QDC-09-13-001 v1.0 (2024-09-11)

User Guide

Isomet Modular Synthesiser (iMS)

Developing C# Applications with iMS

© 2024 Isomet (UK) Limited. All rights are reserved; reproduction in whole or in part is prohibited without written consent of the copyright owners.

Contents

© 2024 Isomet (UK) Limited –Company Confidential

1 Introduction

1.1 Background

The Isomet Modular Synthesiser (iMS) System is a suite of hardware and software components that allow any developer who is integrating Acousto-Optic (AO) components into their system to rapidly design an RF driver to control the AO with precision and sophistication.

Isomet supply free of charge a Software Development Kit (SDK) containing libraries and application software to speed the development and integration of the AO driver into the end system.

1.2 C++ Library

The library code in the iMS SDK is extensively tested and abstracts away from application software all the low level details of implementation and messaging requirements, such as message formatting, CRC error handling, interface status and messaging timeouts. It also supplies a Hardware Abstraction Layer so that application software will work irrespective of whether connection is made to the iMS through USB, Ethernet or RS422.

The library presents a well documented and stable Application Programmers' Interface (API) so that users can write their own programs to control the iMS using easy to understand function calls such as, for example, `ImagePlayer.Play()`.

Figure 1 shows the logical structure of how this works in the end system.

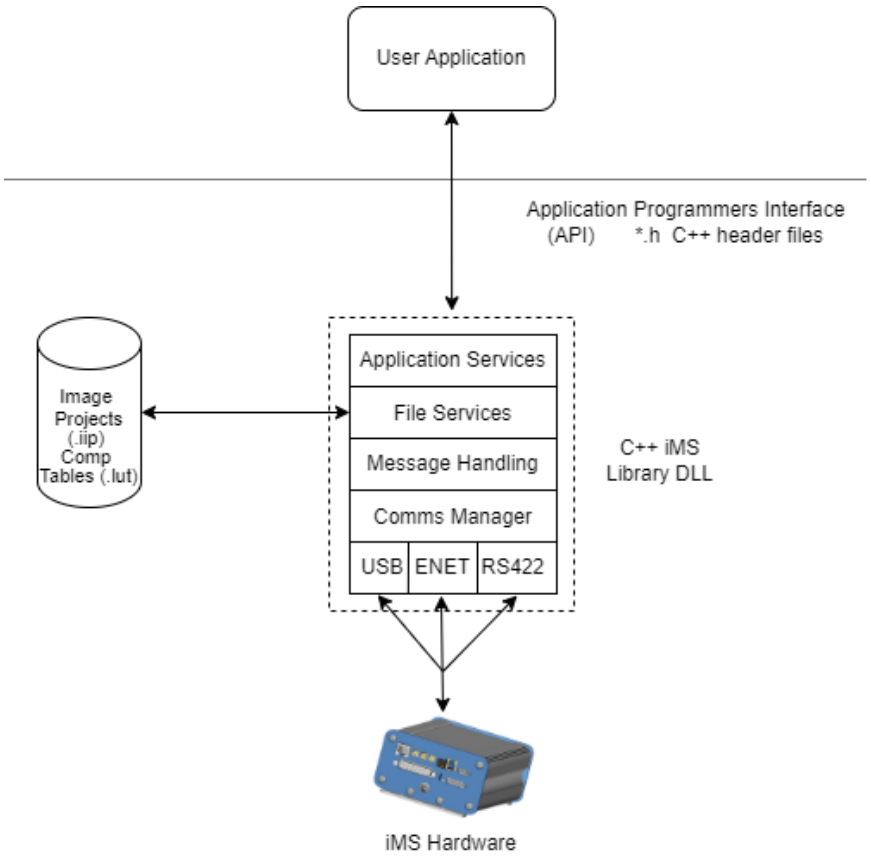

Figure 1 C++ Native Application and Library Usage

The iMS library is written in C++ since this provides the high level of control and interaction with the OS kernel APIs required to drive the iMS hardware in a time-efficient manner.

© 2024 Isomet (UK) Limited –Company Confidential

Consequently, the API is also written in C++, and applications should use C++ natively wherever possible as it offers the most efficient handling of data structures from application, through the library, to the driver code, hardware and back again.

Complex applications that require precise timing or rely on callbacks – especially those using iMS Sequences – are strongly recommended to be written in C++.

However, Isomet understand there are disadvantages to C++ usage and that alternative languages are sometimes preferable. For that reason, we also provide solutions to programming the iMS using the C# language.

1.2.1 Advantages of C++

- Fast and efficient. Because data structures are created and managed by the user and passed natively to the iMS library without requiring any marshalling or translation, C++ is the fastest way to control the iMS. This is especially important for systems that require a high throughput of data to control the AO system, such as when using sequences to rapidly program the AO driver for multiple patterns.

- Cross-platform portable. C++ is not strongly associated with any one host platform or operating system and Isomet make available the iMS library for use in both Microsoft Windows and Ubuntu Linux, compiled from the same source code base. It has even been compiled for embedded RTOS systems such as QNX Neutrino. Application software written on one platform can usually be compiled on others provided any associated libraries and frameworks used by the application author are available on those platforms.

1.2.2 Disadvantages of C++

- Steep learning curve. C++ is not a straightforward language to learn so developers unfamiliar with it may take some time to come up to speed.

- Lack of modern UI frameworks. While some third party commercial and open source C++ GUI libraries do exist, most host operating systems have shifted to using markup language (e.g. XML, XAML) for UI design and managed code for the UI functionality (such as .NET for Microsoft Windows, or Wayland for Ubuntu Linux).

1.3 Using C# / .NET with iMS

To counter the disadvantages of C++ application development as described above – and particularly to make it easier to develop .NET GUI applications under Microsoft Windows for the iMS system, Isomet have designed two approaches for C# application development for iMS:

1. Using a C# library wrapper. Instead of including in your application the .h header files that represent the C++ API and linking with the C++ dll, you just reference the iMSNET.dll and use its classes and methods in almost the same way.

2. Using the iMS Hardware Server. The hardware server is a C++ application that connects to the iMS and provides services to client applications that can be written in C# (or other languages). Remote procedure calls (RPCs) connect the client applications to the hardware server in a language-independent way using a separately documented message API.

The following sections describe the two approaches.

© 2024 Isomet (UK) Limited –Company Confidential

2 iMSNET Managed Wrapper

This is the simpler of the two approaches.

iMSNET.dll is a C# library that can be added directly as a reference into your own custom C# and .NET applications. Internally, iMSNET uses the same native C++ library function calls as if using the C++ library directly. iMSNET adds a wrapper around the C++ library so that each of the C++ functions in the API can be called from C# code using identical or nearly identical semantics.

This is achieved using a Microsoft technology called Platform Invoke (or P/Invoke). You can read more about Platform Invoke here:

https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke

but essentially it is a mechanism for accessing classes and functions in unmanaged (i.e. C++) libraries from managed code (i.e. C#).

iMSNET.dll contains embedded copies of the unmanaged C++ libraries and all of the P/Invoke implementation required to expose the core API as a C# managed API. The resulting managed API has a list of classes and functions that bear the exact same signature as the original C++ unmanaged API. For that reason, the C# API is not separately documented as it can be inferred from the standard API documentation with intuition.

It is also possible to use the Object Browser in Visual Studio to browse the list of available classes and functions (by double clicking the 'iMSNET' reference in the Solution Explorer). For example, see the function definition for `IMSSystem.Connect()` below. Or use IntelliSense when writing code to complete the function for you.

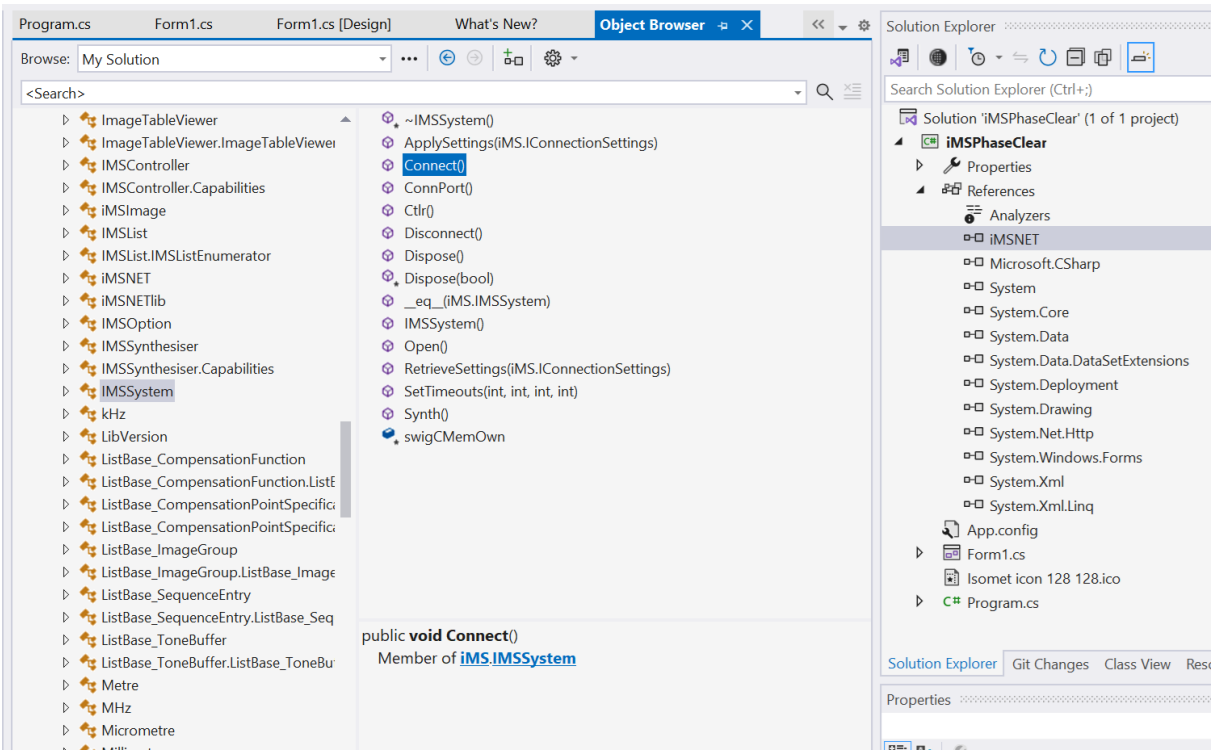

Figure 2 iMSNET Object Browser in Visual Studio

As with the C++ library, all classes and function in the iMSNET library are encapsulated in the iMS namespace.

© 2024 Isomet (UK) Limited –Company Confidential

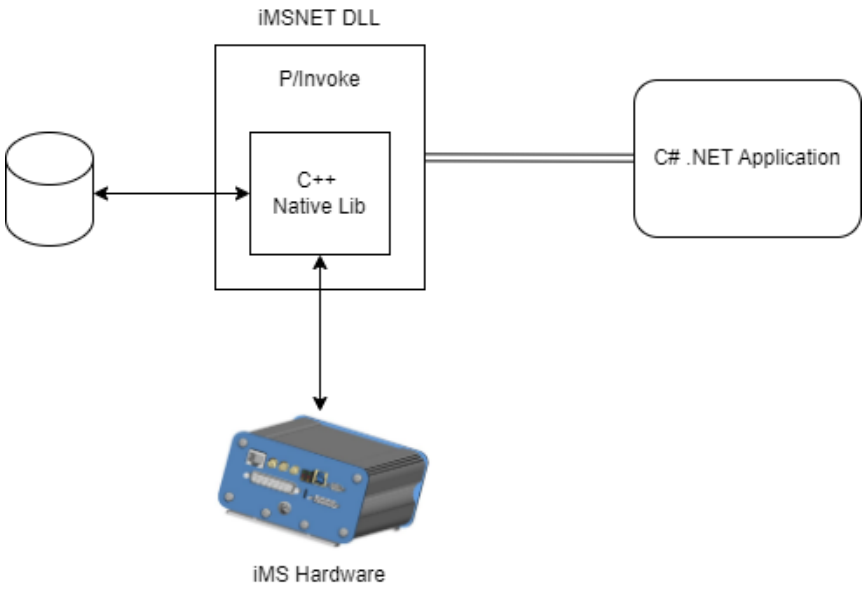

Figure 3 iMSNET Library added as a Reference to C# Application

Figure 3 shows a system that uses the iMSNET library which is added as a Reference into the C# .Net Application Project. The library embeds the C++ Native library and all its features and functions as indicated in Figure 1.

2.1 Initialisation

An important step that must always be performed with the iMSNET library that is not required with the C++ library is to initialize it.

This is because the iMSNET library must first export its embedded copy of the unmanaged DLL to a known location before it can then interact with them (called Interop by Microsoft). In the startup code of your C# application, you must have this line and ensure it is only called once:

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

2.2 iMSImage Class

One important case to note where the class signature is different from the unmanaged C++ library is the iMSImage class which is the C# version of the C++ Image class. The different naming is used to distinguish it from other 'Image' classes which are routinely used especially in graphics libraries.

2.3 Garbage Collection

One of the distinguishing features of a managed environment like .NET is that the responsibility for acquiring and releasing resources such as memory allocation passes from the developer to the system. In traditional native C++ applications, when the developer creates an object in memory, it is down to him/her to remember to release the resource when it is no longer required. Failure to do so results in a memory leak and can eventually crash programs. In .NET, the runtime environment monitors resource usage such that when an object created by a program is no longer required, it is automatically released.

The Garbage Collector (GC) is the mechanism by which this occurs. Every object created by the program has a reference count which is incremented when it is passed to another function and decremented when it goes out

© 2024 Isomet (UK) Limited –Company Confidential

of scope. When the reference count reaches zero and the Garbage Collector is run (which happens at regular intervals), the object is disposed and the resources released.

Unfortunately the GC cannot 'see' into unmanaged code. There is a risk therefore that an object created in managed code is passed into a class method in unmanaged code where it is stored, the GC then runs and can't see any further uses of the object and deletes it. The stored reference in the unmanaged library then has a dangling reference and an exception occurs. There are lots of places in the iMS C++ library where objects are stored in the DLL.

iMSNET C# developers need to be aware of this issue when writing applications and ensure that either a reference to a data object created in managed space is maintained before passing it to a iMSNET function or the object is 'pinned' to prevent the GC from deleting or moving the object.

To pin a managed object, use GCHandle.Alloc(). The objectToBePinned remains pinned until you call Free() on the handle:

```
// Pin "objectToBePinned"
GCHandle handle = GCHandle.Alloc(objectToBePinned, GCHandleType.Pinned);

do_something(objectToBePinned);

// Unpin "objectToBePinned"
handle.Free();
```

Or:

```
GCHandle handle = GCHandle.Alloc(objectToBePinned, GCHandleType.Pinned);

do_something(handle.AddrOfPinnedObject());

// NOTE: Usually you wouldn't free the handle here if "do_something()"
//   stored the pointer to "objectToBePinned".
handle.Free();
```

© 2024 Isomet (UK) Limited –Company Confidential

3 iMS Hardware Server

The alternative approach for writing C# and .NET GUI application in Windows is to make use of the iMS Hardware Server which is supplied as part of the iMS SDK.

The iMS Studio application is an example of a C# Windows GUI (built using WPF) that uses the iMS Hardware Server to communicate with an iMS System.

The iMS Hardware Server is a standalone C++ application that is built upon the C++ native DLL. It is designed to be a background process with no user interaction apart from a terminal display output and it appears to do nothing when it is first started.

However, the server supports a protocol called Google Remote Procedure Call (gRPC – it has nothing to do with the search engine it just happens to be developed by Google engineers!) through which it makes available a range of 'services' which other applications can use to communicate with the server and instruct it to do something with an attached iMS system, for example connect/disconnect or play an image.

Read more about gRPC here:

https://grpc.io/docs/what-is-grpc/introduction/

gRPC uses 'protocol buffers'. These are simply short messages that can be passed between the client and the server application and take the form of request/response pairs. By default, communication on gRPC is always synchronous, but asynchronous messaging is also supported allowing the client to send long sequences of messages efficiently to the server, such as image or compensation data.

The advantage of using protocol buffers is that they are not tied to a single language, they can use any of the languages supported by gRPC[1]. That means that while the hardware server is built in C++, the GUI application can be designed in C# and have full access to the iMS system through the server without any of the problems associated with containing the C++ library in a managed wrapper.

Other advantages of the iMS Hardware Server approach include:

- Multiple client applications (up to 10) can communicate with the server simultaneously. This means you can build different applications for different purposes and they can all talk with an iMS system without having to worry about contention.

- The client and server applications do not need to reside on the same machine. The two parts communicate over a TCP/IP port so as long as the client(s) and server can see each other on the network, the system will work, and remains low latency. This kind of distributed architecture could be useful for systems where the backend is embedded in an industrial system and needs to be controlled by an operator in a different part of the building or even on another device type, e.g. a tablet.

[1] Currently Java, C++, Dart, PHP, Python, Objective-C, C#, Ruby, JavaScript, Node.JS and Go (https://grpc.io/docs/#official-support)

© 2024 Isomet (UK) Limited –Company Confidential

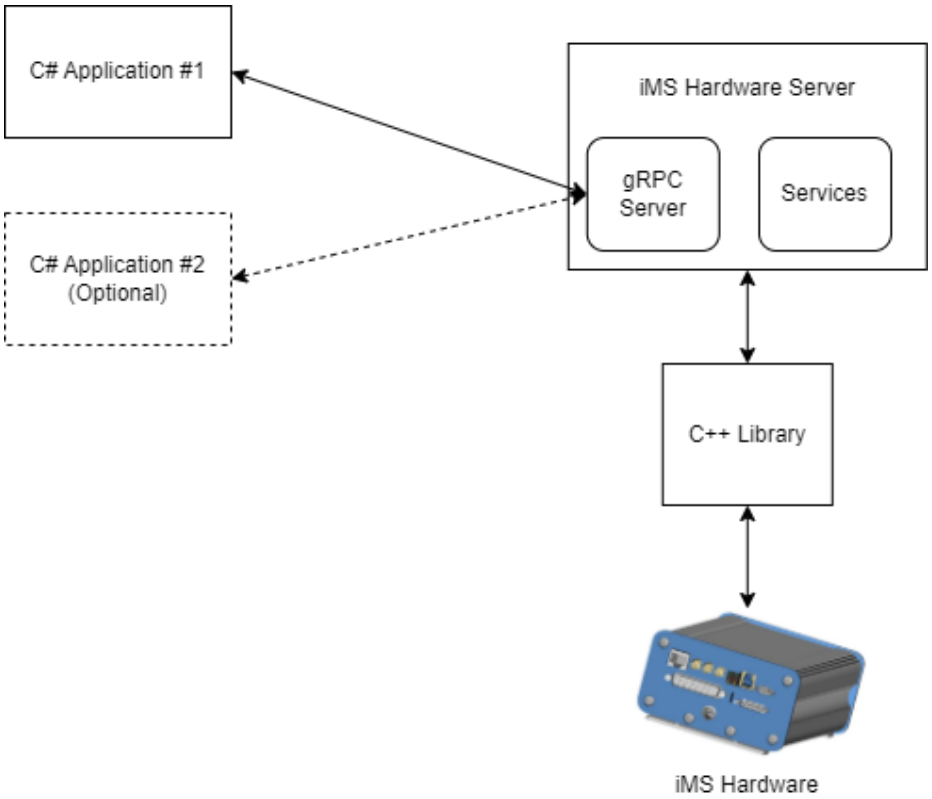

Figure 4 iMS Hardware Server Usage

Services provided by the iMS Hardware Server include:

- A Hardware List service that scans for attached iMS Systems, returns information about them and allows the application code to connect to one of them.
- A Signal Path service that allows users to control such things as output power, amplifier enable, synchronous digital output delay etc.
- An Image Player service that permits user applications to download and play iMS Images
- An Image Table Viewer service that returns information about images present on the iMS
- A Compensation Table service for downloading compensation look up tables
- A Tone Buffer service that downloads and utilizes iMS tone buffers.

All services are documented in HTML and PDF format and included with the iMS SDK.

3.1 Hybrid Applications

Note that it is perfectly possible – and often desirable – to use both approaches, using the iMSNET managed library to read image projects from disk, create images or look up tables, and then use the gRPC service to send the completed data to the iMS HW Server for download to the iMS System.

© 2024 Isomet (UK) Limited –Company Confidential

Annex A Change History

Version	Authors	Date	Status	Comment
1.0	Dave Cowan	11-Sept-2024	In work	

© 2024 Isomet (UK) Limited –Company Confidential