



# *Malibu Library User's Manual*

# *Malibu Library User's Manual*

---

The Malibu Library User's Manual was prepared by the technical staff of Innovative Integration on December 10, 2013.

For further assistance contact:

Innovative Integration  
2390-A Ward Ave  
Simi Valley, California 93065

PH: (805) 578-4260

FAX: (805) 578-4225

email: [techsprt@innovative-dsp.com](mailto:techsprt@innovative-dsp.com)

Website: [www.innovative-dsp.com](http://www.innovative-dsp.com)

This document is copyright 2013 by Innovative Integration. All rights are reserved.

\$/Distributions/Components/Malibu/Documentation/OO\_Manual/Malibu.pdf

Rev 1.4

---

# Table of Contents

<b>Chapter 1. Introduction.....</b>	<b>10</b>
Real Time Solutions!.....	10
Vocabulary.....	10
What is Malibu? .....	10
What is Microsoft MSVC?.....	11
What is Qt?.....	11
What is C++ Builder?.....	11
What kinds of applications are possible with Innovative Integration hardware?.....	11
Why do I need to use Malibu with my Baseboard?.....	12
Finding detailed information on Malibu.....	12
Online Help.....	12
Innovative Integration Technical Support.....	12
Innovative Integration Web Site.....	13
Typographic Conventions.....	13
<b>Chapter 2. A Tour of Malibu.....</b>	<b>14</b>
Malibu Architecture.....	14
High Performance Code.....	14
Synergistic operation with DSP co-processor boards.....	14
A Portable Class Library.....	15
Class Groups In Malibu.....	15
Operating System Independence.....	16
Malibu Namespaces.....	17
Interface Classes in Malibu.....	18
Event Callbacks in Malibu.....	19
UI Thread Synchronization.....	20
Using the Malibu Library.....	20
Creating a Streaming Application in Visual C++.....	21
Creating the Malibu Objects.....	21
Initializing Object Properties and Events.....	22
Event Handler Code.....	23
Loading COFF Files.....	25
Loading Logic Files.....	25
Script Files.....	26
<b>Chapter 3. Creating Applications using an IDE.....</b>	<b>27</b>
Creating a Malibu Project in Microsoft Visual Studio 7.....	27
Enabling Auto-Saving of Projects.....	27
Creating a Malibu Project.....	28
Other Configuration Requirements.....	29
Creating a Malibu Project in Microsoft Visual Studio Vc8/Vc9.....	30
Enabling Auto-Saving of Projects.....	31
Creating a Malibu Project.....	31
Other Configuration Requirements.....	33
Creating a Malibu Application using Nokia QtCreator.....	34
Install and/or Rebuild Qt Library.....	34
Creating a Malibu Project in Borland Developer's Studio/Turbo C++.....	38
Enabling Auto-Saving of Projects.....	38

---

Default Project Options which should be Changed.....	39
Creating Projects in Borland C++ Builder 6.0.....	41
Enabling Auto-Saving of Projects.....	41
Creating a Malibu Project.....	41
<b>Chapter 4. The Malibu Framework Library.....</b>	<b>46</b>
Framework Support Classes.....	46
Thinking.....	46
<b>Chapter 5. The Malibu OS Library.....</b>	<b>47</b>
Thread Support Classes.....	47
Threads.....	47
Signals.....	47
Resource Control.....	48
Inter-Thread Communications .....	48
Operating-System .....	48
<b>Chapter 6. The Malibu Utility Library.....</b>	<b>49</b>
Buffer Classes.....	49
Message Packet Classes.....	49
Disk I/O Classes.....	49
Data Recording and Playback Classes.....	50
System Components.....	50
File Support Methods.....	51
String Support .....	51
Matlab Interface Classes.....	51
Data Set Classes.....	52
<b>Chapter 7. The Malibu Hardware Library.....</b>	<b>53</b>
Target I/O Streaming Classes.....	53
Interface Classes.....	54
Timebase Classes.....	54
Hardware Support Classes.....	54
Hardware Register Classes.....	55
<b>Chapter 8. The Malibu Analysis Library.....</b>	<b>56</b>
Statistical Analysis Classes.....	56
Signal Processing Classes.....	56
Signal Generation Classes.....	57
<b>Chapter 9. The Malibu PCI Library.....</b>	<b>59</b>
PCI Baseboard Classes.....	59
Baseboards and PMC Modules.....	59
PMC Module Classes.....	60
XMC Module Classes.....	60
<b>Chapter 10. The Malibu Ethernet Library.....</b>	<b>62</b>
Baseboard Classes.....	62
<b>Chapter 11. Packet Polling Library.....</b>	<b>63</b>
PacketPollMgr Class.....	63

Con/Destructor.....	63
Events.....	63
Interfaces.....	63
Start()/Stop().....	63
Send()/Recv().....	64
SetFifoDacDelay().....	64
PacketPollDataThread Class.....	64
PacketPollDataEvent Class.....	64
PacketPollQueue Class.....	65
<b>Chapter 12. Writing Custom Applications.....</b>	<b>66</b>
The Snap Example.....	66
Tools Required.....	66
Program Design.....	67
The Host Application .....	67
User Interface.....	67
Configure Tab.....	67
Setup Tab.....	68
Stream Tab.....	70
Host Side Program Organization.....	71
ApplicationIo.....	72
Initialization.....	72
Starting Data flow.....	76
Handle Data Available.....	79
The Wave Example.....	80
User Interface.....	80
The Setup Tab.....	80
The Waveform Tab.....	81
Stream Tab.....	82
ApplicationIo.....	82
Stream Preconfigure.....	82
Start Streaming.....	83
Data Required Event Handler.....	86
FillWaveformBuffer().....	87
<b>Chapter 13. Malibu Buffer Classes.....</b>	<b>90</b>
Buffer Design Decisions.....	90
Design Decision #1 – A “Typeless” Buffer class.....	90
Design Decision #2 – Data Access Datagrams.....	90
Design Decision #3 – Predefined Access Datagram Classes.....	91
Design Decision #4 – IPP Datagram Classes.....	91
Buffer Internals.....	92
Data Buffers : The Innovative::Buffer Class.....	92
Buffer Class (Buffer_Mb.h).....	93
Holding Template (Buffer_Mb.h).....	93
MessageDatagram (Buffer_Mb.h).....	93
Buffer Data Access.....	93
Access Template Features.....	94
Template AccessDatagram<T> (AccessDatagrams_Mb.h).....	94
Template Class DatagramIterator (AccessDatagrams_Mb.h).....	95
Interface Class IDatagrammable (AccessDatagrams_Mb.h).....	97

Interface Class Iterable (AccessDatagrams_Mb.h).....	97
Standard Implementation Classes.....	97
IntegerDG (BufferDatagrams_Mb.h).....	97
UIntegerDG (BufferDatagrams_Mb.h).....	97
FloatDG (BufferDatagrams_Mb.h).....	98
ShortDG (BufferDatagrams_Mb.h).....	98
ComplexDG (BufferDatagrams_Mb.h).....	98
CharDG (BufferDatagrams_Mb.h).....	98
IPP Implementation Classes.....	98
IppCharDG (IppCharDG_Mb.h).....	98
IppComplexDG (IppComplexDG_Mb.h).....	98
IppFloatDG (IppFloatDG_Mb.h).....	98
IppIntegerDG (IppIntegerDG_Mb.h).....	99
IppShortDG (IppShortDG_Mb.h).....	99
Special Purpose Datagrams.....	99
PacketBufferHeader (BufferHeader_Mb.h).....	99
IDatagram Template (Datagram_Mb.h).....	99
MessageDatagram (Buffer_Mb.h).....	99
Internal Datagrams (various CPPs).....	99
Guidelines for Converting to new Buffers.....	100
Translate all buffers to be Innovative::Buffer.....	100
Convert array operators on buffers.....	100
Size Issues.....	100
Datagrams and Iterators are Disposable.....	101
Packet Stream Header Access.....	101
Porting Buffer Access Modes #1 – The Aztec Model.....	101
Porting Buffer Access Modes #2 – Buffer [] operator.....	102
Porting Buffer Access Modes #3 -- Applying a Structure to Buffer Content.....	104

## **Chapter 14. Using the X6 Family Baseboards in Malibu.....105**

Overview.....	105
Buffers and their Type.....	105
Buffer Conversions.....	106
Applying a Type.....	106
Buffer Sizing Template Functions.....	107
Holding<T>() :: Sizing a buffer to hold N elements.....	107
CouldHold<T>() :: Elements in a current buffer.....	107
Buffer Header Datagrams.....	107
Buffer Trailer Datagrams.....	108
Buffer Header/Trailer Utility Functions.....	108
Clear Functions.....	108
Header Correctness Functions.....	108
Trailer Correctness Functions.....	109
Header Size Conversion.....	109
New Streaming Object – VitaPacketStream.....	109
Connection.....	109
Native Buffer Methods.....	109
Send and Recv Methods.....	110
Stream Data Notification Events.....	110
Direct Data Mode.....	110
Working with Vita Packet Streams.....	110

---

VitaPacketParser – Parsing Input Packets.....	110
VitaPacketPacker – Filling Output Packets.....	112
<b>Chapter 15. Vita Packet Format.....</b>	<b>114</b>
Overview.....	114
X6 Velocia Packets.....	114
Packet Header Format.....	114
Packet Data Format.....	115
X6 Vita Packets.....	115
Packet Header Format.....	116
VITA Header IF word.....	116
VITA Header SID word.....	117
VITA Header Class OUI Word.....	117
Vita Header Class Info Word.....	117
Vita Header Timestamp – Integer Seconds Word.....	117
Vita Header Timestamp – Fractional Seconds High and Low Words.....	117
Vita Packet Trailer Format.....	118
VITA Trailer Word.....	118
State and Event Bits and Enable Bits.....	118
Bits 20-23.....	118
Context Packet Count.....	118
Padding.....	118
<b>Chapter 16. Creating a Custom FMC Module.....</b>	<b>120</b>
FMC Modules and Malibu.....	120
The IFmcDaughterCard Interface Class.....	120
BoardStatus().....	121
InitCardPower() and RemoveCardPower().....	122
ConstructCustomParts().....	122
OpenHardware().....	122
PreconfigureHardware().....	122
ConfigureHardware().....	123
StreamStopHardware().....	123
CloseHardware().....	123
Memory Spaces and Registers.....	123
The FICL Interface.....	124
<b>Chapter 17. Interfacing to Software Applications via a DLL.....</b>	<b>126</b>
Overview.....	126
Development Approach.....	126
Example Source.....	126
<b>Chapter 18. Using the embedded FICL interpreter.....</b>	<b>128</b>
Ficl Features.....	128
A Beginners Guide.....	129
Using Ficl.....	129

---

## List of Tables

Table 1. Path Spec Options.....	29
Table 2. Path Spec Options.....	32
Table 3. Path Spec Options.....	44
Table 4. Development Tools for the Windows Snap Example.....	66
Table 5. Basic Buffer Datagram Classes.....	91
Table 6. IPP Function Datagrams.....	91
Table 7. PacketBufferHeader Field Methods.....	99
Table 8. X6 Velocia (Velo) Packet Header.....	114
Table 9. Velocia Header Word.....	114
Table 10. Vita Packet Format.....	115
Table 11. Timestamp Integer Seconds Options .....	116
Table 12. Timestamp Fractional Seconds Options .....	117
Table 13. Padding Example .....	119
Table 14. Maximum Padding for X6 Boards.....	119



---

---

## List of Figures

---

## Chapter 1. *Introduction*

---

### *Real Time Solutions!*

---

Thank you for choosing Innovative Integration, we appreciate your business! Since 1988, Innovative Integration has grown to become one of the world's leading suppliers of DSP and data acquisition solutions. Innovative offers a product portfolio unrivaled in its depth and its range of performance and I/O capabilities .

Whether you are seeking a simple DSP development platform or a complex, multiprocessor, multichannel data acquisition system, Innovative Integration has the solution. To enhance your productivity, our hardware products are supported by comprehensive software libraries and device drivers providing optimal performance and maximum portability.

Innovative Integration's products employ the latest digital signal processor technology thereby providing you the competitive edge so critical in today's global markets. Using our powerful data acquisition and DSP products allows you to incorporate leading-edge technology into your system without the risk normally associated with advanced product development. Your efforts are channeled into the area you know best ... your application.

### *Vocabulary*

---

#### **What is Malibu?**

Malibu is the Innovative Integration-authored component suite, which combines with the Microsoft or Embarcadero (Windows) and GNU C++ compilers (Windows/Linux) and the MS Visual Studio and QtCreator IDEs to support programming of Innovative hardware products under Windows and Linux. Malibu supports both high-speed data streaming plus asynchronous mailbox communications between the DSP and the Host PC, plus a wealth of host functions to visualize and post-process data interchanged with the target DSP.

See the [Malibu User's Guide](#) for detailed information on this comprehensive library.

---

### **What is Microsoft MSVC?**

MSVC is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Malibu extends the MSVC IDE through the addition of dynamically-created MSVC-compatible C++ classes specifically tailored to perform real-time data streaming functions.

### **What is Qt?**

Qt (pronounced officially as "cute (KYOOT)" although commonly referred to as "Q.T. (KYOO-TEE)") is a cross-platform application development framework widely used for the development of GUI programs (in which case it is known as a widget toolkit), and also used for developing non-GUI programs such as console tools and servers. Qt is most notably used in Google Earth, KDE, Opera (before 10.60 version), OPIE, Skype, VLC media player and VirtualBox. It is owned and supported by Digia Qt, which came into being after the acquisition of Qt from (which followed Nokia's acquisition of the Norwegian company Trolltech, the original producer of Qt, on June 17, 2008).

Qt uses standard C++ but makes extensive use of a special pre-processor (called the Meta Object Compiler, or moc) to enrich the language. Qt can also be used in several other programming languages via language bindings. It runs on all major platforms and has extensive internationalization support. Non-GUI features include SQL database access, XML parsing, thread management, network support, and a unified cross-platform API for file handling.

Distributed under the terms of the GNU Lesser General Public License (among others), Qt is free and open source software. All editions support a wide range of compilers, including the GCC C++ compiler and the Visual Studio suite..

So what is special about Qt compared with other cross-platform GUI toolkits? Qt gives you a single, easy-to-use API for writing GUI applications on multiple platforms that still utilize the native platform's controls and utilities. Link with the appropriate library for your platform (Windows/Unix/Mac) and compiler (GNU C++), and your application will adopt the look and feel appropriate to that platform. On top of great GUI functionality, Qt gives you: online help, network programming, streams, clipboard and drag and drop, multi-threading, image loading and saving in a variety of popular formats, database support, HTML viewing and printing, and much more.

### **What is C++ Builder?**

C++ Builder is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Malibu extends the Builder IDE through the addition of a large, powerful family of C++ objects specifically tailored to perform real-time data streaming functions.

### **What kinds of applications are possible with Innovative Integration hardware?**

Data acquisition, data logging, stimulus-response and signal processing jobs are easily solved with Innovative Integration baseboards using the Malibu libraries and software. There are a wide selection of peripheral devices available in the XMC, SBC and DSP product families, for all types of signals from DC to RF frequency applications, video or audio processing. Additionally, multiple Innovative Integration baseboards can be used for a large channel or mixed requirement systems and data acquisition cards from Innovative can be integrated with Innovative's other DSP or data acquisition baseboards for high-performance signal processing.

---

### **Why do I need to use Malibu with my Baseboard?**

One of the biggest issues in using a personal or embedded computer for data collection, control, and communications applications is the relatively poor real-time performance associated with the system. Despite the high computational power of the PC, it cannot reliably respond to real-time events at rates much faster than a few kilohertz. The PC is really best at processing data, not collecting it. In fact, most modern operating systems like Windows and standard distributions of Linux are simply not focused on real-time performance, but rather on ease of use and convenience. Word processing and spreadsheets are simply not high-performance real-time tasks.

The solution to this problem is to provide specialized hardware assistance responsible solely for real-time tasks. Much the same as a dedicated video subsystem is required for adequate display performance, dedicated hardware for real-time data collection and signal processing is needed. This is precisely the focus of our baseboards – a high performance, state-of-the-art, dedicated digital signal processor coupled with real-time data I/O capable of flowing data via a PCI Express bus interface.

The hardware is really only half the story. The other half is the Malibu software tool set which uses state-of-the-art software techniques to bring our baseboards to life in the Windows/Linux environments. These software tools allow you to create applications for your baseboard that encompass the whole job - from high speed data acquisition, to the user interface.

### **Finding detailed information on Malibu**

Information on Malibu is available in a variety of forms:

- Data Sheet (<http://www.innovative-dsp.com/products/malibu.htm>)
- [On-line Help](#)
- Innovative Integration Technical Support
- Innovative Integration Web Site ([www.innovative-dsp.com](http://www.innovative-dsp.com))

### **Online Help**

Help for Malibu is provided in a single file, Malibu.chm which is installed in the Innovative\Documentation folder during the default installation. It provides detailed information about the components contained in Malibu - their Properties, Methods, Events, and usage examples. An equivalent version of this help file in HTML help format is also available online at <http://www.innovative-dsp.com/support/onlinehelp/Malibu>.

### **Innovative Integration Technical Support**

Innovative includes a variety of technical support facilities as part of the Malibu toolset. Telephone hotline supported is available via

Hotline (805) 578-4260 8:00AM-5:00 PM PST.

Alternately, you may e-mail your technical questions at any time to:

[techsprt@innovative-dsp.com](mailto:techsprt@innovative-dsp.com).

---

Also, feel free to register and browse our product forums at <http://forum.iidsp.com/>, which are an excellent source of FAQs and information submitted by Innovative employees and customers. In addition to the forum, please take a few moments to register on the [MyII web portal](#), which will provide access to the latest version of the software libraries, manuals, data sheets and more. The forum and the MyII portal are independent support resources, so it is important to establish an account at both sites.

## Innovative Integration Web Site

Additional information on Innovative Integration hardware and the Malibu Toolset is available via the Innovative Integration website at [www.innovative-dsp.com](http://www.innovative-dsp.com)

## Typographic Conventions

---

This manual uses the typefaces described below to indicate special text.

Typeface	Meaning
Source Listing	Text in this style represents text as it appears onscreen or in code. It also represents anything you must type.
<b>Boldface</b>	Text in this style is used to strongly emphasize certain words.
<i>Emphasis</i>	Text in this style is used to emphasize certain words, such as new terms.
Cpp Variable	Text in this style represents C++ variables
Cpp Symbol	Text in this style represents C++ identifiers, such as class, function, or type names.
<b>KEYCAPS</b>	Text in this style indicates a key on your keyboard. For example, "Press ESC to exit a menu".
Menu Command	Text in this style represents menu commands. For example "Click View   Tools   Customize"

---

## Chapter 2. *A Tour of Malibu*

---

Malibu is a powerful, feature-rich software library designed to meet the challenge of developing software capable of high-speed data flow and real-time signal analysis on PC-compatible platforms. Malibu adds high performance data acquisition and data processing capabilities to Microsoft Visual C++ .NET 2003/2005/2008/2010, Borland/CodeGear/Embarcadero Developers Studio/Turbo C++/BCB6 or GNC C++ applications with a complete set of functions that solve data movement, analysis, viewing, logging and fully take advantage of the object-oriented nature of C++.

Harnessing the expressive power of the standard C++ environments listed above, Malibu offers the most powerful and flexible tools to rapidly integrate high-end data processing in applications. This class library offers an excellent means to application engineers for synchronizing host side data movement and processing with hardware-driven data that is transferred to/from DSP or data acquisition cards. Malibu simply delivers the highest performance data streaming achievable on a desktop or industrial PC.

Rapid application development is achieved using principles such as reusable C++ classes, visual application and form design and full-bandwidth direct hardware access. Malibu is an environment that allows you to create any real-time applications running under Windows or Linux.

### *Malibu Architecture*

---

#### **High Performance Code**

A critically-important feature of Malibu is that it was written from the ground up with demanding real-time applications as the focus of the product. Malibu features a C++ foundation that has powerful data acquisition and analysis features added to it. So when you work with Malibu, you are building on C++ - a language that is unrivaled for its power, ease of development and flexibility. You can code with the tool that suits you, within your preferred visual development environment, supporting rapid prototyping and attractive user interface creation with minimal code.

#### **Synergistic operation with DSP co-processor boards**

Innovative baseboards which are equipped with an on-board Texas Instruments DSP, such as products within the Matador or Velocia family, are provided with an additional library named Pismo which facilitates real-time data acquisition and analysis through embedded DSP programs running on the DSP. Pismo (for target DSP development) coupled with Malibu (for Host development) gives you the power to collect real-time data, analyze it, process it, record it and display it - all within a flexible, yet feature rich tool set. Pismo supports several data capture/playback modes including continuous streaming, transient capture and stimulus/response that allow you to construct your experiment to suit the situation. Complete control over the triggering and data collection/ playback process makes it easier to capture the data you need, all from within one single host application.

When developing an embedded DSP application, Pismo is a tremendous complement of tools for the host side that will greatly facilitate your development. Malibu will help you capture data that has been transferred over the PCI bus by your

---

DSP, and manipulate it, view it and log it. You can even “pass it” through to your own processing code segment, all with ease. Pismo simply equates to an incredible amount of time saved.

An important element within the Malibu library is support for data transfer between the host software and a target baseboard at the highest speed the hardware will support. Obtaining high performance on a PC is a challenging job. Malibu does all of that internally via it's streaming support classes.

## A Portable Class Library

One of the design concepts for Malibu was to allow its library to be used on the most popular IDE frameworks on PCs: Microsoft Visual Studio, Embarcadero Builder C++ products and GNU C++. Malibu is written in standard C++, using standard C++ constructs and libraries. The same source code set is compiled into libraries for each supported compiler, meaning that on any platform the same objects with the same methods and interfaces are supported.

In order to provide the main services of the Malibu library, a number of building-block classes and methods were developed. Many of these classes have uses in the user application as well as in the library, since they provide a portable and tested class or function to perform the sorts of operations that are common in applications.

## *Class Groups In Malibu*

---

The classes in the Malibu suite fall into several functional categories. These are implemented within different library files within Malibu to provide maximum autonomy and keep the organization of the library clear. These categories are outlined in the table and the following paragraphs.

Category	Purpose
Framework	Access to framework-system specific features such as those within the Win32, Win64 or Qt API to accomplish inter-thread messaging and command-line access. Implemented within the <code>Framework_Mb</code> library. This is the only library containing platform-specific code.
OS	Access to operating-system specific features, such as threads, signals resource locks. Implemented within the <code>Os_Mb</code> library. Note: All OS-specific code is isolated into classes within the OSAL layer, described below.
Analysis	Provide access to the common signal processing functions such as filters and FFTs; Logging and playback of waveforms and other classes needed in data acquisition and control applications. Implemented within the <code>Analysis_Mb</code> library.
Utility	Wide variety of common helper classes to manipulate elementary objects such strings and buffers; perform file I/O; accurate timing measurements and delays; implement inter-thread callbacks. Includes a C++ implementation of OpenWire inter-class callback events to allow convenient data processing of a data stream. Implemented within the <code>Utility_Mb</code> library.

---

---

Category	Purpose
Hardware	Provide software interface to generic hardware devices used on Innovative baseboards. Provision for COFF file parsing and downloads, HPI DSP bus access, message I/O structures, XSVF parsing and loading, FPGA loading via SelectMap, access to baseboard calibration ROM and debug scripts. Implemented within the <code>Hardware_Mb</code> library.
PCI	Provide software interface to PCI- and PCI Express -equipped DSP baseboards and PMC modules. Provisions for peripheral initialization and bus-mastering data transfers between target DSP and/or FPGA peripherals and Host PC. Implemented within the <code>Pci_Mb</code> library.
VPX	Provide software interface to VPX -equipped baseboards and FMC modules. Provisions for peripheral initialization and bus-mastering data transfers between target DSP and/or FPGA peripherals and Host PC. Implemented within the <code>Vpx_Mb</code> library.
Ethernet/Poco	Provide software interface to ethernet-equipped targets, such as Andale and single-board products such as the SBC-K7. Provision for peripheral initialization and TCP/IP communications between target DSP and Host PC. Implemented within the <code>Ethernet_Mb</code> and <code>Poco_Mb</code> projects.
Application	Classes which simplify creation of example code, such as static waveform generator, INI file writer, etc.

Each library has its contents summarized in a later chapter in this document. In addition, Malibu has an online help file that provides further information on the classes in the library and their use.

## *Operating System Independence*

---

Malibu currently runs under nearly all versions of Windows 32 and 64 bit, most Linux 32 and 64-bit variants including those using the Xenomai and Red Hat MRG RTOS extensions. Malibu has been ported to Wind River VxWorks 6.8, but that port is no longer actively supported. Contact Innovative if you require current VxWorks support.

Malibu has been designed for portability and migrating to a new operating system is straightforward. All OS dependencies in Malibu are isolated into the [OS abstraction layer](#) (OSAL). OSAL is a pure, abstract C++ interface. To port Malibu to a new OS, one need only implement a concrete version of the OSAL interface, then link accordingly.

The OSAL interface consists of six classes which provide access to primitive OS objects such as mutexes, events, threads, debug tracing and hardware enumeration, mapping and interrupt hooking. The complete interface specification is listed in the header `Osal_Mb.h`, located in the Innovative/Malibu folder after software installation.

To port Malibu to a new OS, implement concrete versions of each of the classes in namespace `OsalSupport`. For instance, your mutex class should derive from `IOsMutex` and must implement a `Lock` and `Unlock` method.



---

Once all methods for all five classes have been implemented and tested, the standard example code provided with each board which uses Malibu will work without change. Simply subsume the entire ApplicationIo object into your application code, and call it's public methods similar to the supplied example to control the card.

## *Malibu Namespaces*

---

Malibu uses C++ name spaces to distinguish its classes and methods from those of other libraries. The majority of the classes within Malibu reside within the `Innovative` name space. Another common name space is `OpenWire` for classes that make up the OpenWire data transfer and connection library in Malibu. There are other name spaces in Malibu that are used internally and not usually involved at the application level.

Like any C++ library, to use Malibu objects you must include the appropriate header that defines the structure of the object and its methods. If this object is in a namespace, the class name has to include the namespace to provide the full name of the class. For instance:

```
#include <Quadia.h>
...
MyClass::DoWork()
{
    Innovative::Quadia Dsp;
    Dsp.Target(0);
    Dsp.Open().
    ...
}
```

Since `Quadia` is in the `Innovative` namespace, its fully qualified name is `Innovative::Quadia`. To avoid having to include the namespace, a using directive can be used to tell the compiler to search the `Innovative` namespace automatically:

```
#include <Quadia.h>
using namespace Innovative;
...
MyClass::DoWork()
{
    Quadia Dsp;
    Dsp.Target(0);
    Dsp.Open().
    ...
}
```

These directives should be used with caution, since names shared in two namespaces may create errors in compilation. Also, be aware that it is poor form to employ the `using namespace` directive within a header file.

Refer to the `Malibu.chm` on-line help file for detailed descriptions of any of the classes or components in the Malibu library suite.

---

## *Interface Classes in Malibu*

---

An interface is a software technique that helps organize the methods of a class into functional groups. Malibu uses interface classes extensively to help manage the complexity of our objects.

What does this mean? Consider these two classes controlling radios:

```
class OldStyleRadio
{
    enum KnobDirection { knobLeft, knobRight };
    void PlugIn();
    void Unplug();
    int TurnStationKnob(KnobDirection turn);
    float TunedFrequency();
};

class NewRadio
{
    void PowerButton(bool state);
    void SetBand( BandType band );
    int StationUpButton();
    int StationDownButton();
    float TunedFrequency();
    int PressPreset(int which);
};
```

Now an application can't easily use both sorts of radio, because the methods differ. Each radio supports similar actions, but in unique ways. Also, it isn't apparent which methods belong with what 'action set' in the radio. So lets define the actions our common radio must perform:

```
class IRadioPower
{
    virtual void Power(bool state) = 0;
};

class IRadioTuning
{
    virtual void TuneUp() = 0;
    virtual void TuneDown() = 0;
    virtual float TunedFrequency() = 0;
};
```

These interface classes define a set of methods that we need to have in order to support an operation. Note that this interface says nothing at all about how an object will actually get the job done; just what method we can call to do a defined task. If our application is written to use the IRadioPower and IRadioTuning interface classes, it will be able to operate any radio that supports the two interfaces.

So here we change the radios to implement the interfaces:

```
class OldStyleRadio : public IRadioPower, public IRadioTuning
{
    enum KnobDirection { knobLeft, knobRight };
    // IRadioPower implementation
    virtual void Power(bool state)
    {
        if (state)
            PlugIn();
        else
            Unplug();
    }
};
```

---

```

    }
    // IRadioPower implementation
    virtual void TuneUp()
    {
        TurnStationKnob( knobRight );
    }
    virtual void TuneDown()
    {
        TurnStationKnob( knobLeft );
    }
    float TunedFrequency();

    // basic functions
    void PlugIn();
    void Unplug();
    int PressPreset(int which);
    int TurnStationKnob(KnobDirection turn);
};

class NewRadio : public IRadioPower, public IRadioTuning
{
    // IRadioPower implementation
    virtual void Power(bool state)
    {
        PowerButton(state);
    }
    // IRadioPower implementation
    virtual void TuneUp()
    {
        StationUpButton();
    }
    virtual void TuneDown()
    {
        StationDownButton();
    }
    float TunedFrequency();

    // basic functions
    void PowerButton(bool state);
    void SetBand( BandType band );
    int StationUpButton();
    int StationDownButton();
};

```

Not only can an application now control both the radios, but the interface classes themselves provide a definition of a subsystem of a device that can aid in reducing the complexity of a complex system. In the above example, the dozen or so methods are reduced to two subsystems – power management and station channel management.

In the baseboard objects that control Innovative's co-processor boards, there are many of these subsystems defined to manage logic loading, loading of code to a target device, and board I/O. As each of these systems is more complicated than this simple example, the value of defining interfaces increases all the more.

---

## *Event Callbacks in Malibu*

It is often the case in a complicated library that a procedure in a library may have to be customized for a particular application or that the application will need to be notified of certain events in a procedure.

---

An example of the former case is data processing. The Malibu library contains means for getting messages and data from a target baseboard, but it obviously has no way of knowing how the application wishes to process the command. In this case, the application needs to insert custom code in this place to complete the process.

An example of the latter is progress messages. If a process such as COFF downloading or logic downloading takes a considerable amount of time, an application may wish to display some feedback to the user giving the current progress. An event can perform this notification as part of the download process.

In order to support event callbacks, a class needs to create an instance of the `OpenWire::EventHandler` template. The template parameter is the `Event` data class, which is the parametric information passed into the installed callback handler when an event is called. The application provides a handler for an event by calling the `SetEvent()` method.

## UI Thread Synchronization

One additional aspect of event callbacks involves user-interface (UI) functions. An event handler often is triggered in a different thread than the main user-interface thread. The use of background threads allows time-consuming tasks to work without interfering with the responsiveness of the main program. But this leads to a problem if event handlers are executed from within the context of a background thread and the handler are expected to update a user-interface (UI) element such as a progress bar, or edit control. Since user interfaces are built atop APIs such as Win32 and Qt which are not thread-safe, such UI control updates are not thread-safe and can cause mysterious, unpredictable failures in an application at runtime.

To avoid this, an event handler can be “thunked” or “synchronized” with the main thread by using the `Thunk()` or `Synchronize()` method. Even though invoked from within a background thread, the installed, user-specific event handler will be executed within the context of the main UI thread, albeit at a slight efficiency penalty. Note that most of the event handlers built into Malibu objects which are routinely used for UI updates are thunked or synchronized by default. However, the synchronization behavior of any event may be overridden using these methods within application code, if desired.

## *Using the Malibu Library*

---

The Malibu library is a library of standard C++ classes. Its classes are created and used in a similar fashion to the classes of the standard library. Versions of the library are built for Visual C++ (v7, v8 and v9), for Borland C++ (BDS2006/TurboC++ and BCB6) and for GNC C++ under Linux. The code that interacts with Malibu classes is identical on all versions – the differences actually come when interacting with the different APIs for the visual portion of the application.

The Malibu library provides a simple means of accessing the features of the Innovative baseboards, and streaming data between a Host application and target peripherals. By using Malibu, you can easily process and analyze data in real-time, as it is moved to and from the hardware.

The Malibu system uses a number of classes to perform data acquisition and analysis functions. Depending on the operations to be performed, you may need a streaming class, one or more baseboard classes, analysis classes and so on. The properties of the baseboard classes are used to define the system configuration. The properties of the analysis classes and especially the connections to other analysis components are crucial in defining the data analysis.

Event handler callbacks are another major part of creating an application in Malibu. Malibu objects provide 'Events' that the user can install a handler for that provide feedback or to customize processing.

---

## Creating a Streaming Application in Visual C++

### Creating the Malibu Objects

First we will declare the necessary objects. In this case we are developing an MFC application and we have selected a dialog-based application in the Visual C++ wizard, so that we can have a visual means of laying out the main window. This is a common technique in Visual C++.

The best place for the declarations is the dialog class that was auto-created by the application wizard. Here is how the code will look like if the code if we have given the name `CAppDlg` to our dialog class:

```
namespace Innovative
{
    class Uwb;
    class Quadia;
    class C64xDsp;
    class DataLogger;
}

class CAppDlg : public CDialog
{
...
...
private:
    Innovative::Uwb *          Uwb[2];
    bool                    UwbOpened[2];
    Innovative::Quadia *      Quadia;
    bool                    QuadiaOpened;
    Innovative::C64xDsp *     Dsp[4];
    bool                    DspOpened[4];
    Innovative::TiBusmasterStream * Stream[4];
    bool                    StreamConnected[4];

    Innovative::IntegerBuffer BB2;
    Innovative::DataLogger *  Log;
...
...
protected:

    void CoffLoadProgressHandler( Innovative::ProcessProgressEvent & event);
    void CoffLoadCompleteHandler( Innovative::ProcessCompletionEvent & event);
    void MailAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event);
    void PacketAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event);
};
```

In this application we will be creating several baseboard objects. The Quadia baseboard has 4 C64x Dsps on it, each of which has its own baseboard. In addition there may be 2 UWB Ultra Wideband PMC baseboards on the Quadia. The header only contains pointers to the objects. The actual objects will be created later.

Later in the declaration are several event handler functions. Each handler has the signature of the event it handles, which is a single class that holds parameters for the handler.

Now it's time to initialize the objects. The `OnInitDialog` member function is a good place for initialization, since the dialog controls are available but the window is not visible.

```
BOOL CAppDlg::OnInitDialog()
{
```

---

```

...
...
//
// Create devices (but don't open!)
Quadia = new Innovative::Quadia();

...
...

Uwb[0] = new Innovative::UwbCs;
Uwb[1] = new Innovative::UwbCs;

//
// Coff File progress events
for (int i=0; i<4; i++)
{
    Dsp[i] = new Innovative::C64xDsp;

    Dsp[i]->Cpu().OnCoffLoadProgress.SetEvent(this, &CApplDg::CoffLoadProgressHandler);
    Dsp[i]->Cpu().OnCoffLoadProgress.Synchronize();
    Dsp[i]->Cpu().OnCoffLoadComplete.SetEvent(this, &CApplDg::CoffLoadCompleteHandler);
    Dsp[i]->Cpu().OnCoffLoadComplete.Synchronize();
    Dsp[i]->SdramCE = SdramCE;
}

for (int i=0; i<4; i++)
{
    Stream[i] = new Innovative::TiBusmasterStream();
    Stream[i]->OnMailAvailable.SetEvent(this, &CApplDg::MailAvailableHandler);
    Stream[i]->OnMailAvailable.Synchronize();
    Stream[i]->OnPacketAvailable.SetEvent(this, &CApplDg::PacketAvailableHandler);
    Stream[i]->OnPacketAvailable.Synchronize();
}

return TRUE; // return TRUE unless you set the focus to a control
}

```

### Initializing Object Properties and Events

The code immediately after the constructor of the C64xDsp and TiBusmasterStream objects are to attach handlers to events contained in the baseboard and its subsystems. In the case of the C64xDsp object, the COFF loading interface returned by the Cpu() member function has the OnCoffLoadProgress event.

This event will be called during the downloading of code to the Dsp in order to give a completion percentage of the download. The handler usually updates a progress bar with this data to give visual feedback. Because this handler will update the GUI, it needs to be synchronized with the GUI main thread. This is done by the call to the Synchronize() member function of the event handler object.

Below that code is the initialization of the streams. Each DSP will have its own stream object to manage. These objects have events associated with data arriving from the target. The two event handlers are attached to functions and set to be synchronized here.

This code also shows setting a property of a baseboard. SdramCE is a property that sets which addressing space on the target the SDRAM is located. For the Quadia, it needs to be initialized to 0.

---

In order to use a baseboard, it must be associated with an actual device. Each device in the system is given a unique index known as the Target ID. After being assigned a target number, the device can be attached to the hardware with a call to `Open()`:

```
// Open Cpus 0 and 1 & connect their streams
Dsp[0]->Target(0);
Dsp[0]->Open();
DspOpened[0] = true;
Stream[0]->ConnectTo(Dsp[0]);
StreamConnected[0] = true;

Dsp[1]->Target(1);
Dsp[1]->Open();
DspOpened[1] = true;
Stream[1]->ConnectTo(Dsp[1]);
StreamConnected[1] = true;
AppendToLog("C64x Pair #0, #1 Opened...");
```

In order to perform I/O with a baseboard, a stream object needs to be connected to it. This is done by the `ConnectTo()` method. If a baseboard does not support a type of streaming, the `ConnectTo()` call will not compile.

### Event Handler Code

Data comes from the target via stream event handlers. 'Mail' messages are small (16 word) packets of data intended for command and control information exchange. Two words of the message is a header that is divided into standard fields. The `TypeCode` field is usually used for distinguishing different types of messages:

```
//-----
//  CAppDlg::MailAvailableHandler() --
//-----

void CAppDlg::MailAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event)
{
    //
    //  Read the mail message packet
    Innovative::MatadorMessage Msg;
    event.Sender->Recv(Msg);

    CString Txt;
    Txt.Format("Dsp Target %d Message:", event.Sender->Target());
    AppendToLog(Txt);

    switch (Msg.TypeCode())
    {
        case kChannelInitMsg:
        {
            //TargetLogin = true;
            int Ver = Msg.Data(0) - 0x100;
            CString Txt;
            Txt.Format("Target logged in OK - Ver: %d\r\n", Ver);
            AppendToLog(Txt);

            AppendToLog("Blocks Rcvd: 0");
        }
        break;

        case kDInInfo:
        {
            CString Txt;
            Txt.Format("Ev/Buf: %d\r\n", Msg.Data(0));
```

---

```

        AppendToLog(Txt);
        Txt.Format("Actual: %d\r\n", Msg.Data(1));
        AppendToLog(Txt);
        Txt.Format("Burst: %d\r\n", Msg.Data(2));
        AppendToLog(Txt);
        Txt.Format("Actual: %d\r\n", Msg.Data(3));
        AppendToLog(Txt);
    }
    break;

case kThresholdAlert:
{
    AppendToLog("ALERT");
    CString Txt = "Threshold Alert Rcvd";
    AppendToLog(Txt);
}
break;

case kOverflowAlert:
{
    AppendToLog("ALERT");
    CString Txt = "Overflow Alert Rcvd";
    AppendToLog(Txt);
}
break;

default:
{
    AppendToLog("Invalid DSP message received");
}
break;
}

    MessageBeep(MB_OK);
}

```

The event handler argument contains parameters for the event. In this case, the event data structure contains a pointer to the stream that generated the event. This pointer is used to actually extract the message via the `Recv()` method.

Handling the packet data event is similar: the buffer is extracted using the `Recv()` method and processed. In this case the data is logged using the `LogDataBlock()` function.

```

//-----
//  CAppDlg::PacketAvailableHandler() --
//-----

void  CAppDlg::PacketAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event)
{
    static int PacketCount(0);
    // Since we got this message we know a buffer is available. So read it now.
    // Buffer will be sized to fit the incoming data.
    event.Sender->Recv(BB2);
    //
    // Find which Cpu is our target
    int DspIdx;
    for (int i=0; i<4; i++)
        if (DspOpened[i])
        {
            if (event.Sender->Target() == CaptureInfo[i].Target)
            {
                DspIdx = i;
                break;
            }
        }
    }
}

```



---

```

        }
    }
    // ...DspIdx is which Dsp # to use
    //
    // Increment
    CaptureInfo[DspIdx].CaptureBlocks++;
    //
    // Log the data block
    LogDataBlock(DspIdx, BB2);
    //
    // Update message showing data arrival.
    CString Text;
    Text.Format("Dsp %d, Packet %d with %d words arrived", DspIdx, ++PacketCount, BB2.IntSize());
    AppendToLog(Text);
}

```

### Loading COFF Files

Operations such as downloading COFF files to a DSP are grouped in an interface class so that the methods used to perform them and the events presented are the same from board to board. This code initiates a download to all four CPUs on a Quadia. Events can be hooked to provide feedback on the progress of the download.

```

void CAppDlg::OnBnClickedDownloadCoff()
{
    CString filename;
    CoffFileNameEdit.GetWindowText(filename);
    std::string FileName(filename);

    for (int i=0; i<4; i++)
        if (DspOpened[i])
        {
            AppendToLog("-----");
            CString Txt;
            Txt.Format("-- COFF Load Dsp #%d", i);
            AppendToLog(Txt);
            AppendToLog("-----");

            Dsp[i]->Cpu().DownloadCoff(FileName);
        }
}

```

### Loading Logic Files

Many baseboards have down-loadable logic to provide customized behavior. Loading this logic is also grouped into an interface class. In the code below, one of the Quadia's two logic chips is being loaded. The interface class also contains events that can be hooked to provide feedback in the user interface.

```

//-----
// BaseboardLogicLoadDialog::OnBnClickedQfpgalCfgbtn() --
//-----

void BaseboardLogicLoadDialog::OnBnClickedQfpgalCfgbtn()
{
    if (!Owner->QuadiaOpened)
    {
        Owner->AppendToLog("No Quadia Installed");
        return;
    }

    CString ExoFilename;

```

---

```

        FpgalFileName.GetWindowText(ExoFilename);

        if (! Innovative::IIFileExists(ExoFilename))
            throw Innovative::IException("Exo file not found!");

        Owner->AppendToLog("-----\r\nParsing FPGA 1");
        Owner->UpdateWindow();
        Owner->Quadia->Logic(1).ConfigureFpga(std::string(ExoFilename));
        Owner->AppendToLog("-----\r\n");
        Owner->UpdateWindow();
    }

```

## Script Files

Many PMC modules feature user-reprogrammable FPGA logic. As the behavior of this logic is subject to change to accommodate user requirements, it is commonplace to map registers into the User FPGA on these modules to support configuration and control.

To facilitate rapid prototyping of new logic, Malibu features two script interpreter classes `Scripter` and `GcScripter`. These classes parse the contents of a text file at application runtime, calling predefined event handlers during the process. By overloading these handlers within application code, it is possible to read and write to custom user logic registers at strategic times during application execution.

For instance, the Digital Receiver PMC module supports four GrayChip 5016 down-converter ICs. These are sophisticated devices with a large complement of mapped registers used to configure the down-conversion process. Rather than building one particular initialization pattern for these devices into the `DigitalReceiver` class, Malibu defers the initialization process for these IC devices into the application domain.

The application program instantiates a `GcScripter` object, and calls the `Execute` method on this object at the inception of analog data flow, within the `OnStreamStart` event handler of the `DigitalReceiver` object. In turn, the `GcScripter` object parses a user-authored text file which contains initialization commands targeting the GC5016 devices addressable through the command-channel (PCI bus). The initialization command file may be created manually, or the Texas Instruments -supplied utility for creation of 5016-compliant initialization files may be used. Regardless of the origin of this file, its contents will be parsed and used to initialize the 5016 devices dynamically at application runtime, without requiring recompilation of the Malibu libraries.

Later, once the initialization sequence is finalized, the contents of the script can be subsumed into the application directly, and explicit calls to `PokeDdcReg` used to initialize the 5016 IC devices, eliminating need for the `GcScripter` object.

---

## Chapter 3. *Creating Applications using an IDE*

---

Developing an application will more than likely involve using an integrated development environment (IDE), also known as an integrated design environment or an integrated debugging environment. This is a type of computer software that assists computer programmers in developing software.

The following sections will aid in the initial set-up of these applications in describing what needs to be set in Project Options or Project Properties for each of the supported development environments.

---

### *Creating a Malibu Project in Microsoft Visual Studio 7*

---

Creating a project that will successfully build a Malibu project requires a few extra steps beyond making a new, empty form project.

---

### *Enabling Auto-Saving of Projects*

---

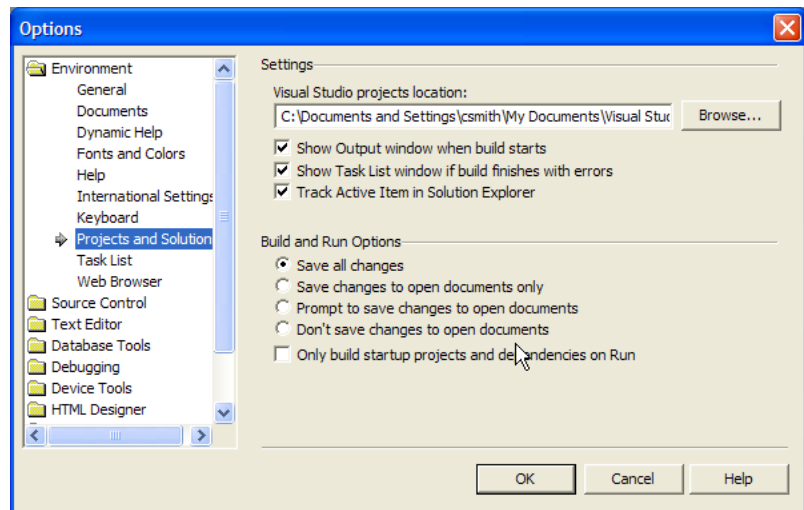
Files and project settings in MSVC should be set to be whenever the program is compiled. This avoids the problem when a program crash causes the loss of much programming effort.

To change the setting, open the Environment Options by selecting Tools | Options from the main Menu.

Selecting this will open the Options dialog.

Select the Environment | Projects and Solutions entry in the list on the left. The Build and Run Options section should be set to Save all Changes ensure that everything will be saved before each project compilation.

This option is set by default.



---

## Creating a Malibu Project

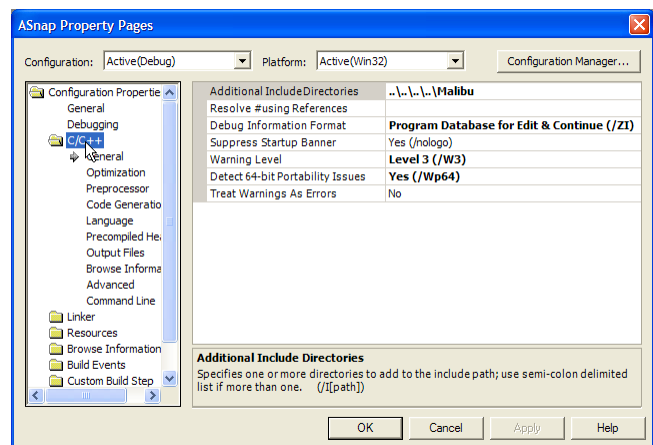
---

Since MSVC knows nothing about Malibu, the basic project options need to be modified to allow the compiler to find Malibu. The Project Option dialog is displayed when the menu Project | <ProjectName> Properties... is selected.

There are two places where the system needs to be informed of the location of library files. These are

- 1) The “Include Path”, which allows header source files to be found in the compilation process.
- 2) The “Library Path”, which allows the linker to find the libraries to search for code modules the application requires.

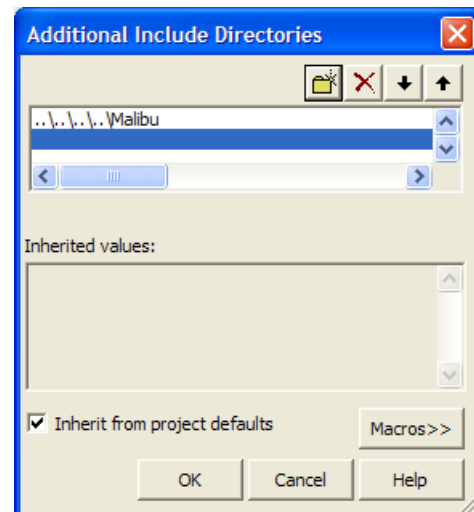
First we will change the Include Path. On the C/C++ | General page of the Property Page dialog, The Additional Include Directories entry determines the extra directories where source files will be searched for. You can just edit the path itself, but an easier way is to select the path, and then press the '...' button that appears in the edit control to display a path editing control dialog.



The path editing control allows directories to be rearranged, and each entry can be edited by selecting it, browsing to a directory by pressing the '...' button and replacing the result in the list.

Here, we wish to add the Malibu source directory to the list. By default, this is installed at C:\Innovative\Malibu. In this case, we use a relative path to indicate its location.

There are several ways that you can define these paths, each with advantages and disadvantages.



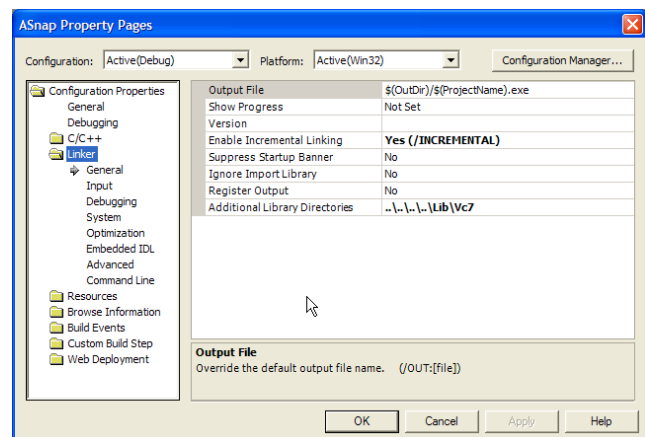
**Table 1. Path Spec Options**

Type of Path Spec	Advantages	Disadvantages	Example
Relative Path	Doesn't need to be changed if project moves only at same level below source directory. Doesn't require the project know where Malibu source directory is or what name the install directory has.	Lots of “..”s. Hard to set up. Requires projects be under the Innovative tree.	..\..\..\Malibu
Absolute Path	Project doesn't have to be located under the Innovative source tree. Project can be moved after creation without change.	Project must be on same drive as Malibu source directory. Project has to know the name of the Malibu install directory.	\Innovative\Malibu
Full Path plus Drive Letter	Project can be anywhere in system.	Requires that Malibu source directory never moves. Project has to know the name and drive of the Malibu install directory.	C:\Innovative\Malibu

The Innovative Examples use relative paths, since we wish to have to specify the name and location of the Malibu source. User projects may have other constraints that make one of the other options more desirable.

To set the library path, on the Linker | General page of the Property Page dialog, the Additional Library Directories entry determines the extra directories that will be searched to find libraries. You can just edit the path itself, but an easier way is to select the path, and then press the '...' button that appears in the edit control to display a path editing control dialog.

Here, we wish to add the Malibu Library directory to the list. There are several directories for libraries, since the source must be built for each compiler. For MSVC 7.0, this is installed at C:\Innovative\Lib\Vc7. In this case, we use a relative path to indicate its location.



## Other Configuration Requirements

Since Malibu applications are multi-threaded, your application should be configured to use Multi-threaded libraries, via the Configuration Properties | C/C++ | Code Generation | Runtime Library option. When building console or unmanaged applications, select Multi-threaded Debug (/MTd) or Multi-threaded (/MT). When building managed code, select the DLL variants of these libraries.

---

Disable use of precompiled headers by setting Configuration Properties | C/C++ | Precompiled Headers | Create/Use Precompiled Header to Not Using Precompiled Headers.

When the C Run-Time (CRT) library and Microsoft Foundation Class (MFC) libraries are linked in the wrong order, you may receive a LNK2005 error such as:

```
nafxcwd.lib(afxmem.obj) : error LNK2005:
```

```
"void * __cdecl operator new(unsigned int)"(??2@YAPAXI@Z) already defined in  
LIBCMTD.lib(new.obj)
```

The CRT libraries use weak external linkage for the new, delete, and DllMain functions. The MFC libraries also contain new, delete, and DllMain functions. These functions require the MFC libraries to be linked before the CRT library is linked.

When you use the MFC libraries, you must make sure that they are linked before the CRT library is linked. You can do this by making sure that every file in your project includes Msdev\Mfc\Include\Afx.h first, either directly (#include <Afx.h>) or indirectly (#include <Stdafx.h>). The Afx.h include file forces the correct order of the libraries, by using the #pragma comment (lib, "<libname>") directive.

If the source file has a .c extension, or the file has a .cpp extension but does not use MFC, you can create and include a small header file (Forcelib.h) at the top of the module. This new header makes sure that the library search order is correct. Visual C++ does not contain this header file. To create this file, follow these steps:

1. Open Msdev\Mfc\Include\Afx.h.
2. Select the lines between #ifndef \_AFX\_NOFORCE\_LIBS and #endif // !\_AFX\_NOFORCE\_LIBS.
3. Copy the selection to the Windows Clipboard.
4. Create a new text file.
5. Paste the contents of the Clipboard into this new file.
6. Save the file as Msdev\Mfc\Include\Forcelib.h.

See <http://support.microsoft.com/default.aspx/kb/148652> for details.

---

## *Creating a Malibu Project in Microsoft Visual Studio Vc8/Vc9*

Creating a project that will successfully build a Malibu project requires a few extra steps beyond making a new, empty Windows Form project.

---

## Enabling Auto-Saving of Projects

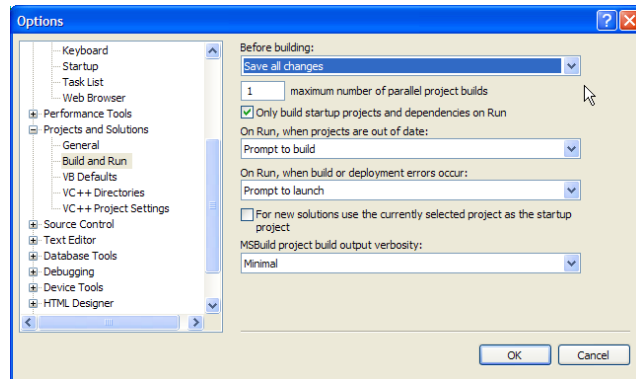
---

Files and project settings in Visual Studio should be set to be whenever the program is compiled. This avoids the problem when a program crash causes the loss of much programming effort.

To change the setting, open the Environment Options by selecting Tools | Options from the main Menu.

Selecting this will open the Options dialog.

Select the Environment | Projects and Solutions entry in the list on the left. The Build and Run page has a Before building: combo box at the top. It should be set to Save all Changes ensure that everything will be saved before each project compilation.



This option is set by default.

---

## Creating a Malibu Project

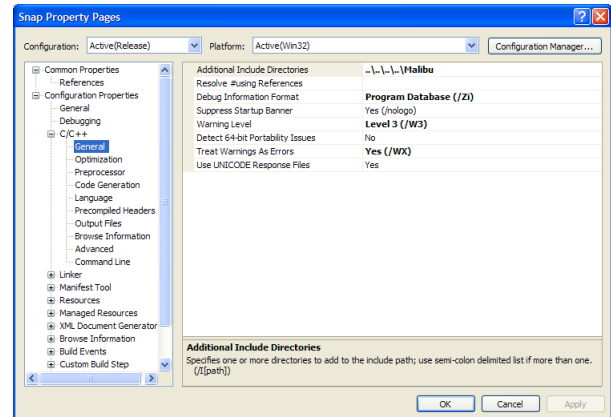
---

Since Visual Studio knows nothing about Malibu, the basic project options need to be modified to allow the compiler to find Malibu. The Project Option dialog is displayed when the menu Project | <ProjectName> Properties... is selected.

There are two places where the system needs to be informed of the location of Malibu support files. These are

- 3) The “Include Path”, which allows header source files to be found in the compilation process.
- 4) The “Library Path”, which allows the linker to find the libraries to search for code modules the application requires.

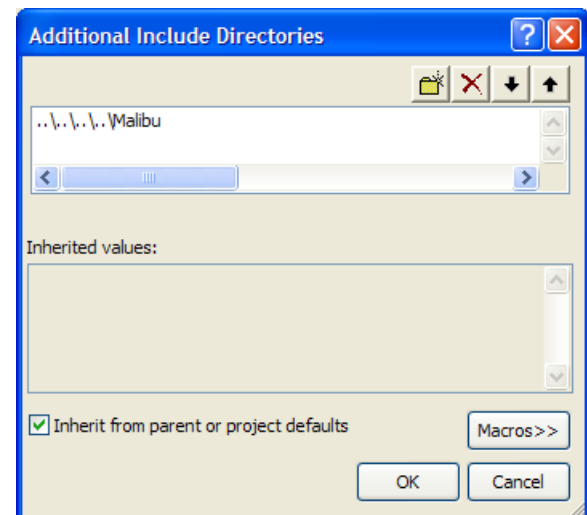
First we will change the Include Path. On the C/C++ | General page of the Property Page dialog, The Additional Include Directories entry determines the extra directories where source files will be searched for. You can just edit the path itself, but an easier way is to select the path, and then press the '...' button that appears in the edit control to display a path editing control dialog.



The path editing control allows directories to be rearranged, and each entry can be edited by selecting it, browsing to a directory by pressing the '...' button and replacing the result in the list.

Here, we wish to add the Malibu source directory to the list. By default, this is installed at C:\Innovative\Malibu. In this case, we use a relative path to indicate its location.

There are several ways that you can define these paths, each with advantages and disadvantages.



**Table 2. Path Spec Options**

Type of Path Spec	Advantages	Disadvantages	Example
Relative Path	Doesn't need to be changed if project moves only at same level below source directory. Doesn't require the project know where Malibu source directory is or what name the install directory has.	Lots of ".."s. Hard to set up. Requires projects be under the Innovative tree.	..\..\..\Malibu
Absolute Path	Project doesn't have to be located under the Innovative source tree. Project can be moved after creation without change.	Project must be on same drive as Malibu source directory. Project has to know the name of the Malibu install directory.	\Innovative\Malibu

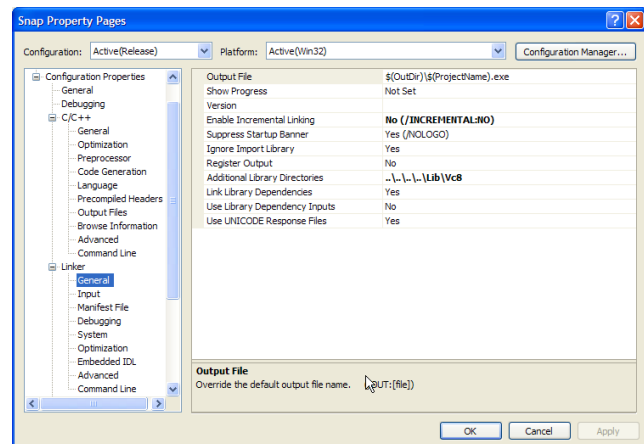


Type of Path Spec	Advantages	Disadvantages	Example
Full Path plus Drive Letter	Project can be anywhere in system.	Requires that Malibu source directory never moves. Project has to know the name and drive of the Malibu install directory.	C:\Innovative\Malibu

The Innovative Examples use relative paths, since we wish to have to specify the name and location of the Malibu source. User projects may have other constraints that make one of the other options more desirable.

To set the library path, on the Linker | General page of the Property Page dialog, the Additional Library Directories entry determines the extra directories that will be searched to find libraries. You can just edit the path itself, but an easier way is to select the path, and then press the '...' button that appears in the edit control to display a path editing control dialog.

Here, we wish to add the Malibu Library directory to the list. There are several directories for libraries, since the source must be rebuilt for each version of the compiler and platform. For Visual Studio 2005, the libraries are installed at C:\Innovative\Lib\Vc8. For Visual Studio 2008, C:\Innovative\Lib\Vc9. The 64-bit platform libraries are located at C:\Innovative\Lib\Vc9\_x64.



## Other Configuration Requirements

Since Malibu applications are multi-threaded, your application should be configured to use Multi-threaded libraries, via the Configuration Properties | C/C++ | Code Generation | Runtime Library option. When building console or unmanaged applications, select Multi-threaded Debug (/MTd) or Multi-threaded (/MT). When building managed code, select the DLL variants of these libraries.

Disable use of precompiled headers by setting Configuration Properties | C/C++ | Precompiled Headers | Create/Use Precompiled Header to Not Using Precompiled Headers.

When the C Run-Time (CRT) library and Microsoft Foundation Class (MFC) libraries are linked in the wrong order, you may receive a LNK2005 error such as:

```
nafxcwd.lib(afxmem.obj) : error LNK2005:
```

```
"void * __cdecl operator new(unsigned int)"(??2@YAPAXI@Z) already defined in
LIBCMDT.lib(new.obj)
```

The CRT libraries use weak external linkage for the new, delete, and DllMain functions. The MFC libraries also contain new, delete, and DllMain functions. These functions require the MFC libraries to be linked before the CRT library is linked.

---

When you use the MFC libraries, you must make sure that they are linked before the CRT library is linked. You can do this by making sure that every file in your project includes `Msdev\Mfc\Include\Afx.h` first, either directly (`#include <Afx.h>`) or indirectly (`#include <Stdafx.h>`). The `Afx.h` include file forces the correct order of the libraries, by using the `#pragma comment (lib, "<libname>")` directive.

If the source file has a `.c` extension, or the file has a `.cpp` extension but does not use MFC, you can create and include a small header file (`Forcelib.h`) at the top of the module. This new header makes sure that the library search order is correct. Visual C++ does not contain this header file. To create this file, follow these steps:

1. Open `Msdev\Mfc\Include\Afx.h`.
2. Select the lines between `#ifndef _AFX_NOFORCE_LIBS` and `#endif // !_AFX_NOFORCE_LIBS`.
3. Copy the selection to the Windows Clipboard.
4. Create a new text file.
5. Paste the contents of the Clipboard into this new file.
6. Save the file as `Msdev\Mfc\Include\Forcelib.h`.

See <http://support.microsoft.com/default.aspx/kb/148652> for details.

Since Malibu is compiled as native code, pure .NET applications cannot be built. However, you may build mixed-mode applications. Set the Configuration Properties | General | Common Language Runtime Support to Common Language Runtime Support (/clr).

Mixed-mode code, .NET applications manage two heaps, one for the .NET code and another for the native code. To work-around an initialization bug within the .NET runtime libraries which fail to initialize both heaps properly, it is essential to force a reference to a C runtime initialization module within the Microsoft-supplied libraries. To do this, you must add a symbol to Configuration Properties | Linker | Input | Force Symbol References. For 32-bit code, add the symbol [\\_DllMainCRTStartup@12](#). For 64-bit code, add the symbol `_DllMainCRTStartup`.

---

## *Creating a Malibu Application using Nokia QtCreator*

Creating a project that will successfully build a Malibu application requires a few extra steps beyond making a new, empty QtForm project.

---

## *Install and/or Rebuild Qt Library*

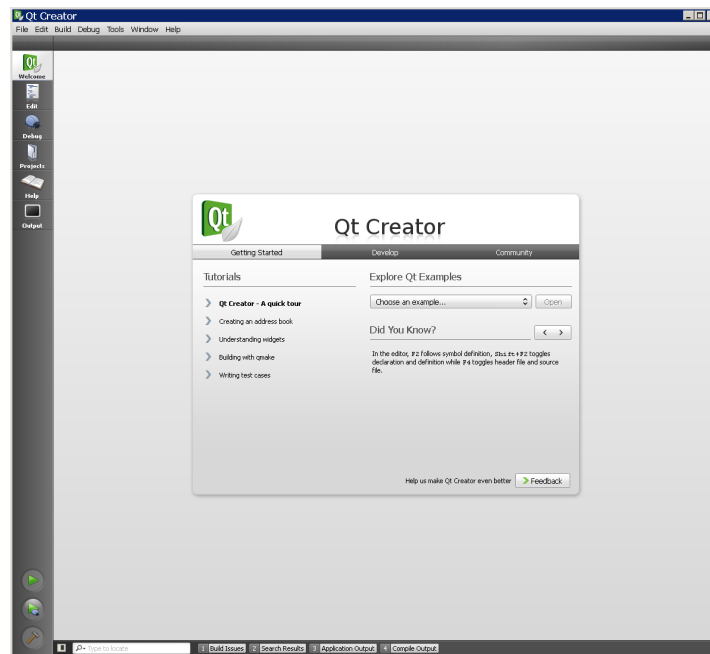
QtCreator projects rely on the open-source Qt controls to implement user interface elements. Download the latest, all-inclusive Qt package from [Nokia QtCreator Website](#). Then, install following the instructions provided on the site. If using one of the provided binary installers, this is accomplished by executing

---

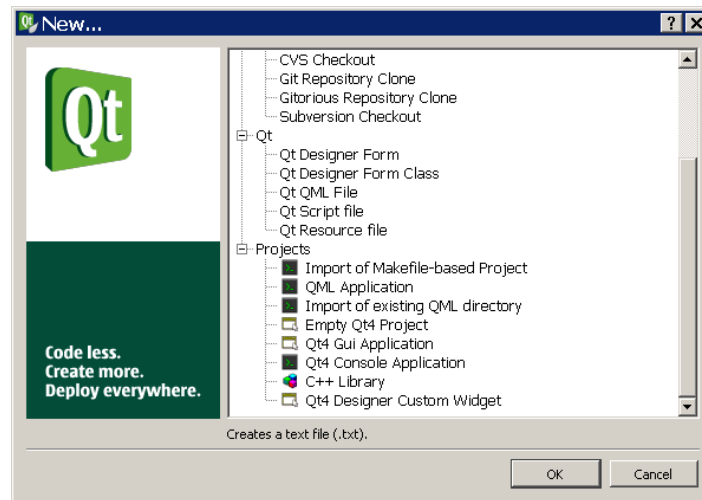
`./qt-sdk-linux-x86-opensource-2009.05.bin`  
from the command line.

Malibu is dependent on the Jungo WinDriver device-driver libraries. These libraries may be downloaded from <http://www.innovative-dsp.com/ftp/Linux/WinDriver.tar.gz> and should be extracted into a user-accessible location on your hard disk, (typically `/home/WinDriver`). Due to the volatility of the Linux kernel and its multiple distributions, it is necessary to rebuild the Jungo WinDriver and the associated Innovative kernel plug-in prior to running applications. Instructions to accomplish this are located in the [Linux Notes](#) on our website.

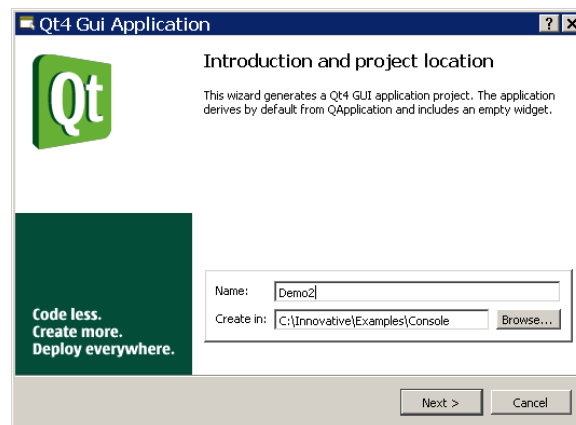
Following successful installation, launch QtCreator.



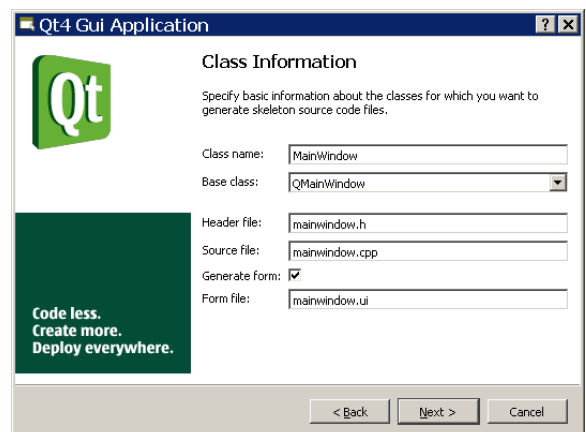
Create a new project by clicking the File | New File or Project menu item.

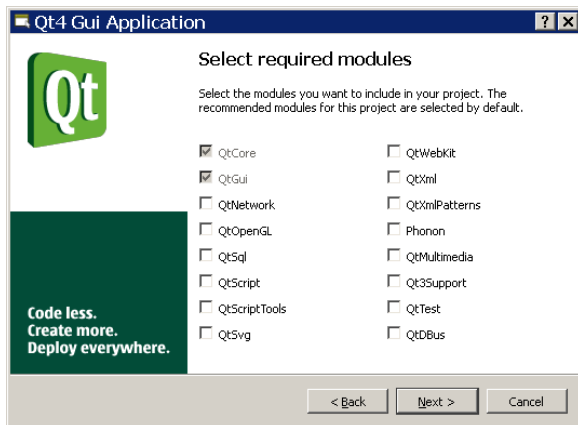


Select Qt4 Gui Application or Qt4 Console Application, depending on the type of project required. Browse to a location in which to store the new project:

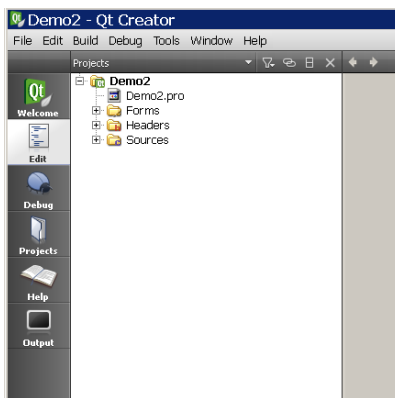


Accept the defaults for the Qt library dependencies:





This creates a new, empty project.



Next, edit the .pro project file to incorporate the Malibu libraries. The addition of the LIBS directive in the .pro file automatically links in all required Malibu libraries.

```
# -----
# Project created by QtCreator 2009-12-17T08:49:47
# -----
TARGET = Snap
TEMPLATE = app
SOURCES += main.cpp \
    mainwindow.cpp \
    ../Common/ApplicationIo.cpp \
    ../Common/ModuleIo.cpp

HEADERS += mainwindow.h

FORMS += mainwindow.ui

DEFINES += GCC

unix:HOMEDIR = "/usr/Innovative"
unix: DEFINES += LINUX
```

---

```
win32:HOMEDIR = $$ (INNOVATIVECOMMON)
CONFIG(debug, debug|release) {
    OBJECTS_DIR = Debug
    DESTDIR = Debug
}
else {
    OBJECTS_DIR = Release
    DESTDIR = Release
}

LIBS += -L$$HOMEDIR/Lib/Gcc \
        -L$$HOMEDIR/Lib/Gcc/$ (OBJECTS_DIR) \
        -Wl,@$$HOMEDIR/Lib/Gcc/Malibu_Qt.lcf

INCLUDEPATH = $$HOMEDIR/Malibu ../Common
```

This project will now compile and run, but does not yet incorporate any Innovative board control functionality. It's typically most efficient to subsume the entire ApplicationIo class from an example program provided with your card directly into the new application in order to insure that you have all of the necessary initialization and event closures required.

Creation of a headless or console-style application is also supported. To build such an application, stub out the functionality of the UserInterface class which is passed by pointer to the ApplicationIo object. And change the .pro file above to link against Malibu\_Con.lcf instead of Malibu\_Qt.lcf. See the [zen console forum thread](#) for details.

---

## *Creating a Malibu Project in Borland Developer's Studio/Turbo C++*

Developing an application will more than likely involve using an integrated development environment (IDE), also known as an integrated design environment or an integrated debugging environment. This is a type of computer software that assists computer programmers in developing software.

Creating a project that will successfully build a Malibu project requires a few extra steps beyond making a new, empty VCL-Form-based project.

The following sections will aid in the initial set-up of these applications in describing what needs to be set in Project Options.

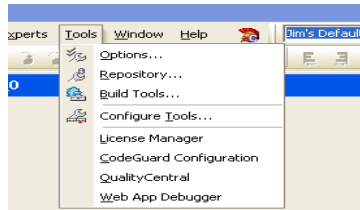
---

## *Enabling Auto-Saving of Projects*

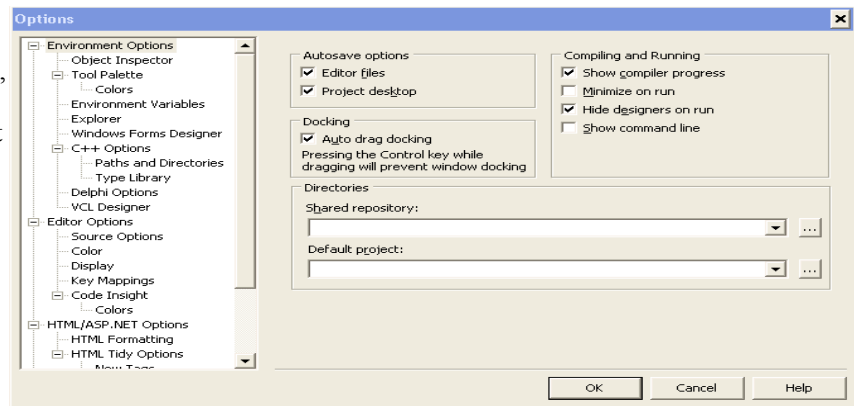
By default, files and project settings in Builder are only saved when you manually save them. In practice, this can be risky as a crashing program can cause the loss of much programming effort. You can change BDS to save all files and project settings whenever a project is compiled. This takes little time, and increases the safety of using the compiler.

To change the setting, open the Environment Options by selecting Tools | Environment Options from the main Builder Menu.

Selecting this will open the Environment Options dialog.



Select the Preferences tab. In the top left side, select the Autosave option's two check boxes to ensure that both the Editor files and Project desktop will be saved before each project compilation.



## *Default Project Options which should be Changed*

### **BCB10 (Borland Turbo C++) Project Settings**

When creating a new application with File, New, VCL Forms Application - C++ Builder

#### **Change the Project Options for the Compiler:**

##### **Project Options**

++ Compiler (bcc32)

C++ Compatibility

**Check 'zero-length empty base class (-Ve)'**

**Check 'zero-length empty class member functions (-Vx)'**

Failure to change these options may result in an access violation when attempting to enter any OpenWire Event function.

i.e.

Access Violation OnLoadMsg.Execute – Load Message Event

Because of statement

Board->OnLoadMsg.SetEvent( this, &ApplicationIo::DoLoadMsg );

---

## Change the Project Options for the Linker:

### Project Options

#### Linker (ilink32)

Linking – **unchecked** ‘Use Dynamic RTL’

In our example Host Applications, if not unchecked, this will cause the execution to fail before the Form is constructed.

Error: First chance exception at \$xxxxxxx. Exception class EAccessViolation with message “Access Violation!”  
Process ????.exe (nnnn)

## Other considerations:

### Project Options

#### ++ Compiler (bcc32)

##### Output Settings

**check** – Specify output directory for object files(-n)

(release build) Release

(debug build) Debug

##### Paths and Defines

**add Malibu**

##### Pre-compiled headers

**unchecked everything**

#### Linker (ilink32)

##### Output Settings

**check** – Final output directory

(release build) Release

(debug build) Debug

##### Paths and Defines

(ensure that Build Configuration is set to All Configurations)

**add Lib/Bcb10**

(change Build Configuration to Release Build)

**add lib\bcb10\release**

(change Build Configuration to Debug Build)

**add lib\bcb10\debug**

(change Build Configuration back to All Configurations)

### Packages

**unchecked** - Build with runtime packages



---

## *Creating Projects in Borland C++ Builder 6.0*

---

Creating a project that will successfully build a Malibu project requires a few extra steps beyond making a new, empty form project.

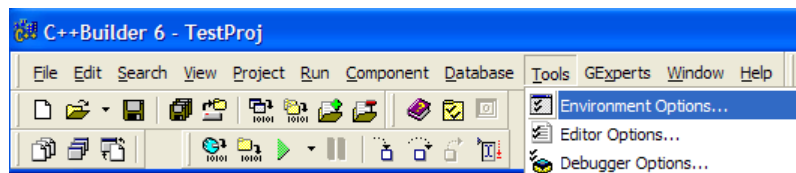
### *Enabling Auto-Saving of Projects*

---

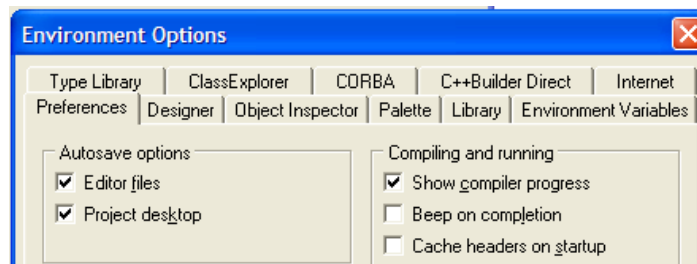
By default, files and project settings in Builder are only saved when you manually save them. In practice, this can be risky as a crashing program can cause the loss of much programming effort. You can change Borland to save all files and project settings whenever a project is compiled. This takes little time, and increases the safety of using the compiler.

To change the setting, open the Environment Options by selecting Tools | Environment Options from the main Builder Menu.

Selecting this will open the Environment Options dialog.



Select the Preferences tab. In the top left side, select the Autosave option's two check boxes to ensure that both the Editor files and Project desktop will be saved before each project compilation.



### *Creating a Malibu Project*

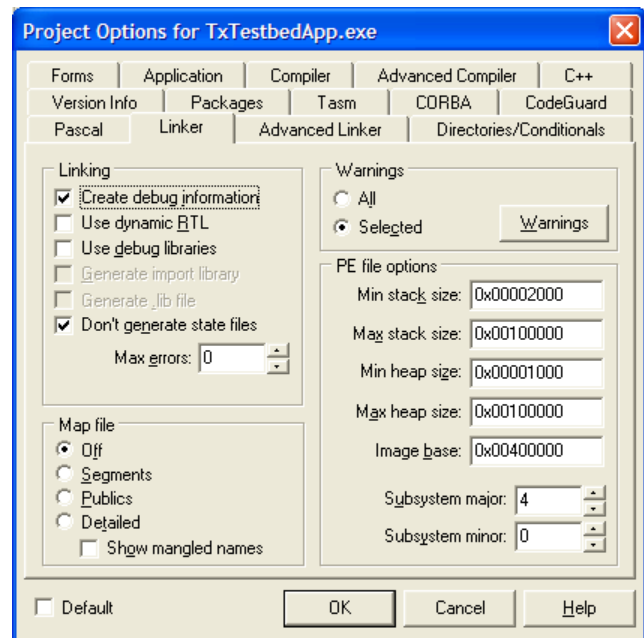
---

Since Borland knows nothing about Malibu, the basic project options need to be modified to allow the compiler to find Malibu. The Project Option dialog is displayed when the menu Project | Options... is selected.

While it is not a required setting, and is not Malibu specific, we recommend that the use of the dynamic C++ run-time library be turned off.

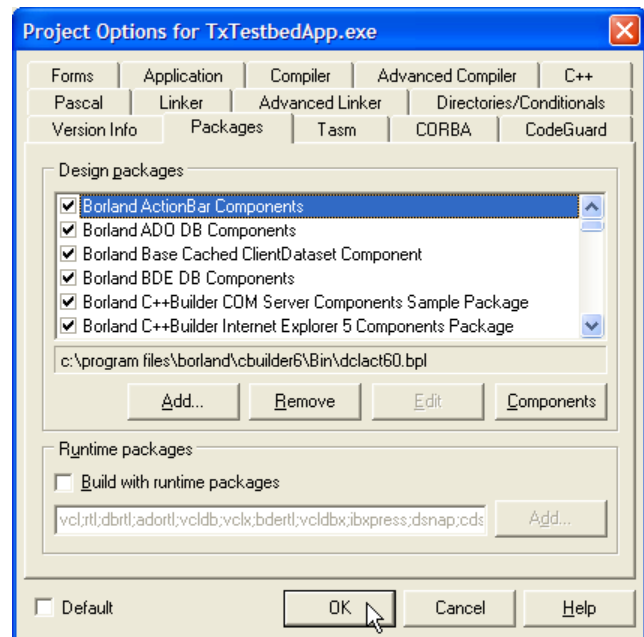
The advantage of using this setting is a somewhat smaller EXE file. The disadvantage of using it is that the DLL file has to accompany the EXE or be installed on the target computer for the program to work. This usually is more trouble than the size decrease is worth.

To disable this option, select the Linker tab. The check box is the second selection in the upper right: Use dynamic RTL.



For similar reasons, Borland's packages should also be static bound. This increases the size of the executable, but allows it to run with a minimum of extra files.

In the Runtime Packages group box in the lower portion of the Packages tab, uncheck the Build with runtime packages check box.



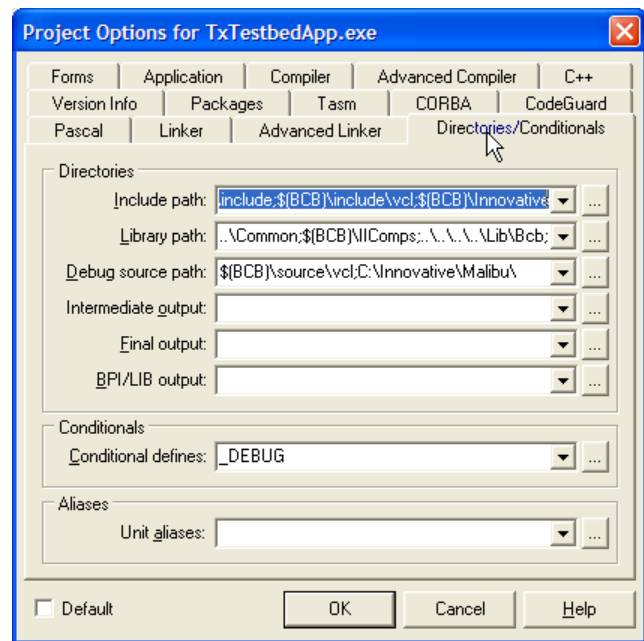
---

The compiler now needs to be informed of Malibu's location. There are three places where the system needs to be informed of the location of library files. These are

- 5) The “Include Path”, which allows header source files to be found in the compilation process.
- 6) The “Library Path”, which allows the linker to find the libraries to search for code modules the application requires.
- 7) The “Debug Source Path”, which is used in debugging to locate code that is being stepped through.

Each of these paths is set by the Directories/Conditionals tab of the Project Options dialog.

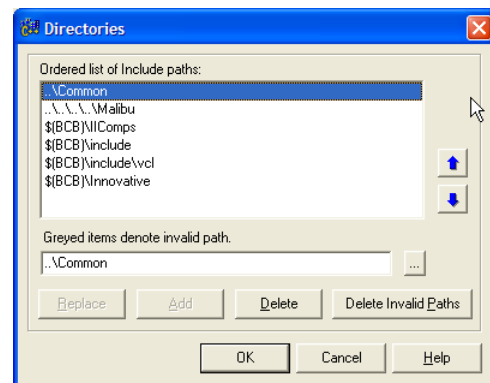
First we will change the Include Path. You can just edit the path itself, but an easier way is to press the '...' button next to the edit control to display a path editing control dialog.



The path editing control allows directories to be rearranged, and each can be edited by selecting it, changing it in the dialog below, and replacing the result in the list.

Here, we wish to add the Malibu source directory to the list. By default, this is installed at C:\Innovative\Malibu. In this case, we use a relative path to indicate its location.

There are several ways that you can define these paths, each with advantages and disadvantages.



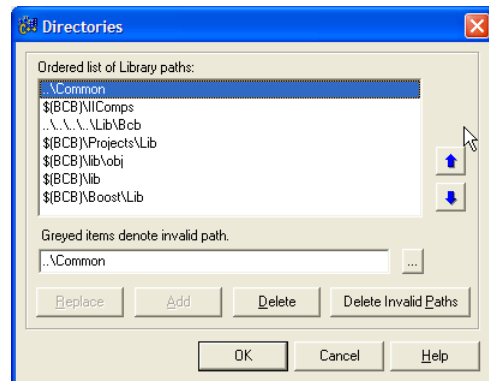
**Table 3. Path Spec Options**

Type of Path Spec	Advantages	Disadvantages	Example
Relative Path	Doesn't need to be changed if project moves only at same level below source directory. Doesn't require the project know where Malibu source directory is or what name the install directory has.	Lots of “.”s. Hard to set up. Requires projects be under the Innovative tree.	..\..\..\Malibu
Absolute Path	Project doesn't have to be located under the Innovative source tree. Project can be moved after creation without change.	Project must be on same drive as Malibu source directory. Project has to know the name of the Malibu install directory.	\Innovative\Malibu
Full Path plus Drive Letter	Project can be anywhere in system.	Requires that Malibu source directory never moves. Project has to know the name and drive of the Malibu install directory.	C:\Innovative\Malibu

The Innovative Examples use relative paths, since we wish to have to specify the name and location of the Malibu source. User projects may have other constraints that make one of the other options more desirable.

To set the library path, select the path editing option button next to the library path edit control.

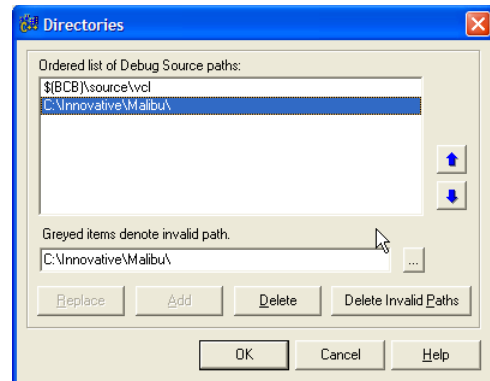
Here, we wish to add the Malibu Library directory to the list. There are several directories for libraries, since the source must be built for each compiler. For Borland C++ Builder 6.0, this is installed at C:\Innovative\Lib\Bcb. In this case, we use a relative path to indicate its location.



---

To set the debug source path, select the path editing option button next to the debug source edit control.

Here, we wish to add the Malibu Source directory to the list. Again, this is installed by default at `C:\Innovative\Malibu\`. In this case, we use an absolute path and drive letter to indicate its location.



---

## Chapter 4. *The Malibu Framework Library*

---

The Malibu framework library `Framework_Mb.lib` contains classes which interact with the operating-system at a low-level to allow inter-thread notification used to synchronize execution via the Malibu `OpenWire::Event` mechanism. Also, a provision to access the application command-line is available, which facilitates authoring of more-portable programs.

The contents of the framework library differ, depending on the application programming interface in use. Under Windows using Borland or Microsoft IDEs, methods within the Win32 API are used to provide inter-thread messaging which forms the basis for thinking between threads. Under Linux, use of the wxWidgets API compiled using GNC C++ is used to accomplish similar thinking. Consequently, be aware that the specific source files in the `Framework_Mb` project differ as a function of the tool chain in use.

---

### *Framework Support Classes*

---

#### **Thinking**

Special care must be taken within multi-threaded applications operating in the context of a user-interface. Specifically, simultaneous calls to UI code from one or more background threads is illegal. To circumvent this limitation, Malibu includes provisions for automatically marshaling (thinking) code execution from a background into the foreground GUI thread context when using `OpenWire::Event` objects.

In order to provide portability amongst various OS frameworks, Malibu implements different versions of the code needed to think between threads. All supported frameworks implement an object of type `ThunkerIntf` within the `Framework_Mb` library to accomplish this. .

Class	Description
<code>ThunkerIntf</code>	Abstract base class from which API-specific thinking class is derived. Provides means of inter-thread notification, usually via a posted message of some sort sent from within the (Malibu) background thread context and dispatched within the foreground (UI) thread context.
<code>CommandLineArguments</code>	Abstract base class from which API-specific concrete class is derived. Provides means of retrieving command-line from operating system using framework-specific methods.

---

## Chapter 5. *The Malibu OS Library*

---

The Malibu utility library `Os_Mb.lib` contains classes which interact with the operating-system at a low-level to allow creation of threads and to synchronize their execution via events, semaphores and mutexes. These features are built atop the Jungo WinDriver package to provide portability between operating systems such as Windows and Linux.

### *Thread Support Classes*

---

#### Threads

It is often useful to run tasks in a separate background thread of execution. Malibu provides class `Innovative::Thread` that simplifies the creating and using of threads, as well as derived classes that are used in Malibu for some commonly used variants. For example, `StartStopThread` adds the ability to freeze a thread by command and the ability to wait on several conditions.

Class	Description
Thread	Abstract base class from which most application threads are derived. Embedded MultipleObjects manages multiple-condition synchronization.
StartStopThread	Extension of Thread base class which implements bipolar execution model: Thread may be running or suspended, but remains live and usable in memory.
Sleep	Block for specified number of milliseconds
uSleep	Block for specified number of microseconds

#### Signals

When using threads, it is essential to provide efficient and safe ways to block until one or more conditions occur. This is generally followed by processing based the signaled condition(s). Malibu includes classes to simplify use of these mainstay building blocks.

Class	Description
Semaphore	Managed counter. When counter is non-zero, thread is signaled.
Event	Boolean state signal. When active, thread is signaled. Both persistent or single-shot modes supported.
MultipleObjects	Smart container for Semaphores and Events. Provides ability to block until either these synchronization objects signals, then unblocks thread, identifies the condition which unblocked the thread to facilitate processing.

---

## Resource Control

When using threads, applications must carefully govern access to shared resources such as memory or hardware devices. Malibu provides a variety of support classes to provide exclusive control to data structures, peripherals or code sections.

Class	Description
Mutex	Basic mutual exclusion.
AtomicAccess	Thread-safe value increment, decrement and exchange
CriticalSection	Exclusive code access via critical section. Method-wise.
CriticalSectionArea	Exclusive code access via critical section. Region-wise.
ThreadSafeQueue	Template class providing thread-safe access to a queue object.

---

## Inter-Thread Communications

A means of passing data efficiently between tasks within an application is commonplace. These classes support such operations.

Class	Description
MailSlot	Inter-process messaging via OS mail slots (Windows-only)
ServerPipe	Inter-process messaging via named pipes (server side)
ClientPipe	Inter-process messaging via named pipes (client side)

## Operating-System

The classes below allow access to process or application-level information maintained by the operating system.

Class	Description
Application	Information about the running application, such as path and name of executable
Registry	Manipulation of registry variables (Windows-only)



---

## Chapter 6. *The Malibu Utility Library*

---

The Malibu utility library `Utility_Mb.lib` contains a wide variety of common helper classes to manipulate elementary objects such strings and buffers; perform file I/O; accurate timing measurements and delays and implement inter-object callbacks.

### *Buffer Classes*

---

The main purpose of the buffer class is to allow the blocks of data transferred around the Malibu system to be handled in chunks freely as objects. The data inside can be accessed by an indexed access just like a C array. In addition, iterators are provided for C++ STL-like iteration over the buffer.

The buffer class is a simple container of data and does not provide advanced access methods such as vector signal processing functions and analysis functions needed in real-time data acquisition and control applications or for post-processing operations. Rather, those features are present in the datagram objects. Most of the classes utilize MMX and SIMD-optimized code using the Intel Performance Primitive libraries that offer the highest performance.

The Malibu buffer classes implement copy-on-write to maximize performance. Malibu's internal, proprietary buffer manager has been designed for optimal real-time performance with minimal runtime heap thrashing and superfluous copy operations.

Class	Description	Application
<code>Buffer</code>	Basic aligned buffer class with built-in header.	Data movement between target hardware and Host PC memory

### **Message Packet Classes**

In addition to the large block buffer classes, there is often a need for a 'command' packet to exchange commands and parameters with a baseboard. The `MatadorMessage` class encapsulates a small 16 word message format used for command I/O on Matador baseboards and C64x DSPs. It is used by convention for other message transfer modes as well, as it provides a good balance of small size with room for parametric data.

---

### **Disk I/O Classes**

Many applications make extensive use of disk files in order to log or analyze collected data. Malibu features a number of classes and stand-alone methods specifically-tailored to aid in these situations.

Class	Description
IniFile	Read/write access to local configuration (INI) files. Useful for persistent application storage.
BinFile	Motorola S-record file reader

## Data Recording and Playback Classes

Malibu provides built-in support and extensive examples for data logging and playback applications. You can record data to and playback data from standard Windows file system disks at up to 50 Mb/s with the components supplied with Malibu. You can also record to network drives for system integration.

Class	Description	Application
BinView	Binview INI file generator class. Useful to create binary data description files which providing formatting information for data display within the Innovative BinView applet.	Tag binary data files via secondary descriptor file. Interface to binary viewer application.
DataLogger	Records raw data received from any input device to Windows local or network disks.	High bandwidth data recording.
RamDataLogger	Records raw data received from any input device to Windows local or network disks.	High bandwidth data recording.
DataPlayer	Retrieves raw data previously stored to Windows local or network disk for real-time output.	High bandwidth data playback.

The `DataPlayer` class may be used to read signals from a binary data file to be sent downstream. The downstream chain could be as simple as a direct connection to a hardware output pin such as a module DAC or a baseboard output pin, or a complex chain of analysis components, each processing the data in an elaborate, application-specific manner. The component automatically fetches data from the disk as needed to sustain the real-time data flow to downstream components. A special property, `Mode`, allows continuous replay of the data contained in the file when the end-of-file condition is reached.

The `DataLogger` class may be used to store signals received from upstream into a binary data file. The class automatically stores received data blocks to disk as needed to sustain the real-time data flow from upstream components. A special property, `Ceiling`, allows capping of the total amount of data logged to the data file.

## System Components

A useful set of system components saves development time. Classes and functions are provided for precision profiling and delays, automatically marshal event processing into the foreground thread A stop watch allows for quick application profiling while other components give direct access to data in RAM, facilitate the numeric display of data arrays and simplify the use of registered Windows messages.

Class	Description	Application
StopWatch	Precision sub-microsecond elapsed time component for code profiling	Application profiling, precise delays.

OpenWire::Event	Inter-class notification	Implementing callbacks within libraries. May be synchronized (marshaled) to main thread or run in caller's context
MalibuException	Exception base class	Error handling
PathSpec	Class used to extract and insert components of file path specifications	Construction/analysis of path specifications by parts

## File Support Methods

Malibu includes some stand alone functions for common file operations.

Method	Description
FileExists	Determine file presence
FileSize	Determine file size

## String Support

The following classes allow management of collection of strings in Malibu:

Class	Description
StringList	Quick text file parser object
StringVector	Quick text file parser object

In addition there are conversion functions between numeric values and text for I/O to the user interface of an application.

Function	Description
BinToHex	Efficiently converts a binary array to hex string equivalent.
Endian	Endian reversal
FloatToString	Returns the string representation of a double value.
HexToBin	Efficiently converts a hex string to byte equivalent.
IntToString	Returns the string representation of an integer value.
StringToFloat	Convert a string into a floating point value
StringToHex	Convert a string representing a hex string into an integer.
StringToInt	Converts a string into an integer value.

## Matlab Interface Classes

Mathworks MatLab and Simulink are powerful analysis and simulation tools. Malibu provides tools to remotely control instances of MatLab, and to transfer data between a C++ application and the Matlab workspace at rates beyond 100 MB/s.

---

---

Method	Description
MatlabMatrix	Manipulate Matlab-compatible vectors of various types within C++ programs
MatlabFile	Read or write vectors from standard Matlab .m files.
MatlabEngine	Launch or close a Matlab instance. Allows use of Matlab as a C++ coprocessor.

## Data Set Classes

Generally, data flow between target hardware and host system memory is organized as interleaved data from all enabled channels in module-specific binary format. This is done to maintain the highest data flow rates. Malibu's DataSet objects provide channelized access to interleaved data stored in standard disk files, to simplify post-analysis or pre-calculation of output data.

Method	Description
DataSet	Channelized read/write methods on data set containing interleaved data in binary file format. Automatic translation between native format data and integer, floating point or u/A-law compressed data sets.
FileDataSet	Data set access to interleaved data stored in disk file.
PacketFileDataSet	Similar to FileDataSet, but specialized to accommodate buffer-prefaced data buffers within a disk file. This packet buffers are produced by all PMC/XMC modules, the M6713, P25M and and other baseboards.
RamDataSet	Similar to FileDataSet, but specialized to accommodate interleaved data stored in a RAM buffer.

---

## Chapter 7. *The Malibu Hardware Library*

---

The Malibu hardware library `Hardware_Mb.lib` contains software interfaces and support classes for the generic hardware devices used on Innovative baseboards. It includes provision for COFF file parsing and downloads, HPI DSP bus access, message I/O structures, XSVF parsing and loading, FPGA loading via SelectMap, access to baseboard calibration ROM and debug scripts.

### *Target I/O Streaming Classes*

---

Data I/O between the target and the host is a major component of many applications. It is also one of the most complicated tasks, involving interrupts on both target and host, busmastering, DMA, data buffering and buffer management, among other issues. In Malibu, each particular style of I/O is packaged into a separate Stream class. When associated with a baseboard class, the stream can provide the methods and events needed for efficient I/O to and from the target.

Before being used, a stream must be attached to a baseboard with the `ConnectTo()` method. Only if this method of streaming is supported on a baseboard will the `ConnectTo()` compile. The `DisconnectFrom()` method removes the connection.

A limitation on all busmaster communications that streams commonly used is that single packet size is limited to what can fit into the allocated busmaster region. This region must be reserved for use by the Innovative ReserveMemDsp applet and is subsequently allocated by the device driver at O.S. startup. The maximum size this buffer can be sized to can depend on the system BIOS or Windows. In any event, it is often relatively-easy to send large amounts of data in multiple packets rather than depend on a single transfer.

Stream	Usage
PacketStream	Packet based streaming, with data from separate data sources in individual packets.
TiBusmasterStream	Packet based streaming from TI CPUs with PCI bus-mastering.
BlockStream	Matador style streaming, with no header and interleaved channels.

`Innovative::PacketStream` provides packet based streaming to the newer PMC cards and the M6713 baseboard. Packets may be of different sizes, the size being inserted into the packet header. A baseboard may have a number of 'peripheral' devices that can source or consume data. For instance, a TX PMC module features four D/A channels addressed as two device pairs. Each is accessible via Peripheral ID #0 and #1. Data is marked by a Peripheral ID field to allow routing according to the source or destination of the data.

By contrast, `Innovative::BlockStream` on the Toro, Conejo and Delfin baseboards are designed for analog processing and produce more typical data streams containing interleaved data from all enabled analog channels. All blocks are of uniform size, and all data is of a uniform format for that run.

The stream `Innovative::TiBusmasterStream` supports both command packets and buffers directly to the TI C64x CPU. There are no headers, and data packets may be of any size.

---

## Interface Classes

---

Interface Object	Subsystem
IUsesOnboardCpu	CPU related functions such as Booting and COFF Downloading.
IUsesVirtexFpgaLoader	User Logic Loading.
IUsesVirtexJtagLoader	Logic EEPROM programming via JTAG scan path.
PacketDeviceMap	Packet-based, bus mastering transfers. Used by all PMC modules and M6713
IUsesCalibration	Storage/retrieval of analog cal coefficients in PMC EEPROM
IUsesFpgaLogicVersionInfo	Standardized logic version information retrieval

The Interface Object classes include the methods to perform the subsystem tasks, and they also include the events that can be hooked by the application in the subsystem. For example, in the COFF loading there are events that allow the intercepting of error and status messages produced during the load, and a progress event that can be used to provide user feedback during the process.

---

## Timebase Classes

---

Class	Description
Ics8442	Phase-lock loop timebases for high-speed clock generation
Ics8402	

Certain baseboards have high precision timebases on board. These classes are available in the baseboard object to program these timers.

---

## Hardware Support Classes

---

Additional support classes.

Class	Description
ClockBase	Clock settings management class
GcScripter	Script interpreter for GrayChip devices
Scripter	Add scripting to a class
HpiEngine	Memory-region access object
PmcIdrom	Base class for Flash ROM Block
ZbtRam	Class to interact with ZBT RAM on a device

---

## *Hardware Register Classes*

---

In interacting with the memory mapped registers of the hardware, some support classes for the different characteristics of a register were created. Usually these will only be used inside of a baseboard support class.

Class	Description
AddressingSpace	Memory-region access object
ReadOnlyRegister	Read-only register access
ReadOnlyRegisterBit	Read-only register bit access
ReadOnlyRegisterBitGroup	Register field access
Register	Read/write register access
RegisterArray	Register array access
RegisterBit	Register bit access
RegisterBitGroup	Register field access
ShadowRegister	Control a memory location as a register with a shadow showing the current state
CachedShadowRegister	Deferred-update ShadowRegister object

---

## Chapter 8. *The Malibu Analysis Library*

---

The Malibu analysis library `Utility_Mb.lib` provides classes that perform common signal processing functions such as filters and FFTs, logging and playback of waveforms and other classes needed in data acquisition and control applications. These routines make use of the Intel Performance Primitives libraries in order to achieve optimal performance. Consequently, use of the classes within this library create a runtime dependency on the IPP shared object codes, which are packaged as DLLs under Windows and .sa files under Linux.

If this dependency is problematic in your application, do not use any of the classes within this library, exclude `#include <Analysis_Mb.h>` from your application source and avoid calling the `Init::UsePerformanceMemoryFunctions()` which forces binding to the IPP libraries.

### *Statistical Analysis Classes*

---

The analysis classes provide access to common DSP algorithms and analysis functions. Most of the components are MMX and SIMD optimized code from the Intel libraries that offer the highest performance.

Class	Description	Application
Stats	Statistics: Min, max, mean, std dev, dynamic range, integrals	Signal analysis
AdcStats	A/D statistics: Signal-Noise, SINAD, total-harmonic distortion, harmonic analysis User application data pump. Channelized data available on events.	A/D and D/A characterization

### *Signal Processing Classes*

---

Common signal processing operations such as FFTs, and filters are implemented as components within the Malibu package. These operations have been implemented using the Intel IPP library for performance. The IPP library uses the full features of Pentium processors to make analysis even more efficient.

Class	Description	Application
-------	-------------	-------------



BandPass	Band-pass filter, variable # taps, automatic digital filter designer.	Waveform filtering. >100 MTaps/sec on Pentium IV 3GHz
BandStop	Band-stop filter, variable # taps, automatic digital filter designer.	
Highpass	High-pass filter, variable # taps, automatic digital filter designer	
Lowpass	Low-pass filter, variable # taps, automatic digital filter designer	
Fir	Generic FIR filter. Variable # taps	
Iir	Generic IIR filter. Variable # taps	
Fourier	Time to frequency domain transformations, adjustable size, numerous window functions.	>100,000 1K-point transforms/sec on Pentium IV 3 GHz
InverseFourier	Frequency to time domain transformations, adjustable size.	

The `Fourier` class may be used to convert signals between the frequency and time domains. Properties control the number of points in the FFT frame, from 128 to 512K points. The `InverseFourier` class performs inverse transformations (from frequency to time domain). A property is available to enable windowing of time-series input data prior to transformation using common windows such as Hanning and Blackman.

The `LowPass`, `HighPass`, `BandPass`, `BandStop`, `Iir` and `Fir` classes perform filtering operations on data blocks. Properties control the number of filter taps to be used to implement the filter, the cutoff frequencies and the sampling rate. The `Filter()` method performs a convolution on a data block using filter coefficients, which are automatically calculated using the specified properties. As with the FFT component, a property is available to enable windowing of time-series input data prior to transformation using common windows such as Hanning and Blackman.

## Signal Generation Classes

The `SignalGen` class generates contiguous sinusoidal, triangular or square waves in block format suitable for consumption by other processing functions, or to be sent to target hardware as block data. A single `SignalGen` object can provide blocks of data to multiple independent streaming output channels within an application, if so desired.

The `GaussGen` class generates random noise, distributed in a Gaussian distribution about a mean value. This mean value and its standard deviation can be changed to suit the needs of the application.

The `RandomGen` class also generates a random noise source, but with a different distribution. This noise distribution is flat, a uniform distribution between an upper and lower boundary.

### Digital Signal Processing Classes

Class	Description	Application
<code>CommonGen</code>	Base class for all signal generators.	Allows creation of user-defined filters

---

---

GaussGen	User-adjustable Gaussian noise source	Frequency response testing, vibration
SignalGen	User-adjustable arbitrary signal source. Sin, Cos, Triangle, Square waves	
RandomGen	User adjustable random noise source	

---

---

## Chapter 9. *The Malibu PCI Library*

---

The Malibu ethernet library `Pci_Mb.lib` provides support for baseboards that use the PCI bus and busmastering as the primary means of communication between target and host.

### *PCI Baseboard Classes*

---

A major part of the purpose of the Malibu library is to provide easy interaction with Innovative hardware products. These products all require means of loading logic, software to CPUs present, configuration and control, and providing the transfer of data and commands to and from the board.

In the Malibu library, most of the details of these procedures is contained inside the library so that the application writer does not need to concern themselves with low level details. This means that it is possible for boards with different means of performing a function can be used in similar or identical ways by an application, simplifying the learning curve for the user.

#### **Baseboards and PMC Modules**

The DSP baseboard components listed below encapsulate the capabilities of the baseboard hardware. For more information about any baseboard class, see the hardware manual for the baseboard. It includes a chapter giving an overview of the object.

Object	Product
Matador	Toro, Delfin, Conejo, Lobo, Oruga DSP baseboards
C64xDsp	TMS320C6416 DSP hosted on Quadia and Quixote baseboards
M6713	M6713 PCI DSP baseboard
Quadia	Quadia and Duet baseboard features (not including the four C64x CPUs).
Quixote	Quixote baseboard features (not including the one C64x CPU).

Baseboard objects are created in a one-to-one relationship with hardware. To associate a baseboard with a hardware device, each device in a system is given a unique index, known as the target number. These indexes are unique for each type of baseboard. Once the target number has been assigned, the baseboard can be attached to the hardware with an `Open()` command. If the target is not present, this method will throw an exception. Otherwise, the baseboard is ready for use. To detach from hardware, use the `Close()` method.

Baseboard objects also have methods to allow access to the features of the board. Some of these are unique to a particular baseboard, and are implemented as simple methods. Other board features are more complex or are shared on several baseboards. These are called subsystems. Logic loading and COFF file loading are examples of subsystems.

---

Subsystems are implemented as an interface class that can be shared from baseboard to baseboard, even if the implementation differs internally. Each baseboard can provide the subsystems that it requires. For example, the Quadia baseboard class has interfaces to load each of the twin user-programmable Virtex II FPGAs.

## *PMC Module Classes*

---

The PMC Module classes provide application access to Innovative's PMC module family. Like the regular baseboards, these modules all require means of loading logic, configuration and control, and providing the transfer of data and commands to and from the board. For more information about any PMC module class, see the hardware manual for the module. It includes a chapter giving an overview of the object and a detailed annotated example.

Object	Product
Uwb	Ultra wide-band analog capture
Sio	High-speed serial I/O
Tx	High-speed analog waveform playback and streaming
DigitalReceiver	Wide-band analog capture and hardware down-conversion

## *XMC Module Classes*

---

The XMC Module classes provide application access to Innovative's X-series module family. Like the regular baseboards, these modules all require means of loading logic, configuration and control, and providing the transfer of data and commands to and from the board. For more information about any XMC module class, see the hardware manual for the module. It includes a chapter giving an overview of the object and a detailed annotated example.

Object	Product
X5-400M	400 MHz A/D and D/A
X3-10M	Eight-channel, 10 MHz A/D
X3-SD	Sixteen-channel, sigma-delta A/D
X3-SDF	Four-channel, high-speed, instrumentation-grade sigma-delta A/D
X3-Servo	Twelve-channel, high-speed, instrumentation-grade successive-approximation A/D for low-latency servo applications

---

---

---

## Chapter 10. *The Malibu Ethernet Library*

---

The Malibu ethernet library `Ethernet_Mb.lib` provides support for baseboards that use ethernet as the primary means of communication between target and host.

### *Baseboard Classes*

---

At present, only a single baseboard uses the Ethernet interface to communicate. For more information about the baseboard class, see the hardware manual for the baseboard. It includes a chapter giving an overview of the object.

Object	Product
Sbc6713e	Supports the SBC6713e ethernet single board processor.

---

## Chapter 11. *Packet Polling Library*

---

The packet polling classes in Malibu library enables packetized data transfers between user applications and Innovative hardware without interrupt signaling. The *PacketPollMgr* class is the primary application interfacing class supplying this capability. Two sub-classes of *PacketPollThread* class perform work of moving data packets. Whenever a data packet is received or is required for transfer, these I/O threads signals the application by triggering an event of the type *PacketPollDataEvent* class. Data packets are buffered in the *PacketPollQueue* class.

Source Files: *PacketPoll\_Mb.h* and *PacketPoll\_Mb.cpp*.

---

### *PacketPollMgr Class*

---

The *PacketPollMgr* class encapsulates all packet polling functionalities. Object of this class encapsulate all functionalities required to perform polling operations.

#### Con/Destructor

On creation, the *PacketPollMgr* class constructor expects a reference to the base board class, such as *X3Servo*.

```
PacketPollMgr(X3Servo & board);  
~PacketPollMgr();
```

#### Events

To interact with application level, *PacketPollMgr* exposes two events. These events are triggered at each iteration of the polling threads. Applications must register event handling functions in order to extract or provide new data packets to *PacketPollMgr* for processing.

```
OpenWire::EventHandler<PacketPollDataEvent> OnDataRequired;  
OpenWire::EventHandler<PacketPollDataEvent> OnDataAvailable;
```

#### Interfaces

##### Start()/Stop()

To start or stop the polling operation, application calls the *start()* or *stop()* functions. The *start()* function need to know the size of packets for transmission. Packets must be fix sized.

```
void Start(ii32 pktSize);  
void Stop();
```

---

### Send()/Recv()

To add a packet to the send queue, call `Send()`. To extract received packet, call `Recv()`. It is recommended that these calls be made in the registered event handlers for optimum performance.

```
bool Send( Buffer & packet );
bool Recv( Buffer & packet );
```

### SetFifoDacDelay()

This function set the amount of processing delay offset between ADC and DAC, in nanoseconds. This time should correspond to the turn around time of the incoming samples.

```
ii32 SetFifoDacDelay(ii32 ns);
```

---

## PacketPollDataThread Class

*PacketPollDataThread* class is a sub-class of *PacketPollThread* class. Each instance of this class create and manage a high priority thread. The thread begin and end execution when *start()* and *end()* functions are called. For more detail, see documentation on *StartStopThread* class. With each iteration of the thread's execution loop, one data packet is received and send between the DSP board and a host-side software queue.

---

## PacketPollDataEvent Class

With each data packet received or required for transfer, these I/O threads classes signals the application by triggering an event of the type *PacketPollDataEvent* class. The application should register two event handling functions with this class in order to process the requests.

```
// Register packet handlers with Poll Manager
PollMgr.OnDataRequired.SetEvent(this, &ApplicationIo::HandlePollDataRequired);
PollMgr.OnDataRequired.Unsynchronize();
PollMgr.OnDataAvailable.SetEvent(this, &ApplicationIo::HandlePollDataAvailable);
PollMgr.OnDataAvailable.Unsynchronize();
```

*PacketPollDataEvent* class has a pointer to the *PacketPollMgr* class. The application can access / transfer pending data packet by calling the `Recv()/Send()` via this pointer.

```
//-----
// ApplicationIo::HandlePollDataRequired() - Handle new data requests
//-----
void ApplicationIo::HandlePollDataRequired(Innovative::PacketPollDataEvent & Event)
{
    //pull packet from queue
    if (pollQ.Pop(TxPacket))
    {
        if (Event.Sender->Send(TxPacket))
        {
            ...
        }
    }
}
```



---



---

```

//-----
// ApplicationIo::HandlePollDataAvailable() - Handle new data from PollMgr
//-----
void ApplicationIo::HandlePollDataAvailable(Innovative::PacketPollDataEvent & Event)
{
    if (Event.Sender->Recv(RxPacket))
    {
        ...
    }
}

```

## *PacketPollQueue Class*

---

This class provide data storage and retrival in FIFO order. It stores data packets of class Buffer in a thread safe queue of class ThreadSafeQueue. The capacity of the queue can be adjusted or retrieved by calling QueueCapacity().

```

void          QueueCapacity(size_t cap_in_blocks);
size_t        QueueCapacity() const;

```

Packets can be added or removed rom the queue by calling Push() or Pop(). True is return if these operations succeed; false otherwise.

```

bool          Push(Buffer & packet);
bool          Pop(Buffer & packet);

```

Current queue depth, i.e. the number of paackets kept in the queue, is returned by calling Depth().

```

size_t        Depth();

```

Call Clear() to remove all packets from the queue and reset the queue depth to zero.

```

void          Clear();

```

---

## Chapter 12. *Writing Custom Applications*

---

Most scientific and engineering applications require the acquisition and storage of data for analysis after the fact. Even in cases where most data analysis is done in place, there is usually a requirement that some data be saved to monitor the system. In many cases a pure data that does no immediate processing is the most common application.

The X6 XMC card family contains high bandwidth analog capture and playback modules with an advanced architecture that provides ultimate flexibility and speed for the most advanced hardware-assisted signal processing and ultrasonic signal capture. The maximum data rate from these module are often in the hundreds of MSPS. This means that a simple logger that saves all of the data to the host disk is not feasible using standard operating system disk I/O calls, as the slower disk writes eventually cause overflow and data loss in the streaming system.

Some modules support decimation so that long duration samples can be taken without data. Also quick snapshots of analog data can be taken without loss as long as the amount of data is less than the net capacity of system memory and what the baseboard holds. The example program provided for nearly all cards is this limited capacity data logger, called the Snap example

### *The Snap Example*

---

The Snap example in each software distribution demonstrates this logging functionality. It consists of a host program in Windows, which works with the logic provided on the board's flash to stream data to the host. It uses the Innovative Malibu software libraries to accomplish the tasks.

### **Tools Required**

In general, writing applications for the XMC requires the development of host program. This requires a development environment, a debugger, and a set of support libraries from Innovative.

**Table 4. Development Tools for the Windows Snap Example**

Processor	Development Environment	Innovative Toolset	Project Directory
Host PC	Embarcadero Developers Studio C++	Malibu	Examples\Snap\Bcb11
			Examples\Snap\VC9
	Microsoft Visual Studio		Examples\Snap\Qt
	QtCreator		Examples\Snap\Common
	Common Host Code		

On the host side, the Malibu library is provided in source form, plus pre-compiled Microsoft, Borland or GCC libraries. The application code that implements the entirety of the board-specific functionality of example is factored into the

---

`ApplicationIo.cpp/h` unit. All user interface aspects of the program are completely independent from the code in `ApplicationIo`, which contains code portable to any compilation environment (i.e., it is common code). While each compiler implements the GUI differently, each version of the example project uses the same code file to interact with the hardware and acquire data.

## Program Design

The Snap example is designed to allow repeated data reception operations on command from the host. As mentioned earlier, received data can be saved as Host disk files. When using modest sample rates, data can be logged to standard disk files. However, full bandwidth storage of multiple A/D channels can require up more capacity, so a dedicated RAID0 drive array for data storage may be required, or data may have to be cached online and stored after stopping data flow. The example application software is written to perform minimal processing of received data and is a suitable template for high-bandwidth applications.

The example uses various configuration commands to prepare the module for data flow. Parametric information is obtained from a Host GUI application, but the code is written to be GUI-agnostic. All board-specific I/O is performed within the `ApplicationIo.cpp/.h` unit. Data is transferred from the module to the Host as packets of `Buffer` class objects.

## *The Host Application*

---

The picture to the right shows the main window of an X6 example (for the X6-RX). This form is from the designer of the Embarcadero CBuilder -version of the example, but the Microsoft and Qt versions are similar. It shows the layout of the controls of the User Interface.

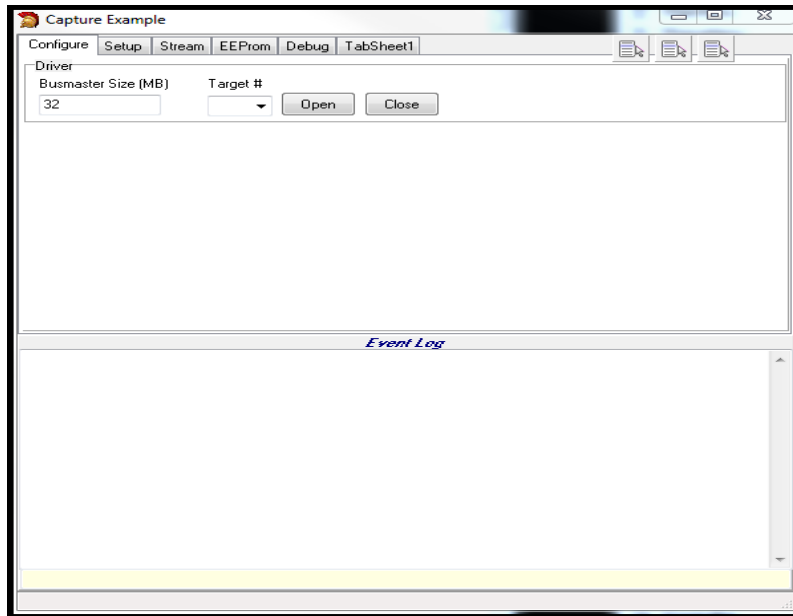
## User Interface

This application has five tabs. Each tab has its own significance and usage, though few are interrelated. All these tabs share a common area, which displays messages and feedback throughout the operation of the program.

### Configure Tab

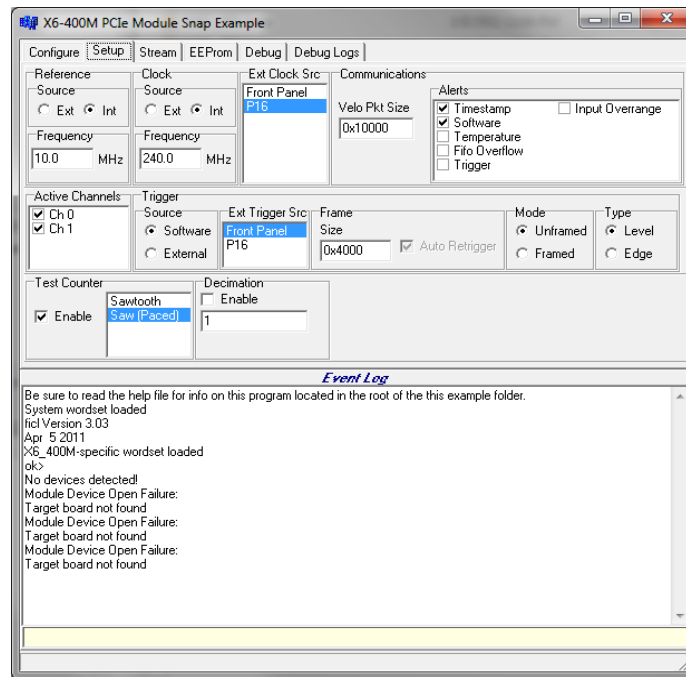
As soon as the application is launched, the Configure tab is displayed. In this tab, a combo box is available to allow the selection of the device from those present in the system. All XMC family devices of whatever type share a sequence of target number identifiers. The first board found is Target 0, the second Target 1, and so on.

Click the Open button to open the driver. To change targets, click the Close button to close the driver, select the number of the desired target using the `Target #` combo box, then click `Open` to open communications with the specified target module. The order of the targets is determined by the location in the PCI bus, so it will remain unchanged from run to run unless the board is moved to a different slot or another target is installed.



### Setup Tab

This tab has a set of controls that hold the parameters for transmission. These settings are delivered to the target and configure the target accordingly. This tab has several sections.



The controls in the Clock group offer configurations and routing of the clock. The clock for the FPGA can come from an external clock (EXT) or from an onboard phase-locked loop (INT). The selection can be made at upper right corner of this section. The output sample rate of the selected clock source is specified in the Output field in MHz.

The controls in the Reference group specify the source and frequency of the reference clock. The behavior of the reference clock and the meaning of these controls are a function specified Clock Source specified above. If the clock source is internal, the Reference controls specify the source and rate of the reference clock routed to the onboard PLL. If the clock source is external, the Reference controls specify the source and rate of the sample clock, bypassing the PLL. In this mode, the ratio of the Reference Frequency/Clock Frequency is used to control an onboard clock divider, supporting sampling at integer submultiple rates of the reference clock. The table below summarizes the behavior.

The physical line driving the external clock/reference signal has a multiplexer on it that can select one of two sources for the signal. One is the front panel input, the second choice is to use the P16 connector clock as the source. This signal is driven by Innovative's SBC-Comex product line.

Reference Source	Clock Source	PLL Reference	Sample Clock	Sample Rate
EXT	EXT	N/A	EXT CLK	External rate / (REF_FREQ/CLK_FREQ)
INT	EXT	N/A	Onboard Oscillator	100 MHz / (REF_FREQ/CLK_FREQ)
EXT	INT	EXT CLK	PLL	CLK_FREQ
INT	INT	Onboard Oscillator	PLL	CLK_FREQ

---

The Communications section controls the Alert features and the input data packets size. Checking the box next to an alert will allow the logic to generate an alert if that condition occurs during data streaming. This alert can then be left in the data stream, or extracted to notify the application. The Velo Packet Size edit control specifies the size of the Velocia packets that the module will use to

In the Channels section, we can specify number of channels to activate. Selecting a channel will flow data from that data source during streaming.

The Trigger group contains controls that affect the way that I/O starts and stops. Triggers act as a gate for I/O - no data flows until a trigger has been received. Triggers may be initiated via software or hardware, depending on the Trigger | Source control. If software, the application program must issue a command to initiate data flow. If hardware, a signal applied to the external trigger connector controls data flow. The external trigger line may also be routed from one of two sources, either the front panel connector or the P16 clock signal line.

Triggers are modal depending on the Trigger | Mode control. In Unframed mode, triggers are level sensitive and data flow proceeds while the trigger is in the high (active) state and stops while the trigger is in the low (inactive) state. This mode is appropriate for conventional data acquisition applications. In Framed mode, triggers can be selected to be edge sensitive or level sensitive on the X6. Upon detection of each trigger, Trigger | Frame | Count samples are acquired from all active channels, then acquisition terminates until the next trigger edge is detected. This mode is well-suited for applications such as spectral analysis using fixed input buffers submitted to FFTs.

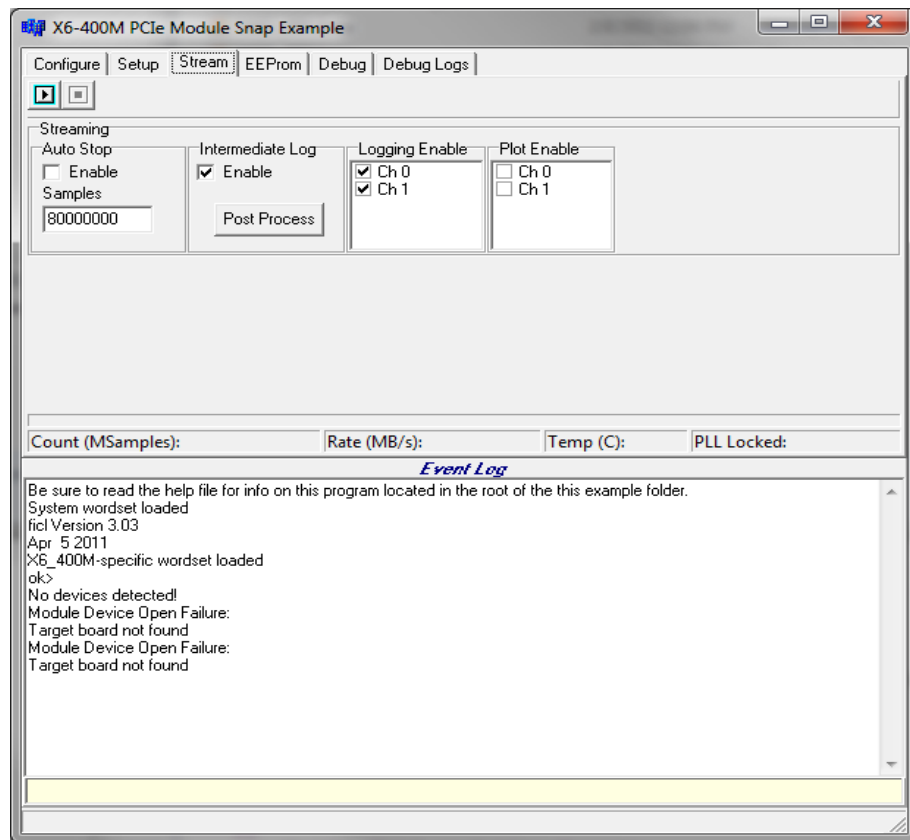
The module supports one or more test modes for module debugging and system test. The Test Counter Enable is used to turn on test mode and substitutes a ramp signal with channel number in the upper byte of the data. The Decimation section sets up the decimation logic to discard data, reducing the incoming data rate.

### **Stream Tab**

The two buttons in the button bar start and stop data streaming. Press the running man button to start streaming data. Press the stop button to stop streaming, unless the stream has stopped itself. When streaming, the status bar data is collected and displayed. This includes a count of the data samples received, the data rate, the measured temperature of the board logic, and the PLL locked indicator bit value.

Data is logged to an intermediate file that holds the Velocia buffers with the Vita Buffer data. This can be read by Binview in native mode for analysis. The Post Processing button will extract the data from the Intermediate file and convert it into X5 compatible data files for use with older versions if desired.

The Auto Stop option determines if data streaming stops after collecting the given number of points (a Snapshot of data) or continues to stream forever until manually stopped.



## Host Side Program Organization

The Malibu library is designed to be built in numerous different host environments: Borland/CodeGear/Embarcadero CBuilder, Microsoft Visual Studio under Windows or QtCreator under Windows/Linux. Because the library is built using and identical source code base in all environments, the code that interacts with Malibu is identical regardless of the toolchain used. Within examples, the portion of the code which interacts with Malibu is factored into a class, *ApplicationIo* in the files *ApplicationIo.cpp* and *.h*. This class acts identically on all frameworks and OS platforms.

It is very important to realize that the most expedient method to create a custom application, regardless of the framework type used (.NET, VCL, Qt, wxWidgets or console mode), is to incorporate the *ApplicationIo* class in its entirety directly into your custom app. *ApplicationIo* contains only portable code, without any framework dependencies whatsoever. It contains all of the essential hardware initialization code, event handlers, etc to support efficient communications with the board. You'll likely need all of the features of *ApplicationIo* in your application - so why reinvent the wheel?

Many users are initially unclear on how *ApplicationIo* can remain agnostic to the framework. This is accomplished by factoring all functions used to relay information between framework-specific code and the *ApplicationIo* into a small separate class called *UserInterface* within the supplied examples.

The concrete *UserInterface* object derives from the pure abstract *IUserInterface* base class. *IUserInterface* simply specifies the functions that must be supplied by a user application to provide a bridge to the stock features of the *ApplicationIo* object. Your application must create a *UserInterface* class and pass it to the constructor of the *ApplicationIo* object, but your implementation of the *IUserInterface* methods may degenerate to trivial no-ops if desired.

---

The Main form of the application creates an *ApplicationIo* to perform the work of the example. The UI can call the methods of the *ApplicationIo* to perform the work when, for example, a button is pressed or a control changed.

Sometimes, however, the *ApplicationIo* object needs to 'call back into' the UI. But since the code here is common, it can't use a pointer to the main window or form, as this would make *ApplicationIo* have to know details of Borland or the VC environment in use.

The standard solution to decouple the *ApplicationIo* from the form is to use an Interface class to hide the implementation. An interface class is an abstract class that defines a set of methods that can be called by a client class (here, *ApplicationIo*). The other class produces an implementation of the Interface by either multiple inheriting from the interface, or by creating a separate helper class object that derives from the interface. In either case the implementing class forwards the call to the UI form class to perform the action. *ApplicationIo* only has to know how to deal with a pointer to a class that implements the interface, and all UI dependencies are hidden.

The predefined *IUserInterface* interface class is defined in *ApplicationIo.h*. The constructor of *ApplicationIo* requires a pointer to the interface, which is saved and used to perform the actual updates to the UI inside of *ApplicationIo's* methods.

## ApplicationIo

### Initialization

The main form creates an *ApplicationIo* object in its constructor. The object creates a number of Malibu objects at once as can be seen from this detail from the header *ApplicationIo.h*.

```
ModuleIo                                Module;
IUserInterface *                        UI;
Innovative::VitaPacketStream            Stream;
Innovative::VitaPacketParser            Vpp;
Innovative::TriggerManager              Trig;
Innovative::SoftwareTimer               Timer;
MultiLoggerManager                      MLM;
Innovative::StopWatch                   RunTimeSW;
Innovative::DataLogger                  IntermediateLogr;
Innovative::DataPlayer                  IntermediatePlayer;

// App State variables
bool                                     Opened;
bool                                     StreamConnected;
bool                                     Stopped;
// App Status variables
double                                  FBlockRate;
ii64                                    FWordCount;
int                                     SamplesPerWord;
ii64                                    WordsToLog;

Innovative::AveragedRate                 Time;
Innovative::AveragedRate                 BytesPerBlock;
...
```

In Malibu, objects are defined to represent units of hardware as well as software units. The *ModuleIo* object represents the board. The *VitaPacketStream* object encapsulates supported, board-specific operations related to I/O Streaming. The *VitaPacketParser* object encapsulates capabilities to extract VITA49 packets from the native Velocia packets supplied by the board during streaming. The *MultiLoggerManager* object provides file storage for each peripheral type parsed from the stream by the parser, facilitating analysis of data received.



---

The ApplicationIo object derives from type FicIo. FICL is an abbreviation for Forth-Inspired Control Language. A simple Forth interpreter is available within the module object which can be used as a simple scripting language, for the purposes of performing hardware initialization during FPGA firmware development.

When the Open button is pressed, the ApplicationIo object begins the process of opening the board driver and setting up the board for a run. The first thing done is to link Malibu software events to callback functions in the applications by setting the handler functions.

Malibu uses *events* to allow functions to be 'plugged into' the library to be called at certain times or in response to certain events detected. Events allow a tight integration between an application and the library:

```
//  
// Configure Trigger Manager Event Handlers  
Trig.OnDisableTrigger.SetEvent(this, &ApplicationIo::HandleDisableTrigger);  
Trig.OnExternalTrigger.SetEvent(this, &ApplicationIo::HandleExternalTrigger);  
Trig.OnSoftwareTrigger.SetEvent(this, &ApplicationIo::HandleSoftwareTrigger);
```

This code attaches event handlers associated with the various trigger conditions. For example, the OnDisableTrigger event fires when triggering is disabled. When Malibu detects that condition, it calls the ApplicationIo::HandleDisableTrigger method, providing a means of user-specific processing under that condition.

In a similar manner, the events hooked below are called at strategic times during data streaming:

```
//  
// Configure Module Event Handlers  
Module().OnBeforeStreamStart.SetEvent(this, &ApplicationIo::HandleBeforeStreamStart);  
Module().OnBeforeStreamStart.Synchronize();  
Module().OnAfterStreamStart.SetEvent(this, &ApplicationIo::HandleAfterStreamStart);  
Module().OnAfterStreamStart.Synchronize();  
Module().OnAfterStreamStop.SetEvent(this, &ApplicationIo::HandleAfterStreamStop);  
Module().OnAfterStreamStop.Synchronize();
```

HandleBeforeStreamStart, HandleAfterStreamStart and HandleAfterStreamStop handle events issued on before stream start, after stream start and after stream stop respectively. These handlers could be designed to perform multiple tasks as event occurs including displaying messages for user. These events are tagged as Synchronized, so Malibu will marshal the execution of the handlers for these events into the main thread context, allowing the handlers to perform user-interface operations.

The hooking of alerts are factored into a method within the Module object, since different modules may provide different alert capabilities.

```
// Alerts  
Module.HookAlerts();
```

This code attaches alert processing event handlers to their corresponding events. Alerts are packets that the module generates and sends to the Host as packets containing out-of-band information concerning the state of the module. For instance, if the analog inputs were subjected to an input over-range, an alert packet would be sent to the Host, interspersed into the data stream, indicating the condition. This information can be acted upon immediately, or simply logged along with analog data for subsequent post-analysis.

```
//  
// Configure Stream Event Handlers  
Stream.OnVeloDataAvailable.SetEvent(this, &ApplicationIo::HandleDataAvailable);
```

---

```
Stream.DirectDataMode(false);
```

The Stream object manages communication between the application and a piece of hardware. Separating the I/O into a separate class clarifies the distinction between an I/O protocol and the implementing hardware.

In Malibu, high rate data flow is controlled by one of a number of streaming classes. In this example we use the events of the VitaPacketStream class to alert us when a packet arrives from the target. When a data packet is delivered by the data streaming system, OnDataAvailable event will be issued to process the incoming data. This event is set to be handled by HandleDataAvailable. After processing, the data will be discarded unless saved in the handler. Similarly, “OnDataRequired” event is handled by HandleDataRequired. In such a handler, packets would be filled with data for output to the baseboard. The Snap application does not generate output, so the HandleDataRequired event is left unhandled.

```
Timer.OnElapsed.SetEvent(this, &ApplicationIo::HandleTimer);  
Timer.OnElapsed.Thunk();
```

In this example, a Malibu SoftwareTimer object has been added to the ApplicationIo class to provide periodic status updates to the user interface. The handler above serves this purpose.

An event is not necessarily called in the same thread as the UI. If it is not, and if you want to call a UI function in the handler you have to have the event synchronized with the UI thread. A call to Synchronize() directs the event to call the event handler in the main UI thread context. This results in a slight performance penalty, but allows us to call UI methods in the event handler freely. The Timer uses a similar synchronization method, Thunk(). Here the event is called in the main thread context, but the issuing thread does not wait for the event to be handled before proceeding. This method is useful for notification events.

Creating a hardware object does not attach it to the hardware. The object has to be explicitly opened. The Open() method of the baseboard activates the board for use. It opens the device driver for the baseboard and allocates internal resources for use. The next step is to call Reset() method which performs a board reset to put the board into a known good state. Note that reset will stop all data streaming through the busmaster interface and it should be called when data taking has been halted.

The size of the busmaster region is changeable by using the BusMasterSize() property before opening the board. Larger values provide more buffering during streaming, at the cost of slower allocation at startup time. Under Windows, up to 32 MB can generally be allocated in each stream direction. Under Linux, only up to 4 MB can generally be allocated in each stream direction.

```
// Insure BM size is a multiple of four MB  
const int Meg = 1024 * 1024;  
const int BmSize = std::max(Settings.BusmasterSize/4, 1) * 4;  
Module().IncomingBusMasterSize(BmSize * Meg);  
Module().OutgoingBusMasterSize(1 * Meg);  
Module().Target(Settings.Target);  
//  
// Open Device  
try  
{  
    Module().Open();  
  
    std::stringstream msg;  
    msg << "Bus master size: " << BmSize << " MB";  
    Log(msg.str());  
}  
  
catch(Innovative::MalibuException & e)  
{  
    UI->Log("Module Device Open Failure:");  
}
```

---

```

        UI->Log(e.what());
        return;
    }

    catch(...)
    {
        UI->Log("Module Device Open Failure: Unknown Exception");
        Opened = false;
        return;
    }

    Module().Reset();
    UI->Log("Module Device opened successfully...");
    Opened = true;

```

This code shows how to open the device for streaming. Each baseboard has a unique code given in a PC. For instance, if there are three boards in a system, they will be targets 0, 1 and 2. The order of the targets is determined by the location in the PCIe bus, so it will remain unchanged from run to run. Moving the board to a different PCIe slot may change the target identification. The `Led()` property can be used to determine the association between a target number and the physical board in a configuration.

```

//
// Connect Stream
Stream.ConnectTo(&(Module.Ref()));
StreamConnected = true;
UI->Status("Stream Connected...");

```

Once the object is attached to actual physical device, the streaming controller associates with a baseboard by the `ConnectTo()` method. The variable `Module` is a class, not a baseboard, to allow quicker porting. The `Module.Ref()` is the admittedly ugly way of getting a pointer to the internal `Module` pointer for connection – `Ref` returns a reference, and the address-of operator (`&`) turns this into a pointer.

Once connected, the object is able to call into the baseboard for board-specific operations during data streaming. If an object supports a stream type, this call will be implemented. Unsupported stream types will not compile.

Lastly we capture and display some information to the screen. This includes the logic version, PCI bus version information, etc.

```

        DisplayLogicVersion();
    }

```

Similarly, the `Close()` method closes the hardware. Inside this method, first we logically detach the streaming subsystem from its associated baseboard using `Disconnect()` method. `Malibu` method `Close()` is then used to detach the module from the hardware and release its resources.

```

//-----
// ApplicationIo::Close() -- Close Hardware & set up callbacks
//-----

void ApplicationIo::Close()
{
    if (!Opened)
        return;

    Stream.Disconnect();
    StreamConnected = false;

    Module().Close();
    Opened = false;
}

```

---

```

        UI->Log("Stream Disconnected...");
    }

```

### Starting Data flow

After downloading interface logic user can setup clocking and triggering options. The stream button then can be used to start streaming and thus data flow.

```

//-----
// ApplicationIo::StartStreaming() -- Initiate data flow
//-----

void ApplicationIo::StartStreaming()
{
    if (!StreamConnected)
    {
        Log("Stream not connected! -- Open the boards");
        return;
    }

    //
    // Set up Parameters for Data Streaming
    // ...First have UI get settings into our settings store
    UI->GetSettings();
}

```

Before we start streaming, all necessary parameters must be checked and loaded into option object. UI-> GetSettings() loads the settings information from the UI controls into the Settings object within the ApplicationIo class.

```

if (Settings.SampleRate*1.e6 > Module().Input().Info().MaxRate())
{
    Log("Sample rate too high.");
    StopStreaming();
    UI->AfterStreamAutoStop();
    return;
}

```

We insure that the sample rate specified by the GUI is within the capabilities of the module.

```

if (Settings.Framed)
{
    // Granularity is firmware limitation
    int framesize = Module().Input().Info().TriggerFrameGranularity();
    if (Settings.FrameSize % framesize)
    {
        std::stringstream msg;
        msg << "Error: Frame count must be a multiple of " << framesize;
        Log(msg.str());
        UI->AfterStreamAutoStop();
        return;
    }
}

```

The module supports both framed and continuous triggering. In framed mode, each trigger event, whether external or software initiated, results in the acquisition of a fixed number of samples. In continuous mode, data flow continues whenever the trigger is active, and pauses while the trigger is inactive. The code above issues a warning if the trigger mode is framed and ill-formed.

---

```

FWordCount = 0;
unsigned int SamplesPerWord = Module().Input().Info().SamplesPerWord();
WordsToLog = Settings.SamplesToLog / SamplesPerWord;

FBlockRate = 0;

```

The class variables above are used to maintain counts of blocks received, reception rate and whether the module is currently triggered. These values are initialized prior to each streaming run.

```

// Disable triggers initially (done by library)
// Set external trigger?
Trig.AtConfigure();
//
// Channel Enables
Module().Input().ChannelDisableAll();
for (unsigned int i = 0; i < Module().Input().Channels(); ++i)
{
    bool active = Settings.ActiveChannels[i] ? true : false;
    if (active==true)
        Module().Input().ChannelEnabled(i, true);
}

```

The number of channels supported varies from board to board. The code uses standard functions like `Module().Input().Channels()` to obtain the max channel count from a board. The `ModuleIo` also has a standard function `AnalogInChannels()` that can be used before the board is opened and other methods are valid. The previous call to `GetSettings` populated the `Settings` object with the number of channels to be enabled on this run. That information is used to enable the required channels via the `Channels` object within the `Module.Input()` object.

The clock source is also programmed using the associated methods within the `Module` object:

```

// Route reference.
IX6ClockIo::IIReferenceSource ref[] = { IX6ClockIo::rsExternal, IX6ClockIo::rsInternal };
Module().Clock().Reference(ref[Settings.ReferenceClockSource]);
Module().Clock().ReferenceFrequency(Settings.ReferenceRate * 1e6);

// Route clock
IX6ClockIo::IIClockSource src[] = { IX6ClockIo::csExternal, IX6ClockIo::csInternal };
Module().Clock().Source(src[Settings.SampleClockSource]);
Module().Clock().Frequency(Settings.SampleRate * 1e6);

// Readback Frequency
double freq_actual = Module().Clock().FrequencyActual();
{
    std::stringstream msg;
    msg << "Actual PLL Frequency: " << freq_actual ;
    Log(msg.str());
}

```

The size of the data packets sent from the module to the Host during streaming is programmable. This is helpful during framed acquisition, since the packet size can be tailored to match a multiple of the frame size, providing application notification on each acquired frame. In other applications, such as when an FFT is embedded within the FPGA, the packet size can be programmed to match the processing block size from the algorithm within the FPGA.

```

//
// Always preconfigure
Stream.Preconfigure();

```

In order to support modes where the clock is not disturbed from run to run, some configuration (notably clock configuration) is performed in a preconfiguration step. Here we call this step automatically to program the clock for the run.

---

```
//
// Velocia Packet Size
Module.SetInputPacketDataSize(Settings.PacketSize);
```

The I/O data can be replaced with FPGA-generated test data by enabling one of the test modes. This is useful when developing custom logic. The code snippet below applies the test mode stored in the Settings.TestGenMode variable to the hardware:

```
//
// Input Test Generator Setup
Module.SetTestConfiguration( Settings.TestCounterEnable, Settings.TestGenMode );
```

I/O samples can be decimated using a feature within the stock firmware. The code snippet below applies the specified decimation factor to the hardware:

```
// Set Decimation Factor
int factor = Settings.DecimationEnable ? Settings.DecimationFactor : 0;
Module().Input().Decimation(factor);
```

The streamed data is logged without change to an intermediate file that can be analyzed with BinView. If this logging is enabled, this starts the logger.

```
if (Settings.IntermediateLogEnable)
{
    IntermediateLogr.Start();
}
```

The code below applies the user-specified trigger and and alert configuration settings to the hardware.

```
// Frame Triggering and other Trigger Config
Module().Input().Trigger().FramedMode(Settings.Framed);
Module().Input().Trigger().Edge(Settings.EdgeTrigger);
Module().Input().Trigger().FrameSize(Settings.FrameSize);

Module.ConfigureAlerts(Settings.AlertEnable);

Trig.AtStreamStart();
```

Samples will not be acquired until the channels are triggered. Triggering may be initiated by a software command or via an external input signal to the Trigger SMA connector. The module supports framed triggering, where a single trigger enables many data samples to be taken before rechecking the trigger. This code enables framed mode, or disables it depending on the settings.

The Stream Start command applies all of the above configuration settings to the module, then enables PCI data flow. The software timer is then started as well.

```
//
// Start Streaming
Stream.Start();
UI->Log("Stream Mode started");
UI->Status("Stream Mode started");

FTicks = 0;
Timer.Enabled(true);
}
```

---

## Handle Data Available

Once streaming is enabled and the module is triggered, data flow will commence. Samples will be accumulated into the onboard FIFO, then they are bus-mastered to the Host PC into page-locked, driver-allocated memory following a two -word header (data packets). Upon receipt of a data packet, Malibu signals the Stream.OnDataAvailable event. By hooking this event, your application can perform processing on each acquired packet. Note, however, that this event is signaled from within a background thread. So, you must not perform non-reentrant OS system calls (such as GUI updates) from within your handler unless you marshal said processing into the foreground thread context. This marshaling behavior can be automatically enabled by calling the event Thunked() or Synchronized() method, as was done in the Open() call earlier.

```
//-----  
// ApplicationIo::HandleDataAvailable() -- Handle received packet  
//-----  
  
void ApplicationIo::HandleDataAvailable(VitaPacketStreamDataEvent & Event)  
{  
    if (Stopped)  
        return;  
  
    VeloBuffer Packet;  
  
    //  
    // Extract the packet from the Incoming Queue...  
    Event.Sender->Recv(Packet);  
  
    FWordCount += Packet.SizeInInts();  
  
    if (Settings.IntermediateLogEnable)  
        if (FWordCount < WordsToLog)  
        {  
            IntermediateLogr.LogWithHeader(Packet);  
        }  
  
    IntegerDG Packet_DG(Packet);  
  
    TallyBlock(Packet_DG.size()*sizeof(int));  
  
    // Per block triggering actions  
    Trig.AtBlockProcess(Packet_DG.size()*sizeof(int));  
}
```

When the event is signaled, the data buffer must be copied from the system bus-master pool into an application buffer. The preceding code copies the packet into the local Buffer called Packet. Since data sent from the hardware can be of arbitrary type (integers, floats, or even a mix, depending on the board and the source), Buffer objects have no assumed data type and have no functions to access the data in them. In the case of X6 series XMC modules, the data contained within the body of the buffer is a list of VITA49 subpackets. These must be parsed and processed individually. This is done as a post-processing step, since in normal cases the Binview application can display the contents of the data file without splitting it up.

As each Velocia packet is received and processed within the HandleDataAvailable method, the TallyBlock method is called. This routine calculates and reports the data flow rate through the user interface.

```
//-----  
// ApplicationIo::TallyBlock() -- Finish processing of received packet  
//-----  
  
void ApplicationIo::TallyBlock(size_t bytes)
```

---

```

{
    double Period = Time.Differential();
    double AvgBytes = BytesPerBlock.Process(static_cast<double>(bytes));
    if (Period)
        FBlockRate = AvgBytes / (Period*1.0e6);

    ...
}

```

In this example, each received packet is logged to a disk file. The packet header and the body are written into the file, which implies that a post-analysis tool (such as BinView) must be used to parse packetized data from the file. Alternately, custom applications may use the Innovative::VitaPacketFileDataSet object to conveniently extract channelized data from a Vita formatted packet data source. Packets are processed until a specified amount of data is logged or the GUI Stop button is pressed.

```

//
// Stop streaming when both Channels have passed their limit
if (Settings.AutoStop && IsDataLoggingCompleted() && !Stopped)
{
    // Stop counter and display it
    double elapsed = RunTimeSW.Stop();

    StopStreaming();
    Log("Stream Mode Stopped automatically");
    Log(std::string("Elapsed (S): ") + FloatToString(elapsed));
}
}

```

## *The Wave Example*

---

The Wave example in the software distribution demonstrates output streaming. It will only be included if the board supports streaming out to a DAC or similar device or devices.

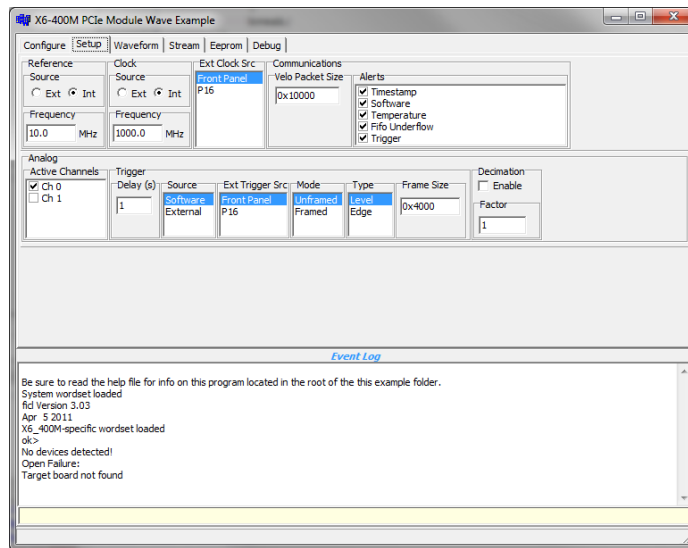
In many ways the Wave example is similar to the Snap example. Differences are highlighted in this section.

### **User Interface**

#### **The Setup Tab**

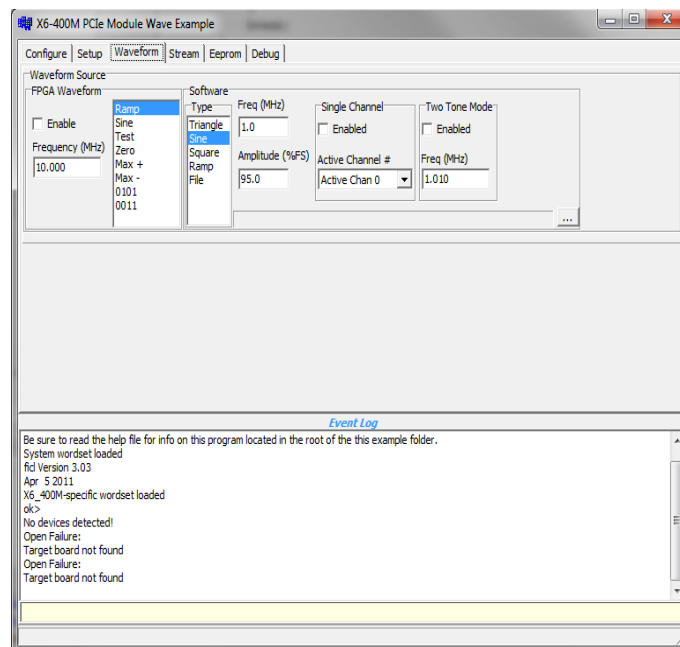
This tab, like that for the Snap application, allows the user to set up the configuration of the board before starting streaming. An additional option on this tab is the Trigger Delay setting. This postpones the software start trigger for a given time to allow the application ample time to stream data into the card before starting output.





## The Waveform Tab

This tab collects the configuration of the output waveform into one area. If the FPGA Waveform is enabled, then it plays the time of waveform selected. If a Sine wave is chosen, then the frequency value is used as the rate of the sine wave. In this mode, no data is streamed by the software.

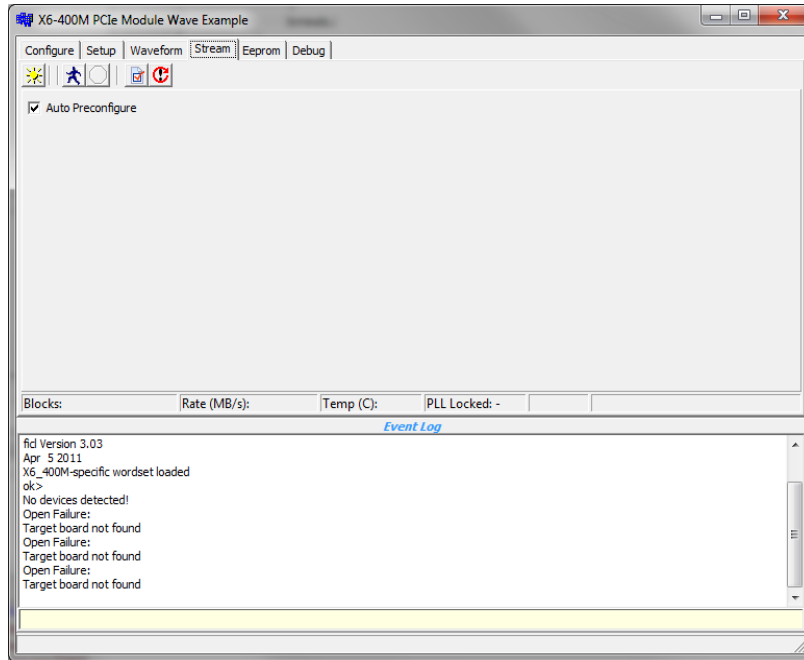


The software waveform generator uses the configuration selected to generate a single buffer's worth of data (indicated by the size of the Velocia packet on the setup tab). All active channels are filled with the same data, unless single channel is enabled. In that case the indicated channel of those active is filled with data, and the remainder are zero filled. Two tone mode fills the buffer with the sum of two waves of the indicated frequencies.

---

## Stream Tab

The Wave example supports the Preconfigure button. This allows the DAC timebase and calibration to occur once before starting streaming, leaving the clock undisturbed during the run. If the Auto Preconfig checkbox is checked, then it acts as if you pressed the Preconfigure button every run.



## ApplicationIo

### Stream Preconfigure

In order to support DAC systems that need calibration to tune the synchronization of their outputs, the channel selection and timebase definition is moved out of the 'normal' location to a special preconfiguration stage. This stage must be manually triggered by the application by calling `Stream.Preconfigure()`. The Wave example allows this by the addition of a preconfiguration button. In addition, the Auto Preconfigure mode will automatically call `Stream.Preconfigure()` whenever the stream is started.

All items that are initialized in the preconfiguration need to have their settings applied to the board object here. In general, the items that need to be set up are the clock/timebase configuration and the DAC configuration, particularly the enables.

```
//-----  
// ApplicationIo::StreamPreconfigure()  
//-----  
  
void ApplicationIo::StreamPreconfigure()  
{  
    Log("Preconfiguring Stream...");  
    //  
    // Make sure if Wave Generator is in single channel mode, we have a valid  
    // active channel filled in  
    if (Builder.Settings.SingleChannelMode)  
    {  
        int active_channels = 0;  

```

---

```

        for (unsigned int i = 0; i < Settings.ActiveChannels.size(); ++i)
            if ((Settings.ActiveChannels[i] ? true : false))
                active_channels++;
        if (active_channels==3)
            active_channels = 4;    // can't do a true 3 channel run, promote to 4.
        if (Builder.Settings.SingleChannelChannel >= active_channels)
        {
            Log("Error: Invalid Active Channel selected in Single Channel Mode");
            return;
        }
    }

    //
    // Set Channel Enables
    Module().Output().ChannelDisableAll();
    for (unsigned int i = 0; i < Module().Output().Channels(); ++i)
    {
        bool active = Settings.ActiveChannels[i] ? true : false;
        if (active==true)
            Module().Output().ChannelEnabled(i, true);
    }

    //
    // Clock Configuration
    // Route ext clock source
    IX6ClockIo::IIClockSelect cks[] = { IX6ClockIo::cslFrontPanel, IX6ClockIo::cslP16 };
    Module().Clock().ExternalClkSelect(cks[Settings.ExtClockSrcSelection]);
    // Route reference.
    IX6ClockIo::IIReferenceSource ref[] = { IX6ClockIo::rsExternal, IX6ClockIo::rsInternal };
    Module().Clock().Reference(ref[Settings.ReferenceClockSource]);
    Module().Clock().ReferenceFrequency(Settings.ReferenceRate * 1e6);
    // Route clock
    IX6ClockIo::IIClockSource src[] = { IX6ClockIo::csExternal, IX6ClockIo::csInternal };
    Module().Clock().Source(src[Settings.SampleClockSource]);
    Module().Clock().Frequency(Settings.SampleRate * 1e6);
    // Readback Frequency
    double freq_actual = Module().Clock().FrequencyActual();
    {
        std::stringstream msg;
        msg << "Actual PLL Frequency: " << freq_actual ;
        Log(msg.str());
    }

    Stream.Preconfigure();
}

```

## Start Streaming

Setup of the stream is much like the Snap example. We have similar error checking code and rate guarding.

```

//-----
// ApplicationIo::StartStreaming()
//-----

void ApplicationIo::StartStreaming()
{
    //
    // Set up Parameters for Data Streaming
    // ...First have UI get settings into our settings store
    UI->GetSettings();

    // if auto-preconfiging, call preconfig here.
    if (Settings.AutoPreconfig)
        StreamPreconfigure();
}

```

---

```

if (!FStreamConnected)
{
    Log("Stream not connected! -- Open the boards");
    return false;
}
if (Settings.SampleRate*1.e6 > Module().Output().Info().MaxRate())
{
    Log("Sample rate too high.");
    return false;
}

```

The first difference is that we configure the Output() sub-object instead of Input(). The X6 Family divides the interface functions for Input and Output devices into separate configuration sub-objects. This allows Input and Output to be independently configured.

```

if (Settings.Framed)
{
    // Granularity is firmware limitation
    int framesize = Module().Output().Info().TriggerFrameGranularity();
    if (Settings.FrameSize % framesize)
    {
        std::stringstream msg;
        msg << "Error: Frame count must be a multiple of " << framesize;
        Log(msg.str());
        UI->AfterStreamStop();
        return false;
    }
}

```

This code checks that the frame size is valid for the particular board being used. The granularity of the hardware frame size control could be different on different cards. But as long as there is a standard means of getting the correct value, the example can work for any card with the same code.

```

// Configure Trigger Manager
Trig.DelayedTriggerPeriod(Settings.TriggerDelayPeriod);
Trig.ExternalTrigger(Settings.ExternalTrigger);
Trig.AtConfigure();

FBlockCount = 0;
FBlockRate = 0;
//
// Check Channel Enables
int ActiveChannels = Module().Output().ActiveChannels();
if (!ActiveChannels)
{
    Log("Error: Must enable at least one channel");
    UI->AfterStreamStop();
    return false;
}

```

The trigger manager is an object defined in the Application portion of Malibu to handle how and when the application manages the software trigger during a run. Separating this kind of boilerplate code makes the ApplicationIo object cleaner, and also makes finding out how the examples use the trigger easier to discover as well.

In this example, if using a software trigger the trigger will be applied after the number of seconds given by the Trigger Delay period. This gives the streaming engine time to deliver substantial data to the card before activating the analog. If using an external trigger, the software trigger will not be applied.

```

//

```

---

```

// Trigger Configuration
// Frame Triggering
Module().Output().Trigger().FramedMode((Settings.Framed)? true : false);
Module().Output().Trigger().Edge((Settings.EdgeTrigger)? true : false);
Module().Output().Trigger().FrameSize(Settings.FrameSize);
// Route External Trigger source
IX6IoDevice::AfeExtSyncOptions syncsel[] = { IX6IoDevice::essFrontPanel, IX6IoDevice::essPl6 };
Module().Output().Trigger().ExternalSyncSource( syncsel[ Settings.ExtTriggerSrcSelection ] );
//
// Velocia Packet Size
Module.SetOutputPacketDataSize(Settings.PacketSize);
//
// Output Test Generator Setup
Module.SetTestConfiguration( Settings.TestGenEnable, Settings.TestGenMode );
Module().Output().TestFrequency( Settings.TestFrequencyMHz * 1e6 );

// Set Decimation Factor
int factor = Settings.DecimationEnable ? Settings.DecimationFactor : 0;
Module().Output().Decimation(factor);

```

This code sets up the trigger logic and routing, the packet size, decimation and the FPGA test generator. This last calls down into the ModuleIo object to handle any board-specific customization.

```

//
// Configure Alert Enables
Module.ConfigureAlerts(Settings.AlertEnable);

```

Alerts and starting the Stream are the same as in Input only mode. We call a board specific function to handle the details.

```

// Disable prefill if in test mode
Stream.PrefillPacketCount(Settings.TestGenEnable ? 0 : PrefillPacketCount);

// Fill Waveform Buffer (if streaming)
if (Settings.TestGenEnable == false)
    FillWaveformBuffer();

```

Like all of our Wave examples, streaming mode is handled by precalculating a wave pattern into a single buffer that is retransmitted each time data is required. The calculation of the buffer itself is a little more involved, but in the end there is a single Velo buffer that is repeatedly sent.

```

Trig.AtStreamStart();

// Start Streaming
Stream.Start();
Log("Stream Mode started");

return true;
}

//-----
// ApplicationIo::StopStreaming()
//-----

void ApplicationIo::StopStreaming()
{
    if (!IsStreaming())
        return;
}

```

---

```

    if (!FStreamConnected)
    {
        Log("Stream not connected! -- Open the boards");
        return;
    }

    //
    // Stop Streaming
    Stream.Stop();

    Timer.Enabled(false);

    // Disable test generator
    if (Settings.TestGenEnable)
        Module.SetTestConfiguration( false, Settings.TestGenMode );

    Trig.AtStreamStop();
}

```

The stream stop process remains simple. The trigger manager, periodic timer, and data stream objects are all stopped. In addition, the test generator is also turned off.

### Data Required Event Handler

When the output stream needs additional data, the Data Required event is signalled. The Wave application uses this call to send the template block to the output via the SendOneBlock() method.

```

//-----
// ApplicationIo::HandleDataRequired()
//-----

void ApplicationIo::HandleDataRequired(PacketStreamDataEvent & Event)
{
    SendOneBlock(Event.Sender);
}

//-----
// ApplicationIo::SendOneBlock()
//-----
const int HeaderTagValuePostPacketizer = 0x00000000;
const int HeaderTagValueOriginal = HeaderTagValuePostPacketizer;

void ApplicationIo::SendOneBlock(PacketStream * PS)
{
    ShortDG Packet_DG(WaveformPacket);

    // Calculate transfer rate in kB/s
    double Period = Time.Differential();
    if (Period)
        FBlockRate = Packet_DG.SizeInBytes() / (Period*1.0e6);

    //
    // No matter what channels are enabled, we have one packet type
    // to send here
    PS->Send(0, WaveformPacket);

    ++FBlockCount;
}

```

For speed, the packet is created on the first call only. After that, the same data wave is sent to all channels. Note that it is allowed to send more than one output packet per notification. If no packets are sent, however, it is possible that further notifications may stop until the application starts sending data again. This decoupling of notification from sending allows

---

different models of data generation to exist in Malibu. An application may send packets asynchronously and not handle notifications at all.

### FillWaveformBuffer()

The Application section of the Malibu library has a waveform generator that will fill a packet with an interleaved set of channels. Sadly, this packet is not in the proper format for sending to the X6 cards, as it needs to be placed into the appropriate number of Vita packets with proper Stream Ids. These packets then need to be inserted into a Velocia buffer to send.

```
//-----  
// ApplicationIo::FillWaveformBuffer() -- Fill buffer with waveform data  
//-----  
  
void ApplicationIo::FillWaveformBuffer()  
{  
    //  
    // Builds a N channel buffer  
    int channels = Module().Output().ActiveChannels();  
    int bits = Module().Output().Info().Bits();  
    int samples = static_cast<int>(Settings.PacketSize);  
  
    // calculate scratch packet size in ints  
    int scratch_pkt_size;  
    if (bits <= 8)  
        scratch_pkt_size = channels * Holding<char>(Settings.PacketSize);  
    else if (bits <= 16)  
        scratch_pkt_size = channels * Holding<short>(Settings.PacketSize);  
    else  
        scratch_pkt_size = channels * Settings.PacketSize;  
  
    Innovative::Buffer ScratchPacket;  
    ScratchPacket.Resize(scratch_pkt_size);  
  
    Builder.SampleRate(Settings.SampleRate*1e6);  
    Builder.Format(channels, bits, samples);  
    Builder.BuildWave(ScratchPacket);  
}
```

The first step is to generate the actual wave data. The approach taken is to have the original wave generator construct a standard wave file in a scratch buffer, which we will then use to construct the correct Waveform packet. So we need to calculate the size of the scratch buffer and load the generator parameters. Then the BuildWave() call constructs the wave in the scratch buffer.

The second step is to copy the data from the scratch buffer into an array of Vita buffers. In this case we can do this as integers, as the data format is the same for our one stream ID. The maximum size of a Vita packet is relatively small, so we have to support paging into a number of them. We choose to make all the packets the same large size until the final one that holds the remainder.

```
//  
// We now have the data in a regular buffer. We need to copy it into  
// VITA buffers, and then those VITAs into the Waveform packet  
std::vector<Innovative::VitaBuffer> VitaQ;  
Innovative::UIntegerDG ScratchDG(ScratchPacket);  
//  
// Bust up the scratch buffer into VITA packets  
unsigned int words_remaining = ScratchPacket.SizeInInts();  
unsigned int offset = 0;  
while (words_remaining)
```

---

```

{
// calculate size of VITA packet
const unsigned int MaxVitaSize = 0xF000;    // Vita limited to 16 bit size

unsigned int VP_size = std::min(MaxVitaSize, words_remaining);

// Get a properly cleared/init'd Vita Header
VitaBuffer VBuf = Stream.NativeVitaBuffer( VP_size );
Innovative::UIntegerDG VitaDG(VBuf);

```

The `NativeVitaBuffer()` function produces a buffer with a properly initialized header and trailer. This is particularly important here since the trailer's padding field, if not correct, will not allow you to completely fill the packet using our Datagrams.

What is padding? Padding is a response to the need to be able to issue a Vita packet at any time. To be sent across the busmaster interface, the packet must have a size that is divisible by 4 (in 32 bit words). Yet we need to be able to issue a Vita packet in cases, such as when a trigger lowers, that can happen at any time. If we hold the data, it will be issued at the wrong time. To send it, we need to pad the packet to an even 128 bit boundary. The padding field is used to tell us which bytes in a packet are invalid pad bytes instead of actual data. The datagrams check this field and reduce the size of the datagram to avoid using this dead space for reading or writing.

```

// Copy Data to Vita
for (unsigned int idx=0; idx<VitaDG.size(); idx++)
    VitaDG[idx] = ScratchDG[offset + idx];

// Init Vita Header
VitaHeaderDatagram VitaH( VBuf );
VitaH.StreamId( 0 );
// fix up loop counters
words_remaining -= VP_size;
offset += VP_size;

// save the buffer
VitaQ.push_back( VBuf );
}

```

Now that we have our array of Vita packets, we can copy them into a large Velo packet for sending. The `VitaPacketPacker` object was made to facilitate this operation. You can set an output packet size, and then as you Pack the packets in, whenever a new packet is filled, the `OnDataAvailable()` event is called for you to process the Velo packet before continuing. The expectation is that you would call `Send()` in this function, but in this case we will tweak the settings to make sure we end up with one buffer containing all the data.

The first step is to set the output size to be considerably larger than the total amount of data we have. Here I use a factor of 2. The real need is to have enough room for the packet data and the 8 words of header/trailer data for each packet. The rough doubling is good enough, since we only will have the one packet.

```

//
// Use a packer to load full VITA packets into a velo packet
// ...Make the packer output size so big, we will not fill it before finishing
VitaPacketPacker VPPk(scratch_pkt_size*2);
VPPk.OnDataAvailable.SetEvent(this, &ApplicationIo::HandlePackedDataAvailable);

// ...Shove in our VITA packets
for (unsigned int i=0; i<VitaQ.size(); i++)
{
    VPPk.Pack( VitaQ[i] );
}

VPPk.Flush();    // outputs the one waveform buffer into Waveform

```



---

At the end of the for loop above, all the data is present in the packer, with a large excess as well. The packer has the ability to force the output of residual data by truncating the packet to exactly fit. This Flush() operation forces a final call to the OnDataAvailable event, which we have made sure is the only one we get.

Then we clean up and init the Waveform headers and we have a valid packet ready for sending when needed.

```
        // WaveformPacket is now correctly filled with data...

        ClearHeader(WaveformPacket);
        InitHeader(WaveformPacket);    // make sure header packet size is valid...
    }
```

This is the event handler for the Packer. In this case, we just do a simple assignment to save the data for later use. You can also add the buffer to a data structure, or Send() it at this time if the application is designed for that kind of I/O.

```
//-----
//  ApplicationIo::HandlePackedDataAvailable() -- Packer Callback
//-----

void  ApplicationIo::HandlePackedDataAvailable(Innovative::VitaPacketPackerDataAvailable & event)
{
    WaveformPacket = event.Data;
}
```

---

## Chapter 13. *Malibu Buffer Classes*

---

The Buffer Classes in Malibu have been designed for efficiency in the context of real-time data processing and flow requirements. This document provides an overview of the design decisions made in their architecture.

### *Buffer Design Decisions*

---

Previous versions of Malibu utilized specialized buffers for each type of data contained. The resulting class family design suffered from a number of issues: The different classes were defined primarily by the type of the data contained in the buffer, namely is it floating point data, integers, and so on. This was problematic for buffers that have no clearly defined type, such as message packets, or if the data type is unknown.

The need to port Malibu to other platforms drove these problems to the surface. The old buffer classes relied on the Intel IPP library, which is not available on all platforms. Since the IPP requirement had to be removed, this opened the door to reworking the class family entirely.

#### **Design Decision #1 – A “Typeless” Buffer class**

In the present design, a buffer is a very simple thing. It is a package of data of undetermined type and format. Its primary purpose is to act as a container to simplify movement of data between application and target hardware. From this requirement, the natural element-size within a buffer is 32-bit integers because that is the smallest size of data that can be bus-mastered on Innovative hardware. All buffers have a header block and a data block. The header information may be ignored if the data streaming method used does not understand or make use of headers.

This reduces the number of buffer classes that our streaming classes needed to deal with down to one. Packet streams use the header. The others ignore it. This class is called `Innovative::Buffer`.

Since a buffer is agnostic as to the size of the elements it holds, the only size method is called `SizeInInts()`.

#### **Design Decision #2 – Data Access Datagrams**

Though Malibu uses typeless buffers, it is still important to be able to access the contents of the buffer simply and easily. To accommodate this need, access to the data in buffers is performed by a wrapper class that is linked to the buffer just as access is needed. In most applications, there are two kinds of buffers in general use:

- “Command” messages, in which the data is a set heterogeneous argument values
- “Data” packets where the all the data is likely to be of the same, if undetermined, type. For example one buffer might be all 16 bit `short` data. Another might be floating point data.

---

The former type of message is supported by the `IDatagram` interface and the `MessageDatagram` class which derives from it. The latter type is supported by the `AccessDatagram` class.

Since the `AccessDatagram` needs to support many different data types, it is implemented as a template class - `AccessDatagram<T>`. It provides typed, random-access iterators, STL-like `begin()` and `end()` methods and array operators. Each instance of a datagram provides a `size()` method that returns the size of the buffer in units of the data type accessed. The template assures that any new operations will be available to all data types without cutting and pasting code. This datagram has no dependencies on the IPP library.

An additional benefit of this design is that the template works on any data type as well as any structure that is defined by the user. If the buffer contains an array of records, parsing the data is then very simple without adding any code to the library.

### Design Decision #3 – Predefined Access Datagram Classes

While an access datagram can be simply built up for any data type, there are some data types that are commonly in use. For simplicity's sake, numerous datagrams have been pre-defined in wrapper classes for these common types in `BufferDatagrams_Mb.cpp`. Classes are provided for these data types:

**Table 5. Basic Buffer Datagram Classes**

Class Name	Data Type
<code>IntegerDG</code>	<b>int</b>
<code>UIntegerDG</code>	<b>unsigned int</b>
<code>FloatDG</code>	<b>float</b>
<code>ComplexDG</code>	<b>Complex</b>
<code>ShortDG</code>	<b>short</b>
<code>CharDG</code>	<b>char</b>

### Design Decision #4 – IPP Datagram Classes

Malibu uses the Intel Performance Primitives under operating systems that support it to accelerate signal processing and vector operations. As a consequence, Malibu implements IPP-enabled datagrams which wrap buffer objects and allow high-speed manipulations of their contents. These datagram classes reside in the `Analysis_Mb` library and also derive from the `AccessDatagram` template. In addition to IPP-specific functionality, all basic access methods are supported as well.

**Table 6. IPP Function Datagrams**

Class Name	Data Type	File
<code>IppCharDG</code>	<b>char</b>	<code>IppCharDG_Mb.h</code>

---

---

Class Name	Data Type	File
IppComplexDG	<b>Complex</b>	IppComplexDG_Mb.h
IppFloatDG	<b>float</b>	IppFloatDG_Mb.h
IppIntegerDG	<b>int</b>	IppIntegerDG_Mb.h
IppShortDG	<b>short</b>	IppShortDG_Mb.h

## *Buffer Internals*

---

Buffers are designed to be the primary transport vehicle for data moving from an application to Innovative hardware boards. This means that there is an advantage in being able to create and move buffers about in our system without copying data when it is not required, as copying large amounts of data will degrade performance.

Since data transfers to the target are done at least in units of 32 bit words, the internal buffer size and pointers are integer pointers. Even if the data type is shorter, such as a short or byte, the size still must be an integral number of 32-bit words. `PacketStream` buffers have an additional requirement that the header and body be an integral number of 64-bit words, meaning that the size of each in 32-bit words must be an even number.

In addition, the IPP library has some alignment restrictions on where the data buffers must begin for optimal performance. To insure that buffers are compatible with this library, Malibu insures suitable buffer alignment.

The buffer class minimizes the cost of copying data by using a handle-body approach. When a buffer is copied, two 'handle' class instances are created, each pointing to the same header and data body information. This is a faster operation than bulk copying the large amount of data, especially if the data is only rarely-changed. There are in fact two handles present, one to the header data and one to the packet data. Both handles manage properly aligned data blocks for use with the IPP library.

If the data body is changed, however, all handles will be affected. This breaks the simplistic logical model. Therefore Malibu implements a 'copy on write' scheme in which any write to a data region will force the body to be separated from all other handles and copied. This can be a relatively expensive process. Data access datagrams will properly force this to happen when used. Using raw pointers to buffer data regions will not, and should therefore be avoided.

A final optimization is that the buffer classes use a shared pool to cache blocks to reduce the time to allocate and free buffer data blocks. If a buffer of the correct size has been previously freed it will be reused from the cache rather than reallocated. Provisions are made to pre-allocate buffers of a specified size in order to mitigate allocation time prior to real-time activities.

## *Data Buffers : The Innovative::Buffer Class*

---

The Buffer class is the class used for all bulk data transfers to Innovative boards. All stream classes exchange Buffer objects.

---

### Buffer Class (Buffer\_Mb.h)

The Buffer class contains a header block and a data block. The header block is only used and transmitted/received on Packet Stream boards. Other streams ignore the header, although it is always present and sized to hold at least 2 words.

Like the previous buffers, Buffer uses managed aligned blocks, and is reference counted for fast copying as long as the data is unchanged.

Unlike before, the Buffer class assumes no type to its contents. The size of the buffer's contents are returned in units of 32-bit ints. The pointers to the base of the data and header region are available, but access to the contents are best done with datagrams.

The use of non-const methods such as `Data()` and `Resize()` will force a copy of the contents if the Buffer contents are shared with another buffer. This may invalidate datagrams associated with this buffer.

### Holding Template (Buffer\_Mb.h)

Because the Buffer class is logically typeless, sizing presents a small problem. With STL containers, such as vectors, one can create a buffer sized to a specified number of elements. For example:

```
std::vector<int>( 1000 );
```

would make a buffer that is 1000, 32-bit words long. But the Buffer class has no notion of the size of the elements that it contains. For this reason, Malibu includes the `Holding` template. This template performs the conversion of a size in elements of a type to a size in integers needed by the Buffer constructor. So in the case above where we need to hold 1000 short integers:

```
Innovative::Buffer KiloBuf( Holding<int>(1000) );
```

This sizes the Buffer to be large enough to hold the 1000 integer elements that will be accessed later using a datagram class.

### MessageDatagram (Buffer\_Mb.h)

A specialty access datagram class interface has been created to simplify filling packet stream buffers with command parameters similar to those used in the message packets used on Matador cards and C64x streaming. This interface, called `IDatagram`, allows access to the data as a heterogeneous collection of data – for example one argument can be an integer and the next a float.

Previous versions of Malibu employed `PmcBuffers` which had an implementation of this interface to support sending packets containing commands to boards with DSPs such as the M6713 and P25 called `PmcBufferDatagram`. Within the current Malibu, this has been renamed `MessageDatagram` to more closely follow the use of the object.

---

## *Buffer Data Access*

The data access requirements seem to require contradictory features: Support for many types of data quickly and easily is required, but a minimal code base is desired. Templates solve this problem very cleanly. A template class can be instantiated for many data types from a single code base. If a feature is added to the template, it is added to them all.

In fact, the template allows the user to apply his own structure to a buffer as easily as any that we provide.

---

The data access template provides a view of a buffer as an array of same-typed data. So an integer datagram accesses the buffer as an array of integers.

## Access Template Features

### Template `AccessDatagram<T>` (`AccessDatagrams_Mb.h`)

The access datagram uses an interface as its view of the buffer to on which to operate. This decouples the template from the `Buffer` class itself and makes the template more general. The buffer class implements the interface by deriving from `IDatagrammable`, so all buffers can be accessed by the template easily:

```
Buffer A(128);
AccessDatagram<unsigned int> A_dg(A);    // accesses buffer A

for (int i=0; i<A_dg.size(); i++)
    A_dg[i] = i;
```

The for-loop in the above code fills the buffer with a ramp. The `size()` method returns the size of the data in elements. The datagram array operator accesses the data in the buffer as an array of `unsigned int`. This version is not range checked. The `at()` method performs the same access with range checking.

There are some additional methods for returning sizes. The `size()` method returns the size in elements.

`SizeInElements()` is an alias for that method. `SizeInInts()` returns the size in integers, and `SizeInBytes()` returns the size in bytes. `ElementSizeBytes()` returns the size of the access element in bytes.

The access datagram supports resizing the associated buffer.

An access datagram can be constructed from any structure. For example:

```
struct FourSamples
{
    unsigned short sample[4];
}

Buffer B(100);
AccessDatagram<FourSamples> B_4Sample_dg(B);    // accesses buffer B

for (int i=0; i<B_4Sample_dg.size(); i++)    // size will return 50 here
{
    B_4Sample_dg[i].sample[0] = i;
    B_4Sample_dg[i].sample[1] = i + 100;
    B_4Sample_dg[i].sample[2] = i + 200;
    B_4Sample_dg[i].sample[3] = i + 300;
}
```

Since the size of the element is 2, 32 bit words, the buffer only fits 50 elements in the 100 words.

`AccessDatagram` supports an STL iterator over the data. This iterator is a random access iterator. Forward and reverse iteration is supported using the standard `begin()`, `end()`, `rbegin()`, and `rend()` methods. Constant versions of iterators allow read-only access.

```
Buffer C( Holding<float>(20) );
AccessDatagram<float> C_dg(C);    // accesses buffer C
```

---

```

// write
for (AccessDatagram<float>::iterator iter = C_dg.begin(); iter != C_dg.end(); ++iter)
    *iter = i;

// read - outputs 0.0, 1.0, 2.0...
for (AccessDatagram<float>::const_iterator iter = C_dg.begin(); iter != C_dg.end(); ++iter)
    Output(*iter);

// read backward - outputs 19.0, 18.0, 17.0...
for (AccessDatagram<float>::reverse_iterator iter = C_dg.rbegin(); iter != C_dg.rend(); ++iter)
    Output(*iter);

```

The availability of these iterators also allows STL algorithm templates to be used on buffers via datagrams. The following code fills a buffer with 0 using the `std::fill` algorithm.

```

Buffer D( Holding<unsigned int>(20) );
AccessDatagram<unsigned int> D_dg(D);
std::fill(D_dg.begin(), D_dg.end(), 0);

```

**Note:** A datagram object can be made invalid by certain operations on the buffer. Since the datagram cache the information about the data for speed, if the buffer changes the iterator will no longer point to its assumed buffer, and may point nowhere. Similarly, any iterators created from a datagram can be invalidated by these operations.

```

Buffer E( Holding<unsigned int>(20) );
Buffer F;

F = E; // F shares E's buffer

AccessDatagram<unsigned int> F_dg(F);

F.MakeUnique(); // F_dg now invalid!

```

In the above code sample, two buffers share the same data block after the assignment. When F is split away via the `MakeUnique()` method, `F_dg` is no longer pointing to F's buffer. (In this case it is probably pointing to E's buffer). Similar issues can occur with multiple datagrams:

```

Buffer E( Holding<unsigned int>(20) );

AccessDatagram<unsigned int> E_dg(E);
AccessDatagram<unsigned short> E_short_dg(E);

E_short_dg.Resize( 500 ); // E_dg now invalid!

```

In the above code, when the second datagram changes the internal buffer by resizing it, the `E_short_dg` datagram is updated to match the new block, but `E_dg` is not and is invalidated. To mitigate these problems, datagrams should be constructed as close to the point of use as possible. Also, a datagram can be revalidated with the `renew` call:

```

E_dg.Renew(); // E_dg now valid again.

```

`Renew()` does not re-validate any iterators created by the datagram that also were invalidated. These remain invalid.

### Template Class `DatagramIterator` (`AccessDatagrams_Mb.h`)

This template provides the iterator objects for the access datagram. It is a standard random-access iterator supporting forwards and backwards iteration.

```

// Iterator Test
Log("Iterator Test!");

```

---

---

```
Buffer A(100);

AccessDatagram<int> A_dg(A);

AccessDatagram<int>::iterator Iter1 = A_dg.begin();
AccessDatagram<int>::iterator Iter2 = A_dg.begin();
```

Iterators can be compared with each other.

```
Log("Compare equal Iterators");
{
    std::stringstream msg;
    msg << " ==: " << (Iter1==Iter2) << " !=: " << (Iter1!=Iter2) <<
        " <: " << (Iter1<Iter2) << " <=: " << (Iter1<=Iter2) <<
        " >: " << (Iter1>Iter2) << " >=: " << (Iter1>=Iter2) ;
    Log(msg.str());
}
```

Subtracting iterators gives the 'distance' between them in elements.

```
Log("Iterator Difference");
++Iter1; ++Iter1; ++Iter1;
int delta = Iter1 - Iter2;    // delta is 3
{
    std::stringstream msg;
    msg << "Pointer Difference " << delta ;
    Log(msg.str());
}
```

Iterators can be assigned, pointing them to the same location. They can be offset like pointers

```
Log("Iterator Assign");
AccessDatagram<int>::iterator Iter3 = A_dg.begin() + 10;
int delta2 = Iter3 - Iter1;    // delta2 is 7
Iter3 = Iter2;
int delta3 = Iter3 - Iter1;    // delta3 is -3
{
    std::stringstream msg;
    msg << "Delta2 " << delta2 << " Delta3 " << delta3; ;
    Log(msg.str());
}

Log("Compare Unequal Iterators (A>B)");
{
    std::stringstream msg;
    msg << " ==: " << (Iter1==Iter2) << " !=: " << (Iter1!=Iter2) <<
        " <: " << (Iter1<Iter2) << " <=: " << (Iter1<=Iter2) <<
        " >: " << (Iter1>Iter2) << " >=: " << (Iter1>=Iter2) ;
    Log(msg.str());
}
```

Iterators can use the bracket notation just like a pointer or array can. It adjusts the location without moving the iterator.

```
for (int i=0; i<100; i++)
    Iter2[i] = i;
```

Datagram iterators can be bound to any class that supports the `IIteratable` interface. This allows the code to be reused if new datagrams are developed.



---

### Interface Class `IDatagrammable` (`AccessDatagrams_Mb.h`)

This interface allows the access datagram to bind to a buffer class. The buffer class derives from `IDatagrammable` allowing access to the data portion of the buffer. Users can implement this interface to allow the access template to work on another class. There are several examples of this in the library, one being `AlignedBlockDatagram` which builds an interface for the `AlignedBlock` class.

```
//=====
// CLASS IDatagrammable -- Interface required to support datagrams
//=====

class IDatagrammable
{
public:
    virtual ~IDatagrammable() {}

    virtual unsigned int    DatagramSize() = 0;
    virtual int *           DatagramBasePtr() = 0;
    virtual bool            MakeWritable() = 0;           // returns 'true' if buffer renewed
    virtual void            Resize(unsigned int size_in_ints) = 0;
};
```

### Interface Class `IIteratable` (`AccessDatagrams_Mb.h`)

This interface allows the Datagram Iterator template to bind to a `Datagram` class. Any class supporting `IIteratable` can be iterated-through with a `DatagramIterator`.

```
//=====
// CLASS IIteratable -- Interface required to support iteration over data
//=====

class IIteratable
{
public:
    virtual char *    Base() = 0;
    virtual size_t    SizeInBytes() = 0;
};
```

### Standard Implementation Classes

The Malibu library provides some standard implementations of access datagrams. These provide shorter names than the full template syntax, and also allow a reference to the original buffer to be returned by the `WrappedBuffer()` method.

#### `IntegerDG` (`BufferDatagrams_Mb.h`)

Provides access as integers.

#### `UIntegerDG` (`BufferDatagrams_Mb.h`)

Provides access as unsigned integers.

---

### **FloatDG (BufferDatagrams\_Mb.h)**

Provides access as floating point data.

### **ShortDG (BufferDatagrams\_Mb.h)**

Provides access as short integers.

### **ComplexDG (BufferDatagrams\_Mb.h)**

Provides access as complex numbers.

### **CharDG (BufferDatagrams\_Mb.h)**

Provides access as characters.

## **IPP Implementation Classes**

The original buffer classes provided methods to use IPP functions to manipulate the buffer. These datagrams preserve the functions so that code using them can be easily ported to the new library. One difference from the old classes is that datagram methods that create a new buffer return a `Buffer`, not a datagram. This buffer then must be wrapped in a datagram to be manipulated further.

```
// Assume A, B, C are buffers containing float data
IppFloatDG A_dg(A);
IppFloatDG B_dg(B);
IppFloatDG C_dg(C);

Buffer D = A + B + C;           // will not compile! Can't Add buffers

Buffer D = A_dg + B_dg + C_dg; // will not compile! Can't use the temporary

// correct way to sum A, B, and C
Buffer AB = A_dg + B_dg;
IppFloatDG AB_dg(AB);
Buffer D = AB_dg + C_dg;
```

All the calculations supported before can be performed in the new system, but it may well be more verbose due to the requirement to explicitly assign temporary sums to a buffer handle and wrapping them in a datagram before continuing.

### **IppCharDG (IppCharDG\_Mb.h)**

Character IPP vector functions.

### **IppComplexDG (IppComplexDG\_Mb.h)**

Complex IPP vector functions.

### **IppFloatDG (IppFloatDG\_Mb.h)**

Float IPP vector functions.

---

### **IppIntegerDG (IppIntegerDG\_Mb.h)**

Integer IPP vector functions

### **IppShortDG (IppShortDG\_Mb.h)**

Short IPP vector functions.

## **Special Purpose Datagrams**

### **PacketBufferHeader (BufferHeader\_Mb.h)**

This datagram is crafted to access the header of a buffer rather than the data. In addition, it defines some additional methods to access the header information fields used by Packet Stream buffers.

**Table 7. PacketBufferHeader Field Methods**

Method Name	Description
PeripheralId()	Access PID field of header (Packet Type)
PacketSize()	Size of entire packet in words (data and header)
DataSize()	Size of data region in words

### **IDatagram Template (Datagram\_Mb.h)**

This datagram is designed to allow access to a buffer as a heterogeneous collection of arguments, like a command packet. This is especially useful for Packet Stream boards with co-processors, like the P25 and M6713 as it is common for the host and target software to have to communicate via a command protocol. This datagram provides a similar interface than that used by Matador message packet and the TI Bus-master stream mail packets. This aids in porting code between these platforms.

### **MessageDatagram (Buffer\_Mb.h)**

This datagram implements the IDatagram interface on a Buffer.

### **Internal Datagrams (various CPPs)**

The original buffer classes had functions that were used internally by stream classes to move data to and from data packets to the hardware. These have been separated out into separate datagram classes, removing them from the interface of the user classes.

---

## *Guidelines for Converting to new Buffers*

---

### **Translate all buffers to be `Innovative::Buffer`**

Previous code may have used `PmcBuffers`, `IntegerBuffers`, `MemoryBuffers`, or something else. All these references must be replaced by `Buffers`.

The original buffer type may give you a guideline on which type of data access datagram you will need, if any.

### **Convert array operators on buffers**

The old buffer classes used to provide a native array operator method. These no longer exist. You have to create a access datagram to do the same thing.

```
// Original Code:

static PmcBuffer Packet;
//
// Extract the packet from the Incoming Queue...
Event.Sender->Recv(Packet);

for (int i=0; i<Packet.Size(); i++)
    Log(Packet[i]);

// Ported Code

static Buffer Packet;
//
// Extract the packet from the Incoming Queue...
Event.Sender->Recv(Packet);

IntegerDG Packet_DG(Packet);
for (int i=0; i<Packet_DG.size(); i++)
    Log(Packet_DG[i]);
```

An `IntegerDG` datagram is used since the `PmcBuffer`'s native size was integer. A `FloatBuffer` would have used a `FloatDG` to produce the same effect.

Note that the datagram was created as late as practical, to avoid any issues with invalidation by the `Recv()` method of the stream.

### **Size Issues**

The original buffers used `Size()` to give the size in elements. In order to force a reexamination of various size calls, the new buffers and datagrams do not have a `Size()` method.

The `Buffer` class has a `SizeInInts()` method to return the raw size of the data region. `HeaderSizeInInts()` returns the size of the header. `FullSizeInInts()` returns the sum of these.

Datagram wrappers provide a `size()` method for compatibility with STL that is size in elements. There is also `SizeInElements()` that is more explicit. For calculating rates, `SizeInBytes()` or `SizeInInts()` can give the values for any type directly.

---

### Datagrams and Iterators are Disposable

As discussed above, there can be cases where datagrams can be made invalid by changes to their underlying buffer. The best way to reduce this risk is to consider a datagram as a very volatile entity. Create a datagram at the last minute before use. Limit its lifetime as much as possible.

Creating a new datagram is inexpensive, so this technique will not cost much in computation time.

### Packet Stream Header Access

Normal datagrams only access the data portion of a buffer. To access the header region, use a `PacketBufferHeader` datagram.

```
// ...Old way
//
void ApplicationIo::HandleDataAvailable(PacketStreamDataEvent & Event)
{
    static PmcBuffer Packet;
    //
    // ...Get the packet from the system
    Event.Sender->Recv(Packet);
    //
    // ...Process the packet
    short PacketType = Packet.Header()->PeripheralId();
    switch (PacketType)
    {
        case ccLogin:
            UI->Log("Dsp logged in: " + IntToString(++LoginTally));
            UI->OnLoginCommand();
            break;
        // ...continues
    }
}

// ...New way
//
void ApplicationIo::HandleDataAvailable(PacketStreamDataEvent & Event)
{
    static Buffer Packet;
    //
    // ...Get the packet from the system
    Event.Sender->Recv(Packet);
    //
    // ...Process the packet
    PacketBufferHeader PktHeader(Packet);

    short PacketType = PktHeader.PeripheralId();
    switch (PacketType)
    {
        case ccLogin:
            UI->Log("Dsp logged in: " + IntToString(++LoginTally));
            UI->OnLoginCommand();
            break;
        // ...continues
    }
}
```

The `PacketBufferHeader` has methods to set or get the fields of a buffer to be used in packet streaming applications.

### Porting Buffer Access Modes #1 – The Aztec Model

Here we look at how buffers are accessed in applications and how the new style can improve matters. In this mode of access the programmer pulls all the interesting parts out of the object and works with them. The main reason for this was that we wanted to use a different data type. This is far better done with an access datagram.

---

```

//-----
//  BlockChecker::Ramp16BitCheck() -- verify loopback
//-----

bool  BlockChecker::Ramp16BitCheck(PmcBuffer & pda, unsigned int block_idx)
{
    //
    //  we have short data so recast the pointer
    short * Data = pda.ShortPtr();
    const int ShortScaleFactor = sizeof(int)/sizeof(short);

    bool is_error;
    if (Data[0] != (short)NextBlockStartVal)
        is_error = true;
    else
        is_error = false;

    NextBlockStartVal = Data[0] + pda.Size()*ShortScaleFactor;
    if (is_error)
        return true;
    // ...in-block continuity
    for (unsigned int i=1; i<pda.Size()*ShortScaleFactor; i++)
        if (Data[i] != Data[i-1]+1)
            return true;

    return false;
}

```

Since we wish to access as short integers, we will create a short datagram. This choice makes all the code to convert sizes unnecessary, as the short datagram knows how many elements it has in it.

```

//-----
//  BlockChecker::Ramp16BitCheck() -- verify loopback
//-----

bool  BlockChecker::Ramp16BitCheck(Buffer & pda, unsigned int block_idx)
{
    ShortDG pda_dg(pda);

    bool is_error;
    if (pda_dg[0] != (short)NextBlockStartVal)
        is_error = true;
    else
        is_error = false;

    NextBlockStartVal = pda_dg[0] + pda_dg.size();
    if (is_error)
        return true;
    // ...in-block continuity
    for (unsigned int i=1; i<pda_dg.size(); i++)
        if (pda_dg[i] != pda_dg[i-1]+1)
            return true;

    return false;
}

```

### Porting Buffer Access Modes #2 – Buffer [] operator

This code is a bit nicer, in that we used the array access methods of the buffer to access the contents. Of course, these methods no longer exist.

---

```

//-----
//  BlockChecker::LoopbackCheck() -- verify loopback
//-----

bool  BlockChecker::LoopbackCheck(PmcBuffer & pda, unsigned int block_idx)
{
    //
    //  check for gaps in between packets
    bool is_error;
    if ((unsigned int)pda[0] != NextBlockStartVal)
        is_error = true;
    else
        is_error = false;

    NextBlockStartVal = pda[0] + pda.Size();
    if (is_error)
        return true;
    // ...in-block continuity
    for (unsigned int i=1; i<pda.Size(); i++)
        if (pda[i] != pda[i-1]+1)
            return true;

    return false;
}

```

The IntegerDG Access datagram is a fast substitute:

```

//-----
//  BlockChecker::LoopbackCheck() -- verify loopback
//-----

bool  BlockChecker::LoopbackCheck(Buffer & pda, unsigned int block_idx)
{
    IntegerDG pda_dg(pda);
    //
    //  check for gaps in between packets
    bool is_error;
    if ((unsigned int)pda_dg[0] != NextBlockStartVal)
        is_error = true;
    else
        is_error = false;

    NextBlockStartVal = pda_dg[0] + pda_dg.size();
    if (is_error)
        return true;
    // ...in-block continuity
    for (unsigned int i=1; i<pda_dg.Size(); i++)
        if (pda_dg[i] != pda_dg[i-1]+1)
            return true;

    return false;
}

```

As an alternative, you can use iterators. Note the use of operator `[]` in the loop to look backwards at the previous sample from the current location. As an alternative, `*(iter-1)` could have been used. In use, iterators act much as a pointer would and code written for pointers converts naturally to using an iterator in place of the pointer.

```

//-----
//  BlockChecker::LoopbackCheck() -- verify loopback
//-----

bool  BlockChecker::LoopbackCheck(Buffer & pda, unsigned int block_idx)
{

```

---

```

IntegerDG pda_dg(pda);
IntegerDG::iterator iter = pda_dg.begin();
//
// check for gaps in between packets
bool is_error;
if ((unsigned int)(*iter) != NextBlockStartVal)
    is_error = true;
else
    is_error = false;

NextBlockStartVal = (*iter) + pda_dg.size();
if (is_error)
    return true;
// ...in-block continuity
iter++;
for ( ; iter < pda_dg.end() ; iter++)
    if ((*iter) != iter[-1]+1)
        return true;

return false;
}

```

### Porting Buffer Access Modes #3 -- Applying a Structure to Buffer Content

In some cases example programs use this technique to overlay a structure on a message packet:

```

// Display capture stats
ResultsInfo Info;
memcpy(&Info, Packet.IntPtr(), ResultsInfoSize);

UI->Log( "PreOverflow flag: " + IntToString(Info.PreOverflow));
UI->Log( "PostOverflow flag: " + IntToString(Info.PostOverflow));

```

Interface datagrams allow this code to do the same thing by laying the structure over the data using the access datagram. Then, the data can be copied using structure assignment rather than `memcpy()`:

```

// Display capture stats
ResultsInfo Info;
AccessDatagram<ResultsInfo> PacketDG(Packet); // apply structure
Info = PacketDG[0]; // copy out

UI->Log( "PreOverflow flag: " + IntToString(Info.PreOverflow));
UI->Log( "PostOverflow flag: " + IntToString(Info.PostOverflow));

```

In fact, in this case there is no need for the copy at all, as the data could be read from the datagram directly:

```

// Display capture stats
AccessDatagram<ResultsInfo> PacketDG(Packet); // apply structure

UI->Log( "PreOverflow flag: " + IntToString(PacketDG[0].PreOverflow));
UI->Log( "PostOverflow flag: " + IntToString(PacketDG[0].PostOverflow));

```



---

## Chapter 14. *Using the X6 Family Baseboards in Malibu*

### *Overview*

---

The X6 family of baseboards a number of changes from the X5 and X3 family baseboards. These differences will impact the application using an X6.

The most profound change is that the X6 introduces a new style of data streaming. The raw data is enclosed in VITA standard formatted packets, that are enclosed in a packet similar, but not identical to the X5 and X3 buffers. This means that there is no longer a single kind of Buffer, but now three different flavors of buffers may be in use.

### *Buffers and their Type*

---

The concept of the Innovative::Buffer class is that of a container of generic data. The type and format of the data contained in a buffer is defined by the datagram object used to view it. What had been consistent, the presence of a header of a fixed size, is required to be different for these new types of data. A class is needed to manage these differences.

Buffer class	Header Size	Trailer Size	Description
Buffer	2 words	n/a	Original buffer, plus the common base for all buffers
PmcBuffer	2 words	n/a	X3/X5 buffer formats
VeloBuffer	4 words	n/a	X6 stream buffer wrapper
VitaBuffer	7 words	1 word	X6 stream buffer data packet

The Vita buffers require a trailer in addition to the header and data sections. To accommodate this, all buffers now include a Trailer section in addition to the Header and Data. Since older code will not access this section, the presence of the trailer will not harm legacy applications.

Most software processing buffers only are interested in the data portion of the buffer. This code will be able to work identically on all kinds of buffers without problems. For example, a buffer filling function coded like this:

```
void  MyBufferStuffer( Buffer & buf_to_fill );
```

MyBufferStuffer can handle any buffer type without any problems

```
VeloBuffer  Vbuf;  
PmcBuffer  Pbuf;  
Buffer      Buf;  
  
MyBufferStuffer(Vbuf);  
MyBufferStuffer(Pbuf);
```

---

```
MyBufferStuffer(Buf);
```

Note that Pbuf and Buf actually have exactly the same format, although they are not identical types. But a function can be written to only allow particular types to be passed in:

```
void VeloBuffersOnly( VeloBuffer & buf );

MyBufferStuffer(Vbuf);    // works
MyBufferStuffer(Pbuf);    // does not compile - PmcBuffers not allowed.
```

This technique is used on the X6 Streaming object to only allow Vita and Velo packets to be streamed.

## Buffer Conversions

At times, these types can introduce some difficulties. For example, lets say you already have a generator function that creates buffer objects full of data for output. So this function returns a buffer object, and has a signature like this:

```
Buffer BufGen();

Buffer Dbuf = BufGen();
```

But if you use this function with a VitaBuffer:

```
VitaBuffer Vbuf = BufGen();
```

You have a problem. The assignment copies all the parts of the buffer – header, trailer, and body. But the header and trailer of the Buffer are incorrectly sized. Code (such as streaming) that relies on properly formatted packets will fail.

To help with this situation, a template function ConvertData has been defined to create a correct buffer of one type from another, copying only the data.

```
VitaBuffer Vbuf;
Buffer Dbuf = BufGen();
Vbuf = ConvertData<VitaBuffer>(Dbuf);    // Vbuf shares Dbuf's data only.
```

The header and trailer of Vbuf will be overwritten with those in the temporary VitaBuffer returned. If you want to preserve your VitaBuffer's header and trailer for some reason do this:

```
VitaBuffer Vbuf;
... set up Vbuf header/trailer
Buffer Dbuf = BufGen();
VitaBuffer scratch = ConvertData<VitaBuffer>(Dbuf);
Vbuf.SwapData(scratch);    // header and trailer unaffected
```

## Applying a Type

At times you can have the opposite situation – you have a proper buffer, header and all, but it has the wrong type. This happens in the library when we share code with X3/X5 streaming systems, and it can happen in this situation in an application: assume you have a piece of code that you want to only accept PmcBuffers, and a generator that creates Buffer objects from legacy code:

```
void PmcBuffersOnly( PmcBuffer & buf );

Buffer Dbuf = BufGen();
```

---

So even though the buffer Dbuf is perfectly formatted to be passed into PmcBuffersOnly(), its incorrect type makes it impossible. For this, we have another template function Convert<T>(), which shares all sections with the original buffer and changes its type:

```
Buffer Dbuf = BufGen();
PmcBuffer Pbuf = Convert<PmcBuffer>(Dbuf);
PmcBuffersOnly(Pbuf);    // works
```

These conversions are all fast, since they only involve moving pointers around. No data is copied by any of these conversions.

## Buffer Sizing Template Functions

These functions already existed before, but this is a good place to mention it. Since Buffers are generic, the size arguments in the constructors and in Resize() use the size in integers as an argument. Writing unit conversion code from scratch is error prone, so these template functions exist:

### Holding<T>() :: Sizing a buffer to hold N elements

This template function is intended for use in a constructor or Resize() call to make sure the resulting buffer can hold N elements. For example, to make a buffer hold 1000 doubles, use”

```
Buffer DoubleStore( Holding<double>(1000) );    // DoubleStore.SizeInInts() will be 2000
```

### CouldHold<T>() :: Elements in a current buffer

This is more or less the reverse operation. Given a buffer, how many elements could it hold without resizing. (In effect this is what the size would be if the buffer were wrapped in a datagram of that type).

```
unsigned int shorts_available = CouldHold<short>( DoubleStore.SizeInInts() );    // result is 4000
```

## Buffer Header Datagrams

The increase in the number of buffer types also means there needs to be Header Datagrams to access them.

Buffer class	Header Datagram	Description
Buffer	PacketBufferHeader, PacketHeaderDatagram	Original datagram for the X3/X5, plus an alias to follow new naming convention.
PmcBuffer	PmcHeaderDatagram	X3/X5 buffer formats
VeloBuffer	VeloHeaderDatagram	X6 stream buffer wrapper
VitaBuffer	VitaHeaderDatagram	X6 stream buffer data packet

The datagrams for all buffer types except the Vita buffer have the same interface. The first word of the header is the standard PeripheralId() field, and the remainder is the packet size. A DataSize() field allows setting the size without caring about the size of any header or trailer size.

---

The Vita Buffer's header has a different format. There is a 16 bit size field, a StreamId that acts much like the PeripheralId does for the other packets. Any other information in the header will be supported by methods of this datagram.

## Buffer Trailer Datagrams

The addition of trailers means there be Trailer Datagrams to access them.

Buffer class	Trailer Datagram	Description
VitaBuffer	VitaTrailerDatagram	X6 stream buffer data packet

Since the VitaBuffer is the only buffer to use trailers, this is the only one with an explicit trailer datagram defined. The main function of interest in this case is the Padding() field. Vita buffers must be an even multiple of four words long. The system supports the truncation of a packet when data ends (for example when a trigger falls) so that all data before the boundary is transmitted. If the full four word packet is incomplete, the Padding() field of the trailer is set to show what part of the last 16 byte double word is “padding” – that is, not full of data. The value of this field is 0-15, where 0 means all data is valid, and 15 means that only the first byte is valid.

The AccessDatagram template and all classes derived from it understand the Padding() call and will truncate the effective size of the buffer to avoid the padding. So assume a Short datagram is wrapped around a buffer with a padding of 1. The datagram will ignore the last short integer (2 bytes) because one of them is invalid. An integer datagram would ignore the last four bytes. Only a char datagram would allow viewing of all valid data and still ignore the invalid data.

## Buffer Header/Trailer Utility Functions

These functions perform some useful operations on buffer headers and trailers.

### Clear Functions

```
void ClearHeader(Buffer & buffer);           // fill with 0s
void ClearTrailer(Buffer & buffer);
```

These functions clear the header or trailer of a buffer. They work with any buffer type.

### Header Correctness Functions

```
void InitPmcHeader(Buffer & buffer);          // "correct" Header
void InitHeader(PmcBuffer & buffer);          // (make size correct and
void InitHeader(VeloBuffer & buffer);         // other required fields
void InitHeader(VitaBuffer & buffer);
```

These functions make sure the headers are correct for each type. An important check is to fill the size field with the current size of the Data region plus headers and trailers. The Vita packet has some additional required fields that are initialized correctly by this function.

---

## Trailer Correctness Functions

```
void InitTrailer(VitaBuffer & buffer); // "correct" Trailer
```

This function makes sure all required fields of the trailer are initialized correctly.

## Header Size Conversion

```
void ConvertHeader(VeloBuffer & buffer); // if Velo Buffer has wrong type of header, fix it
```

This function corrects a buffer whose header was resized by some legacy operation to be the correct size for Velo packets. It copies the contents of the original header, as much as possible. It also calls `Clear()` and `InitHeader()` in the process.

---

## *New Streaming Object – VitaPacketStream*

The new streaming format of the X6 family is supported by a new streaming object, `VitaPacketStream`. The `VitaPacketStream` object connects to an X6 board exactly as the `PacketStream` object connected to an X5 or X3 board to implement the stream functionality.

The most fundamental difference between the `VitaPacketStream` is the format of the packet data. In `PacketStream`, each packet contains raw data from the source device. These packets were `PmcBuffers`. In `VitaPacketStream`, the basic packet streamed is a `VeloBuffer`. The contents of the `VeloBuffer` is a stream of `VitaBuffer` data. This stream is not necessarily aligned on `VitaBuffer` boundaries.

There are Malibu objects to simplify the construction of these packets, as described in the next section.

## Connection

A `VitaPacketStream` is associated with a particular X6 board object with the `ConnectTo()` method. If `Module` is an X6 baseboard and `Stream` a `VitaPacketStream`, the following code shows the connection and disconnection process:

```
Stream.ConnectTo(Module);
FStreamConnected = true;
Log("Stream Connected...");

... code that uses the Stream and board ...

Stream.Disconnect();
FStreamConnected = false;
Module.Close();
```

## Native Buffer Methods

```
VeloBuffer NativeBuffer(size_t data_size, bool autoinit=true);
VitaBuffer NativeVitaBuffer(size_t data_size, bool autoinit=true);
```

These methods produce a properly typed buffer for use with the streaming object. The `autoinit` parameter, if true, will initialize the buffer header and trailer for the buffer to zeros and for required values. This is done by calling the standard buffer `InitHeader` and `ClearHeader` functions. The `VitaBuffer` version also clears and initializes the trailer using `InitTrailer` and `ClearTrailer`.

---

### Send and Recv Methods

```
virtual void Send( short PeriphId, VeloBuffer & packet );  
virtual void Send( VeloBuffer & packet );  
virtual void Recv( VeloBuffer & packet );
```

Like PacketStream, these methods are used to dispatch a new packet to the output, or read a packet from the input. These can be used alone, asynchronously, or in conjunction with the stream data notification callback events for a more demand driven interface.

### Stream Data Notification Events

```
OpenWire::EventHandler<VitaPacketStreamDataEvent> OnVeloDataRequired;  
OpenWire::EventHandler<VitaPacketStreamDataEvent> OnVeloDataAvailable;
```

These notification events allow the user to install a handler method that will be called when a data packet arrives as input (OnVeloDataAvailable) or when a new buffer should be sent to the output ( OnVeloDataRequired). The application then uses the Send() method to issue a completed output buffer or the Recv() method to obtain the next input buffer.

Each event that is handled needs to be hooked to an application function or method to be used, as shown below. An unhooked event is silently ignored.

```
Stream.OnVeloDataRequired.SetEvent( this, &ApplicationIo::HandleDataRequired );  
Stream.OnVeloDataAvailable.SetEvent( this, &ApplicationIo::HandleDataAvailable );
```

### Direct Data Mode

VitaPacketStream also supports Direct Data Mode, where raw slabs of data are processed without being broken up into Velo or Vita packets. This allows higher rate logging applications, since a minimal amount of processing is being done on the data. This is done in an identical fashion to PacketStream, since it is a function of the common base class to both stream objects, PacketStreamBase.

---

## *Working with Vita Packet Streams*

The data in a Velo buffer is part of a stream of Vita packet data. This stream does not need to be aligned on packet boundaries. This makes parsing more difficult, as all packets need to be accounted for to properly extract the data. Malibu contains a pair of objects to insert Vita packets into Velocia packets (VitaPacketPacker) or to extract Vita packets from the stream (VitaPacketParser).

### VitaPacketParser – Parsing Input Packets

The VitaPacketParser object is used to extract Vita packets from the Velo packets received. To set up, you need to create the object in your application class:

```
Innovative::VitaPacketParser Vpp;
```

Then as part of the initialization, add a handler to the OnImageAvailable event. This handler is called when the parser finds a complete Vita packet in the data stream.

---

```
Vpp.OnImageAvailable.SetEvent(this, &ApplicationIo::Handle_VPP_ImageAvailable);
```

In the Snap example application, we read in a previously logged Velo packet stream in and extract the Vita packets in as a post processing step. This could also be done along with logging, but the extra processing reduces the data rate of the application.

```
// ...Reset Vita Parser
Vpp.Clear();
//
// Start Playback
IntermediatePlayer.Start();

VeloBuffer PB_Buffer;
while (l==1)
{
    // Extract Velo Blocks, Quit if done
    unsigned int good = IntermediatePlayer.PlayWithHeader(PB_Buffer);
    if (!good)
        break;

    // Parse Blocks
    Vpp.Append(PB_Buffer);
    Vpp.Parse();
}
```

Each Velo packet is appended to the parser and processed when Parse() is called. Parse() will walk through the data in the parser, finding all packets and calling the OnImageAvailable() for each one. All buffers must be passed in to allow the parsing to process the entire data stream.

```
// Quit if Parser has an error?
}
IntermediatePlayer.Stop();
```

For efficiency, the parser returns an object pointing to the image of the data in the original buffer. This allows the handler to only copy buffers with data of interest rather than all buffers streamed. The image class allows the inspection of the header region to return the Vita Stream identification word to determine the data type. If the data is found to be of interest, the Image class has the CopyDataTo() method to create a VitaBuffer from the image data. This buffer can then be analyzed using standard Malibu tools.

```
//-----
// ApplicationIo::Handle_VPP_ImageAvailable() -- Parser callback
//-----

void ApplicationIo::Handle_VPP_ImageAvailable(Innovative::VitaPacketParserImageAvailable & event)
{
    // Create Vita Packet
    VitaBuffer Vita;
    // Fill with Image Data
    event.Image.CopyDataTo( Vita );

    // ...Process Vita Packet Data
}
```

---

## VitaPacketPacker – Filling Output Packets

The corresponding helper class for output packets is VitaPacketPacker. To use this, you push Vita packets into the packer, and when packets of a particular size are accumulated, an event is fired to allow this Velo packet to be processed (presumably by sending it to the VitaPacketStream output with Send()).

This illustration is for a Wave example filling a Velo buffer full of a list of Vita packets holding a waveform. This is done once at start, and the Waveform buffer sent repeatedly when wave data is needed during the run.

```
//  
// We now have the data in a regular buffer. We need to copy it into  
// VITA buffers, and then those VITAs into the Waveform packet  
std::vector<Innovative::VitaBuffer> VitaQ;  
  
...copy buffer into queue of Vita Packets...
```

At this point we have our Vita packets pre-made, so we can enter them all at once. You could generate them and Pack each one at a time if desired.

The normal use of the packer is to insert the buffers after we have entered the packet data size to be the size of output buffer we want. In this case, its handier for us to just have one resulting buffer that fits the entire pattern in it so that we can send it over and over, the same as the X5 Wave did. So we intentionally make the “automatic” size too large, so no packets will be output as we go. Note that the size of the fill pattern includes the size of all the headers and trailers of the Vita packets, as well as the data itself.

We also hook the callback function before starting the packing.

```
//  
// Use a packer to load full VITA packets into a velo packet  
// ...Make the packer output size so big, we will not fill it before finishing  
VitaPacketPacker VPPk(scratch_pkt_size*2);  
VPPk.OnDataAvailable.SetEvent(this, &ApplicationIo::HandlePackedDataAvailable);  
  
// ...Shove in our VITA packets  
for (unsigned int i=0; i<VitaQ.size(); i++)  
{  
    VPPk.Pack( VitaQ[i] );  
}
```

All the data is packed, but the buffer isn't full so it can't be sent. To push out these kind of fractional packets, the Flush() command is provided that truncates the remaining data and outputs it as packet exactly sized for the data. InitHeader() then called to make sure the size field of the header is properly set up.

```
VPPk.Flush();    // outputs the one waveform buffer into Waveform  
  
// WaveformPacket is now correctly filled with data...  
  
InitHeader(WaveformPacket);    // make sure header packet size is valid...
```

This is the handler called when the Flush() function truncates and emits the packet. Since we constructed this example to only make one output packet, we just copy it to our Buffer. Other applications could call Send() here to output it, or add the buffer to a list or vector.

```
//-----  
// ApplicationIo::HandlePackedDataAvailable() -- Packer Callback  
//-----
```



---

```
void ApplicationIo::HandlePackedDataAvailable (Innovative::VitaPacketPackerDataAvailable & event)
{
    WaveformPacket = event.Data;
}
```

---

## Chapter 15. *Vita Packet Format*

### *Overview*

---

The X6 family of baseboards introduces a new form of data streaming that changes the format of the data packets streamed to and from the card. This chapter describes the format and the changes from previous use.

### *X6 Velocia Packets*

---

#### **Packet Header Format**

Just as in the X5, data busmastered between the board and the application program is enclosed in a data packet. For the most part, this packet has not changed. What has changed is that the packet header is now 4 words long, rather than 2. Also, the total packet size now must be divisible by 4 words rather than 2 as well. This is a reflection of the new internal bus being made wider for increased efficiency.

**Table 8. X6 Velocia (Velo) Packet Header**

<b>0</b>	<b>Velocia Header Word</b>
<b>1</b>	(reserved)
<b>2</b>	(reserved)
<b>3</b>	(reserved)

As before, the first word in the header contains the information used to identify the packet and determine its size. The Peripheral ID is a tag value used to identify the data source – all packets with the same Peripheral ID are defined as making up a single stream of data. The size allows the parsers to find the next header.

**Table 9. Velocia Header Word**

<b>3</b>	<b>3</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
<b>1</b>	<b>0</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>										
Peripheral ID												Packet size (in 32 bit words)																			

In the Malibu library there are additional classes to support these packets – VeloBuffer to hold packet data and VeloHeaderDatagram to access the header. The standard access datagrams will operate on VeloBuffers identically with the original buffers to access the data portion of the packet.

---

## Packet Data Format

The contents of an X6 Velocia packet is not raw data as in earlier systems, but a stream of VITA-49 formatted packets that are processed individually. These packets can be split across Velo packets, so the start of a Velo packet does not mean that a Vita packet header follows. The initial data could be part of a previous Vita packet's data.

### *X6 Vita Packets*

---

Like Velocia packets, Vita packets require information included with them in addition to the data to indicate the data source and other useful information. This header is seven words long. There is an additional trailer word for each packet that increases the total to 8 words. The total size of a Vita packet on the X6 must be a multiple of 4 words long.

The table below shows a minimal Vita packet – it contains only 4 words of data with its 8 words of header information.

**Table 10. Vita Packet Format**

<b>0</b>	<b>Header IF Word</b>
<b>1</b>	<b>Header SID Word</b>
<b>2</b>	<b>Header Class OUI Word</b>
<b>3</b>	<b>Header Class Info Word</b>
<b>4</b>	<b>Header Timestamp – Integer Seconds Word</b>
<b>5</b>	<b>Header Timestamp – Fractional Seconds High Word</b>
<b>6</b>	<b>Header Timestamp - Fractional-Seconds Low Word</b>
<b>7</b>	Packet Data 0
<b>8</b>	Packet Data 1
<b>9</b>	Packet Data 2
<b>10</b>	Packet Data 3
<b>11</b>	<b>Trailer Word</b>

The requirement of the Vita packet to remain aligned to a 4 word boundary conflicts with the need to dispatch a packet at once when events occur such as the trigger signal going off. If the packet remains unsent, part of the data will not arrive and even if it will eventually come in, part of the packet will have occurred at one time and part at potentially a much later time. To allow the packet to be sent in the absence of valid data, the concept of Padding was added.

A field in the trailer can be set to indicate to the destination which bytes in the last 16 bytes are not valid data. The application must ignore this data when processing the buffer. So in the case a trigger ends in the middle of a 4 word block, the packet can be padded to achieve alignment and the padding field in the trailer set to indicate this to the analysis routines.

The Access Datagram classes all recognize this padding value and reduce the size of the packet accordingly. This means the applications must initialize the Padding field before attempting to fill a packet.

## Packet Header Format

The above table shows the arrangement of the seven header words in the header. The remainder of this section will describe the fields for each word

### VITA Header IF word

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0									
Packet Type			C	T	R	R	TSI	TSF	Packet Count			Packet Size																		
0001			1	1	0	0																								

Packet type: set to IF Data packet with Stream Id, binary 0001.

C (bit 27): Optional Class field, enabled.

T (bit 26): Optional Trailer field, enabled.

TSI (bits 23-22): Timestamping format for integer seconds field, configurable.

**Table 11. Timestamp Integer Seconds Options**

TSI code	Meaning
00	No Integer-seconds Timestamp field included (Not supported)
01	UTC: seconds elapsed since January 1, 1970 GMT.
10	GPS: seconds elapsed since January 6, 1980 GMT.
11	Other: seconds elapsed since some documented starting time.

TSF (bits 21-20): Timestamping format for fractional seconds field, configurable between 01 (sample count) or 11 (free running).

**Table 12. Timestamp Fractional Seconds Options**

TSF code	Meaning
00	No Fractional-seconds Timestamp field included (Not supported)
01	Sample count timestamp: fractional seconds since last integer-seconds event, counting in samples.
10	Real time timestamp: counts in increments of 1 picosecond since last integer-seconds event. (Not supported)
11	Free running count timestamp: No relation to the integer-seconds field. Counts in samples.

Packet count: increments on each subpacket with the same Stream Id (PDN). It's allowed to roll over from 1111 to 0000.

Packet size: in 32-bit words, including the header; the packet size is always a multiple of 4 due to how the packets are handled internally.

#### VITA Header SID word

3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0								
Destination Mask																Stream ID													

The VITA-49 Stream Id field is split into two 16-bit fields: Destination Mask and Stream Id.

The Destination Mask will help route packets to their destinations (ie. PCIE, Aurora 0 or 1 etc.).

The Stream ID will be used much like the peripheral ID is for Velocia packets – to identify the source or meaning of the data in the packet.

#### VITA Header Class OUI Word

This word is reserved.

#### Vita Header Class Info Word

This word is reserved.

#### Vita Header Timestamp – Integer Seconds Word

This word contains the integer portion of the timestamp data.

#### Vita Header Timestamp – Fractional Seconds High and Low Words

These words contain the fractional portion of the timestamp data.

---

## Vita Packet Trailer Format

### VITA Trailer Word

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	
Enables								1111				State and Event indicators								Padding in bytes				E	Context Packet count							

### State and Event Bits and Enable Bits

Enable bit position	State and event Indicator bit position	Indicator name
31	19	Calibrated Time
30	18	Valid Data
29	17	Reference lock
28	16	AGC/MGC Indicator
27	15	Detected Signal indicator
26	14	Spectral inversion
25	13	Over-range indicator
24	12	Sample Loss (over-run)

Setting the appropriate bit in the Enable field will cause the State/Event bit to follow the signals given in the table. These will flag if the enabled conditions occur in the current Vita packet.

### Bits 20-23

These should always be set to 1

### Context Packet Count

This number tells which context packet is associated with this packet. Context packets can give slowly changing information, such as temperature readings, into a data stream without burdening each data packet with data that is unlikely to be different from packet to packet.

### Padding

The Padding field indicate how many padding bytes were added to align the current packet to a 4 word boundary. Padding values can be from 0 (no padding) to 15 (1 data byte valid in the last 4 words).

---

**Table 13. Padding Example**

Data n-3	Data n-4	Data n-5	Data n-6
Padding 1	Data n	Data n-1	Data n-2
Padding 5	Padding 4	Padding 3	Padding 2
Padding 9	Padding 8	Padding 7	Padding 6
Trailer (Padding Field Value = 9)			

The above table shows an example of padding. The logic had accumulated the data in grey when the trigger was disabled – seven bytes of data. The packet can not be sent because the data would not be properly aligned. The logic therefore adds the nine byte remainder to the packet as padding, and loads the value 9 into the padding field of the trailer.

Analysis routines that check the padding value will properly limit themselves to reading the valid data and ignore the padding bytes.

**Table 14. Maximum Padding for X6 Boards**

Source FIFO parallel samples	Event size (bits)	Maximum Padding (bytes)
1x16 (RX)	16	14
2x16 (400M)	32	12
4x8 (GSPS)	32	12
8x8 (GSPS DES)	64	8

The above table shows the maximum expected padding value for a particular X6 board. The RX, which has one 16 bit sample per packet for each period, could possibly have to pad 14 of 16 bytes in a 4 word block. Devices which send more data per sample period will have smaller maximum padding values.

---

## Chapter 16. *Creating a Custom FMC Module*

---

The Malibu Class Library provides classes that implement the functioning of Innovative FMC modules, as well as that of the Carrier cards that use them. Customers that want to develop their own module to attach to our cards are obviously not supported directly with classes. However, there is a way for those customers to develop software that will integrate with the library just as native Innovative modules do.

### *FMC Modules and Malibu*

---

First, we need to consider how an FMC module and carrier fit in the Malibu scheme. Previous boards were unified, and one class object performed all the duties of the card – integration with data streaming, initialization of streaming and analog properties, configuration of any clocks and triggers, and so on. (Internally we divided up the duties into a trio of Board/Driver/DeviceMap classes, but the idea remains the same).

From the user's perspective a board needs to be Opened and Closed, and be able to attach to the Streaming object that handles data streaming. The initiation of streaming is done in the separate data streaming class (here, VitaPacketStream) and the stream class calls over to the board at times in the stream process to allow the board to initialize itself. The final job that a board class must do is to provide a clean interface for all of the parameters needed to configure the board – clocks, triggers, channel enables, status readback registers, stream packet configuration, and so on. This part tends to be the bulk of the code, as you have to translate in and out the information for not only each register, but each field and bit in a register. In practice we tend to reuse these clusters of registers from product to product, a practice that was regularized in X6 and later products as a 'subdevice' class – a collection of registers and access methods and implementation functions bound together as a unit.

The FMC device and carrier design required a change in thinking, and the best software model was to divide up the memory map and assign the implementation of the aspects of the FMC card to a FMC board class. Just as the physical device is plugged into the board, the FmcBoard object is plugged into the FMC Carrier card board object. Since this attachment needs to work for any future product from Innovative, we need an Interface class to define the view a baseboard has of a module. This interface is the IFmcDaughterCard class.

What this means for makers of custom boards is that if you make your class implement the IFmcDaughterCard interface, your class will work in a Malibu application as well as an Innovative board will.

### *The IFmcDaughterCard Interface Class*

---

```
//=====
// CLASS IFmcDaughterCard  -- FMC Daughter Card Interface
//=====

class IFmcDaughterCard
{
public:
    enum DaughterCardStatusStates
    { dsOk, dsCardPoweredDown, dsNoSoftwareObjectPresent, dsNoCardPresent,
```



```

        dsCardPowerFailure
    };

    virtual DaughterCardStatusStates BoardStatus() { return FBoardStatus; }

    virtual void ConstructCustomParts() = 0;    // call in most derived ctor
    virtual void InitCardPower() = 0;
    virtual void OpenHardware() = 0;
    virtual void PreconfigureHardware() = 0;
    virtual void ConfigureHardware() = 0;
    virtual void StreamStopHardware() = 0;
    virtual void CloseHardware() = 0;
    virtual void RemoveCardPower() = 0;

    // Ficl Interface
    virtual void AfterFiclStart(FiclSystem & engine) = 0;

    virtual int Do_DFatch(int dev, int addr) = 0;
    virtual void Do_DStore(int dev, int addr, int value) = 0;
    virtual void Do_DFreq(int dev, float freq) = 0;
    virtual float Do_DFreqActual(int dev) = 0;

    IFmcDaughterCard() : Owner(0), FBoardStatus(dsCardPoweredDown)
    {}
    virtual ~IFmcDaughterCard()
    {}

    // Memory Spaces
    AddressingSpace PortMemory;
    AddressingSpace LogicMemory;

protected:
    class FmcCarrierBoard * Owner;
    DaughterCardStatusStates FBoardStatus;

    void BoardStatus(DaughterCardStatusStates status)
    { FBoardStatus = status; }

};

```

## BoardStatus()

```

enum DaughterCardStatusStates
{ dsOk, dsCardPoweredDown, dsNoSoftwareObjectPresent, dsNoCardPresent,
  dsCardPowerFailure
};

```

The initialization of standard FMC cards will set the Board Status enumeration value which can be retrieved by an application from the BoardStatus() method. To avoid issues with a custom card, the software supporting it should do the same.

Enumeration	Description
dsOk	FMC module is properly set up and ready.
dsCardPoweredDown	FMC module is powered down by the operator
dsNoSoftwareObjectPresent	No FMC software object is connected.
dsNoCardPresent	No physical board could be detected.

---

---

Enumeration	Description
dsCardPowerFailure	Card power could not be applied due to a resource mismatch.

The FMC standard requires information about the power requirements to be stored in a standard format in a ROM. This rom will be examined and checked against what the baseboard can deliver. If the requirements of the module are too extreme, then the module will not be powered up. The last two cases in the above table are ways that this can fail – first if the ROM cannot be read, and secondly if the information shows that this board will not function on the carrier.

Parts of this process are still being worked out and are subject to change.

### **InitCardPower() and RemoveCardPower()**

```
virtual void    InitCardPower() = 0;
virtual void    RemoveCardPower() = 0;
```

These methods are part of the FMC card power management interface, which is still in progress.

### **ConstructCustomParts()**

```
virtual void    ConstructCustomParts() = 0;    // call in most derived ctor
```

These next eight or so methods are the way that the FMC card, and a custom card tie into the library and the streaming system. At key points in the lifetime of the baseboard, the library will call the FMC card's methods to give each connected card a chance to perform the required function. A custom card just has to follow the same rules that Innovative cards do to be fully functional.

### **OpenHardware()**

```
virtual void    OpenHardware() = 0;
```

OpenHardware() is called when resources are first given to the board during the carrier's Open() process. The FMC card can use this call to set up its software elements in a known good state.

### **PreconfigureHardware()**

```
virtual void    PreconfigureHardware() = 0;
```

Some analog devices take a lot of time to set themselves up. Doing this every data run might be expensive, especially if the setup is not changing in a significant way. Devices that perform in this way can put the key initialization in the Preconfigure() step, and then perform multiple streaming operations using the same settings. If this is the case for an FMC device, this call is the place to perform this sort of initialization.

---

## ConfigureHardware()

```
virtual void    ConfigureHardware() = 0;
```

This is for pre-streaming configuration. Most board set up is done at this stage.

## StreamStopHardware()

```
virtual void    StreamStopHardware() = 0;
```

This is for post-streaming cleanup.

## CloseHardware()

```
virtual void    CloseHardware() = 0;
```

CloseHardware() is called during the carrier's Close() process. Any cleanup that the FMC card needs to perform can be done here.

## Memory Spaces and Registers

```
// Memory Spaces
AddressingSpace    PortMemory;
AddressingSpace    LogicMemory;
```

Innovative boards use memory-mapped areas to use as registers for sending parameters to the physical card and reading status information back from the card. They are not intended for bulk data transfer, which is done via the streaming engine. The carrier card implements two such spaces, but only the Logic Memory space will be used.

Note that this space gives access to the full memory map, including the carrier card's addresses. This isn't what you want normally, so our base class creates WishboneBusSpaces to divide up the base region. The only difference is that the WishboneBusSpace adds a base to turn an FMC relative address into the absolute offset needed for the physical access. A custom board could do likewise.

```
//
// Data
WishboneBusSpace    WB_FMC_0;
```

The point of having memory spaces is that the registers can be defined with space and offset as parameters, allowing them to be reused effectively. Here is a sample register object:

```
//~~~~~
// CLASS FmcAFE_AdcCommon::AdcTriggerCfgRegister -- Configure ADC Trigger
//~~~~~
class AdcTriggerCfgRegister : public Register
{
    typedef Register inherited;

public:
    RegisterBitGroup    AdcFrameSize;
    RegisterBit          AdcRisingEdge;
    RegisterBit          AdcFramedMode;
```

---

```

        RegisterBit      AdcExternalTrigger;

public:
    AdcTriggerCfgRegister( IAddressingSpace &space, int offset )
        : inherited(space, offset),
          AdcFrameSize(*this, 0, 24),    AdcRisingEdge(*this, 29),
          AdcFramedMode(*this, 30),     AdcExternalTrigger(*this, 31)
        {
        }
};

```

So the bit and bit group locations can be defined here and modified individually without worry that changing one will trash the others. In the constructor for the object owning the register (here, it is a subdevice class) you can see the space and register offsets being loaded into each register at construction time.

```

FmcAFE_AdcCommon::FmcAFE_AdcCommon(IRequires_FmcCommonParts * mdc, IAddressingSpace & space,
MapRegisters & regs)
    : MDC(mdc), Dev(0),
      AdcEnable_Lo_Reg(space, regs.AfeAdcEnable_Lo_Addr),
      AdcEnable_Hi_Reg(space, regs.AfeAdcEnable_Hi_Addr),
      AdcPower_Lo_Reg(space, regs.AfeAdcPower_Lo_Addr),
      AdcPower_Hi_Reg(space, regs.AfeAdcPower_Hi_Addr),
      AdcTriggerCfgReg(space, regs.AfeAdcTriggerCfg_Addr),
      AdcDecimationReg(space, regs.AfeAdcDecimation_Addr),
      FDevices(0), FChannelsPerDevice(0)
{
}

```

While this may seem a bit much at first, from experience formalizing the register map makes maintenance vastly easier down the road as you develop. And much of the effort in the software part of supporting this hardware are the interface from the user to the board object and the corresponding interface from the board object down to the hardware register or registers.

## The FICL Interface

```

// Ficl Interface
virtual void    AfterFiclStart(FiclSystem & engine) = 0;

virtual int     Do_DFatch(int dev, int addr) = 0;
virtual void    Do_DStore(int dev, int addr, int value) = 0;
virtual void    Do_DFreq(int dev, float freq) = 0;
virtual float   Do_DFreqActual(int dev) = 0;

```

Innovative cards implement a threaded interpretive command language in them for debugging purposes. Functions are defined to access registers 'live' during a program, and programs can be written in the language. One especially useful aspect of the interface is the device fetch and store methods (d@ and d!). These words are mapped by the board object into calls to the Do\_DFatch() and Do\_DStore() ca..s of the carrier card. The carrier card will forward the call to the daughter card if the device code does not match one defined on the carrier card, thus calling these functions.

A common 'device' defined is that of an I<sup>2</sup>C device. The raw mechanics of this interface are complicated, and the library needs to implement them anyway. So attaching a FICL device to this allows you to access this device for debugging as easily as if it were memory mapped.

Clock devices are also common, and have two additional device functions defined – one to set the clock frequency and one to report the actual clock frequency.

The AfterFiclStart() function is used to predefine any FICL words needed for support. Constants for the devices is a common one here. That way a user can use “AdcSpi0 100 d@” to fetch the data from address 100 on the AdcSpi0 device.

```
//-----
//  FmcAdc20::AfterFiclStart() --
//-----

void  FmcAdc20::AfterFiclStart(FiclSystem & engine)
{
//    engine.Evaluate(" 0 constant Fmc ");    ...in baseboard class
engine.Evaluate(" 100 constant AdcSpi0 101 constant AdcSpi1 102 constant AdcSpi2");
engine.Evaluate(" 103 constant AdcSpi3 104 constant AdcSpi4 105 constant P11 ");
engine.Evaluate(" 106 constant TempI2C 107 constant IdromI2C  ");
engine.Evaluate(" 108 constant Vga0I2C 109 constant Vga1I2C 110 constant Vga2I2C  ");
engine.Evaluate(" 111 constant Vga3I2C 112 constant Vga4I2C 113 constant VgaIdromI2C  ");

engine.Evaluate(": ?FmcAdc20 "
    " cr .\" --- FMC SubDevice Constants for d@ d! dFreq dFreqActual --- \" "
    " cr .\" (starting at 100) \" "
    " cr .\" AdcSpi0 AdcSpi1 AdcSpi2 AdcSpi3 AdcSpi4 P11 \" "
    " cr .\" TempI2C IdromI2C \" "
    " cr .\" Vga0I2C Vga1I2C Vga2I2C Vga3I2C Vga4I2C VgaIdromI2C \" "
    " cr "
    " cr .\" ?FmcAdc20 : display this message \" "
    " cr ; ");

engine.Display("FmcAdc20-specific wordset loaded \n" );
}
```

The final word is a debug aid that just prints out the device codes for an FmcModule. Note that there is a special device Fmc that offsets the address to be relative to the module memory space. For example, if a Module starts at address 0xC00, then Fmc 0 d@ is the same as 0xC00 l@ on the carrier card.

---

## Chapter 17. *Interfacing to Software Applications via a DLL*

### *Overview*

---

The desire to control one or more Innovative board level products from within a foreign environment such as NI Labview or Mathworks Matlab is commonplace. Unfortunately, it is virtually impossible to create a universal interface between these environments and the Malibu C++ libraries that meets all of the needs of every possible application. Fortunately, these third-party environments provide extensibility via user-supplied dynamic link libraries (DLLs). A user written DLL acts as a “bridge” between the Malibu C++ and the third party environments.

### *Development Approach*

---

The DLL must be written in C++ and linked to the Malibu libraries – similar to the process used in the creation of a standard executable program. The DLL can incorporate any available Malibu class, user-developed C/C++ classes or libraries or other commercial libraries to implement complex computations in addition to Innovative board control functions. The DLL provides another opportunity to factor code as required to best meet your application requirements. For instance, analog time-series data received via bus mastering from an Innovative XMC module within a DLL could be transformed into frequency domain using the Malibu Fourier class. This frequency domain data could be further analyzed to provide AC quality information such as SNR or THD using the Malibu AdcStats class and the results of this analysis could be made available to external applications or third party environments via C-callable functions. Any application that uses the DLL gains access to this board control and analysis capability.

The most efficient approach to creating such a DLL is to subsume the entire ApplicationIo board control object which is included in each of the Innovative example programs directly into the user-written DLL. Control of Innovative XMC modules within supplied examples are factored such that all board control functions and real-time callback handlers are implemented in the portable ApplicationIo C++ object. Because this class is compiler and user interface independent, we recommend incorporation of this class directly into end-user applications and DLLs, since this guarantees that all required board initialization and control are properly implemented.

### *Example Source*

---

Complete source code illustrating this technique is available on the II forum [here](#). This project uses the above encapsulation strategy to assimilate ApplicationIo into a DLL, which exports various board control functions as plain “C-style” function calls, so that they can be used from environments like LabView or Matlab. The Windows, 32-bit DLL has been written in

---

C++ under MSVC 2008. All exported functions are C-compatible and will be restricted to use of plain C data types and structures for parameters.

The underlying C++ code within the DLL uses the Innovative Malibu libraries to perform all board configuration and control functions. Malibu runs as native machine code, is not dependent on the .NET libraries, but requires that the MSVC runtime libraries be present on the target machine. This particular example exposes registers of a dual-channel DDC which was developed in conjunction with custom firmware for a specific application. While the functions in this DLL are specialized for that application, they are merely representative of the types of functions that could be exposed in any custom static or dynamic library.

---

## Chapter 18. *Using the embedded FICL interpreter*

---

Since Innovative products include user-programmable FPGA devices, the requirement to interact with newly-added peripheral devices is routine. Malibu incorporates an interpreted language facility, named *Ficl*, which is useful when initially testing new devices and resources. Ficl is a programming language interpreter designed to be embedded into other systems as a command, macro, and development prototyping language. Ficl is an acronym for "Ficl Inspired Command Language".

### *Ficl Features*

---

Ficl is written in strict ANSI C and runs natively on 32- and 64-bit processors. It has a small memory footprint - A fully featured Win32 console version takes less than 100K of memory, and a minimal version is less than half that.

Ficl is easy to integrate into custom C++ programs. Where most Ficl's view themselves as the center of the system and expect the rest of the system to be coded in Ficl, Ficl acts as a component of your program. Since Ficl is embedded into Malibu, most C++ applications inherit Ficl with no effort whatsoever.

Ficl is fast, thanks to its "switch-threaded" virtual machine design. Ficl also features blindingly fast "just in time" compiling, removing the "compile" step from the usual compile-debug-edit iterative debugging cycle.

Ficl is a complete and powerful programming language.

Ficl is an implementation of the Ficl language, a language providing a wide range of standard programming language features including integer and floating-point numbers, with a rich set of operators, arrays, file I/O, flow control (if/then/else and many looping structures), subroutines with named arguments and language extensibility.

Ficl is standards-compliant and conforms to the 1994 ANSI Standard for Ficl (DPANS94). See the document [Ficl\\_Primer.pdf](#) for a tutorial introduction to Ficl, as this knowledge is directly applicable to creation and use of Ficl within Malibu-based C++ applications.

Ficl is extensible. Ficl is extensible both at compile-time and at run-time. You can add new script functions, new native functions, even new control structures.

Ficl is interactive. Ficl can be used interactively, like most other Ficl's, Python, and Smalltalk. You can inspect data, run commands, or even define new commands, all on a running Ficl VM. Ficl also has a built-in script debugger that allows you to step through Ficl code as it is executed.

Ficl is open-source and free. The Ficl licence is a BSD-style license, requiring only that you document that you are using Ficl. There are no licensing costs for using Ficl.

Previous versions of Malibu utilized specialized scripting classes such as `Innovative::Scripter`, which were limited in capability. The new integrated Ficl interpreter renders such classes obsolete.

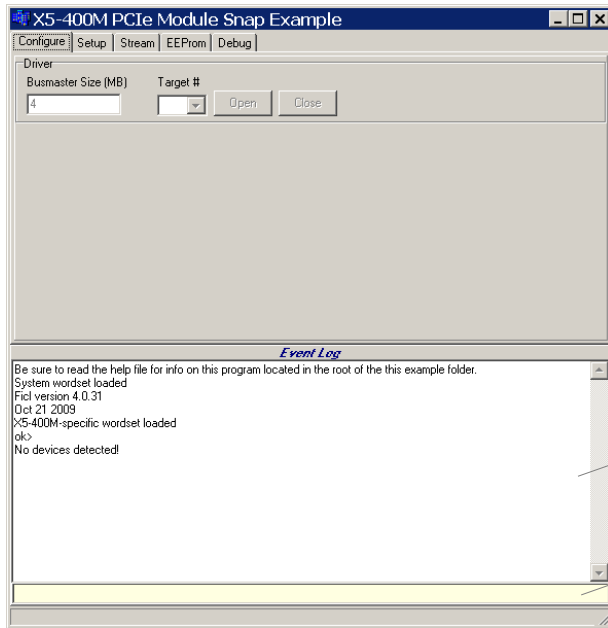


---

## A Beginners Guide

---

Ficl is embedded into the Snap example provided with most X5 and X3 XMC modules. Launching that application will result in a display that resembles the following:



Event log displays Ficl interpreter responses

Command line accepts Ficl commands

You can use upper or lower case to type commands and data, since Ficl is case-insensitive. You can also store commands in a standard text file using your favorite editor, then execute these commands using the `load <filename>` command. Ficl allows execution of the system shell, via the `system` command, which can be used to invoke you editor to add commands to a file. For instance under Windows, the command `system notepad hello.fr`, creates a new text file called *hello.fr* using the Notepad editor. New commands added to this file could be loaded into Ficl using the command `load test.fr`.

### Using Ficl

Ficl is an embedded variant of the interpreted language *Forth*. Review the [Forth Primer](#) to become acquainted with the characteristics and capabilities of this powerful language. The [Forth Standard](#) is the definitive reference on the language features, but most users will find the primer sufficient. The examples illustrated below assume a working knowledge of the material presented in these documents. The particular features and capabilities of FICL are detailed on the [FICL Website](#). Note that Malibu implements FICL 3 rather than FICL 4, to allow operation in both 32 and 64-bit environments.

Ficl incorporates a floating point stack in addition to the conventional parameter and return stacks. Floating point values are recognized via an embedded 'e' within the number. For instance 1.0e3 is interpreted as floating point 1000. Within stack notation, values prefaced with f: are floating point values and reside on the floating point stack rather than the parameter stack.

Ficl-enabled XMC module object derive publicly from `Innovative::IFiclTarget`. Consequently, the following module-oriented words are available for use:

<i>Word</i>	<i>Description</i>	<i>Stack</i>
?<module-name> e.g. ?X5-G12	Display default module-specific wordset	( -- )
l@	Fetch from logic space. Read 32-bit FPGA register contents.	( addr -- data )
l!	Store to logic space. Write 32-bit FPGA register contents.	( data addr -- )
p@	Fetch from pci space	( addr -- data )
p!	Store to pci space	( data addr -- )
d@	Fetch from sub-device space. Read 32-bit value from specified address in specified device. Used to access registers stored within peripheral devices attached to the FPGA via I2C, SPI or other buses.	( addr dev -- data )
d!	Store to sub-device space. Write 32-bit value to specified address in specified device. Used to access registers stored within peripheral devices attached to the FPGA via I2C, SPI or other buses.	( data addr dev -- )
Pll, Vcxo, RefVcxo, Adc0, Adc1	Module-specific constants which name the device values used with d@ and d!. Specific constants available vary by module.	( -- index )
dFreq	Set sample clock frequency	( f:freq dev -- )
dFreqActual	Report actual sample clock frequency	( dev -- f:freq )

Ficl words may be executed interactively at the Ficl command line, or they may be typed into a standard disk file for subsequent re-use. The Ficl word `load` may be used to interpret the entire contents of a file, exactly as though the contents of the file were typed into the Ficl command line. Usage is

```
load <filename>
```

Note that Ficl word definitions may also be added to files, to create utility functions containing loops, register displays and myriad other useful diagnostics.

For example, the word `.regs` below will create a hex dump of the contents of user FPGA logic registers 0..10:

```
: .regs    ( -- )    base @ hex    10 0 do    ." Reg: 0x" i .    ." =" i l@ .    cr    loop    base ! ;
```

Once this definition has been added to the dictionary, it may be executed by typing:

```
.regs
```

at the Ficl command line. Or, this word could be used within other words to create a more sophisticated diagnostic.

## Stream Support

A handful of Ficl words are executed automatically at strategic times during data flow to accommodate automated initialization, finalization or diagnostic display.

---

---

<i><b>Word</b></i>	<i><b>When Executed by Malibu</b></i>	<i><b>Stack</b></i>
OnStreamConfigure	Prior to register configuration in preparation for streaming	( -- )
OnBeforeStreamStart	Before assertion of FPGA RUN bit. Contents of all FPGA registers have been reapplied.	( -- )
OnAfterStreamStart	After assertion of FPGA RUN bit. Streaming is underway.	( -- )
OnBeforeStreamStop	Before deassertion of FPGA RUN bit as streaming is about to terminate	( -- )
OnAfterStreamStop	After deassertion of FPGA RUN bit after streaming terminates	( -- )

Words added to the dictionary named as listed above will be executed automatically at the time specified during data streaming. This can be used to initialize application-specific registers, display state or any other diagnostic function.