functions to correctly format data and messages that are sent to the hardware.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.41.2 Member Function Documentation

17.41.2.1 const std::string& iMS::IMSController::Description ( ) const

Returns a descriptive string for the Controller.

Since

1.0

17.41.2.2 const Capabilities iMS::IMSController::GetCap ( ) const

Returns the Capabilities structure for the Controller.

Since

1.0

17.41.2.3 const FWVersion& iMS::IMSController::GetVersion ( ) const

Returns the firmware version for the Controller.

Since

1.0

17.41.2.4 const ImageTable& iMS::IMSController::ImgTable ( ) const

Returns the Image Index Table for the Controller.

Since

1.2

17.41.2.5 const bool iMS::IMSController::IsValid ( ) const

Returns true if the system scan successfully identified the Controller and initialised this Class.

Returns

true if the class contains valid data representing an attached iMS Controller

Since

1.0

17.41.2.6 const std::string& iMS::IMSController::Model ( ) const

Returns the short model number for the Controller.

Since

1.0

The documentation for this class was generated from the following file:

- IMSSystem.h

17.42 iMS::IMSOption Class Reference

An iMS Synthesiser can support one iMS Option, which adds an additional hardware function to the capabilities of the Synthesiser.

```
#include <include/IMSSystem.h>
```

17.42.1 Detailed Description

An iMS Synthesiser can support one iMS Option, which adds an additional hardware function to the capabilities of the Synthesiser.

One example of an iMS Option is a Frequency Doubler, the iMS-FX2, which doubles the available range of frequencies reproducible by the Synthesiser RF output.

Note

This class has not yet been implemented

Author

Dave Cowan

Date

2015-11-03

Since

1.0

The documentation for this class was generated from the following file:

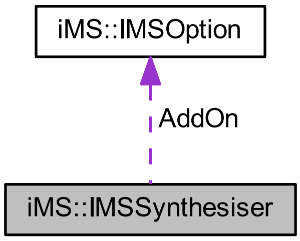
- IMSSystem.h

17.43 iMS::IMSSynthesiser Class Reference

Stores Capabilities, Description, Model & Version Number of an iMS Synthesiser.

`#include <include/IMSSystem.h>`

Collaboration diagram for iMS::IMSSynthesiser:



Classes

- struct Capabilities

  Returns information about the capabilities of the Synthesiser hardware.

Public Member Functions

- const Capabilities GetCap () const

  Returns the Capabilities structure for the Synthesiser.
- const std::string & Description () const

  Returns a descriptive string for the Synthesiser.
- const std::string & Model () const

  Returns the short model number for the Synthesiser.
- const FWVersion & GetVersion () const

  Returns the Firmware version for the Synthesiser.
- const bool IsValid () const

  Returns true if the system scan successfully identified the Synthesiser and initialised this Class.
- const FileSystemTable & FST () const

  Returns the FileSystemTable for the Synthesiser.

Public Attributes

- IMSOption ∗ AddOn

  If there are any Options attached to the Synthesiser, these are accessed here, else a null pointer is returned.

17.43.1 Detailed Description

Stores Capabilities, Description, Model & Version Number of an iMS Synthesiser.

An IMSSynthesiser class is a member of the IMSSystem class and contains valid information about an iMS Synthesiser if the ConnectionList::scan() function was able to successfully identify it.

The fields that can be read back to describe the Synthesiser can be used in Application code to select between Synthesisers, display information about them or determine capabilities. The information is also used by internal library functions to correctly format data and messages that are sent to the hardware.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.43.2 Member Function Documentation

17.43.2.1 const std::string& iMS::IMSSynthesiser::Description ( ) const

Returns a descriptive string for the Synthesiser.

Since

1.0

17.43.2.2 const FileSystemTable& iMS::IMSSynthesiser::FST ( ) const

Returns the FileSystemTable for the Synthesiser.

Since

1.1

17.43.2.3 const Capabilities iMS::IMSSynthesiser::GetCap ( ) const

Returns the Capabilities structure for the Synthesiser.

Since

1.0

17.43.2.4 const FWVersion& iMS::IMSSynthesiser::GetVersion ( ) const

Returns the Firmware version for the Synthesiser.

Since

1.0

17.43.2.5 const bool iMS::IMSSynthesiser::IsValid ( ) const

Returns true if the system scan successfully identified the Synthesiser and initialised this Class.

Returns

true if the class contains valid data representing an attached iMS Synthesiser

Since

1.0

17.43.2.6 const std::string& iMS::IMSSynthesiser::Model ( ) const

Returns the short model number for the Synthesiser.

Since

1.0

The documentation for this class was generated from the following file:

- IMSSystem.h

17.44 iMS::IMSSystem Class Reference

An object representing the overall configuration of an attached iMS System and permits applications to connect to it.

```
#include <include/IMSSystem.h>
```

Public Member Functions

- IConnectionManager ∗const Connection () const

  returns a pointer to an object which is the Connection through which all messages to the hardware go
- void Ctlr (const IMSController &)

  Add an iMS Controller to the System. Intended for internal library use.
- void Synth (const IMSSynthesiser &)

  Add an iMS Synthesiser to the System. Intended for internal library use.
- const IMSController & Ctlr () const

  Retrieve data about the iMS Controller.
- const IMSSynthesiser & Synth () const

  Retrieve data about the iMS Synthesiser.
- const std::string & ConnPort () const

  Returns a descriptive string representing the connection port on which the iMS System was discovered.
- bool operator== (IMSSystem const &rhs) const

  Tests for equality between two IMSSystem's.

Connect to / Disconnect from iMS Hardware

- void Connect ()

  Attempts to establish a Connection to an iMS System.
- void Disconnect ()

  Breaks a connection to an iMS System.
- bool Open () const

  Tests Connection Status.

17.44.1 Detailed Description

An object representing the overall configuration of an attached iMS System and permits applications to connect to it.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.44.2 Member Function Documentation

17.44.2.1 void iMS::IMSSystem::Connect ( )

Attempts to establish a Connection to an iMS System.

Apart from scanning to identify attached iMS Systems (see ConnectionList::scan()), no interaction can occur with an iMS System until a connection has been established to it. This can be done by calling the Connect() function. Once established, the connection will remain open until Disconnect() is called.

Since

1.0

17.44.2.2 IConnectionManager* const iMS::IMSSystem::Connection ( ) const

returns a pointer to an object which is the Connection through which all messages to the hardware go

Warning

This function may be removed in a future release. Avoid using.

17.44.2.3 const std::string& iMS::IMSSystem::ConnPort ( ) const

Returns a descriptive string representing the connection port on which the iMS System was discovered.

Since

1.0

17.44.2.4 void iMS::IMSSystem::Ctlr (const IMSController &)

Add an iMS Controller to the System. Intended for internal library use.

Since

1.0

17.44.2.5 const IMSController& iMS::IMSSystem::Ctlr ( ) const

Retrieve data about the iMS Controller.

Returns

a const reference to the IMSSystem's Controller class

Since

1.0

17.44.2.6 void iMS::IMSSystem::Disconnect ( )

Breaks a connection to an iMS System.

Any existing connection to an iMS System can be terminated by calling the Disconenct() function. Any messages that are pending but not yet sent will be completed before closing the connection, so the application can be sure that any immediately preceding commands will be run to completion before the connection is closed.

Since

1.0

17.44.2.7 bool iMS::IMSSystem::Open ( ) const

Tests Connection Status.

If an open connection exists to the iMS System, this function will return true

Since

1.3

17.44.2.8 bool iMS::IMSSystem::operator== (IMSSystem const & *rhs*) const

Tests for equality between two IMSSystem's.

Since

1.3

17.44.2.9 void iMS::IMSSystem::Synth (const IMSSynthesiser &)

Add an iMS Synthesiser to the System. Intended for internal library use.

Since

1.0

17.44.2.10 const IMSSynthesiser& iMS::IMSSystem::Synth ( ) const

Retrieve data about the iMS Synthesiser.

Returns

a const reference to the IMSSystem's Synthesiser class

Since

1.0

The documentation for this class was generated from the following file:

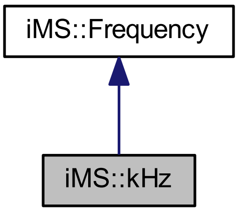
- IMSSystem.h

17.45 iMS::kHz Class Reference

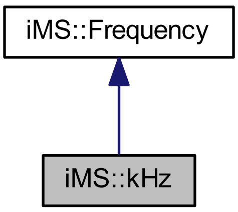
Type Definition for all operations that require a frequency specification in kiloHertz.

`#include <include/IMSTypeDefs.h>`

Inheritance diagram for iMS::kHz:



Collaboration diagram for iMS::kHz:



Public Member Functions

- kHz (double arg)

  Construct a kHz object from a double argument representing kiloHertz.
- kHz & operator= (double arg)

  Assignment of a double argument in kiloHertz to an existing Frequency object.
- operator double () const

  Return a double representing the Frequency value in kiloHertz.

Additional Inherited Members

17.45.1 Detailed Description

Type Definition for all operations that require a frequency specification in kiloHertz.

kHz inherits from Frequency, which internally stores the value in Hertz.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.45.2 Constructor & Destructor Documentation

17.45.2.1 iMS::kHz::kHz (double *arg*) [inline]

Construct a kHz object from a double argument representing kiloHertz.

Parameters

in	*arg*	Frequency in kiloHertz

Since

1.0

17.45.3 Member Function Documentation

17.45.3.1 iMS::kHz::operator double ( ) const [inline]

Return a double representing the Frequency value in kiloHertz.

```
Frequency f1(3750.0);
kHz f2 = f1();
std::cout << "f2's Frequency is: " << f2() << "kHz" << std::endl;
```

prints:

```
f2's Frequency is 3.75kHz
```

Since

1.0

17.45.3.2 kHz& iMS::kHz::operator= (double *arg*) [inline]

Assignment of a double argument in kiloHertz to an existing Frequency object.

```
kHz f;
f = 1.0;
// f contains 1000Hz
```

Since

1.3

The documentation for this class was generated from the following file:

- IMSTypeDefs.h

17.46 iMS::LibVersion Class Reference

Access the version information for the API.

`#include <include/LibVersion.h>`

Static Public Member Functions

Version Numbers

- static int GetMajor ()
  Return the major version number, e.g., 1 for "1.2.3".
- static int GetMinor ()
  Return the minor version number, e.g., 2 for "1.2.3".
- static int GetPatch ()
  Return the patch version number, e.g., 3 for "1.2.3".
- static std::string GetVersion ()
  Return the full version number.

Version Number Maths

- static bool IsAtLeast (int major, int minor, int patch)
  Compare the current version number against a specific version.

Feature Tags

- static bool HasFeature (const std::string &name)
  Test whether a feature is implemented by this API.

17.46.1 Detailed Description

Access the version information for the API.

For example, you can get the current version number as a string using `GetVersion`, or you can get the separate major, minor and patch integer values by calling `GetMajor`, `GetMinor`, or `GetPatch`, respectively.

This class also provides some basic version comparison functionality and lets you determine if certained named features are present in your current build.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.46.2 Member Function Documentation

17.46.2.1 static int iMS::LibVersion::GetMajor ( ) [static]

Return the major version number, e.g., 1 for "1.2.3".

Returns

The major version number as an integer

Since

1.0

17.46.2.2 static int iMS::LibVersion::GetMinor ( ) [static]

Return the minor version number, e.g., 2 for "1.2.3".

Returns

The minor version number as an integer

Since

1.0

17.46.2.3 static int iMS::LibVersion::GetPatch ( ) [static]

Return the patch version number, e.g., 3 for "1.2.3".

Returns

The patch version number as an integer

Since

1.0

17.46.2.4 static std::string iMS::LibVersion::GetVersion ( ) [static]

Return the full version number.

Returns

The version string, e.g., "1.2.3"

Since

1.0

17.46.2.5 static bool iMS::LibVersion::HasFeature (const std::string & *name*) [static]

Test whether a feature is implemented by this API.

New features that change the implementation of API methods are specified as "feature tags." This method lets you query the API to find out if a given feature is available.

Parameters

in	*name*	The feature tag name, e.g., "IMAGE_FILE"

Returns

Returns true if the named feature is available in this version

Since

1.0

17.46.2.6 static bool iMS::LibVersion::IsAtLeast (int *major*, int *minor*, int *patch*) [static]

Compare the current version number against a specific version.

This method lets you check to see if the current version is greater than or equal to the specified version. This may be useful to perform operations that require a minimum version number.

Parameters

in	*major*	The major version number to compare against
in	*minor*	The minor version number to compare against
in	*patch*	The patch version number to compare against

Returns

Returns true if the current API version >= (major, minor, patch)

Since

1.0

The documentation for this class was generated from the following file:

- LibVersion.h

17.47 iMS::ListBase< T > Class Template Reference

Template Class encapsulating a list object and acting as a base list class for other classes in the library to inherit from.

```
#include <include/Containers.h>
```

Public Member Functions

- bool operator== (ListBase const &rhs) const

  Equality Operator checks ListBase object for equivalence.

Constructors & Destructor

- ListBase (const std::string &Name="[no name]", const std::time_t &modified_time=std::time(nullptr))

  Create a default empty List with optional name parameter.
- ~ListBase ()

  Destructor.
- ListBase (const ListBase &)

  Copy Constructor.
- ListBase & operator= (const ListBase &)

  Assignment Constructor.

ListBase Unique Identifier

- const std::array< std::uint8_t, 16 > GetUUID () const

  Returns a vector representing the Unique Identifier assigned to the ListBase object.

Timestamping

- const std::time_t & ModifiedTime () const

  Returns Time at which the Container was last modified.
- std::string ModifiedTimeFormat () const

  Returns Human-readable string for the time at which the Container was last modified.

Container Description

- const std::string & Name () const

  A string stored with the Container to aid human users in identifying its purpose.
- std::string & **Name** ()

Modifiers

- void assign (size_t n, const T &val)

  Assign new content to ImageSequence list.
- void push_front (const T &val)

  Insert ImageSequenceEntry at beginning.
- void pop_front ()

  Delete first ImageSequenceEntry.
- void push_back (const T &val)

  Add ImageSequenceEntry at end.
- void pop_back ()

  Delete last ImageSequenceEntry.
- iterator insert (iterator position, const T &val)

  Insert ImageSequenceEntry.
- iterator insert (iterator position, const_iterator first, const_iterator last)

  Insert Range Of ImageSequenceEntry's.

- iterator erase (iterator position)
  Erase ImageSequenceEntry.
- iterator erase (iterator first, iterator last)
  Erase a range of ImageSequenceEntry's.
- void resize (size_t n)
  Change Size.
- void clear ()
  Clear Content.

Helper Functions

- bool empty () const
  Returns True if the ListBase is empty.
- std::size_t size () const
  Returns the Number of Entries in the ListBase.

Iterator Specification

Use these iterators when you want to iteratively read through or update the entries stored within a ListBase. Iterators can be used to access elements at an arbitrary offset position relative to the element they point to.

Two types of iterators are supported; both are random access iterators. Dereferencing const_iterator yields a reference to a constant entry in the ListBase(const ListBase&).

- typedef std::list< T >::iterator iterator
  Iterator defined for user manipulation of ListBase.
- typedef std::list< T >::const_iterator const_iterator
  Const Iterator defined for user readback of ListBase.
- iterator begin ()
  Returns an iterator pointing to the first element in the ListBase container.
- iterator end ()
  Returns an iterator referring to the past-the-end element in the ListBase container.
- const_iterator begin () const
  Returns a const_iterator pointing to the first element in the ListBase container.
- const_iterator end () const
  Returns a const_iterator referring to the past-the-end element in the ListBase container.
- const_iterator cbegin () const
  Returns a const_iterator pointing to the first element in the ListBase container.
- const_iterator cend () const
  Returns a const_iterator referring to the past-the-end element in the ListBase container.

17.47.1 Detailed Description

template<typename T>class iMS::ListBase< T >

Template Class encapsulating a list object and acting as a base list class for other classes in the library to inherit from.

Date

2016-11-09

Since

1.3

17.47.2 Member Function Documentation

17.47.2.1 template<typename T> void iMS::ListBase< T >::assign (size_t *n*, const T & *val*)

Assign new content to ImageSequence list.

Assigns new contents to the ImageSequence list container, replacing its current contents, and modifying its size accordingly. the new contents are `n` elements, each initialized to a copy of `val`.

Parameters

in	*n*	New size for the container.
in	*val*	ImageSequenceEntry to fill the ImageSequence with. Each of the n elements in the container will be initialized to a copy of this value.

17.47.2.2 template<typename T> iterator iMS::ListBase< T >::begin ( )

Returns an iterator pointing to the first element in the ListBase container.

Returns

An iterator to the beginning of the ListBase container.

17.47.2.3 template<typename T> const_iterator iMS::ListBase< T >::begin ( ) const

Returns a const_iterator pointing to the first element in the ListBase container.

Returns

A ListBase to the beginning of the ListBase container.

Since

1.2.5

17.47.2.4 template<typename T> const_iterator iMS::ListBase< T >::cbegin ( ) const

Returns a const_iterator pointing to the first element in the ListBase container.

Returns

A const_iterator to the beginning of the ListBase container.

17.47.2.5 template<typename T> const_iterator iMS::ListBase< T >::cend ( ) const

Returns a const_iterator referring to the past-the-end element in the ListBase container.

Returns

A const_iterator to the element past the end of the ListBase.

17.47.2.6 template<typename T> void iMS::ListBase< T >::clear ( )

Clear Content.

Removes all elements from the list container (which are destroyed), and leaving the ImageSequence with a size of 0.

17.47.2.7 template<typename T> bool iMS::ListBase< T >::empty ( ) const

Returns True if the ListBase is empty.

Returns

True if the ListBase is empty

17.47.2.8 template<typename T> iterator iMS::ListBase< T >::end ( )

Returns an iterator referring to the past-the-end element in the ListBase container.

The past-the-end element is the theoretical element that would follow the last element in the ListBase container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function can be used in combination with ListBase::begin to specify a range including all the elements in the container.

Returns

An iterator to the element past the end of the ListBase

17.47.2.9 template<typename T> const_iterator iMS::ListBase< T >::end ( ) const

Returns a const_iterator referring to the past-the-end element in the ListBase container.

Returns

A const_iterator to the element past the end of the ListBase.

Since

1.2.5

17.47.2.10 template<typename T> iterator iMS::ListBase< T >::erase (iterator *position*)

Erase ImageSequenceEntry.

Removes a single ImageSequenceEntry element (at position) from the list container

Parameters

in	*position*	Iterator pointing to a single element to be removed from the list.

Returns

An iterator pointing to the element that followed the last element erased by the function call. This is the container end if the operation erased the last element in the sequence.

17.47.2.11 template<typename T> iterator iMS::ListBase< T >::erase (iterator *first,* iterator *last*)

Erase a range of ImageSequenceEntry's.

Removes a range of ImageSequenceEntry elements (first,last) from the list container

Parameters

in	*first*	Iterators within the list to be removed.
in	*last*	Iterators within the list to be removed.

Returns

An iterator pointing to the element that followed the last element erased by the function call. This is the container end if the operation erased the last element in the sequence.

17.47.2.12 template<typename T> const std::array<std::uint8_t, 16> iMS::ListBase< T >::GetUUID ( ) const

Returns a vector representing the Unique Identifier assigned to the ListBase object.

Returns

UUID as an array of uint8_t's

17.47.2.13 template<typename T> iterator iMS::ListBase< T >::insert (iterator *position*, const T & *val*)

Insert ImageSequenceEntry.

The ImageSequence container is extended by inserting new elements before the element at the specified position. This effectively increases the ImageSequence list size by one.

Parameters

in	*position*	Position in the container where the new elements are inserted.
in	*val*	ImageSequenceEntry Value to be copied (or moved) to the inserted elements.

Returns

An iterator that points to the first of the newly inserted elements.

17.47.2.14 template<typename T> iterator iMS::ListBase< T >::insert (iterator *position*, const_iterator *first*, const_iterator *last*)

Insert Range Of ImageSequenceEntry's.

The ImageSequence container is extended by inserting new elements before the element at the specified position from a range of ImageSequenceEntries present in another ImageSequence. This effectively increases the ImageSequence list size by the number of entries in the range

Parameters

in	*position*	Position in the container where the new elements are inserted.
in	*first*	Iterator specifying the first of a range of elements.
in	*last*	Iterator specifying the last of a range of elements. All the elements between first and last, including the element pointed by first but not the one pointed by last are inserted to the ImageSequence before position.

Returns

An iterator that points to the first of the newly inserted elements.

17.47.2.15 template<typename T> const std::time_t& iMS::ListBase< T >::ModifiedTime ( ) const

Returns Time at which the Container was last modified.

Any time the container is modified (added to, deleted from, elements updated), the system time is recorded. This happens coincident with the UUID if the container also being updated. This function returns to the user that timestamp.

Returns

a reference to a std::time_t representing the time at which the container was last modified

Since

1.3

17.47.2.16 template<typename T> std::string iMS::ListBase< T >::ModifiedTimeFormat ( ) const

Returns Human-readable string for the time at which the Container was last modified.

Since

1.3

17.47.2.17 template<typename T> const std::string& iMS::ListBase< T >::Name ( ) const

A string stored with the Container to aid human users in identifying its purpose.

Updating the Container Name does not cause the Container UUID to change.

17.47.2.18 template<typename T> bool iMS::ListBase< T >::operator== (ListBase< T > const & *rhs*) const

Equality Operator checks ListBase object for equivalence.

Parameters

in	*rhs*	An ListBase object to perform the comparison with

Returns

True if the supplied ListBase is identical to this one.

17.47.2.19 template<typename T> void iMS::ListBase< T >::pop_back ( )

Delete last ImageSequenceEntry.

Removes the last ImageSequenceEntry in the ImageSequence list container, effectively reducing its size by one. This destroys the removed entry.

17.47.2.20 template<typename T> void iMS::ListBase< T >::pop_front ( )

Delete first ImageSequenceEntry.

Removes the first ImageSequenceEntry in the ImageSequence list container, effectively reducing its size by one. This destroys the removed entry.

17.47.2.21 template<typename T> void iMS::ListBase< T >::push_back (const T & *val*)

Add ImageSequenceEntry at end.

Adds a new ImageSequenceEntry at the end of the ImageSequence list container, after its current last element. The content of val is copied (or moved) to the new element. This effectively increases the Sequence size by one.

Parameters

in	*val*	ImageSequenceEntry to be copied (or moved) to the new element.

17.47.2.22 template<typename T> void iMS::ListBase< T >::push_front (const T & *val*)

Insert ImageSequenceEntry at beginning.

Inserts a new element at the beginning of the list, right before its current first element. The content of val is copied (or moved) to the inserted element. This effectively increases the Sequence size by one.

Parameters

in	*val*	ImageSequenceEntry to be copied (or moved) to the inserted element.

17.47.2.23 template<typename T> void iMS::ListBase< T >::resize (size_t *n*)

Change Size.

Resizes the ImageSequence container so that it contains n elements. If n is smaller than the current container size, the content is reduced to its first n elements, removing those beyond (and destroying them). If n is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of n. The new ImageSequenceEntry 's are default-initialized. Notice that this function changes the actual content of the container by inserting or erasing elements from it.

Parameters

in	*n*	New container size, expressed in number of elements.

17.47.2.24 template<typename T> std::size_t iMS::ListBase< T >::size ( ) const

Returns the Number of Entries in the ListBase.

Returns

std::size_t representing the number of elements in the ListBase

The documentation for this class was generated from the following file:

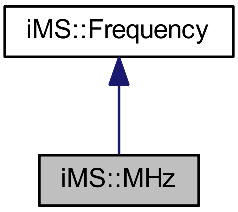
- Containers.h

17.48 iMS::MHz Class Reference

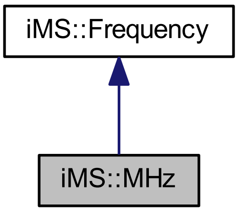
Type Definition for all operations that require a frequency specification in MegaHertz.

`#include <include/IMSTypeDefs.h>`

Inheritance diagram for iMS::MHz:



Collaboration diagram for iMS::MHz:



Public Member Functions

- MHz (double arg)

    Construct a MHz object from a double argument representing MegaHertz.
- MHz & operator= (double arg)

    Assignment of a double argument in MegaHertz to an existing Frequency object.
- operator double () const

    Return a double representing the Frequency value in MegaHertz.

Static Public Member Functions

- static unsigned int RenderAsImagePoint (const IMSSystem &, const MHz)

    Used internally by the library to convert a Frequency object into a hardware-dependent integer representation used by the Image for RF Output frequency.

17.48.1 Detailed Description

Type Definition for all operations that require a frequency specification in MegaHertz.

MHz inherits from Frequency, which internally stores the value in Hertz.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.48.2 Constructor & Destructor Documentation

17.48.2.1 iMS::MHz::MHz (double *arg*) [inline]

Construct a MHz object from a double argument representing MegaHertz.

Parameters

in	*arg*	Frequency in MegaHertz

Since

1.0

17.48.3 Member Function Documentation

17.48.3.1 iMS::MHz::operator double ( ) const [inline]

Return a double representing the Frequency value in MegaHertz.

```
Frequency f1(1234567.0);
MHz f2 = f1();
std::cout << "f2's Frequency is: " << f2() << "MHz" << std::endl;
```

prints:

```
f2's Frequency is 1.234567MHz
```

Since

1.0

17.48.3.2 MHz& iMS::MHz::operator= (double *arg*) [inline]

Assignment of a double argument in MegaHertz to an existing Frequency object.

```
MHz f;
f = 1.0;
// f contains 1,000,000Hz
```

Since

1.3

17.48.3.3 static unsigned int iMS::MHz::RenderAsImagePoint (const IMSSystem & , const MHz) [static]

Used internally by the library to convert a Frequency object into a hardware-dependent integer representation used by the Image for RF Output frequency.

Not intended for use in application code

The documentation for this class was generated from the following file:

- IMSTypeDefs.h

17.49 iMS::Percent Class Reference

Type Definition for all operations that require a percentage specification.

`#include <include/IMSTypeDefs.h>`

Public Member Functions

- Percent ()
  Default Constructor assigns 0.0%.
- Percent (double arg)
  Construct a Percent object from a double argument and check its value is within the range 0.0 <= arg <= 100.0. If not, the object is still constructed, but the value is clipped to the upper or lower bound.
- Percent & operator= (double arg)
  Assignment of a double argument in percent to an existing Percent object.
- operator double () const
  Return a double representing the Percent object's value.

Static Public Member Functions

- static unsigned int RenderAsImagePoint (const IMSSystem &, const Percent)
  Used internally by the library to convert a Percent object into a hardware-dependent integer representation used by the Image for RF Output amplitude.
- static unsigned int RenderAsCompensationPoint (const IMSSystem &, const Percent)
  Used internally by the library to convert a Percent object into a hardware-dependent integer representation used by the Compensation Table for Compensation amplitude.
- static unsigned int RenderAsCalibrationTone (const IMSSystem &, const Percent)
  Used internally by the library to convert a Percent object into a hardware-dependent integer representation used by the Calibration Tone for Single Tone amplitude.

17.49.1 Detailed Description

Type Definition for all operations that require a percentage specification.

Internally, the Percent value is stored as a double precision variable and is bounds-limited to 0.0 <= Percent <= 100.0.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.49.2 Constructor & Destructor Documentation

17.49.2.1 iMS::Percent::Percent ( ) [inline]

Default Constructor assigns 0.0%.

Since

1.1

17.49.2.2 iMS::Percent::Percent (double *arg*) [inline]

Construct a Percent object from a double argument and check its value is within the range 0.0 <= arg <= 100.0. If not, the object is still constructed, but the value is clipped to the upper or lower bound.

Parameters

in	*arg*	The percentage value

Since

1.0

17.49.3 Member Function Documentation

17.49.3.1 iMS::Percent::operator double ( ) const [inline]

Return a double representing the Percent object's value.

Since

1.0

17.49.3.2 Percent& iMS::Percent::operator= (double *arg*) [inline]

Assignment of a double argument in percent to an existing Percent object.

The double argument of the assigner must be within the range 0.0 <= arg <= 100.0 else it will be limited to those bounds.

```
// In a group of 7 children, 3 of them have dark hair
Percent ChildrenWithDarkHair = (3.0 / 7.0) * 100.0;
std::cout << ChildrenWithDarkHair << "% of the group have dark hair" << std::endl;
```

prints:

```
42.8571% of the group have dark hair
```

Since

1.0

17.49.3.3 static unsigned int iMS::Percent::RenderAsCalibrationTone (const IMSSystem & , const Percent) [static]

Used internally by the library to convert a Percent object into a hardware-dependent integer representation used by the Calibration Tone for Single Tone amplitude.

Not intended for use in application code

Since

1.1.0

17.49.3.4 static unsigned int iMS::Percent::RenderAsCompensationPoint (const IMSSystem & , const Percent) [static]

Used internally by the library to convert a Percent object into a hardware-dependent integer representation used by the Compensation Table for Compensation amplitude.

Not intended for use in application code

17.49.3.5 static unsigned int iMS::Percent::RenderAsImagePoint (const IMSSystem & , const Percent) [static]

Used internally by the library to convert a Percent object into a hardware-dependent integer representation used by the Image for RF Output amplitude.

Not intended for use in application code

The documentation for this class was generated from the following file:

- IMSTypeDefs.h

17.50 iMS::ImagePlayer::PlayConfiguration Struct Reference

This struct sets the attributes for the ImagePlayer to use when initiating an Image Playback.

`#include <include\ImageOps.h>`

Public Types

- using post_delay = std::chrono::duration< std::uint16_t, std::ratio< 1, 10000 > >

  This type is used internally to define the correct scaling between std::chrono classes and the hardware delay counter. Min Resolution is 0.1msec.

Public Member Functions

Constructors

- PlayConfiguration ()

  Empty Constructor. All attributes take on their default values.
- PlayConfiguration (PointClock c)

  Constructor with Clock Initialisation. Use this to set the Clock to be supplied from an External signal.
- PlayConfiguration (PointClock c, ImageTrigger t)

  Constructor with Clock & Trigger Initialisation. Use this to set the Clock, Trigger or both to be supplied from External signals.
- PlayConfiguration (PointClock c, std::chrono::duration< int > d)

Constructor with Clock Initialisation and Post-Delay. Use this for a configurable delay between images.

- PlayConfiguration (PointClock c, std::chrono::duration< int > d, Repeats r, int n_rpts)

  Constructor with Clock Initialisation, Post-Delay and Image Repeats. Use this to configure the Clock source, Delay between Image repeats and the number of Repeats per Image.
- PlayConfiguration (Repeats r)

  Constructor with Indefinite Repeats. Use this to set the Image to Repeat Always until Stopped by User Command.
- PlayConfiguration (Repeats r, int n_rpts)

  Constructor with Programmable Repeats. Use this to set the Image to Repeat a programmable number of times.

Public Attributes

- PointClock int_ext { PointClock::INTERNAL }

  Use Internal NCO or External Clock signal.
- ImageTrigger trig { ImageTrigger::CONTINUOUS }

  Trigger Next Image Immediately, after programmable delay, External Trigger signal or software Trigger.
- Repeats rpts { Repeats::NONE }

  Run Image Once, Always until stopped, or a Programmable number of times.
- int n_rpts { 0 }

  If Repeats set to Repeats::PROGRAM, this field sets the number of repeats to trigger (not including first pass, i.e. n_rpts = 3 => 4 playbacks in total)
- Polarity clk_pol { Polarity::NORMAL }

  Sets the active edge of the External Clock signal (Polarity::NORMAL = rising edge)
- Polarity trig_pol { Polarity::NORMAL }

  Sets the active edge of the External Trigger signal (Polarity::NORMAL = rising edge)
- post_delay del { 0 }

  When ImageTrigger is set to ImageTrigger::POST_DELAY, this field defines the length of time between the end of one image (or repeat) and the start of the next. Use SetPostDelay(std::chrono::milliseconds(...)) or an associated std::chrono class.

17.50.1 Detailed Description

This struct sets the attributes for the ImagePlayer to use when initiating an Image Playback.

Author

Dave Cowan

Date

2015-11-11

Since

1.0

The documentation for this struct was generated from the following file:

- ImageOps.h

17.51 iMS::RFChannel Class Reference

Type that represents the integer values 1, 2, 3 and 4, one each for the RF Channels of an iMS Synthesiser.

```
#include <include/IMSTypeDefs.h>
```

Public Member Functions

- RFChannel ()

  Default construct an RF Channel object initialised to the first RF Channel.
- RFChannel (int arg)

  Construct an RF Channel object and check that it is being created with an integer value within the range 1 <= arg <= 4. If not, the object is still constructed, but the RF Channel value is set to 1 and an invalid_argument exception is thrown.
- RFChannel & operator= (int arg)

  Assignment of an integer argument to an existing RF Channel object.
- operator int () const

  Return an integer representing the RF Channel that the object references.

- RFChannel & operator++ ()

  Prefix and Postfix operators for (dec)incrementing through channels.
- RFChannel **operator++** (int)
- RFChannel & **operator--** ()
- RFChannel **operator--** (int)

17.51.1 Detailed Description

Type that represents the integer values 1, 2, 3 and 4, one each for the RF Channels of an iMS Synthesiser.

The type is used to ensure that incorrect channel specifications cannot be passed to functions requiring an argument referencing an RF output channel. Attempting to use an integer outside the range 1 <= arg <= 4 will result in RFChannel = 1 and an invalid_argument exception being thrown.

Exceptions

std::invalid_argument("Invalid	RF Channel Number") Attempted to use an integer specification not tied to an RF Output Channel

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.51.2 Constructor & Destructor Documentation

17.51.2.1 iMS::RFChannel::RFChannel ( ) [inline]

Default construct an RF Channel object initialised to the first RF Channel.

Since

1.1

17.51.2.2 iMS::RFChannel::RFChannel (int *arg*) [inline]

Construct an RF Channel object and check that it is being created with an integer value within the range 1 <= arg <= 4. If not, the object is still constructed, but the RF Channel value is set to 1 and an invalid_argument exception is thrown.

Parameters

in	*arg*	The channel specification

Exceptions

std::invalid_argument("↩ Invalid	RF Channel Number") Attempted to use an integer specification not tied to an RF Output Channel

Since

1.0

17.51.3 Member Function Documentation

17.51.3.1 iMS::RFChannel::operator int ( ) const [inline]

Return an integer representing the RF Channel that the object references.

Since

1.0

17.51.3.2 RFChannel& iMS::RFChannel::operator++ ( ) [inline]

Prefix and Postfix operators for (dec)incrementing through channels.

Since

1.1

17.51.3.3 RFChannel& iMS::RFChannel::operator= (int *arg*) [inline]

Assignment of an integer argument to an existing RF Channel object.

Checks that it is being created with an integer value within the range 1 <= arg <= 4. If not, the object is still constructed, but the RF Channel value is set to 1 and an invalid_argument exception is thrown

Parameters

in	*arg*	The channel specification

Exceptions

std::invalid_argument("↩ Invalid	RF Channel Number") Attempted to use an integer specification not tied to an RF Output Channel

Since

1.0

The documentation for this class was generated from the following file:

- IMSTypeDefs.h

17.52 iMS::SequenceManager::SeqConfiguration Struct Reference

This struct sets the attributes for the Sequence to use when initiating an Sequence Playback.

`#include <include\ImageOps.h>`

Public Member Functions

Constructors

- SeqConfiguration ()
  Empty Constructor. All attributes take on their default values.
- SeqConfiguration (PointClock c)
  Constructor with Clock Initialisation. Use this to set the Clock to be supplied from an External signal.
- SeqConfiguration (PointClock c, ImageTrigger t)
  Constructor with Clock & Trigger Initialisation. Use this to set the Clock, Trigger or both to be supplied from External signals.

Public Attributes

- PointClock int_ext {PointClock::INTERNAL }
  Use Internal NCO or External Clock signal.
- ImageTrigger trig { ImageTrigger::CONTINUOUS }
  Trigger Next Image Immediately, after programmable delay, External Trigger signal or software Trigger.
- Polarity clk_pol { Polarity::NORMAL }
  Sets the active edge of the External Clock signal (Polarity::NORMAL = rising edge)
- Polarity trig_pol { Polarity::NORMAL }
  Sets the active edge of the External Trigger signal (Polarity::NORMAL = rising edge)

17.52.1 Detailed Description

This struct sets the attributes for the Sequence to use when initiating an Sequence Playback.

Author

Dave Cowan

Date

2016-05-05

Since

1.2.4

The documentation for this struct was generated from the following file:

- ImageOps.h

17.53 iMS::SequenceDownload Class Reference

This class is a worker for transmitting an ImageSequence to an iMS Controller and joining it to the back of the sequence queue.

`#include <include\ImageOps.h>`

Public Member Functions

Constructor & Destructor

- SequenceDownload (IMSSystem &ims, const ImageSequence &seq)
  Constructor for SequenceDownload Object.
- ∼SequenceDownload ()
  Destructor.

Download Trigger

- bool Download ()
  Adds a new sequence to the end of the iMS Controller Sequence Queue.

17.53.1 Detailed Description

This class is a worker for transmitting an ImageSequence to an iMS Controller and joining it to the back of the sequence queue.

Author

Dave Cowan

Date

2016-05-05

Since

1.2.4

17.53.2 Constructor & Destructor Documentation

17.53.2.1 iMS::SequenceDownload::SequenceDownload (IMSSystem & *ims*, const ImageSequence & *seq*)

Constructor for SequenceDownload Object.

The pre-requisites for an SequenceDownload object to be created are: (1) - an IMSSystem object, representing the iMS target to which the ImageSequence is to be downloaded. (2) - a complete ImageSequence object to download to the iMS target.

SequenceDownload stores references to both. This means that both must exist before the SequenceDownload object, and both must remain valid (not destroyed) until the SequenceDownload object itself is destroyed. Because they are stored as references, the IMSSystem and Image objects themselves may be modified after the construction of the SequenceDownload object.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A reference to the iMS System which is the target for downloading the Image↩ Sequence
in	*seq*	A const reference to the ImageSequence which shall be downloaded to the target

Since

1.2.4

17.53.3 Member Function Documentation

17.53.3.1 bool iMS::SequenceDownload::Download ( )

Adds a new sequence to the end of the iMS Controller Sequence Queue.

Calling this function will program the list of ImageSequenceEntry's and the termination action/value from the ImageSequence object reference into a new sequence added to the end of the Sequence Queue.

Returns

True to indicate Sequence has been successfully added to the queue

The documentation for this class was generated from the following file:

- ImageOps.h

17.54 iMS::SequenceEvents Class Reference

All the different types of events that can be triggered by the SequenceManager class.

```
#include <include\ImageOps.h>
```

Public Types

- enum Events { SEQUENCE_START, SEQUENCE_FINISHED, SEQUENCE_ERROR, **Count** }

  List of Events raised by the Image Downloader.

17.54.1 Detailed Description

All the different types of events that can be triggered by the SequenceManager class.

Some events contain integer parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2016-05-04

Since

1.2.4

17.54.2 Member Enumeration Documentation

17.54.2.1 enum iMS::SequenceEvents::Events

List of Events raised by the Image Downloader.

Enumerator

SEQUENCE_START Event raised at the beginning of playback of each sequence.

SEQUENCE_FINISHED Event raised after the final image of a sequence has completed and there are no more sequences in the queue, or the sequence was programmed to stop.

SEQUENCE_ERROR Event raised when an error occurs in processing the sequence queue (typically if the sequence queue was cleared during playback)

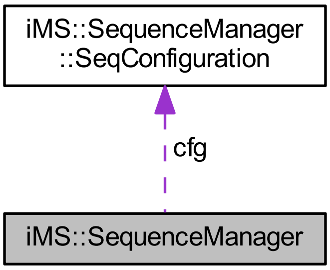
The documentation for this class was generated from the following file:

- ImageOps.h

17.55 iMS::SequenceManager Class Reference

`#include <include\ImageOps.h>`

Collaboration diagram for iMS::SequenceManager:



Classes

- struct SeqConfiguration

  This struct sets the attributes for the Sequence to use when initiating an Sequence Playback.

Public Types

- using PointClock = ImagePlayer::PointClock

  Defines Internal Oscillator / External Clock operation.
- using ImageTrigger = ImagePlayer::ImageTrigger

  Defines Image Trigger function.
- using Polarity = ImagePlayer::Polarity

  Defines polarity of external clock / trigger signals.

Public Member Functions

Constructors & Destructor

- SequenceManager (const IMSSystem &)

  Default Constructor.
- ∼SequenceManager ()

Destructor.

Playback Operations

- bool StartSequenceQueue (const SeqConfiguration &cfg=SeqConfiguration(), ImageTrigger start_↩ trig=ImageTrigger::CONTINUOUS)

    Begins playback through the sequence queue.
- void SendHostTrigger ()

    Software trigger for sequence Image propagation When either SeqConfiguration::trig or start_trig are set to ImageTrigger::HOST, the application software must send a signal to the hardware to begin playback of either the next Image in the sequence, or the first image in the sequence respectively.

Queue Modification

- std::uint16_t QueueCount ()

    Number of Sequences programmed into the Queue.
- bool GetSequenceUUID (int index, std::array< std::uint8_t, 16 > &uuid)

    Returns the identity of a particular sequence via its index in the Controller sequence queue.
- bool QueueClear ()

    Remove all sequences from the queue.
- bool RemoveSequence (const ImageSequence &seq)

    Remove an individual sequence from the queue.
- bool RemoveSequence (const std::array< std::uint8_t, 16 > &uuid)

    Remove an individual sequence from the queue.
- bool UpdateTermination (ImageSequence &seq, SequenceTermAction action, int val=0)

    Update the termination behaviour of a specific sequence.
- bool UpdateTermination (const std::array< std::uint8_t, 16 > &uuid, SequenceTermAction action, int val=0)

    Update the termination behaviour of a specific sequence.

Public Attributes

- struct LIBSPEC iMS::SequenceManager::SeqConfiguration cfg

    Defines the configuration for Sequence Playback.

Sequence Event Signalling

If the user application requires that new sequences are created and downloaded to the Sequence Queue while playback is operational, it may be advantageous to configure the SequenceManager to inform the application when sequences complete or when the sequence queue has emptied. To do that, create an EventHandler derived class in your code and subscribe it to the SequenceManager using these functions and one of the SequenceEvent messages. The handler will be called at the relevant time which will allow the application to synchronously update the Controller sequence queue with new information.

Warning

Subscribing to sequence events turns on the interrupt sending mechanism in the Controller in order to guarantee minimum latency from the event occurrence. Since this involves sending Controller initiated messages to the SDK, it is inadvisable to subscribe to the SEQUENCE_START event if sequences are expected to be started at a rate greater than approx once every 10msec, as buffer overruns can occur which will lead to the breakdown of communications between the SDK and the iMS System.

- void SequenceEventSubscribe (const int message, IEventHandler ∗handler)

    Subscribe a callback function handler to a given SequenceManager event.
- void SequenceEventUnsubscribe (const int message, const IEventHandler ∗handler)

    Unsubscribe a callback function handler from a given SequenceManager event.

17.55.1 Detailed Description

Author

Dave Cowan

Date

2016-05-04

Since

1.2.4

17.55.2 Constructor & Destructor Documentation

17.55.2.1 iMS::SequenceManager::SequenceManager (const IMSSystem &)

Default Constructor.

Requires a reference to an iMS System in order to carry out communications with the Sequence Queue in the Controller

17.55.3 Member Function Documentation

17.55.3.1 bool iMS::SequenceManager::GetSequenceUUID (int *index*, std::array< std::uint8_t, 16 > & *uuid*)

Returns the identity of a particular sequence via its index in the Controller sequence queue.

Every ImageSequence has a unique ID, which is downloaded to the Controller sequence queue so that, although it is not possible to readback the configuration of every sequence in the queue, it is possible to match the UUID of every sequence with the UUID of an ImageSequence object stored in the application.

Parameters

`in`	*index*	The offset from the front of the queue from which to retrieve the UUID (0 = Sequence currently playing or next to play if stopped)
`in`	*uuid*	A reference to a 16-byte array in which to store the UUID

Returns

True if the operation completed successfully

17.55.3.2 bool iMS::SequenceManager::QueueClear ( )

Remove all sequences from the queue.

If the sequence queue is currently playing, this function call will fail and return false

Returns

True if the queue was successfully cleared

17.55.3.3 std::uint16_t iMS::SequenceManager::QueueCount ( )

Number of Sequences programmed into the Queue.

Returns

the number of sequences that are currently in the Controller queue

17.55.3.4 bool iMS::SequenceManager::RemoveSequence (const ImageSequence & *seq*)

Remove an individual sequence from the queue.

Removes an ImageSequence from anywhere within the sequence queue. If attempting to remove the sequence from the front of the queue, while it is playing, the operation will still succeed but subsequent behaviour may be undefined. If multiple identical sequences exist in the queue, the sequence most recently added (or most recently played) will be deleted. The function must be called multiple times to remove multiple sequences

Parameters

in	*seq*	A reference to an ImageSequence object to find in the queue and remove

Returns

True if the removal was carried out successfully

17.55.3.5 bool iMS::SequenceManager::RemoveSequence (const std::array< std::uint8_t, 16 > & *uuid*)

Remove an individual sequence from the queue.

Removes an ImageSequence from anywhere within the sequence queue. If attempting to remove the sequence from the front of the queue, while it is playing, the operation will still succeed but subsequent behaviour may be undefined. If multiple identical sequences exist in the queue, the sequence most recently added (or most recently played) will be deleted. The function must be called multiple times to remove multiple sequences

Parameters

in	*uuid*	An identifier that can be returned from GetSequenceUUID to mark a sequence for deletion

Returns

True if the removal was carried out successfully

17.55.3.6 void iMS::SequenceManager::SequenceEventSubscribe (const int *message*, IEventHandler ∗ *handler*)

Subscribe a callback function handler to a given SequenceManager event.

SequenceManager can callback user application code when an event occurs in the sequence playback process. Supported events are listed under SequenceEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to an SequenceManager event. For the period that a callback is subscribed, each time an event in the Controller sequence playback occurs that would trigger the subscribed SequenceManager event, the user function callback will be executed.

Parameters

in	*message*	Use the SequenceEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

17.55.3.7 void iMS::SequenceManager::SequenceEventUnsubscribe (const int *message*, const IEventHandler ∗ *handler*)

Unsubscribe a callback function handler from a given SequenceManager event.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the Controller sequence playback following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the SequenceEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

17.55.3.8 bool iMS::SequenceManager::StartSequenceQueue (const SeqConfiguration & *cfg* = SeqConfiguration (), ImageTrigger *start_trig* = ImageTrigger::CONTINUOUS)

Begins playback through the sequence queue.

The iMS Controller will start playing the ImageSequenceEntry that exists at the front of the Sequence Queue. If the queue is empty, the call will have no effect, but may still return true. Use the QueueCount() function if it is necessary to check for queue contents prior to playback. Image playback will continue through each ImageSequenceEntry and ImageSequence in turn until it either encounters an ImageSequence with a 'STOP_∗' termination action or the queue becomes empty.

The queue behaviour may be controlled by the SeqConfiguration struct, which specifies whether the Sequence is clocked by the internal NCO oscillator or an external clock and what method is used to propagate the start of the next ImageSequenceEntry in the list.

The start_trig parameter may be used to control how the Sequence playback begins. If set to CONTINUOUS (or not specified) Sequence playback will start immediately. If set to EXTERNAL, Sequence playback will start when an External trigger is detected. If set to HOST, Sequence playback will begin when a software trigger is sent (Using SendHostTrigger()).

Parameters

in	*cfg*	The clocking and trigger configuration for propagating through images in the sequence
in	*start_trig*	the type of trigger used to begin the sequence playing back

Returns

true whe the Sequence playback has been initiated

17.55.3.9 bool iMS::SequenceManager::UpdateTermination (ImageSequence & *seq*, SequenceTermAction *action*, int *val* = 0)

Update the termination behaviour of a specific sequence.

This function is used to change the behaviour of a sequence after it has completed playback. It can be used for example to modify a sequence set to RECYCLE mode so that next time it plays back, it is instead DISCARDed. If multiple identical sequences exist in the queue, the sequence most recently added (or most recently played) will be updated.

Parameters

in	*seq*	A reference to an ImageSequence object to find in the queue and update

in	*action*	The new action value to program the ImageSequence with
in	*val*	An optional Termination value to apply to the sequence

Returns

True if the update was carried out successfully

17.55.3.10 bool iMS::SequenceManager::UpdateTermination (const std::array< std::uint8_t, 16 > & *uuid*, SequenceTermAction *action*, int *val* = 0)

Update the termination behaviour of a specific sequence.

This function is used to change the behaviour of a sequence after it has completed playback. It can be used for example to modify a sequence set to RECYCLE mode so that next time it plays back, it is instead DISCARDed. If multiple identical sequences exist in the queue, the sequence most recently added (or most recently played) will be updated.

Parameters

in	*uuid*	An identifier that can be returned from GetSequenceUUID to mark a sequence for update
in	*action*	The new action value to program the ImageSequence with
in	*val*	An optional Termination value to apply to the sequence

Returns

True if the update was carried out successfully

The documentation for this class was generated from the following file:

- ImageOps.h

17.56 iMS::SignalPath Class Reference

Controls Signal routing and other parameters related to the RF output signals.

```
#include <include\SignalPath.h>
```

Public Types

- enum AmplitudeControl { AmplitudeControl::OFF, AmplitudeControl::EXTERNAL, AmplitudeControl::WIPER_1, AmplitudeControl::WIPER_2 }

  Selects Amplitude Control source for each of the 4 RF Channel outputs.
- enum ToneBufferControl { ToneBufferControl::HOST, ToneBufferControl::EXTERNAL, ToneBufferControl::EXTERNAL_EXTENDED, ToneBufferControl::OFF }

  Selects Control Source for the Local Tone Buffer.
- enum Compensation : bool { Compensation::ACTIVE = true, Compensation::BYPASS = false }

  Controls whether to use the Compensation Look-Up Table path for pixel data.
- enum SYNC_SRC {
  FREQUENCY_CH1, **FREQUENCY_CH2**, **FREQUENCY_CH3**, **FREQUENCY_CH4**,
  AMPLITUDE_CH1, **AMPLITUDE_CH2**, **AMPLITUDE_CH3**, **AMPLITUDE_CH4**,
  AMPLITUDE_PRE_COMP_CH1, **AMPLITUDE_PRE_COMP_CH2**, **AMPLITUDE_PRE_COMP_CH3**, **AMPLITUDE_PRE_COMP_CH4**,
  PHASE_CH1, **PHASE_CH2**, **PHASE_CH3**, **PHASE_CH4**,
  LOOKUP_FIELD_CH1, **LOOKUP_FIELD_CH2**, **LOOKUP_FIELD_CH3**, **LOOKUP_FIELD_CH4**,
  IMAGE_ANLG_A, **IMAGE_ANLG_B**, **IMAGE_DIG** }

Selects a source of Synchronous Output Data.
- enum SYNC_SINK { **ANLG_A**, **ANLG_B**, **DIG** }

  The Synchronous Output to which to assign Synchronous Data.
- enum ENCODER_MODE { **QUADRATURE**, **COUNT_DIRECTION** }

  Selects the type of encoder connected to the Synthesiser.
- enum VELOCITY_MODE { **FAST**, **SLOW** }

  Selects the method of velocity calculation.
- enum ENCODER_CHANNEL { **CH_X**, **CH_Y** }

  Selects which of two available encoder channels.

Public Member Functions

Constructor & Destructor

- SignalPath (const IMSSystem &ims)

  Constructor for SignalPath Object.
- ∼SignalPath ()

  Destructor for SignalPath Object.

RF Output Control

- bool UpdateDDSPowerLevel (const Percent &power)

  Scales the DDS device (Direct Digital Synthesis RF signal generator) power up & down.
- bool UpdateRFAmplitude (const AmplitudeControl src, const Percent &ampl)

  Scales the Digital Potentiometer mixer drive level up & down.
- bool SwitchRFAmplitudeControlSource (const AmplitudeControl src)

  Selects the amplitude control source for all 4 RF channels.
- bool UpdatePhaseTuning (const RFChannel &channel, const Degrees &phase)

  Applies a constant Phase offset to one of the 4 RF Channels.
- bool SetChannelReversal (bool reversal)

  Reverses the channel order of the 4 RF Outputs.
- bool EnableImagePathCompensation (SignalPath::Compensation amplComp, SignalPath::Compensation phaseComp)

  Enables / Disables the programmed amplitude and phase Compensation Functions for Image Playback.
- bool EnableXYPhaseCompensation (bool XYCompEnable)

  Configures Beam Steering Phase Compensation for X/Y Deflector Mode.

Calibration Functions

- bool SetCalibrationTone (const FAP &fap)

  Bypasses Controller Data and Compensation Tables and plays a fixed tone for calibration purposes.
- bool ClearTone ()

  Stops the tone playback and restores the signal path configuration to the Controller and Compensation Tables.

Synchronous Output Control

- bool AssignSynchronousOutput (const SYNC_SINK &sink, const SYNC_SRC &src) const

  Selects the source of data for the 2 Analog and 12 Digital output signals that operate synchronously with the Image Pixel Clock.
- bool ConfigureSyncDigitalOutput (::std::chrono::nanoseconds delay=::std::chrono::nanoseconds↩ ::zero(),::std::chrono::nanoseconds pulse_length=::std::chrono::nanoseconds::zero())

  Configures the Synchronous Digital Output data.

Local Tone Buffer Functions

- bool UpdateLocalToneBuffer (const ToneBufferControl &tbc, const unsigned int index, const SignalPath::Compensation AmplitudeComp=SignalPath::Compensation::ACTIVE, const SignalPath::Compensation PhaseComp=SignalPath::Compensation::ACTIVE)

  Use these functions to output tones from the Local Tone Buffer, control their selection and compensation.
- bool UpdateLocalToneBuffer (const ToneBufferControl &tbc)
- bool UpdateLocalToneBuffer (const SignalPath::Compensation AmplitudeComp, const SignalPath::Compensation PhaseComp)
- bool UpdateLocalToneBuffer (const unsigned int index)

Velocity / Encoder Compensation Functions

Some iMS Synthesisers include dual optical encoder inputs and built in tracking filters that can be used to monitor the velocity of a moving object in two dimensions, compensate the RF frequency by a scaled amount to alter the AOD deflection angle and hence remove distortion from the target feature.

Each of the 2 encoder inputs has a pair of RS422 receivers and can be configured to work with both quadrature (for best precision) and clock + direction style encoder signals. The encoder inputs are passed through a glitch filter to remove any excursions $<$ 30ns before being decoded to extract a pulse train and to identify direction of travel.

This information is fed into a tracking loop filter that both attenuates noise from the signal and calculates an estimate for the encoder velocity (in encoder ticks per second). The filter has a number of parameters that can be adjusted for optimum performance. The transfer function of the filter is:

H(s) = ((kp / I.ki).s + 1) / ((1 / I.ki).s^2 + (kp / I.ki).s + 1)

where:

- *kp = the proportion gain coefficient*
- *ki = the integral gain coefficient*
- *I = a constant correction factor = 65535 / 687 = 95.393*
- *s = the Laplace operator*

The resulting X and Y velocity estimates are applied to the pixel subsystem where they are scaled by a gain coefficient and used to offset the RF channel output frequency from the value requested by Image data, Single Tone or Tone Buffer. The offset is applied as follows:

- *(1) If X/Y Phase compensation is enabled (see EnableXYPhaseCompensation), offsets from Encoder input X are applied to RF Channels 1 and 2, offsets from Encoder input Y are applied to RF Channels 3 and 4.*
- *(2) If X/Y Phase compensation is not enabled, offsets from Encoder input X are applied to all RF Channels and Encoder input Y is ignored.*

Note that negative gains are allowed which result in frequency offsets in the opposite direction.

- bool UpdateEncoder (const VelocityConfiguration &velcomp)

  UpdateEncoder enables the Encoder velocity offset correction and updates the parameters.
- bool DisableEncoder ()

  Turns off the Velocity Compensation process.
- bool ReportEncoderVelocity (ENCODER_CHANNEL chan)

  Retrieves the current angular velocity of the requested encoder channel.

Event Notifications

- void SignalPathEventSubscribe (const int message, IEventHandler *handler)

  Subscribe a callback function handler to a given SignalPathEvents event.
- void SignalPathEventUnsubscribe (const int message, const IEventHandler *handler)

  Unsubscribe a callback function handler from a given SignalPathEvents event.

17.56.1 Detailed Description

Controls Signal routing and other parameters related to the RF output signals.

The iMS Signal Path consists of Frequency, Amplitude, Phase and Synchronous Output data driven by the Controller, or generated internally by the Synthesiser, passing through the Compensation Tables, and driven by the DDS device to result in 4 RF signal outputs along with analogue and digital synchronous outputs.

This class provides functions that control the routing options of that data and functions that control the attributes of the signals within the signal path.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.56.2 Member Enumeration Documentation

17.56.2.1 enum **iMS::SignalPath::AmplitudeControl** [strong]

Selects Amplitude Control source for each of the 4 RF Channel outputs.

The RF signal outputs from the Synthesiser feature channel bandwidth filtering and an RF mixer with a selectable control source. The mixer input can be routed to one of two digital potentiometers on the Synthesiser, which act as amplitude controls, turned off, or to an external signal input (scalable range from 0 - 15V) for signal modulation.

Enumerator

OFF Turn RF outputs off.

EXTERNAL Route RF mixer inputs to analogue modulation external signals.

WIPER_1 Connect RF mixer inputs to digital pot wiper 1.

WIPER_2 Connect RF mixer inputs to digital pot wiper 2.

17.56.2.2 enum **iMS::SignalPath::Compensation** : bool [strong]

Controls whether to use the Compensation Look-Up Table path for pixel data.

The Synthesiser includes a Compensation system for correcting amplitude non-linearities in the RF signal path, generating inter-channel phase data for beam steered applications, and synchronous digital and analogue output signals all as a function of the current active pixel frequency. The Compensation table can be in circuit (active) or bypassed for both the normal pixel path from the Controller Image and also for the Local Tone Buffer used on the Synthesiser.

Since

1.1

Enumerator

ACTIVE Use the Compensation Look-up Path.

BYPASS Do not use the Compensation Look-up Path.

17.56.2.3 enum iMS::SignalPath::ENCODER_CHANNEL [strong]

Selects which of two available encoder channels.

The Rotary Encoder input has two channels, each comprising a pair of RS422 differential signals. The signal pairs are decoded by the rotary encoder block into a sequence of forward and reverse pulses which are processed to calculate a tick velocity, which can be converted to angular velocity through knowledge of the number of pulses per revolution (ppr) of the encoder.

Normally, only the first encoder is used and the velocity value used to compensate the frequency on all 4 channels of the synthesiser. However, in the case where the Synthesiser is configured for X/Y deflection, the first encoder input affects Synthesiser channels 1 and 2 and the second encoder input affects Synthesiser channels 3 and 4.

Since

1.4

17.56.2.4 enum iMS::SignalPath::ENCODER_MODE [strong]

Selects the type of encoder connected to the Synthesiser.

The preferred mode of operation is quadrature, in which the two encoder signals output a pulse train in which the second electrically leads or lags the other by 90 degrees, depending on the direction of rotation. This mode gives the best resolution.

The alternative mode: Count+Direction ouputs a single pulse train with the second signal indicating the direction of rotation ('1' = forward, '0' = reverse)

Since

1.4

17.56.2.5 enum iMS::SignalPath::SYNC_SINK [strong]

The Synchronous Output to which to assign Synchronous Data.

Since

1.1

17.56.2.6 enum iMS::SignalPath::SYNC_SRC [strong]

Selects a source of Synchronous Output Data.

Since

1.1

17.56.2.7 enum iMS::SignalPath::ToneBufferControl [strong]

Selects Control Source for the Local Tone Buffer.

The Local Tone Buffer (LTB) in the synthesiser contains 256 individually selectable TBEntry Entries. Each entry contains Frequency, Amplitude and Phase data for each of the 4 channels independently. The index into the LTB can be chosen from either software control, or one of two external control modes. In the standard external control mode, the 4 PROFILE input signals are used to index the first 16 TBEntrys in the LTB. In the extended external control mode, the 4 PROFILE input signals are used in conjunction with the GPI1 and GPO1 control signals to enable selection of all 256 LTB TBEntries as 16 "pages" of 16 Entries each.

If none of these 3 modes is selected, the normal Image Path drives the Synthesiser outputs.

Since

1.1

Enumerator

HOST The Local Tone Buffer is routed to the Synthesiser. Index updates are controlled from host software.

EXTERNAL The Local Tone Buffer is routed to the Synthesiser (first 16 entries only). LTB is indexed from PROFILE pin inputs.

EXTERNAL_EXTENDED The Local Tone Buffer is routed to the Synthesiser (all 256 entries available). Index page and Entry select controlled from PROFILE pin inputs.

OFF Local Tone Buffer not used. Synthesiser outputs from Image data.

17.56.2.8 enum iMS::SignalPath::VELOCITY_MODE [strong]

Selects the method of velocity calculation.

The rotary encoder input is connected to a tracking loop filter which calculates the current angular velocity of the encoder shaft, in ticks / second. The loop filter can generate two different estimates of the current velocity with different characteristics, without altering the behaviour of the filter response.

The first is the closest approximation to the filter state and has a fast response but a higher noise profile which may lead to low level frequency modulation on the DDS output signal.

The second has a much slower response (typically 1-2 orders of magnitude) but a cleaner spectrum.

Since

1.4

17.56.3 Constructor & Destructor Documentation

17.56.3.1 iMS::SignalPath::SignalPath (const IMSSystem & *ims*)

Constructor for SignalPath Object.

An IMSSystem object, representing the configuration of an iMS target must be passed by const reference to the SignalPath constructor.

The IMSSystem object must exist before the SignalPath object, and must remain valid (not destroyed) until the SignalPath object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System

Since

1.0

17.56.4 Member Function Documentation

17.56.4.1 bool iMS::SignalPath::AssignSynchronousOutput (const SYNC_SINK & *sink*, const SYNC_SRC & *src*) const

Selects the source of data for the 2 Analog and 12 Digital output signals that operate synchronously with the Image Pixel Clock.

Since

1.1

17.56.4.2 bool iMS::SignalPath::ClearTone ( )

Stops the tone playback and restores the signal path configuration to the Controller and Compensation Tables.

Returns

true if the clear tone request was sent successfully

Since

1.0

17.56.4.3 bool iMS::SignalPath::ConfigureSyncDigitalOutput (::std::chrono::nanoseconds *delay* = `::std::chrono::nanoseconds::zero()`, ::std::chrono::nanoseconds *pulse_length* = `::std::chrono::nanoseconds::zero()`)

Configures the Synchronous Digital Output data.

Synchronous Digital output data is usually time aligned with the update of RF Channel data output and remains valid for the duration of the image pixel clock period. There are two options to this:

(1) The assertion of Synchronous Digital output data can be delayed with respect to the RF signal by any number of nanoseconds that is less than 655360ns and has a minimum resolution of 10ns. (2) The synchronous digital output bits can be set to "pulse mode" - they return to inactive after a defined time period. The period may be any number of nanoseconds that is less than 655360ns and has a minimum resolution of 10ns.

Parameters

`in`	*delay*	the number of nanoseconds to delay the onset of synchronous digital output data
`in`	*pulse_length*	the width of the digital output data pulse (or zero to disable)

Returns

true if the syncronous digital output data configuration request was sent successfully

Since

1.4

17.56.4.4 bool iMS::SignalPath::DisableEncoder ( )

Turns off the Velocity Compensation process.

Since

1.4

17.56.4.5 bool iMS::SignalPath::EnableImagePathCompensation (SignalPath::Compensation *amplComp*, SignalPath::Compensation *phaseComp*)

Enables / Disables the programmed amplitude and phase Compensation Functions for Image Playback.

Image Pixel data pass through a Compensation process in the Synthesiser which performs amplitude corrections and phase adjustment for beam steering applications as a function of the programmed frequency. The Compensation tables must be programmed either from software, or from a look-up table stored in the Synthesiser FileSystem. If no compensation table has been programmed, or the application does not wish to use Compensation, the process can be bypassed by calling this function with the appropriate settings for amplitude and phase.

Parameters

in	*amplComp*	Set to SignalPath::Compensation::BYPASS or SignalPath::Compensation::↵ ACTIVE for amplitude compensation (frequency dependent correction)
in	*phaseComp*	Set to SignalPath::Compensation::BYPASS or SignalPath::Compensation::↵ ACTIVE for phase compensation (frequency dependent beam steering)

Returns

true if the compensation request was sent successfully

Since

1.3

17.56.4.6 bool iMS::SignalPath::EnableXYPhaseCompensation (bool *XYCompEnable*)

Configures Beam Steering Phase Compensation for X/Y Deflector Mode.

Normal phase beam steering configures the 4 RF Channel outputs for incremental phase adjustment so that channel 1 has zero phase, channel 2 has a frequency dependent phase offset with respect to channel 1, channel 3 has twice the phase offset and channel 4 has three times the phase offset.

In an X/Y deflector configuration, the first two channels are assigned to deflector X and the second two channels to deflector Y. In this case, both channels 1 and 3 have zero phase, channels 2 and 4 have a single frequency dependent offset with respect to those channels.

Parameters

in	*XYCompEnable*	Set to true to enable X/Y style phase beam steering (split channels)

Returns

true if the XY Phase Setting request was sent successfully.

Since

1.3

17.56.4.7 bool iMS::SignalPath::ReportEncoderVelocity (ENCODER_CHANNEL *chan*)

Retrieves the current angular velocity of the requested encoder channel.

Whilst enabled, the encoder inputs are continuously monitored for activity and any movement is converted by the tracking loop filter into an estimate of velocity in number of encoder ticks per second. Note that for a quadrature encoder, a single tick is defined as an edge of either type (rising or falling) on either signal input, to guarantee maximum possible resolution, thus there are 4 ticks to a single pulse on one signal input.

This function allows application software to request the current velocity estimate of either encoder channel. The result is reported to the software in the SignalPathEvents::ENC_VEL_CH_X and ENC_VEL_CH_Y events.

Parameters

in	*chan*	which of the two encoder channels to request the velocity from (X or Y).

Returns

true if the encoder velocity report request was sent successfully

Since

1.4

17.56.4.8 bool iMS::SignalPath::SetCalibrationTone (const FAP & *fap*)

Bypasses Controller Data and Compensation Tables and plays a fixed tone for calibration purposes.

In order to calibrate the RF output signal path and the AO Device, it is sometimes useful to play a fixed calibration tone. This can be achieved using this function, which disconnects the Controller from the signal path along with the Compensation Tables and immediately plays a pure tone on all 4 RF Channels simultaneously at the Frequency, Amplitude and Phase Offsets specified by the input argument. The fixed tone will remain on the output until cleared.

Bug In v1.0 SDK calibration tone amplitude would be 25% of value provided in fap. Corrected in 1.1.0.

Parameters

in	*fap*	a FAP triad specifying the output tone to be played back.

Returns

true if the calibration tone request was sent successfully

Since

1.0

17.56.4.9 bool iMS::SignalPath::SetChannelReversal (bool *reversal*)

Reverses the channel order of the 4 RF Outputs.

Sometimes, usually to simplify cable routing, it is desirable to order the 4 RF outputs in 4-3-2-1 configuration instead of 1-2-3-4. This can be achieved by setting the channel reversal configuration bit, using this function.

Parameters

in	*reversal*	Set true to enable the channel reversal (Channel 1 outputs Channel 4 data and vice versa)

Returns

true if the reversal update request was sent successfully

Since

1.0

17.56.4.10 void iMS::SignalPath::SignalPathEventSubscribe (const int *message*, IEventHandler ∗ *handler*)

Subscribe a callback function handler to a given SignalPathEvents event.

SignalPath can callback user application code when an event occurs that affects the signal path. Supported events are listed under SignalPathEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to a SignalPathEvents event. For the period that a callback is subscribed, each time an event in SignalPath occurs that would trigger the subscribed SignalPathEvents event, the user function callback will be executed.

Parameters

in	*message*	Use the SignalPathEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.0

17.56.4.11 void iMS::SignalPath::SignalPathEventUnsubscribe (const int *message*, const **IEventHandler** ∗ *handler*)

Unsubscribe a callback function handler from a given SignalPathEvents event.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the SignalPath object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the SignalPathEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.0

17.56.4.12 bool iMS::SignalPath::SwitchRFAmplitudeControlSource (const **AmplitudeControl** *src*)

Selects the amplitude control source for all 4 RF channels.

Selects the analogue control source to apply to the RF mixer in the output signal conditioning for all 4 RF Channels: digital pot 1, digital pot 2, external analogue modulation or turned off

Parameters

in	*src*	The Amplitude Control Source selection

Returns

true if the source select update request was sent successfully

Since

1.0

17.56.4.13 bool iMS::SignalPath::UpdateDDSPowerLevel (const **Percent** & *power*)

Scales the DDS device (Direct Digital Synthesis RF signal generator) power up & down.

The RF signal generator device on the Synthesiser converts frequency, amplitude and phase data into the 4 RF signals that drive the output of the Synthesiser. The device can be configured to scale the analogue output power up & down. This function performs the power scaling between 0% (minimum power) and 100% (maximum power)

Parameters

in	*power*	the percentage of maximum power at which the DDS should drive RF signals into the output signal conditioning

Returns

true if the power update request was sent successfully

Since

1.0

17.56.4.14 bool iMS::SignalPath::UpdateEncoder (const VelocityConfiguration & *velcomp*)

UpdateEncoder enables the Encoder velocity offset correction and updates the parameters.

Calling this function will enable the velocity correction capability of the Synthesiser or update the parameters of the velocity correction according to the values in the VelocityConfiguration struct

Parameters

in	*velcomp*	Contains the values with which to configure the Velocity Correction process

Returns

true if the Encoder Update request was sent successfully

Since

1.4

17.56.4.15 bool iMS::SignalPath::UpdateLocalToneBuffer (const ToneBufferControl & *tbc*, const unsigned int *index*, const SignalPath::Compensation *AmplitudeComp* = SignalPath::Compensation::ACTIVE, const SignalPath::Compensation *PhaseComp* = SignalPath::Compensation::ACTIVE)

Use these functions to output tones from the Local Tone Buffer, control their selection and compensation.

The Local Tone Buffer in the Synthesiser stores a set of 256 TBEntry's, each comprising of a FAP per each of the 4 output channels. The LTB can be inserted in the Synthesiser output signal path, replacing the Image data deriving from a connected Controller. Multiple ToneBuffers can be stored in Synthesiser non-volatile memory and any one of these can be recalled by host software and if one of them is marked with the filesystem 'default' flag, it will be loaded into the LTB at startup causing the Signal Path to be routed to the LTB.

In order to determine which method is used to provide the tone index for the LTB (Host Software, External 16-entry and External 256-entry), update the LTB buffer using one of the methods containing a ToneBufferControl parameter.

In order to change the currently selected LTB index (only in Host Software control mode), use one of the methods containing the index parameter.

The LTB outputs may be injected into the Synthesiser signal path either before or after the CompensationTable Look-Up Table. If before (true), amplitude compensation is applied to the signal amplitudes, if after (false), use the methods containing the AmplitudeCompensation parameter.

Parameters

in	*tbc*	Select LTB Control Source
in	*AmplitudeComp*	indicates whether to apply LUT Compensation to Tone amplitude data
in	*PhaseComp*	indicates whether to apply LUT Compensation to Tone phase data
in	*index*	In Host Software control mode, select which LTB index to use

Returns

true if update was successful

Since

1.1

17.56.4.16 bool iMS::SignalPath::UpdateLocalToneBuffer (const ToneBufferControl & *tbc*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

17.56.4.17 bool iMS::SignalPath::UpdateLocalToneBuffer (const SignalPath::Compensation *AmplitudeComp,* const SignalPath::Compensation *PhaseComp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

17.56.4.18 bool iMS::SignalPath::UpdateLocalToneBuffer (const unsigned int *index*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

17.56.4.19 bool iMS::SignalPath::UpdatePhaseTuning (const RFChannel & *channel,* const Degrees & *phase*)

Applies a constant Phase offset to one of the 4 RF Channels.

The 4 RF Channels can be 'tuned' to offset phase discrepancies in, for example, cable length differences by setting up a constant phase offset that will be added to the RF signal output of that channel

Parameters

in	*channel*	The RF Channel to apply the offset to
in	*phase*	The amount of constant phase offset to apply, in degrees

Returns

true if the phase offset update request was sent successfully

Since

1.0

17.56.4.20 bool iMS::SignalPath::UpdateRFAmplitude (const **AmplitudeControl** *src*, const **Percent** & *ampl*)

Scales the Digital Potentiometer mixer drive level up & down.

The 2 digital potentiometers on the Synthesiser can be selected to apply a DC drive level to the IF input of a wideband RF mixer in the output channel signal conditioning, thereby acting as an amplitude control voltage.

This function sets the drive level of the 2 digital potentiometers. The AmplitudeControl input determines which potentiometer is updated, if it is set to anything other than WIPER_1 or WIPER_2, the request is ignored and the function returns false.

Parameters

in	*src*	Which of the two digital potentiometers to update
in	*ampl*	the percentage of maximum amplitude scaling to update the potentiometer to

Returns

true if the amplitude update request was sent successfully

Since

1.0

The documentation for this class was generated from the following file:

- SignalPath.h

17.57 iMS::SignalPathEvents Class Reference

All the different types of events that can be triggered by the SignalPath class.

`#include <include\SignalPath.h>`

Public Types

- enum Events { RX_DDS_POWER, ENC_VEL_CH_X, ENC_VEL_CH_Y, **Count** }

  List of Events raised by the Signal Path module.

17.57.1 Detailed Description

All the different types of events that can be triggered by the SignalPath class.

Some events contain integer parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.57.2 Member Enumeration Documentation

17.57.2.1 enum iMS::SignalPathEvents::Events

List of Events raised by the Signal Path module.

Enumerator

RX_DDS_POWER Returns DDS Power setting.

ENC_VEL_CH_X Returns current Encoder X Channel Velocity.

ENC_VEL_CH_Y Returns current Encoder Y Channel Velocity.

The documentation for this class was generated from the following file:

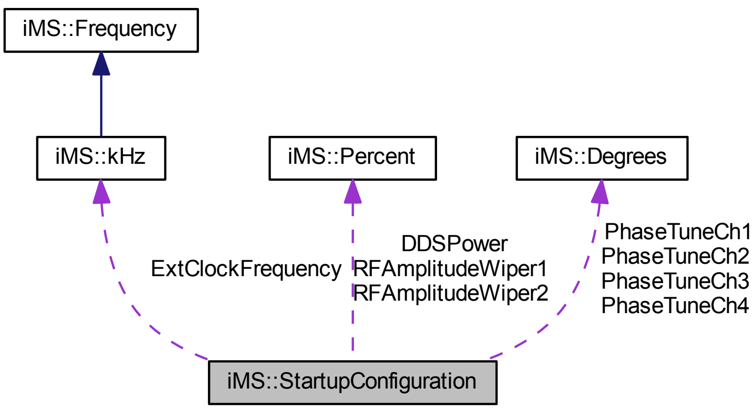
- SignalPath.h

17.58 iMS::StartupConfiguration Struct Reference

The Synthesiser stores in its non-volatile memory a set of configuration values that are preloaded on startup.

`#include <include\SystemFunc.h>`

Collaboration diagram for iMS::StartupConfiguration:



Public Attributes

- Percent RFAmplitudeWiper1 { 0.0 }

  Setting for RF Amplitude Control Wiper 1.
- Percent RFAmplitudeWiper2 { 0.0 }

  Setting for RF Amplitude Control Wiper 2.
- Percent DDSPower { 0.0 }

  Setting for DDS Powere Level.
- SignalPath::AmplitudeControl AmplitudeControlSource { SignalPath::AmplitudeControl::WIPER_1 }

  Select which of the four control sources should be applied to the RF signal amplitude modulation.
- bool RFGate { false }

Switch the RF power amplifier gate signal on/off at startup.

- bool RFBias12 { false }

  Individually enable the bias power for channels 1 and 2 at startup.
- bool RFBias34 { false }

  Individually enable the bias power for channels 3 and 4 at startup.
- bool ExtEquipmentEnable { false }

  Select whether to enable the external equipment optoswitch at startup.
- SignalPath::Compensation LTBUseAmplitudeCompensation { SignalPath::Compensation::ACTIVE }

  Sets whether the LTB should use amplitude compensation from the look-up table.
- SignalPath::Compensation LTBUsePhaseCompensation { SignalPath::Compensation::BYPASS }

  Sets whether the LTB should use phase compensation from the look-up table.
- SignalPath::ToneBufferControl LTBControlSource { SignalPath::ToneBufferControl::OFF }

  Selects the control mode for the LTB: OFF (Image mode), host software control, external drive (16) or extended external (256)
- std::uint8_t LocalToneIndex { 0 }

  In host mode, picks the initial setting for the Tone Buffer index.
- Degrees PhaseTuneCh1 { 0.0 }

  Apply any phase tuning offset coefficient to the RF output on channel 1.
- Degrees PhaseTuneCh2 { 0.0 }

  Apply any phase tuning offset coefficient to the RF output on channel 2.
- Degrees PhaseTuneCh3 { 0.0 }

  Apply any phase tuning offset coefficient to the RF output on channel 3.
- Degrees PhaseTuneCh4 { 0.0 }

  Apply any phase tuning offset coefficient to the RF output on channel 4.
- bool ChannelReversal { false }

  If true, the 4 RF signals will output in reverse channel order.
- SignalPath::Compensation ImageUseAmplitudeCompensation { SignalPath::Compensation::ACTIVE }

  Sets whether Image pixel data should use amplitude compensation from the look-up table.
- SignalPath::Compensation ImageUsePhaseCompensation { SignalPath::Compensation::BYPASS }

  Sets whether Image pixel data should use phase compensation from the look-up table.
- SystemFunc::UpdateClockSource upd_clk { SystemFunc::UpdateClockSource::INTERNAL }

  Configures the DDS update clock source to either be generated internally or be derived from the external Image clock input.
- bool XYCompEnable { false }

  Enables X/Y Deflector mode in which phase compensation is applied independently to each pair of channels.
- Auxiliary::LED_SOURCE LEDGreen { Auxiliary::LED_SOURCE::PULS }

  Configures the Green LED function.
- Auxiliary::LED_SOURCE LEDYellow { Auxiliary::LED_SOURCE::RF_GATE }

  Configures the Yellow LED function.
- Auxiliary::LED_SOURCE LEDRed { Auxiliary::LED_SOURCE::INTERLOCK }

  Configures the Red LED function.
- std::uint8_t GPOutput { 0 }

  The default value to drive on the General Purpose output.
- SystemFunc::NHFLocalReset ResetOnUnhealthy { SystemFunc::NHFLocalReset::NO_ACTION }

  Sets what action to perform if the communications channel enters an "unhealthy" state.
- bool CommsHealthyCheckEnabled { false }

  Turns on/off the communications channel health state check.
- unsigned int CommsHealthyCheckTimerMilliseconds { 500 }

  Timeout between communications messages after which deemed unhealthy.
- SignalPath::SYNC_SRC SyncDigitalSource { SignalPath::SYNC_SRC::IMAGE_DIG }

  Sets the source of synchronous data applied to the digital outputs.

- SignalPath::SYNC_SRC SyncAnalogASource { SignalPath::SYNC_SRC::IMAGE_ANLG_A }

  Sets the source of synchronous data applied to the analog A output.
- SignalPath::SYNC_SRC SyncAnalogBSource { SignalPath::SYNC_SRC::IMAGE_ANLG_B }

  Sets the source of synchronous data applied to the analog B output.
- SystemFunc::PLLLockReference PLLMode { SystemFunc::PLLLockReference::INTERNAL }

  Sets the default system clock mode - internally generated or slave to an external reference clock input.
- kHz ExtClockFrequency { 1000.0 }

  Defines the external supplied reference clock frequency when manual external reference PLL mode is used.

17.58.1 Detailed Description

The Synthesiser stores in its non-volatile memory a set of configuration values that are preloaded on startup.

Modify the values present in this struct and pass the struct by reference to the SystemFunc::StoreStartupConfig() function to overwrite the existing startup configuration parameters

e.g.

```
SystemFunc sys(myiMS);
StartupConfiguration cfg;
cfg.DDSPower = 100.0;
sys.StoreStartupConfig(cfg);
```

Since

1.1

The documentation for this struct was generated from the following file:

- SystemFunc.h

17.59 iMS::SystemFunc Class Reference

Provides System Management functions not directly related to RF signal generation or signal path control.

`#include <include\SystemFunc.h>`

Public Types

- enum UpdateClockSource { UpdateClockSource::INTERNAL, UpdateClockSource::EXTERNAL }

  Determines whether DDS Synthesiser IC should have its update signal driven by the Synthesiser internal circuitry or from an external source (for synchronising the device to a system clock)
- enum TemperatureSensor { TemperatureSensor::TEMP_SENSOR_1, TemperatureSensor::TEMP_SENS←OR_2 }

  There are two available temperature sensors in the Synthesiser System.
- enum PLLLockReference { PLLLockReference::INTERNAL, PLLLockReference::EXTERNAL_FIXED, PLL←LockReference::EXTERNAL_AUTO, PLLLockReference::EXTERNAL_FAILOVER }

  Synthesiser Master Clock Reference Mode.
- enum PLLLockStatus {
  PLLLockStatus::EXTERNAL_NOSIGNAL = 0, PLLLockStatus::INTERNAL_UNLOCKED = 4, PLLLock←Status::INTERNAL_LOCKED = 5, PLLLockStatus::EXTERNAL_VALID_UNLOCKED = 8,
  PLLLockStatus::EXTERNAL_LOCKED = 9 }

  Synthesiser Master Clock Status Cast the status value reported to application code from a GetMasterClockStatus() call to PLLLockStatus to determine the current status of the Synthesiser Master Clock.

Public Member Functions

Constructor & Destructor

- SystemFunc (const IMSSystem &ims)
  Constructor for SystemFunc Object.
- ∼SystemFunc ()
  Destructor for SystemFunc Object.

RF Amplifier Master Switches

These software switches drive signal lines in the Synthesiser which connect through to the RF Amplifier and turn on or off the high power RF Amplifier, and selectively enable pairs of RF Channels within it.

- bool EnableAmplifier (bool en)
  Enables the RF Amplifier.
- bool EnableExternal (bool enable)
  Enables the External Equipment Optoisolator.
- bool EnableRFChannels (bool chan1_2, bool chan3_4)
  Selectively enables channels 1&2 and channels 3&4.

Pixel Interface Checksum Error Counter

The Fast Pixel Interface between the iMS Controller and iMS Synthesiser is protected by a simple checksum. Any errors that accumulate on the interface are recorded in a counter which can be read and reset from software. If the counter is non-zero, an LED can be configured to light on the Synthesiser - see function Auxiliary::AssignLED().

Parameters

in	Reset	*clears the error count to zero, extinguishing the LED (default true)*

Returns

true if the checksum error count request was sent successfully.

Since

1.1

- bool **GetChecksumErrorCount** (bool Reset=true)

DDSUpdateClockSource

- bool SetDDSUpdateClockSource (UpdateClockSource src=UpdateClockSource::INTERNAL)
  Configures DDS Update signal source The Direct Digital Synthesiser engine built into the Synthesiser requires an update signal to initiate the output of an RF signal that was previously programmed to the device from an ImagePoint, ToneBufferEntry or CalibrationTone. Normally this is handled internally by the Synthesiser electronics, in which case this should be left to Internal. In certain advanced usage scenarios (typically where the Synthesiser must be synchronised to a user supplied master clock), the update signal may be sourced externally in which case it is derived from the External Image Clock input.

Startup Configuration Programming

- bool StoreStartupConfig (const StartupConfiguration &cfg)
  Store Synthesiser Default Startup Configuration to Non-volatile Memory.

Temperature Sensing

- bool ReadSystemTemperature (SystemFunc::TemperatureSensor sensor)
  Reads the current temperature of the iMS.

Master Reference Clock

Some iMS Synthesisers feature a PLL (Phase Lock Loop) and high accuracy internal clock oscillator that can either be set to lock the Synthesiser master clock to a precision internal reference (<2ppm) or slave to ane externally supplied reference clock.

If set to slave to an external reference clock, there are three external modes:

1) External Manual: in which the frequency of the externally supplied clock source is programmed into the Synthesiser by application software. 2) External Auto: in which the frequency of the externally supplied clock source is measured by the Synthesiser and the PLL continually updated to lock to that frequency 3) External Failover: a modification of the "Auto" mode in which if the PLL is ever seen to lose its locked state, having previously been locked, it will switch over to the Internal precision crystal oscillator.

In all cases, the externally supplied clock may have any frequency that is a multiple of 10kHz with a minimum supported clock rate of 50kHz and a maximum of 10MHz.

- bool SetClockReferenceMode (SystemFunc::PLLLockReference mode, kHz ExternalFixedFreq=k↩ Hz(1000.0))

  Sets the Master Reference Clock mode of the Synthesiser to either Internal or on of the External modes.
- bool GetClockReferenceStatus ()

  Returns the current status of the master reference clock function.
- bool GetClockReferenceFrequency ()

  Returns the measured frequency of the external reference clock port.
- bool GetClockReferenceMode ()

  Returns the current mode of the reference clock function.

Event Notifications

- void SystemFuncEventSubscribe (const int message, IEventHandler ∗handler)

  Subscribe a callback function handler to a given SystemFuncEvents event.
- void SystemFuncEventUnsubscribe (const int message, const IEventHandler ∗handler)

  Unsubscribe a callback function handler from a given SystemFuncEvents event.

Communications "Not Healthy Flag"

Since

1.4.1

Communications with an iMS System can be monitored using a "Communications Not Healthy" mechanism. The iMS Controller features a timer with a configurable timeout value which resets each time a message is received from the host. If no message is received from the host within the timeout period, the host communications is considered to be in an "unhealthy" state, meaning that messages were expected but haven't arrived. The iMS System will set the Communications Not Healthy flag in any subsequent message responses that do get sent to the host, in case the problem was a temporary one, to indicate the problem to the host. The host can clear the not healthy flag and take any further action, as necessary.

In addition, if configured to do so, the iMS System can cause a local system-wide reset to attempt to re-initialise the communications, once it registers a Not Healthy condition. This will flush any communications buffers and may restart communications if the problem was a local one. This behaviour is turned off by default.

At the host end, two things must be done:

(1) Host software must be sure to send messages to the iMS System regularly, and well within the timeout limit set by the NHF timer. The message can be any simple request for status information or anything else as required. All messages will reset the timer.

(2) Host software must perform a similar type of check, looking for timed out responses to its requests to identify that communications have failed. It should then take appropriate action, resetting its communications interfaces where possible.

The mechanism is intended for high-reliability applications where uptime is important and service access is limited. It can be disabled completely if required.

- enum NHFLocalReset { NHFLocalReset::NO_ACTION = 0, NHFLocalReset::RESET_ON_COMMS_UNH←EALTHY = 1 }

    The action to perform at the iMS System when a Not Healthy condition is registered.
- bool ClearNHF ()

    Clear the Not Healthy Flag once normal service is resumed.
- bool ConfigureNHF (bool Enabled, int milliSeconds, NHFLocalReset reset)

    Configure the Not Healthy Flag mechanism.

17.59.1 Detailed Description

Provides System Management functions not directly related to RF signal generation or signal path control.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

17.59.2 Member Enumeration Documentation

17.59.2.1 enum iMS::SystemFunc::NHFLocalReset [strong]

The action to perform at the iMS System when a Not Healthy condition is registered.

Enumerator

NO_ACTION Do nothing other than set the NHF bit on future responses.

RESET_ON_COMMS_UNHEALTHY Perform a system wide reset.

17.59.2.2 enum iMS::SystemFunc::PLLLockReference [strong]

Synthesiser Master Clock Reference Mode.

Since

1.4.1

Enumerator

INTERNAL Master Clock uses internal precision frequency reference.

EXTERNAL_FIXED Master Clock attempts to phase lock to an externally supplied reference clock with manually configured frequency.

EXTERNAL_AUTO Master Clock attempts to phase lock to an externally supplied reference clock whose frequency is automatically determined.

EXTERNAL_FAILOVER Master Clock attempts to phase lock to an externally supplied reference clock whose frequency is automatically determined. If the reference frequency measurement goes invalid for $>$ 400ms, switch over to the internal frequency source until the clock reference mode is reprogrammed.

17.59.2.3 enum iMS::SystemFunc::PLLLockStatus [strong]

Synthesiser Master Clock Status Cast the status value reported to application code from a GetMasterClockStatus() call to PLLLockStatus to determine the current status of the Synthesiser Master Clock.

Enumerator

EXTERNAL_NOSIGNAL No signal detected on external reference clock input. PLL Unlocked.

INTERNAL_UNLOCKED Master Clock using internal clock reference but PLL Not Locked (this should only occur temporarily when switching from external to internal mode)

INTERNAL_LOCKED Master Clock using internal clock reference and PLL is Locked.

EXTERNAL_VALID_UNLOCKED Reference signal detected on external reference clock input but PLL is Not Locked to it (usually due to a frequency mismatch or non-conformal external signal reference)

EXTERNAL_LOCKED Reference signal detected and PLL is Locked.

17.59.2.4 enum iMS::SystemFunc::TemperatureSensor [strong]

There are two available temperature sensors in the Synthesiser System.

Since

1.4

Enumerator

TEMP_SENSOR_1 Sensor 1 is adjacent to the RF stage.

TEMP_SENSOR_2 Sensor 2 is adjacent to the DC power supplies.

17.59.2.5 enum iMS::SystemFunc::UpdateClockSource [strong]

Determines whether DDS Synthesiser IC should have its update signal driven by the Synthesiser internal circuitry or from an external source (for synchronising the device to a system clock)

Since

1.1

Enumerator

INTERNAL Drive Update Signal internally (default)

EXTERNAL Drive Update Signal from external update source.

17.59.3 Constructor & Destructor Documentation

17.59.3.1 iMS::SystemFunc::SystemFunc (const IMSSystem & *ims*)

Constructor for SystemFunc Object.

An IMSSystem object, representing the configuration of an iMS target must be passed by const reference to the SystemFunc constructor.

The IMSSystem object must exist before the SystemFunc object, and must remain valid (not destroyed) until the SystemFunc object itself is destroyed.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System

Since

1.0

17.59.4 Member Function Documentation

17.59.4.1 bool iMS::SystemFunc::ConfigureNHF (bool *Enabled*, int *milliSeconds*, NHFLocalReset *reset*)

Configure the Not Healthy Flag mechanism.

Parameters

in	*Enabled*	Turns the mechanism on or off (default: on)
in	*milliSeconds*	The timeout interval for the NHF timer (default: 500msec)
in	*reset*	The behaviour to perform when a the Communications is determined to be "Not Healthy" (default: NO_ACTION)

Returns

true if the configuration request was sent successfully

Since

1.0

17.59.4.2 bool iMS::SystemFunc::EnableAmplifier (bool *en*)

Enables the RF Amplifier.

Parameters

in	*en*	true turns on the RF Amplifier, false turns it off

Returns

true if the enable request was sent successfully

Since

1.0

17.59.4.3 bool iMS::SystemFunc::EnableExternal (bool *enable*)

Enables the External Equipment Optoisolator.

Parameters

in	*enable*	true turns on the Optoisolator

Returns

true if the enable request was sent successfully

Since

1.1

17.59.4.4 bool iMS::SystemFunc::EnableRFChannels (bool *chan1_2*, bool *chan3_4*)

Selectively enables channels 1&2 and channels 3&4.

Parameters

in	*chan1_2*	true turns on channels 1 and 2, false turns them off
in	*chan3_4*	true turns on channels 3 and 4, false turns them off

Returns

true if the enable request was sent successfully

Since

1.0

17.59.4.5 bool iMS::SystemFunc::GetClockReferenceFrequency ( )

Returns the measured frequency of the external reference clock port.

The external reference clock port is continually monitored, even if the master clock is set to Internal. This function requests the current frequency of any signal connected to the reference clock port. Note that the reference clock measurement function is limited to measure the input clock only as a multiple of 10kHz so this function will not return a value with any finer resolution than that. The value is returned as a real value (double) event and the user should subscribe to the SystemFuncEvents::MASTER_CLOCK_REF_FREQ event to retrieve the result. The returned double represents the frequency of the external reference clock to the closest value of 10kHz.

Returns

true if the frequency reference request was sent successfully

17.59.4.6 bool iMS::SystemFunc::GetClockReferenceMode ( )

Returns the current mode of the reference clock function.

This function requests from the Synthesiser the current mode in which the reference clock function is operating. The mode is returned as an integer event and the user should subscribe to the SystemFuncEvents::MASTER_CLOCK_REF_MODE event to retrieve the result. The returned integer can be cast to a SystemFunc::PLLLockReference enum to interpret the event integer

Returns

true if the mode request was sent successfully

17.59.4.7 bool iMS::SystemFunc::GetClockReferenceStatus ( )

Returns the current status of the master reference clock function.

This command issues a status request from the Synthesiser's Master Reference Clock. The status is returned as an integer event and the user should subscribe to the SystemFuncEvents::MASTER_CLOCK_REF_STATUS event to retrieve the result. The returned integer can be cast to a SystemFunc::PLLLockStatus enum to interpret the status integer

Returns

true if the status request was sent successfully

17.59.4.8 bool iMS::SystemFunc::ReadSystemTemperature (SystemFunc::TemperatureSensor *sensor*)

Reads the current temperature of the iMS.

Some iMS Synthesisers include onboard temperature sensors to monitor the temperature inside the iMS case (note this is different to the temperature readings available using the Diagnostics class that perform temperature readings on the amplifier and AO Device). Call this function to initiate a temperature reading, specifying which sensor to read from. The temperature value will be reported back to the application code using the SystemFuncEvents::SYNTH↩_TEMPERATURE_1 and SystemFuncEvents::SYNTH_TEMPERATURE_2 events.

Returns

true if the requst to read the iMS temperature was sent successfully

Since

1.4

17.59.4.9 bool iMS::SystemFunc::SetClockReferenceMode (SystemFunc::PLLLockReference *mode*, kHz *ExternalFixedFreq* = kHz (1000.0))

Sets the Master Reference Clock mode of the Synthesiser to either Internal or on of the External modes.

Specify the desired reference clock mode. If using EXTERNAL_FIXED, also specify the external frequency

Parameters

in	*mode*	The reference clock mode to set
in	*ExternalFixed↩Freq*	the frequency of the external reference clock, if using Fixed mode

Returns

true if the mode setting command was sent successfully

Since

1.4.1

17.59.4.10 bool iMS::SystemFunc::SetDDSUpdateClockSource (UpdateClockSource *src* = UpdateClockSource::INTERNAL)

Configures DDS Update signal source The Direct Digital Synthesiser engine built into the Synthesiser requires an update signal to initiate the output of an RF signal that was previously programmed to the device from an Image↩Point, ToneBufferEntry or CalibrationTone. Normally this is handled internally by the Synthesiser electronics, in which case this should be left to Internal. In certain advanced usage scenarios (typically where the Synthesiser must be synchronised to a user supplied master clock), the update signal may be sourced externally in which case it is derived from the External Image Clock input.

Parameters

in	*src*	INTERNAL for most scenarios, set to EXTERNAL for external update signal applications

Returns

true if the request to change update signal source was sent successfully

Since

1.1

17.59.4.11 bool iMS::SystemFunc::StoreStartupConfig (const StartupConfiguration & *cfg*)

Store Synthesiser Default Startup Configuration to Non-volatile Memory.

After every power up and reset event, the Synthesiser will inspect the non-volatile memory to see if a startup configuration is present. If it is, the configuration contents are parsed and assigned to their respective control registers. Combining this process with Default Scripts stored in the Filesystem can result in a fully specified standalone operational Synthesiser system with no software connection required. param[in] cfg A const reference to the required configuration behaviour structure. Pre-define the behaviour by setting the config structure fields to requirements.

Returns

true if the request to program the startup configuration was sent successfully

Since

1.1

17.59.4.12 void iMS::SystemFunc::SystemFuncEventSubscribe (const int *message*, IEventHandler * *handler*)

Subscribe a callback function handler to a given SystemFuncEvents event.

SystemFunc can callback user application code when an event occurs that affects the signal path. Supported events are listed under SystemFuncEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to a SystemFuncEvents event. For the period that a callback is subscribed, each time an event in SystemFunc occurs that would trigger the subscribed SystemFunc↩ Events event, the user function callback will be executed.

Parameters

in	*message*	Use the SystemFuncEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.0

17.59.4.13 void iMS::SystemFunc::SystemFuncEventUnsubscribe (const int *message*, const IEventHandler * *handler*)

Unsubscribe a callback function handler from a given SystemFuncEvents event.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the SystemFunc object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the SystemFuncEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.0

The documentation for this class was generated from the following file:

- SystemFunc.h

17.60 iMS::SystemFuncEvents Class Reference

All the different types of events that can be triggered by the SystemFunc class.

`#include <include\SystemFunc.h>`

Public Types

- enum Events {
  PIXEL_CHECKSUM_ERROR_COUNT, **MASTER_CLOCK_REF_FREQ**, **MASTER_CLOCK_REF_MOD↩E**, **MASTER_CLOCK_REF_STATUS**,
  SYNTH_TEMPERATURE_1, **SYNTH_TEMPERATURE_2**, **Count** }

  List of Events raised by the Signal Path module.

17.60.1 Detailed Description

All the different types of events that can be triggered by the SystemFunc class.

Some events contain integer parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2015-11-11

Since

1.0

17.60.2 Member Enumeration Documentation

17.60.2.1 enum iMS::SystemFuncEvents::Events

List of Events raised by the Signal Path module.

Enumerator

PIXEL_CHECKSUM_ERROR_COUNT The number of Errors accumulated on the Pixel Interface.

The documentation for this class was generated from the following file:

- SystemFunc.h

17.61 iMS::ToneBuffer Class Reference

An array of 4-channel FAP Tones stored in memory on the Synthesiser.

`#include <include/ToneBuffer.h>`

Public Types

Tone Buffer Array

`TBArray` is the internal type definition used for storing a buffer of TBEntry 's in the Image

- using **TBArray** = std::array< TBEntry, 256 >

Iterator Specification

Use these iterators when you want to work with ranges of Tone Buffer entries stored within a tone buffer. Iterators can be used to access elements at an arbitrary offset position relative to the element they point to

Two types of iterators are supported; both are random access iterators. Dereferencing const_iterator yields a reference to a constant element in the ToneBuffer (const TBEntry&).

- typedef TBArray::iterator iterator
  Iterator defined for user manipulation of internal TBArray.
- typedef TBArray::const_iterator const_iterator
  Const Iterator defined for user readback of internal TBArray.

Public Member Functions

Constructors & Destructors

- ToneBuffer (const std::string &name="")
  Empty Constructor.
- ToneBuffer (const TBEntry &tbe, const std::string &name="")
  Fill Constructor.
- ToneBuffer (const int entry, const std::string &name="")
  Non-volatile Memory Constructor.
- ToneBuffer (const ToneBuffer &)
  Copy Constructor.
- ToneBuffer & operator= (const ToneBuffer &)
  Assignment Constructor.
- ∼ToneBuffer ()
  Destructor.

ToneBuffer Boundary Iterators

- iterator begin ()
  Returns an iterator pointing to the first element in the TBArray container.
- iterator end ()
  Returns an iterator referring to the past-the-end element in the TBArray container.
- const_iterator begin () const
  Returns a const_iterator pointing to the first element in the TBArray container.
- const_iterator end () const
  Returns a const_iterator referring to the past-the-end element in the TBArray container.
- const_iterator cbegin () const
  Returns a const_iterator pointing to the first element in the TBArray container.
- const_iterator cend () const
  Returns a const_iterator referring to the past-the-end element in the TBArray container.

TBArray Operators

- const TBEntry & operator[ ] (std::size_t idx) const
  Random Access to a TBEntry in the TBArray.
- TBEntry & operator[ ] (std::size_t idx)
  Random Write Access to a TBEntry in the TBArray.
- bool operator== (ToneBuffer const &rhs) const
  Equality Operator checks ToneBuffer contents for equivalence.

ToneBuffer Size

- const std::size_t Size () const
  Returns the number of elements in the ToneBuffer (non-modifiable)

Tone Buffer Description

- const std::string & Name () const
  A string stored with the Tone Buffer to aid human users in identifying the purpose of the buffer.
- std::string & **Name** ()

17.61.1 Detailed Description

An array of 4-channel FAP Tones stored in memory on the Synthesiser.

Author

Dave Cowan

Date

2016-02-24

Since

1.1

17.61.2 Constructor & Destructor Documentation

17.61.2.1 iMS::ToneBuffer::ToneBuffer (const std::string & *name* = "")

Empty Constructor.

Parameters

in	*name*	The optional descriptive name to apply to the Tone Buffer

17.61.2.2 iMS::ToneBuffer::ToneBuffer (const TBEntry & *tbe*, const std::string & *name* = "")

Fill Constructor.

Use this constructor to generate a Tone Buffer with each entry initialised to the value of `tbe`

Parameters

in	*tbe*	The TBEntry that will fill each of the elements of the TBArray
in	*name*	The optional descriptive name to apply to the Tone Buffer

Since

1.1

17.61.2.3 iMS::ToneBuffer::ToneBuffer (const int *entry*, const std::string & *name* = "")

Non-volatile Memory Constructor.

Use this constructor to preload the ToneBuffer with data recalled from an entry in the Synthesiser FileSystem.

Parameters

in	*entry*	the entry in the FileSystem Table from which to recall a ToneBuffer
in	*name*	The optional descriptive name to apply to the Tone Buffer

Since

1.1

17.61.3 Member Function Documentation

17.61.3.1 iterator iMS::ToneBuffer::begin ( )

Returns an iterator pointing to the first element in the TBArray container.

Returns

An iterator to the beginning of the TBArray container.

Since

1.1

17.61.3.2 const_iterator iMS::ToneBuffer::begin ( ) const

Returns a const_iterator pointing to the first element in the TBArray container.

Returns

A const_iterator to the beginning of the TBArray container.

Since

1.2.5

17.61.3.3 const_iterator iMS::ToneBuffer::cbegin ( ) const

Returns a const_iterator pointing to the first element in the TBArray container.

Returns

A const_iterator to the beginning of the TBArray container.

Since

1.1

17.61.3.4 const_iterator iMS::ToneBuffer::cend ( ) const

Returns a const_iterator referring to the past-the-end element in the TBArray container.

Returns

A const_iterator to the element past the end of the buffer.

Since

1.1

17.61.3.5 iterator iMS::ToneBuffer::end ( )

Returns an iterator referring to the past-the-end element in the TBArray container.

The past-the-end element is the theoretical element that would follow the last element in the TBArray container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function can be used in combination with TBArray::begin to specify a range including all the elements in the container.

Returns

An iterator to the element past the end of the TBArray.

Since

1.1

17.61.3.6 const_iterator iMS::ToneBuffer::end ( ) const

Returns a const_iterator referring to the past-the-end element in the TBArray container.

Returns

A const_iterator to the element past the end of the buffer.

Since

1.2.5

17.61.3.7 const std::string& iMS::ToneBuffer::Name ( ) const

A string stored with the Tone Buffer to aid human users in identifying the purpose of the buffer.

A descriptive string can be set alongside the Tone Buffer to allow users to identify and differentiate between Tone Buffers without having to browse through the data. The description is optional, and if, not used, the description will simply default to null.

17.61.3.8 bool iMS::ToneBuffer::operator== (ToneBuffer const & *rhs*) const

Equality Operator checks ToneBuffer contents for equivalence.

Parameters

in	*rhs*	A ToneBuffer object to perform the comparison with

Returns

True if the supplied ToneBuffer is identical to this one.

Since

1.1

17.61.3.9 const TBEntry& iMS::ToneBuffer::operator[ ] (std::size_t *idx*) const

Random Access to a TBEntry in the TBArray.

The array subscript operator is defined to permit applications to access a TBEntry at any arbitrary position for readback.

Parameters

in	*idx*	Integer offset into the TBArray with respect to the first element in the array (ToneBuffer::cbegin())

Returns

A const reference to a TBEntry.

Since

1.1

17.61.3.10 TBEntry& iMS::ToneBuffer::operator[ ] (std::size_t *idx*)

Random Write Access to a TBEntry in the TBArray.

The array subscript operator is defined to permit applications to access a CompensationPoint at any arbitrary position for modification.

Parameters

in	*idx*	Integer offset into the TBArray with respect to the first element in the array (ToneBuffer::begin())

Returns

A reference to a TBEntry.

Since

1.1

17.61.3.11 const std::size_t iMS::ToneBuffer::Size ( ) const

Returns the number of elements in the ToneBuffer (non-modifiable)

Returns

The number of elements in the ToneBuffer

Since

1.1

The documentation for this class was generated from the following file:

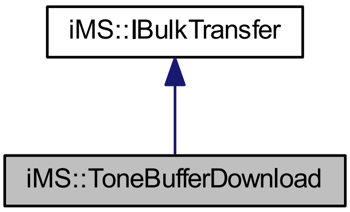
- ToneBuffer.h

17.62 iMS::ToneBufferDownload Class Reference

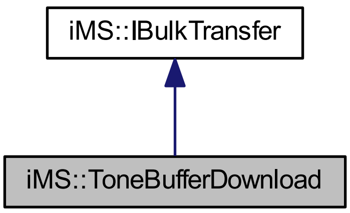
Provides a mechanism for downloading ToneBuffer's to a Synthesiser's LTB memory.

`#include <include\ToneBuffer.h>`

Inheritance diagram for iMS::ToneBufferDownload:



Collaboration diagram for iMS::ToneBufferDownload:



Public Member Functions

Constructor & Destructor

- ToneBufferDownload (IMSSystem &ims, const ToneBuffer &tb)

Constructor for ToneBufferDownload Object.
- ∼ToneBufferDownload ()

  Destructor for ToneBufferDownload Object.

Bulk Transfer Initiation

- bool StartDownload ()

  Begins download of entire ToneBuffer to LTB memory on Synthesiser.
- bool StartDownload (ToneBuffer::const_iterator first, ToneBuffer::const_iterator last)

  Begins download of partial ToneBuffer to LTB memory on Synthesiser beginning at `first` TBEntry and continuing until `last` TBEntry (including first but not including last)
- bool StartDownload (ToneBuffer::const_iterator single)

  Downloads a single TBEntry to LTB memory on Synthesiser.
- bool StartVerify ()

  No Verify is possible. Always returns false.
- int GetVerifyError ()

  No Verify is possible. Always return -1.

Event Notifications

- void ToneBufferDownloadEventSubscribe (const int message, IEventHandler ∗handler)

  Subscribe a callback function handler to a given ToneBufferEvents entry.
- void ToneBufferDownloadEventUnsubscribe (const int message, const IEventHandler ∗handler)

  Unsubscribe a callback function handler from a given ToneBufferEvents entry.

Store in Synthesiser Non-Volatile Memory

- const FileSystemIndex Store (const std::string &FileName, FileDefault def=FileDefault::NON_DEFAULT) const

  Store ToneBuffer contents to non-volatile memory on the synthesiser.

17.62.1 Detailed Description

Provides a mechanism for downloading ToneBuffer's to a Synthesiser's LTB memory.

Author

Dave Cowan

Date

2016-02-24

Since

1.1

17.62.2 Constructor & Destructor Documentation

17.62.2.1 iMS::ToneBufferDownload::ToneBufferDownload (IMSSystem & *ims*, const ToneBuffer & *tb*)

Constructor for ToneBufferDownload Object.

The pre-requisites for an ToneBufferDownload object to be created are: (1) - an IMSSystem object, representing the configuration of an iMS target to which the ToneBuffer is to be downloaded. (2) - a complete ToneBuffer object to download to the iMS target.

ToneBufferDownload stores const references to both. This means that both must exist before the ToneBufferDownload object, and both must remain valid (not destroyed) until the ToneBufferDownload object itself is destroyed. Because they are stored as references, the IMSSystem and ToneBuffer objects themselves may be modified after the construction of the ToneBufferDownload object.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System which is the target for downloading the Image
in	*tb*	A const reference to the ToneBuffer which shall be downloaded to the target

Since

1.1

17.62.3 Member Function Documentation

17.62.3.1 int iMS::ToneBufferDownload::GetVerifyError ( ) [inline], [virtual]

No Verify is possible. Always return -1.

Since

1.1

Implements iMS::IBulkTransfer.

17.62.3.2 bool iMS::ToneBufferDownload::StartDownload ( ) [virtual]

Begins download of entire ToneBuffer to LTB memory on Synthesiser.

Since

1.1

Implements iMS::IBulkTransfer.

17.62.3.3 bool iMS::ToneBufferDownload::StartDownload (ToneBuffer::const_iterator *first*, ToneBuffer::const_iterator *last*)

Begins download of partial ToneBuffer to LTB memory on Synthesiser beginning at `first` TBEntry and continuing until `last` TBEntry (including first but not including last)

Since

1.1

17.62.3.4 bool iMS::ToneBufferDownload::StartDownload (ToneBuffer::const_iterator *single*)

Downloads a single TBEntry to LTB memory on Synthesiser.

Since

1.1

17.62.3.5 bool iMS::ToneBufferDownload::StartVerify ( ) [inline], [virtual]

No Verify is possible. Always returns false.

Since

1.1

Implements iMS::IBulkTransfer.

17.62.3.6 const FileSystemIndex iMS::ToneBufferDownload::Store (const std::string & *FileName*, FileDefault *def* = FileDefault::NON_DEFAULT) const

Store ToneBuffer contents to non-volatile memory on the synthesiser.

The contents of this ToneBuffer can be stored to an area of non-volatile memory on the Synthesiser for retrieval at a future time, including after subsequent power cycles. The data stored can be used to select between alternative ToneBuffers without needing to recalculate or download from Software.

The table can be flagged to be used as a default at startup in which case the Synthesiser will use the contents as a default ToneBuffer program allowing the Synthesiser to be used with no connection to a host system.

Parameters

in	*def*	mark the entry as a default and the Synthesiser will attempt to program the data to the Local Tone Buffer on power up.
in	*FileName*	a string to tag the download with in the File System Table (limited to 8 chars)

Returns

the index in the File System Table where the data was stored or -1 if the operation failed

Since

1.1

17.62.3.7 void iMS::ToneBufferDownload::ToneBufferDownloadEventSubscribe (const int *message*, IEventHandler ∗ *handler*)

Subscribe a callback function handler to a given ToneBufferEvents entry.

ToneBufferDownload can callback user application code when an event occurs in the download process. Supported events are listed under ToneBufferEvents. The callback function must inherit from the IEventHandler interface and override its EventAction() method.

Use this member function call to subscribe a callback function to an ToneBufferEvents entry. For the period that a callback is subscribed, each time an event in ToneBufferDownload occurs that would trigger the subscribed Tone↩BufferEvents entry, the user function callback will be executed.

Parameters

in	*message*	Use the ToneBufferEvents::Event enum to specify an event to subscribe to
in	*handler*	A function pointer to the user callback function to execute on the event trigger.

Since

1.1

17.62.3.8 void iMS::ToneBufferDownload::ToneBufferDownloadEventUnsubscribe (const int *message*, const IEventHandler ∗ *handler*)

Unsubscribe a callback function handler from a given ToneBufferEvents entry.

Removes all links to a user callback function from the Event Trigger map so that any events that occur in the ToneBufferDownload object following the Unsubscribe request will no longer execute that function

Parameters

in	*message*	Use the ToneBufferEvents::Event enum to specify an event to unsubscribe from
in	*handler*	A function pointer to the user callback function that will no longer execute on an event

Since

1.1

The documentation for this class was generated from the following file:

- ToneBuffer.h

17.63 iMS::ToneBufferEvents Class Reference

All the different types of events that can be triggered by the ToneBuffer and ToneBufferDownload classes.

`#include <include\ToneBuffer.h>`

Public Types

- enum Events { DOWNLOAD_FINISHED, DOWNLOAD_ERROR, **Count** }

  List of Events raised by the ToneBuffer Class and ToneBuffer Table Downloader.

17.63.1 Detailed Description

All the different types of events that can be triggered by the ToneBuffer and ToneBufferDownload classes.

Some events contain integer parameter data which can be processed by the IEventHandler::EventAction derived method

Author

Dave Cowan

Date

2016-02-24

Since

1.1

17.63.2 Member Enumeration Documentation

17.63.2.1 enum iMS::ToneBufferEvents::Events

List of Events raised by the ToneBuffer Class and ToneBuffer Table Downloader.

Enumerator

DOWNLOAD_FINISHED Event raised when ToneBufferDownload has confirmed that the iMS Controller received all of the ToneBuffer data.

DOWNLOAD_ERROR Event raised each time the ToneBufferDownload class registers an error in the download process.

The documentation for this class was generated from the following file:

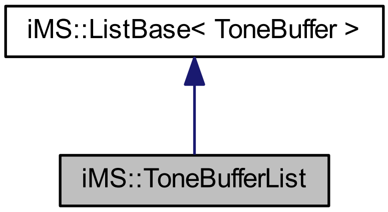
- ToneBuffer.h

17.64 iMS::ToneBufferList Class Reference

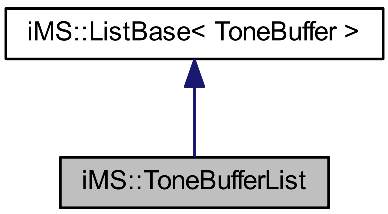
A List of ToneBuffer's used as a container by ImageProject.

`#include <include/Image.h>`

Inheritance diagram for iMS::ToneBufferList:



Collaboration diagram for iMS::ToneBufferList:



Additional Inherited Members

17.64.1 Detailed Description

A List of ToneBuffer's used as a container by ImageProject.

Date

2016-11-09

Since

1.3

The documentation for this class was generated from the following file:

- ImageProject.h

17.65 iMS::UserFileReader Class Reference

Provides a mechanism for retrieving User File data from the Synthesiser FileSystem.

`#include <include\FileSystem.h>`

Public Member Functions

Constructor & Destructor

- UserFileReader (const IMSSystem &ims, const FileSystemIndex index)
  Constructor for UserFileReader Object.
- UserFileReader (const IMSSystem &ims, const std::string &FileName)
  Constructor for UserFileReader Object (referenced by File Name)
- ∼UserFileReader ()
  UserFileReader destructor.

Readback Core Function

- bool Readback (std::vector< std::uint8_t > &data)
  Retrieves User File data into a byte array.

17.65.1 Detailed Description

Provides a mechanism for retrieving User File data from the Synthesiser FileSystem.

Author

Dave Cowan

Date

2016-01-21

Since

1.1

17.65.2 Constructor & Destructor Documentation

17.65.2.1 iMS::UserFileReader::UserFileReader (const IMSSystem & *ims*, const FileSystemIndex *index*)

Constructor for UserFileReader Object.

The UserFileReader object requires an IMSSystem object, which will have had its FileSystemTable read back during initialisation. It must therefore exist before the UserFileReader object, and must remain valid (not destroyed) until the UserFileReader object itself is destroyed. The UserFileReader object is tied to a single FileSystemTableEntry and can only be used for reading back that object. If multiple files need to be read back, new UFRs should be created for each one.

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System whose FileSystemTable should be used for reading back data
in	*index*	the Entry in the FileSystemTable containing USER_DATA file data to readback

Since

1.1

17.65.2.2 iMS::UserFileReader::UserFileReader (const IMSSystem & *ims*, const std::string & *FileName*)

Constructor for UserFileReader Object (referenced by File Name)

Parameters

in	*ims*	A const reference to the iMS System whose FileSystemTable should be used for reading back data
in	*FileName*	a string representing the name of the entry containing USER_DATA file data to readback

Since

1.1

17.65.3 Member Function Documentation

17.65.3.1 bool iMS::UserFileReader::Readback (std::vector< std::uint8_t > & *data*)

Retrieves User File data into a byte array.

Call this function to initiate readback of data from the Synthesiser FileSystem into a byte array allocated by the application

Parameters

out	*data*	A reference to a vector to store the unformatted byte data representing the user file. Any existing contents are overwritten.

Returns

true if the operation was successful

Since

1.1

The documentation for this class was generated from the following file:

- FileSystem.h

17.66 iMS::UserFileWriter Class Reference

Provides a mechanism for committing User File data to the Synthesiser FileSystem.

`#include <include\FileSystem.h>`

Public Member Functions

Constructor & Destructor

- UserFileWriter (IMSSystem &ims, const std::vector< std::uint8_t > &file_data, const std::string file_name)
  Constructor for UserFileWriter Object.
- ∼UserFileWriter ()
  Destructor for UserFileWriter.

File Write Core Function

- FileSystemIndex Program ()
  Stores User File data into a FileSystem and allocates a new FileSystemTableEntry.

17.66.1 Detailed Description

Provides a mechanism for committing User File data to the Synthesiser FileSystem.

Author

Dave Cowan

Date

2016-01-21

Since

1.1

17.66.2 Constructor & Destructor Documentation

17.66.2.1 iMS::UserFileWriter::UserFileWriter (IMSSystem & *ims,* const std::vector< std::uint8_t > & *file_data,* const std::string *file_name*)

Constructor for UserFileWriter Object.

The UserFileWriter object requires an IMSSystem object, which will have had its FileSystemTable read back during initialisation. It must therefore exist before the UserFileWriter object, and must remain valid (not destroyed) until the UserFileWriter object itself is destroyed.

A reference to the User File data wrapped in an unformatted byte array needs to be provided, along with a string representing the file name to allocate to the file in the FileSystemTable.

The File Name may be any sequence of valid ASCII characters, including all special characters ($, %, /, \ etc) but not control characters. It is limited to 8 characters and will be truncated as such. Though not recommended, it is permissible to allocate the same filename to multiple files contained in the FileSystemTable

Once constructed, the object can neither be copied or assigned to another instance.

Parameters

in	*ims*	A const reference to the iMS System whose FileSystemTable should be used for writing new data
in	*file_data*	an unformatted byte array containing the User File contents to program
in	*file_name*	a string representing the name of the file to be allocated in the FileSystemTable

Since

1.1

17.66.3 Member Function Documentation

17.66.3.1 FileSystemIndex iMS::UserFileWriter::Program ( )

Stores User File data into a FileSystem and allocates a new FileSystemTableEntry.

Call this function to initiate writing of the provided User File data into the FileSystem.

The function will first attempt to find sufficient free space and allocate it for the new data. If it cannot do that, it will return an invalid FileSystemIndex (-1). If free space was found, it will start writing the user file data starting at the address that was found by the allocation algorithm (the user application cannot predict where in the Filesystem address space the User data will be stored but this is unlikely to be a problem). A new FileSystemTableEntry is created and added to the FileSystemTable containing the allocated address, the overall file length (including an additional 2-byte marker at the start required by the FileSystem protocol), the type as USER_DATA, a NON_DE←FAULT marker, and the next available index.

Returns

the index in the FileSystemTable that was created by the Programming process, or -1 if it failed.

Since

1.1

The documentation for this class was generated from the following file:

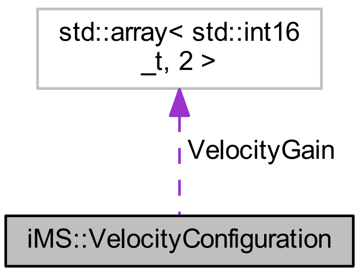
- FileSystem.h

17.67 iMS::VelocityConfiguration Struct Reference

Sets the parameters required to control the operation of the Encoder Input / Velocity Compensation function.

`#include <SignalPath.h>`

Collaboration diagram for iMS::VelocityConfiguration:



Public Member Functions

- void SetVelGain (const IMSSystem &ims, SignalPath::ENCODER_CHANNEL chan, kHz EncoderFreq, MHz DesiredFreqDeviation, bool Reverse=false)

  Sets the amount of frequency deviation gain applied to velocity measurement.

Public Attributes

- SignalPath::ENCODER_MODE EncoderMode { SignalPath::ENCODER_MODE::QUADRATURE }

  Sets the type of encoder signal connected to the Synthesiser inputs.
- SignalPath::VELOCITY_MODE VelocityMode { SignalPath::VELOCITY_MODE::FAST }

  Sets the velocity calculation method used in the tracking filter for frequency compensation.
- std::uint16_t TrackingLoopProportionCoeff { 4000 }

  The Proportion Coefficient (0 - 65535) used in the Tracking Loop Filter.
- std::uint16_t TrackingLoopIntegrationCoeff { 10000 }

  The Integration Coefficient (0 - 65535) used in the Tracking Loop Filter.
- std::array< std::int16_t, 2 > VelocityGain

  Controls the extent to which a given value of velocity causes a deviation in synthesiser frequency. Do not set manually, use SetVelGain.

17.67.1 Detailed Description

Sets the parameters required to control the operation of the Encoder Input / Velocity Compensation function.

Holds parameters for the Encoder type (Quadrature or Clk/Dir), Velocity Estimation method, tracking loop filter parameters and overall output gain - being the amount of deviation applied to the RF frequency generation for a given encoder velocity. Also contains a method for calculating the value of the gain parameter for a desired frequency deviation at a given encoder velocity.

Since

1.4

17.67.2 Member Function Documentation

17.67.2.1 void iMS::VelocityConfiguration::SetVelGain (const IMSSystem & *ims*, SignalPath::ENCODER_CHANNEL *chan*, kHz *EncoderFreq*, MHz *DesiredFreqDeviation*, bool *Reverse* = `false`)

Sets the amount of frequency deviation gain applied to velocity measurement.

Use this function to set the encoder channel gain according to the amount of desired frequency offset (deviation) at a chosen spot encoder angular frequency.

Parameters

in	*ims*	a const reference to the IMSSystem in use
in	*chan*	Which channel (X or Y) to set the encoder gain for
in	*EncoderFreq*	The encoder tick frequency for which we shall define the gain
in	*DesiredFreqDeviation*	The amount of change to the RF Frequency that shall be offset when the encoder is operating at the specified velocity
in	*Reverse*	Causes the RF frequency deviation to effect in the opposite direction

The documentation for this struct was generated from the following file:

- SignalPath.h

Chapter 18

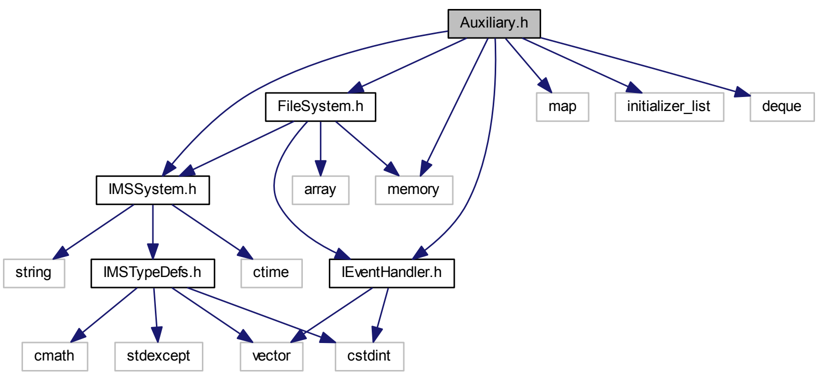
File Documentation

18.1 Auxiliary.h File Reference

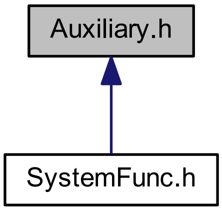
Classes for performing various auxiliary actions not directly related to driving Acousto-Optic devices.

```
#include "IEventHandler.h"
#include "IMSSystem.h"
#include "FileSystem.h"
#include <memory>
#include <map>
#include <initializer_list>
#include <deque>
```

Include dependency graph for Auxiliary.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::AuxiliaryEvents

  All the different types of events that can be triggered by the Auxiliary class.
- class iMS::Auxiliary

  Provides auxiliary additional functions not directly related to Synthesiser operation.
- class iMS::DDSScriptRegister

  Create a register write to send to the DDS IC.
- class iMS::DDSScriptDownload

  Provides a mechanism for transferring DDS Scripts into Filesystem memory.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

Typedefs

- using iMS::DDSScript = std::vector< DDSScriptRegister >

  `DDSScript` *stores the sequence of register writes to be loaded onto the Synthesiser. Can be manipulated using the normal container operations provided by std::vector*

18.1.1 Detailed Description

Classes for performing various auxiliary actions not directly related to driving Acousto-Optic devices.

There are a number of additional functions provided by the Synthesiser which may be used to facilitate integration of the iMS device into the overall system. These features are not fundamental to the operation of the iMS device which is why they are held in a separate 'Auxiliary' file.

Features include:

- assignment of LEDs to indicate specific events
- Reading one of the two external analog inputs
- Writing to the external analog output

- Controlling the 4-bit Profile select signal driving the DDS Synthesiser IC (software control or externally provided)
- Advanced manual control of register contents written to the DDS Synthesiser IC.

Author

Dave Cowan

Date

2016-02-18

Since

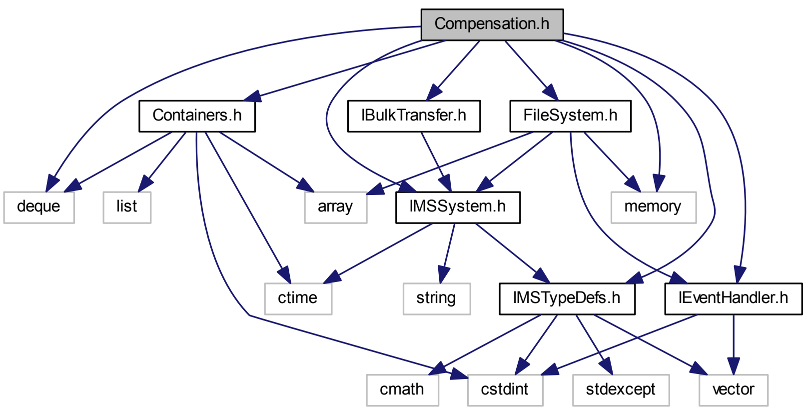
1.1

18.2 Compensation.h File Reference

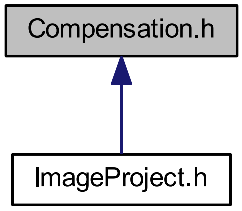
Classes for creating and downloading data that is used in the Compensation tables of the Synthesiser.

```
#include "Containers.h"
#include "IMSSystem.h"
#include "IEventHandler.h"
#include "IMSTypeDefs.h"
#include "IBulkTransfer.h"
#include "FileSystem.h"
#include <memory>
#include <deque>
```

Include dependency graph for Compensation.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::CompensationEvents

  All the different types of events that can be triggered by the Compensation and CompensationTableDownload classes.
- class iMS::CompensationPoint

  Stores 4 data fields containing amplitude, phase, sync analogue and sync digital compensation data.
- class iMS::CompensationPointSpecification

  Completely specifies the desired compensation at a spot frequency.
- class iMS::CompensationFunction

  Class for performing Compensation related functions with the Synthesiser.
- class iMS::CompensationTable

  A table of CompensationPoints storing look-up data that can be transferred to memory in the Synthesiser.
- class iMS::CompensationTableDownload

  Provides a mechanism for downloading and verifying Compensation Tables to a Synthesiser's Look-Up memory.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.2.1 Detailed Description

Classes for creating and downloading data that is used in the Compensation tables of the Synthesiser.

The Compensation Tables are a part of the signal chain in the Synthesiser. There are 4 of them, each serving a different purpose. All 4 are indexed by the signal frequency, spanning the lowest to the highest frequency supported by the Synthesiser, each table consisting of a sequence of look-up entries (typically 2,048) spaced equidistantly in frequency.

The 4 tables are:

(1) Amplitude: used to compensate for frequency-dependent inefficiency in the AO device, as well as in the RF Amplifier and the Synthesiser. The signal amplitude passing through the Synthesiser is multiplied by the compensation output to result in a combined amplitude being passed to the Synthesiser DDS device.

(2) Phase: used in beam-steered AO applications where multiple acoustic columns present in the crystal are offset in phase from each other in a way that is linearly dependent on the frequency offset from a central Bragg Angle adjusted frequency.

(3) Analogue Sync: The output of this table can be routed to the Synchronous DAC output which gives a handy analogue reference signal for either test purposes or for driving external custom circuitry. The advantage of driving this from the look-up table is that custom mappings can be generated which allows great flexibility in configuring the analogue signal in relation to the signal frequency that drives it.

(4) Digital Sync: As with the analogue sync, the output of this table is routed to external synchronous outputs which can be used for test purposes or for driving external custom circuitry. The digital output bits could, for example, be used to tune signal conditioning circuitry as the RF signal passes through certain frequency bands.

Author

Dave Cowan

Date

2015-11-03

Since

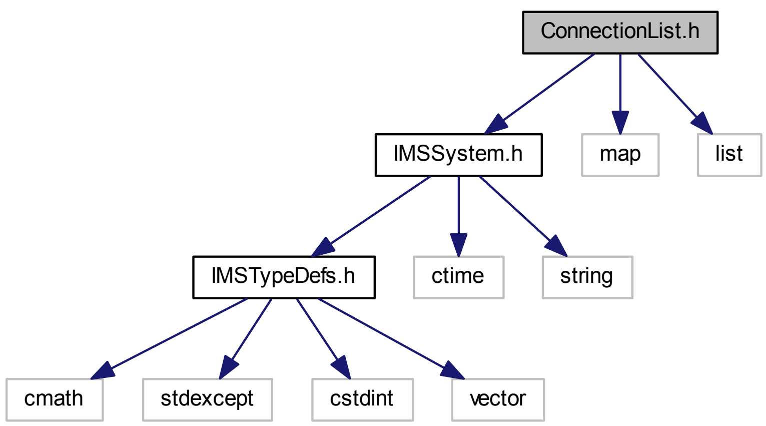
1.0

18.3 ConnectionList.h File Reference

Creates iMS Connection Interfaces for Application Use and scans them to discover all available iMS Systems.

```
#include "IMSSystem.h"
#include <map>
#include <list>
```

Include dependency graph for ConnectionList.h:



Classes

- class iMS::ConnectionList

  Creates iMS Connection Interfaces and scans them to discover available iMS Systems.
- struct iMS::ConnectionList::ConnectionConfig

  Controls the behaviour of a Connection Module during its discovery process.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.3.1 Detailed Description

Creates iMS Connection Interfaces for Application Use and scans them to discover all available iMS Systems.

ConnectionList.h is the starting point for all software interaction with an iMS System. It maintains a list of all the available host to iMS connection types (USB, Ethernet, RS422, etc) allows the application software to search all of them for iMS Systems with one function call, populates the IMSSystem object with details about the attached system and provides it with the internal library interface for communications to occur.

Author

Dave Cowan

Date

2015-11-03

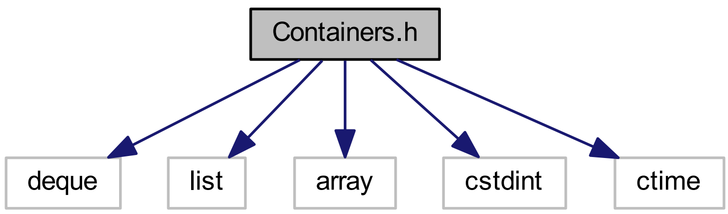
Since

1.0

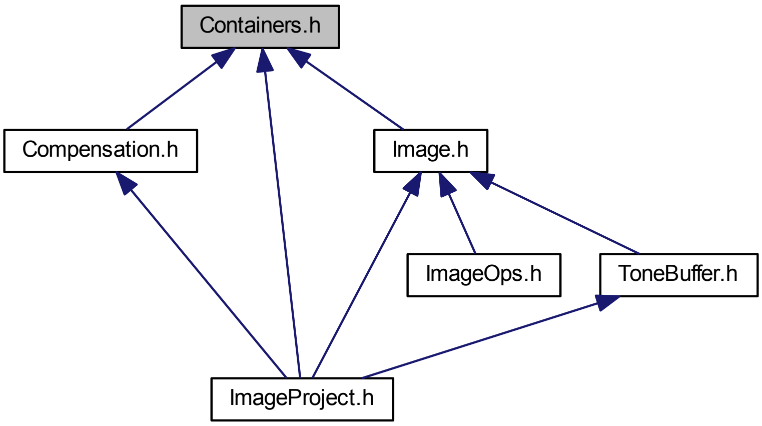
18.4 Containers.h File Reference

Container Classes for storing various types of data related to Image classes and others.

```
#include <deque>
#include <list>
#include <array>
#include <cstdint>
#include <ctime>
```
Include dependency graph for Containers.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::ListBase< T >

  Template Class encapsulating a list object and acting as a base list class for other classes in the library to inherit from.
- class iMS::DequeBase< T >

  Template Class encapsulating a deque object and acting as a base deque class for other classes in the library to inherit from.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.4.1 Detailed Description

Container Classes for storing various types of data related to Image classes and others.

Author

Dave Cowan

Date

2016-10-01

Since

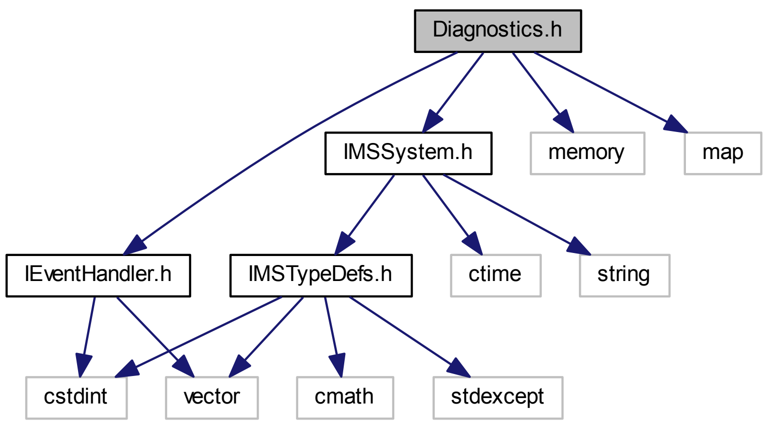
1.3

18.5 Diagnostics.h File Reference

Access diagnostic reporting information about the connected iMS System.

```
#include "IMSSystem.h"
#include "IEventHandler.h"
#include <memory>
#include <map>
```

Include dependency graph for Diagnostics.h:



Classes

- class iMS::DiagnosticsEvents

  All the different types of events that can be triggered by the Diagnostics class.
- class iMS::Diagnostics

  Provides a mechanism for retrieving diagnostics data about the attached iMS System.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.5.1 Detailed Description

Access diagnostic reporting information about the connected iMS System.

The iMS provides a range of diagnostic reporting measures to ensure the continued health and safe function of the Synthesiser, power amplifier and attached acousto-optic devices.

Diagnostics data includes:

- A record of hours recorded while the device was powered up
- The current temperature reading
- Forward current passing through each channel of the amplifier

- Forward power for each amplifier channel
- Reflected power for each amplifier channel

Some of this data may have been stored on the device's non-volatile memory by the factory so the user application can compare against current readings and has a record of how the device performance has changed over time.

Author

Dave Cowan

Date

2016-03-08

Since

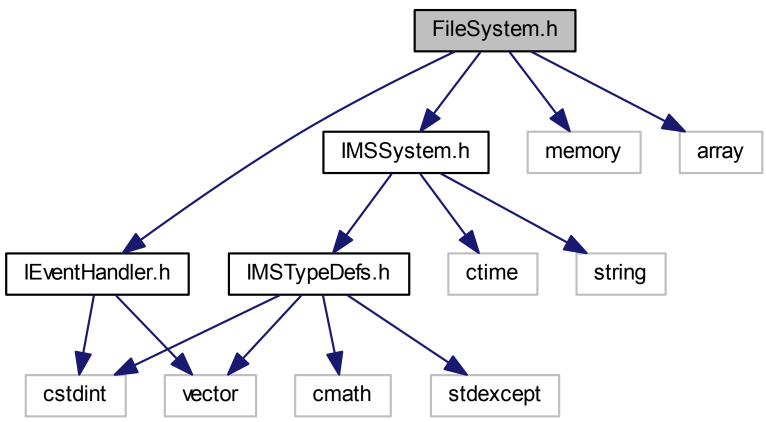
1.1

18.6 FileSystem.h File Reference

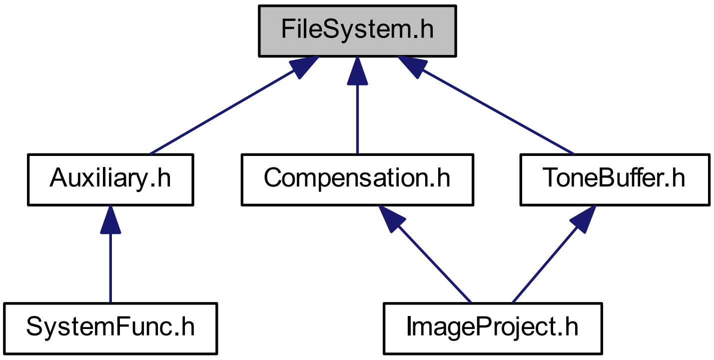
Classes for reading, writing and managing the file system built into an iMS Synthesiser.

```
#include "IMSSystem.h"
#include "IEventHandler.h"
#include <memory>
#include <array>
```

Include dependency graph for FileSystem.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct iMS::FileSystemTableEntry

  Contains all the parameters that uniquely locate a File within the Synthesiser FileSystem.
- class iMS::FileSystemTableViewer

  Provides a mechanism for viewing the FileSystemTable associated with an iMS System.
- class iMS::FileSystemManager

  Provides user management operations for working with Synthesiser FileSystems.
- class iMS::UserFileReader

  Provides a mechanism for retrieving User File data from the Synthesiser FileSystem.
- class iMS::UserFileWriter

  Provides a mechanism for committing User File data to the Synthesiser FileSystem.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

Typedefs

- using iMS::FileSystemIndex = int

  FileSystemIndex represents the entry number for a particular file in the FileSystemTable.

Enumerations

- enum iMS::FileSystemTypes : std::uint8_t {
  iMS::FileSystemTypes::NO_FILE = 0, iMS::FileSystemTypes::COMPENSATION_TABLE = 1, iMS::File←
  SystemTypes::TONE_BUFFER = 2, iMS::FileSystemTypes::DDS_SCRIPT = 3,
  iMS::FileSystemTypes::USER_DATA = 15 }

  All of the different (up to 15) types of file available to the filesystem.
- enum iMS::FileDefault : bool { iMS::FileDefault::DEFAULT = true, iMS::FileDefault::NON_DEFAULT = false }

  Default flag tags a file entry for execution at startup (only one per filetype)

Variables

- const unsigned int iMS::MAX_FST_ENTRIES = 33

  Maximum number of entries that may be stored in the FileSystem.

18.6.1 Detailed Description

Classes for reading, writing and managing the file system built into an iMS Synthesiser.

The Synthesiser includes an area of non-volatile memory which is used for permanent storage of a variety of different data types.

A simple filesystem structure has been defined which arranges and organises the data stored in the memory, allowing the user to keep track of data files and the system to perform relevant functions on the stored data, both on command by the user, and at startup through the setting of default flags.

The filesystem allows up to MAX_FST_ENTRIES different files to be stored in the data area, with each entry being one of 15 different types. Each file can be any size up to the maximum available space in the memory.

The file types so far defined are:

- COMPENSATION_TABLE: contents are used for programming the Compensation Look-Up table
- TONE_BUFFER: contents are used for programming the Local Tone Buffer
- DDS_SCRIPT: contents are DDSScriptRegister sequences for manual programming of the DDS
- USER_DATA: has no functional use on the Synthesiser but can be used for application purposes, e.g. storing application settings, or web pages

The FileSystem has a FileSystemTable associated with it which stores the starting addresses, lengths and types of each file stored in the FileSystem, along with a default flag indicating whether it should be executed at startup and a short (max 8 character) filename for descriptive purposes.

One file of each type may be tagged as a Default, in which case when the Synthesiser initialises, it will attempt to Execute that file. If multiple files are tagged default, the lowest index of each type is executed and any subsequent flags cleared.

File execution as a predictable effect on each type of file, except for USER_DATA, which does nothing (can only by read and written).

At present, the total size of the filesystem on all Synthesiser models is 128kB with 1kB reserved for system use. The FileSystemManager will allocate space in memory for data to be downloaded to but files must always be stored contiguously therefore it is up to the user to ensure the FileSystem does not become excessively fragmented.

All files stored to the FileSystem of all types are prepended with a 2-byte marker symbol which is a requirement of the FileSystem protocol.

When an IMSSystem object is initialised (typically through the ConnectionList::Scan() method), the FileSystemTable is read back and made available for use by classes in this file.

Author

Dave Cowan

Date

2016-01-20

Since

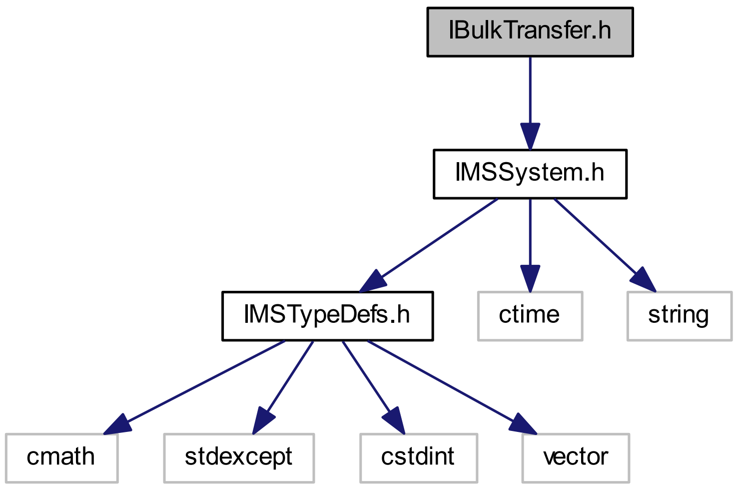
1.1

18.7 IBulkTransfer.h File Reference

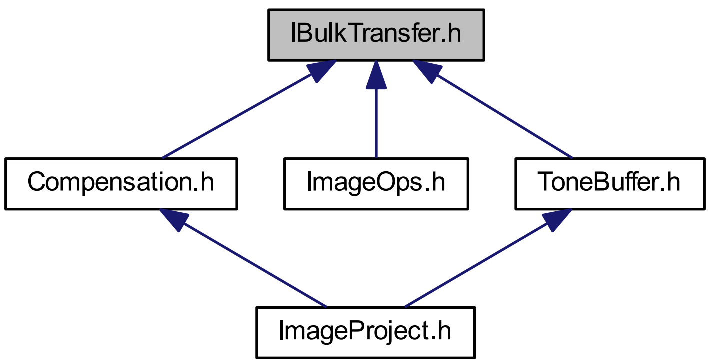
Interface Specification class for sending large binary data objects to the iMS.

```
#include "IMSSystem.h"
```

Include dependency graph for IBulkTransfer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::IBulkTransfer

    Interface Specification class for sending large binary data objects to the iMS.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.7.1 Detailed Description

Interface Specification class for sending large binary data objects to the iMS.

Author

Dave Cowan

Date

2015-11-03

Since

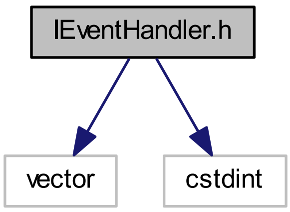
1.0

18.8 IEventHandler.h File Reference

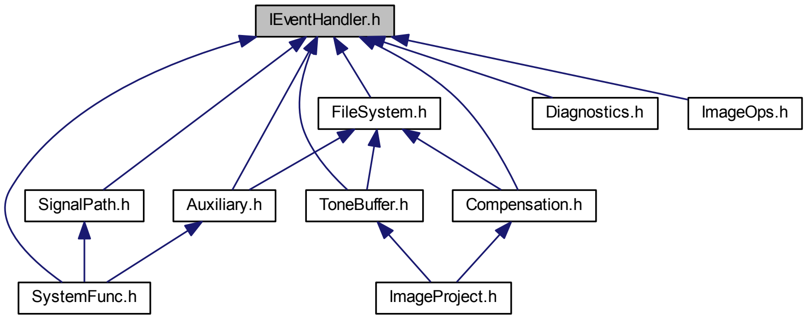
Interface Class for User Application code to receive and process events from the iMS library.

```
#include <vector>
#include <cstdint>
```

Include dependency graph for IEventHandler.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::IEventHandler

  Interface Class for an Event Handler to be defined in User Code and subscribed to library events.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.8.1 Detailed Description

Interface Class for User Application code to receive and process events from the iMS library.

Author

Dave Cowan

Date

2015-11-03

Since

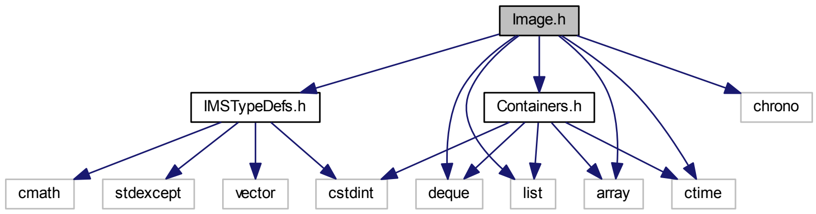
1.0

18.9 Image.h File Reference

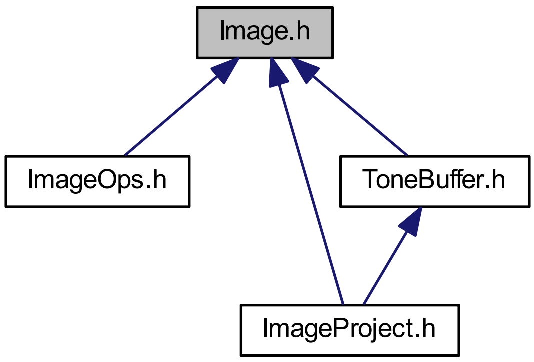
Classes for storing sequences of synchronous multi-channel RF drive data.

```
#include "IMSTypeDefs.h"
#include "Containers.h"
#include <deque>
#include <list>
#include <array>
#include <chrono>
#include <ctime>
```

Include dependency graph for Image.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::ImagePoint

  Stores 4 FAP Triads containing frequency, amplitude and phase data for 4 RF channels.
- class iMS::Image

  A sequence of ImagePoints played out sequentially by the Controller and driven by the Synthesiser.
- struct iMS::ImageTableEntry

  An ImageTableEntry is created by the SDK on connecting to an iMS System, one for each Image that is stored in Controller memory and allocated in the Image Index Table. Further ImageTableEntries are added to the table each time an Image is downloaded to the Controller.
- struct iMS::ImageSequenceEntry

  An ImageSequenceEntry object can be created by application software to specify the parameters by which an Image is played back during an ImageSequence.
- class iMS::ImageSequence

  An ImageSequence object completely defines a sequence to be played back on an iMS Controller in terms by containing a list of ImageSequenceEntry 's plus a terminating action and optional value.
- class iMS::ImageGroup

  An ImageGroup collects together multiple associated images and a single ImageSequence for controlling Image playback order.

Namespaces

- iMS

    The entire API is encapsulated by the iMS namespace.

Typedefs

- using iMS::ImageIndex = int

    Each ImageIndex is an offset into the Image Index Table that uniquely refers to an Image stored in Controller Memory.
- typedef ImageGroup iMS::ImageFile

    For backwards compatibility with code written against SDK 1.2.6 or earlier.

Enumerations

- enum iMS::ImageRepeats { iMS::ImageRepeats::NONE, iMS::ImageRepeats::PROGRAM, iMS::ImageRepeats::FOREVER }

    Each Image can be repeated, either a programmable number of times, or indefinitely.
- enum iMS::SequenceTermAction : std::uint8_t { iMS::SequenceTermAction::DISCARD = 0, iMS::SequenceTermAction::RECYCLE = 1, iMS::SequenceTermAction::STOP_DISCARD = 2, iMS::SequenceTermAction::STOP_RECYCLE = 3, iMS::SequenceTermAction::REPEAT = 4, iMS::SequenceTermAction::REPEAT_FROM = 5 }

    Operation to perform on the completion of the last repeat of the last entry in a Sequence.

18.9.1 Detailed Description

Classes for storing sequences of synchronous multi-channel RF drive data.

Author

Dave Cowan

Date

2015-11-03

Since

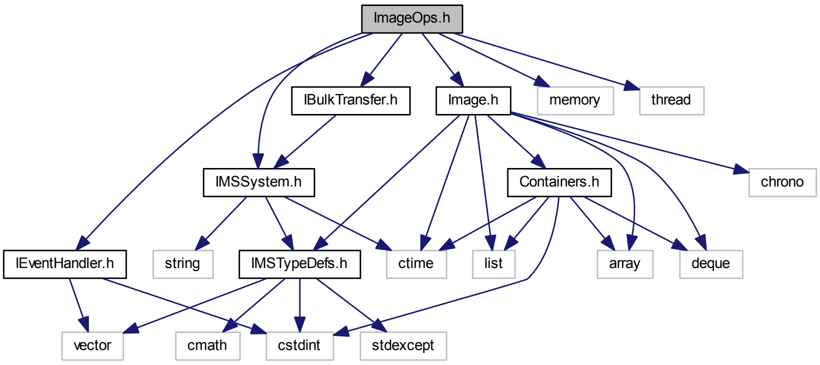
1.0

18.10 ImageOps.h File Reference

Classes for downloading and playback of Image data.

```
#include "IMSSystem.h"
#include "IEventHandler.h"
#include "IBulkTransfer.h"
#include "Image.h"
#include <memory>
#include <thread>
```

Include dependency graph for ImageOps.h:



Classes

- class iMS::ImageDownloadEvents

  All the different types of events that can be triggered by the ImageDownload class.
- class iMS::ImageDownload

  Provides a mechanism for downloading and verifying Images to a Controller's memory.
- class iMS::ImagePlayerEvents

  All the different types of events that can be triggered by the ImagePlayer class.
- class iMS::ImagePlayer

  Once an Image has been downloaded to Controller memory, ImagePlayer can be used to configure and begin playback.
- struct iMS::ImagePlayer::PlayConfiguration

  This struct sets the attributes for the ImagePlayer to use when initiating an Image Playback.
- class iMS::ImageTableViewer

  Provides a mechanism for viewing the ImageTable associated with an iMS System.
- class iMS::SequenceDownload

  This class is a worker for transmitting an ImageSequence to an iMS Controller and joining it to the back of the sequence queue.
- class iMS::SequenceEvents

  All the different types of events that can be triggered by the SequenceManager class.
- class iMS::SequenceManager
- struct iMS::SequenceManager::SeqConfiguration

  This struct sets the attributes for the Sequence to use when initiating an Sequence Playback.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.10.1 Detailed Description

Classes for downloading and playback of Image data.

ImageOps or Image Operations is one of the core features of the iMS Library, providing the user application with the ability to download and verify Images and ImageGroups to an iMS Controller's memory along with the means to configure, start and stop the Controller playback.

Author

Dave Cowan

Date

2015-11-03

Since

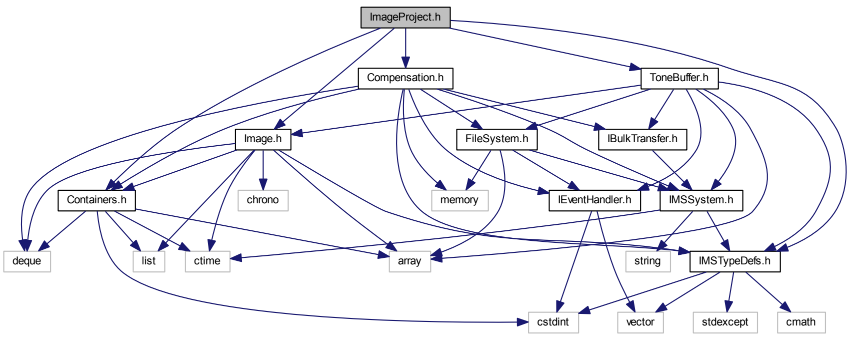
1.0

18.11 ImageProject.h File Reference

Classes for organising Images and associated data.

```
#include "IMSTypeDefs.h"
#include "Containers.h"
#include "Image.h"
#include "Compensation.h"
#include "ToneBuffer.h"
```

Include dependency graph for ImageProject.h:



Classes

- class iMS::ImageGroupList

  A List of ImageGroup's used as a container by ImageProject.
- class iMS::CompensationFunctionList

  A List of CompensationFunction's used as a container by ImageProject.
- class iMS::ToneBufferList

  A List of ToneBuffer's used as a container by ImageProject.
- class iMS::ImageProject

  An ImageProject allows the user to organise their data and store it on the host computer.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.11.1 Detailed Description

Classes for organising Images and associated data.

Author

Dave Cowan

Date

2015-11-03

Since

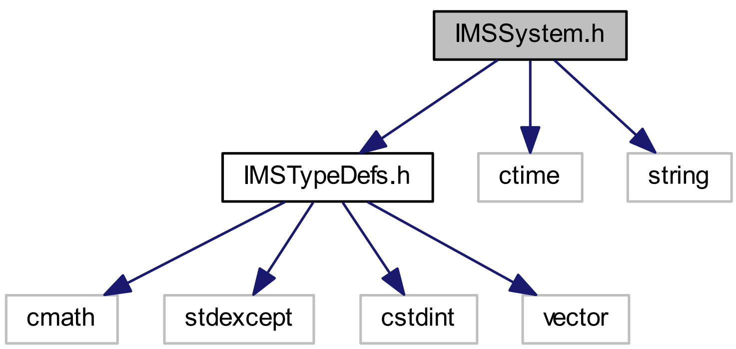
1.0

18.12 IMSSystem.h File Reference

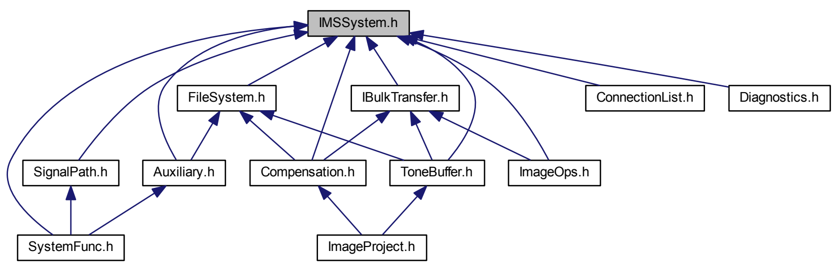
Classes within this group are used to store information about an iMS System and to Connect / Disconnect from it.

```
#include "IMSTypeDefs.h"
#include <ctime>
#include <string>
```

Include dependency graph for IMSSystem.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::IMSOption

  An iMS Synthesiser can support one iMS Option, which adds an additional hardware function to the capabilities of the Synthesiser.
- struct iMS::FWVersion

  Stores the version number of firmware running on iMS hardware.
- class iMS::IMSController

  Stores Capabilities, Description, Model & Version Number of an iMS Controller.
- struct iMS::IMSController::Capabilities

  Returns information about the capabilities of the Controller hardware.
- class iMS::IMSSynthesiser

  Stores Capabilities, Description, Model & Version Number of an iMS Synthesiser.
- struct iMS::IMSSynthesiser::Capabilities

  Returns information about the capabilities of the Synthesiser hardware.
- class iMS::IMSSystem

  An object representing the overall configuration of an attached iMS System and permits applications to connect to it.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.12.1 Detailed Description

Classes within this group are used to store information about an iMS System and to Connect / Disconnect from it.

When a host system is scanned to find attached iMS Systems using ConnectionList::scan(), an IMSSystem object is created for each system that it finds. The system is then probed to discover any Controllers and Synthesisers that belong to it, along with any Option boards that are attached to the Synthesiser (e.g. Frequency doubling). If an AO Deflector or Modulator is connected to the Synthesiser and/or an RF Amplifier, it will also attempt to find out any information it can about those devices.

Once done, the IMSSystem object is returned to the User application. The User can read all of the data that has been created about the iMS System that was discovered, including system structure, capabilities, descriptions, model numbers, serial numbers and firmware versions.

Much of the data that is stored about an iMS System and its components is retrieved from a hardware database which is crossreferenced by identity information read back from the hardware. The hardware database stored as a resource within the library object.

Because the iMS concept is modular in approach, there are many different configurations of an iMS which must all be compatible with the iMS library. Therefore, IMSSystem is vital to many functions within the library to allow them to carry out their objectives according to the capabilities of the attached hardware. As a result, you will see many class constructors which require a const reference to an IMSSystem object so that they have the information about hardware targetting available.

Two further features of IMSSystem are important to notice: the ability to Connect to and Disconnect from a system. No functions can be carried out on an iMS System until it has been identified by the connection scan, and a connection established by calling IMSSystem::Connect().

Author

Dave Cowan

Date

2015-11-03

Since

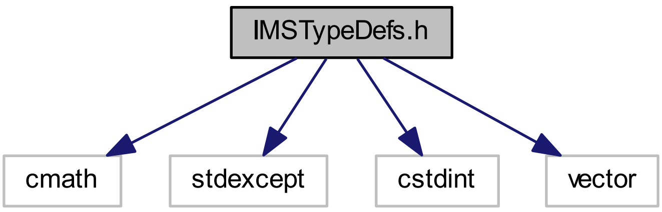
1.0

18.13 IMSTypeDefs.h File Reference

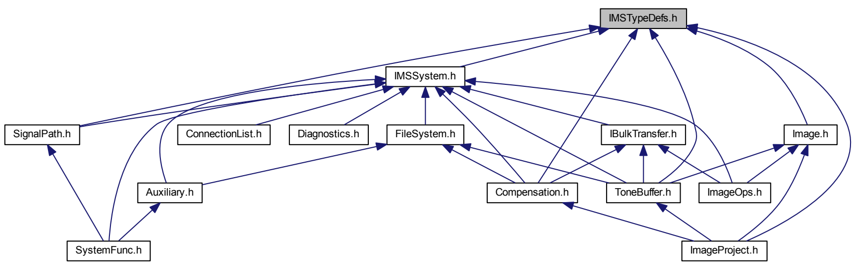
Useful Type Definitions for working with iMS Systems.

```
#include <cmath>
#include <stdexcept>
#include <cstdint>
#include <vector>
```

Include dependency graph for IMSTypeDefs.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::Frequency

  Type Definition for all operations that require a frequency specification.
- class iMS::kHz

  Type Definition for all operations that require a frequency specification in kiloHertz.
- class iMS::MHz

  Type Definition for all operations that require a frequency specification in MegaHertz.

- class iMS::Percent

    Type Definition for all operations that require a percentage specification.
- class iMS::Degrees

    Type Definition for all operations that require an angle specification in degrees.
- struct iMS::FAP

    FAP (Frequency/Amplitude/Phase) triad stores the instantaneous definition of a single RF output.
- class iMS::RFChannel

    Type that represents the integer values 1, 2, 3 and 4, one each for the RF Channels of an iMS Synthesiser.

Namespaces

- iMS

    The entire API is encapsulated by the iMS namespace.

18.13.1 Detailed Description

Useful Type Definitions for working with iMS Systems.

Author

Dave Cowan

Date

2015-11-03

Since

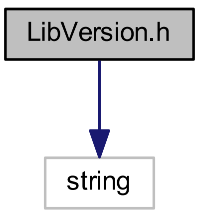
1.0

18.14 LibVersion.h File Reference

Access the API's version information.

```
#include <string>
```

Include dependency graph for LibVersion.h:



Classes

- class iMS::LibVersion

    Access the version information for the API.

Namespaces

- iMS

    The entire API is encapsulated by the iMS namespace.

Macros

- #define IMS_API_MAJOR 1

    Major Version Number.
- #define IMS_API_MINOR 4

    Minor Version Number.
- #define IMS_API_PATCH 2

    Patch Version Number.

18.14.1 Detailed Description

Access the API's version information.

Author

Dave Cowan

Date

2015-11-03

Since

1.0

18.14.2 Macro Definition Documentation

18.14.2.1 #define IMS_API_MAJOR 1

Major Version Number.

The API Major Version number for use in preprocessing directives

18.14.2.2 #define IMS_API_MINOR 4

Minor Version Number.

The API Minor Version number for use in preprocessing directives

18.14.2.3 #define IMS_API_PATCH 2

Patch Version Number.

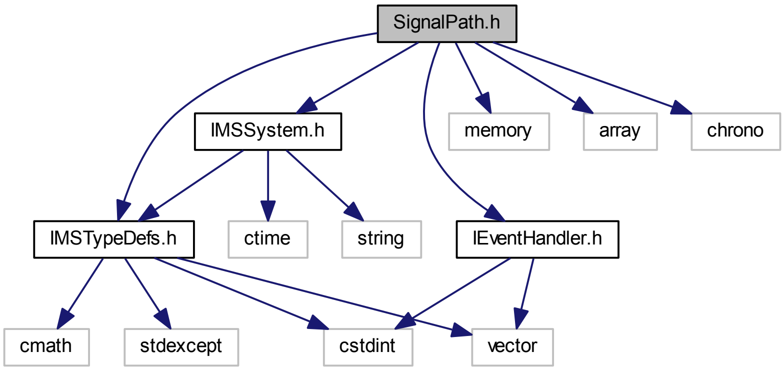
The API Patch Version number for use in preprocessing directives

18.15 SignalPath.h File Reference

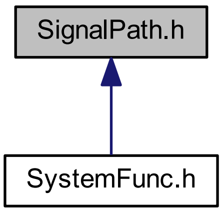
Classes for controlling the flow of data and RF signals through the Synthesiser.

```
#include "IMSSystem.h"
#include "IEventHandler.h"
#include "IMSTypeDefs.h"
#include <memory>
#include <array>
#include <chrono>
```

Include dependency graph for SignalPath.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::SignalPathEvents

  All the different types of events that can be triggered by the SignalPath class.
- class iMS::SignalPath

  Controls Signal routing and other parameters related to the RF output signals.
- struct iMS::VelocityConfiguration

  Sets the parameters required to control the operation of the Encoder Input / Velocity Compensation function.

Namespaces

- iMS

    The entire API is encapsulated by the iMS namespace.

18.15.1 Detailed Description

Classes for controlling the flow of data and RF signals through the Synthesiser.

SignalPath is one of the core features of the iMS Library, providing the user application with the ability to configure the routing of signal data (frequency, amplitude, phase and synchronous output busses), switching in and out functions that affect the signal path, and control RF signal flow, such as DDS output power and modulation control

Author

Dave Cowan

Date

2015-11-03

Since

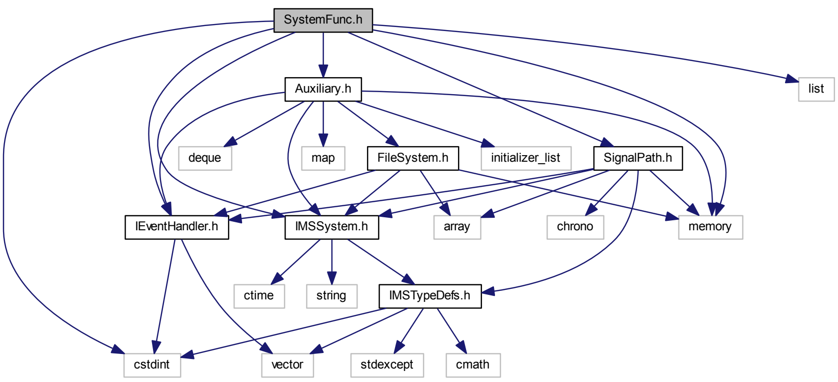
1.0

18.16 SystemFunc.h File Reference

Classes for performing system functions not directly related to RF signal generation and output.

```
#include "IMSSystem.h"
#include "IEventHandler.h"
#include "SignalPath.h"
#include "Auxiliary.h"
#include <cstdint>
#include <list>
#include <memory>
```

Include dependency graph for SystemFunc.h:



Classes

- class iMS::SystemFuncEvents

  All the different types of events that can be triggered by the SystemFunc class.
- class iMS::SystemFunc

  Provides System Management functions not directly related to RF signal generation or signal path control.
- struct iMS::StartupConfiguration

  The Synthesiser stores in its non-volatile memory a set of configuration values that are preloaded on startup.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

18.16.1 Detailed Description

Classes for performing system functions not directly related to RF signal generation and output.

Author

Dave Cowan

Date

2015-11-03

Since

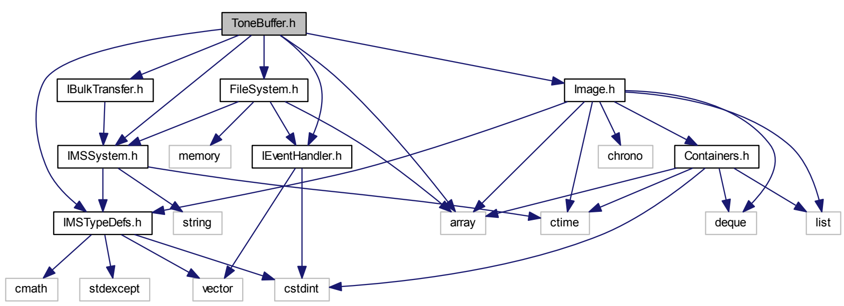
1.0

18.17 ToneBuffer.h File Reference

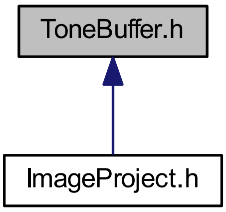
Class for storing an array of Synthesiser tones.

```
#include "IMSSystem.h"
#include "IEventHandler.h"
#include "IMSTypeDefs.h"
#include "IBulkTransfer.h"
#include "Image.h"
#include "FileSystem.h"
#include <array>
```

Include dependency graph for ToneBuffer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class iMS::ToneBufferEvents

  All the different types of events that can be triggered by the ToneBuffer and ToneBufferDownload classes.
- class iMS::ToneBuffer

  An array of 4-channel FAP Tones stored in memory on the Synthesiser.
- class iMS::ToneBufferDownload

  Provides a mechanism for downloading ToneBuffer's to a Synthesiser's LTB memory.

Namespaces

- iMS

  The entire API is encapsulated by the iMS namespace.

Typedefs

- using iMS::TBEntry = ImagePoint

  TBEntry is synonymous with ImagePoint An entry in the Tone Buffer contains four FAPs, one per output channel and is therefore comparable to a single ImagePoint making up one entry in an Image.

18.17.1 Detailed Description

Class for storing an array of Synthesiser tones.

Author

Dave Cowan

Date

2016-02-24

Since

1.0

Index