CARDSHARP

User's Manual

The Cardsharp User's Manual was prepared by the technical staff of Innovative Integration on June 10, 2016.
For further assistance contact:

Innovative Integration
741 Flynn Road
Camarillo, CA 93012

PH: (805) 383-8994
FAX: (805) 482-8470

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

VSS \ Distributions \ Cardsharp \ Documentation \ Manual \ Cardsharp.odm
Rev A

Table of Contents

List of Tables

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.

Terminology

What is the Cardsharp?

Cardsharp is a user-customizable, turnkey embedded instrument that includes two A9 CPU cores. Linux runs in core 0 to provide Ethernet, USB and disk connectivity while core 1 runs real-time stand-alone applications. Cardsharp is compatible with Innovative's wide assortment of ultimate-performance FMC modules. With its modular IO, scalable performance and easy to use CPU core architecture, the Cardsharp reduces time-to-market while providing the performance you need.

What is Vivado?

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis

What kinds of applications are possible with Cardsharp hardware?

Data acquisition, data logging, Distributed Data Acquisition, stimulus-response and signal processing jobs are easily solved

with Innovative Integration baseboards using the Malibu software. There are a wide selection of peripheral devices available in the Innovative product family, 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.

Distributed Data Acquisition – Put the Cardsharp at the data source and reduce system errors and complexity. Available IEEE 1588 network or GPS-synchronized timing, triggering and sample control is available for remote IO. Limitless expansion via multiple nodes.

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.

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)

Online Help
The online help system for Malibu is fully integrated into the excellent OpenHelp system provided with Builder. Help for Malibu is provided in a single file, Malibu.hlp which is installed beneath the main Builder C++ directory tree during the default installation. It provides detailed information about the components contained in Malibu - their Properties, Methods, Events, and usage examples. An

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.

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

Chapter 2: Installation

Malibu for PetaLinux

This is a procedure to install the Malibu for PetaLinux software library on a development host PC running either 64-bit Windows or Linux. This procedure presumes that the Xilinx Software Development Kit (SDK) has been installed on the host. Xilinx project files are included with the library to automate building the library and its example applications for execution on a Innovative Integration Cardsharp target system running Xilinx PetaLinux. The procedure is nearly identically for Windows and Linux hosts.

Preparing Workspace Preferences

Run the Malibu PetaLinux software installer on the development host. In the installation folder, find and extract the PetaLinuxProject.zip archive into a folder, henceforth assumed to be called “PetaLinuxProject” (although the name and location can be chosen freely). Start the Xilinx SDK program and select an existing or new workspace in the “Workspace Launcher” dialog. When the program's workbench IDE has opened, select the **File** menu from the top menu bar and then select **Import** from the File dropdown menu. In the Import wizard, open the **General** list, left-click **Preferences** and then click the **Next** button.

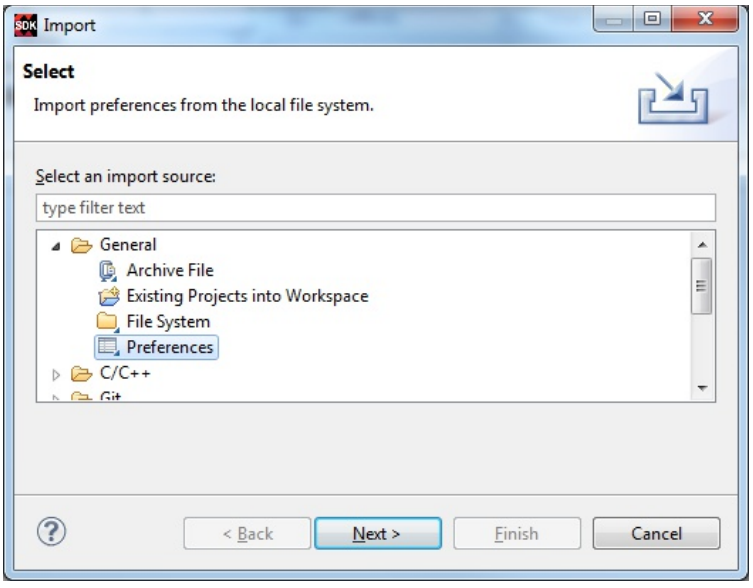

Figure 1. Importing SDK Workspace Preferences

Left-click the **Browse...** button. In the file explorer dialog that opens, navigate to the installation folder and, if on a Linux host, find and select the “LinuxMalibuPreferences.epf” file or otherwise select the “WindowsMalibuPreferences.epf” file, and click the **Open** button. Back in the Import wizard, ensure that the **Import all** checkbox is checked and click the **Finish** button.

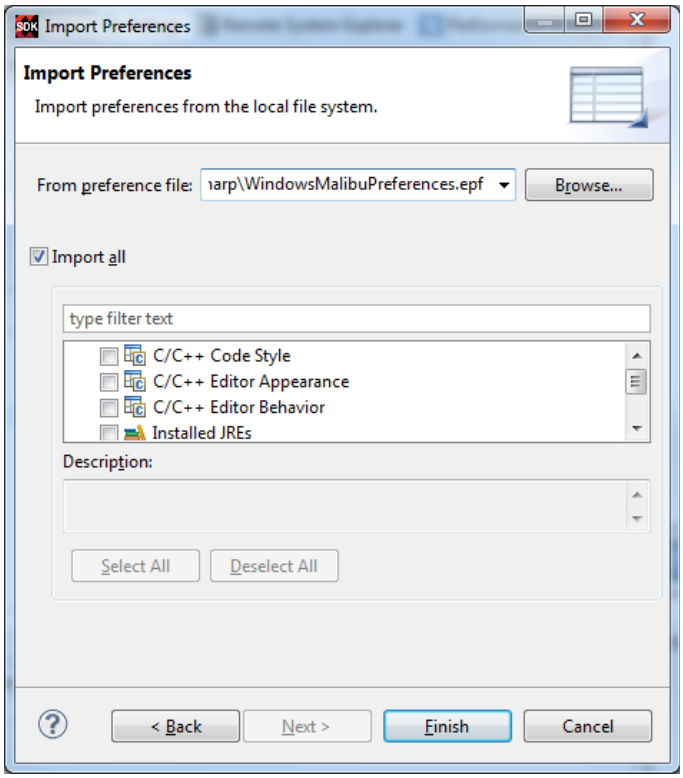

Figure 2. Selecting SDK Workspace Preferences File

If a version of Xilinx SDK other than 2015.4 reports an error reading in the file, the preferences can be created manually but, in any case, some values must be adjusted to match the location of the installation folder. Select the **Window** menu from the workbench top menu bar and then select **Preferences** from the bottom end of the Window dropdown menu. In the Preferences dialog, open the **General** list in the left pane, then in that list open the **Workspace** list, and finally, select **Linked Resources**.

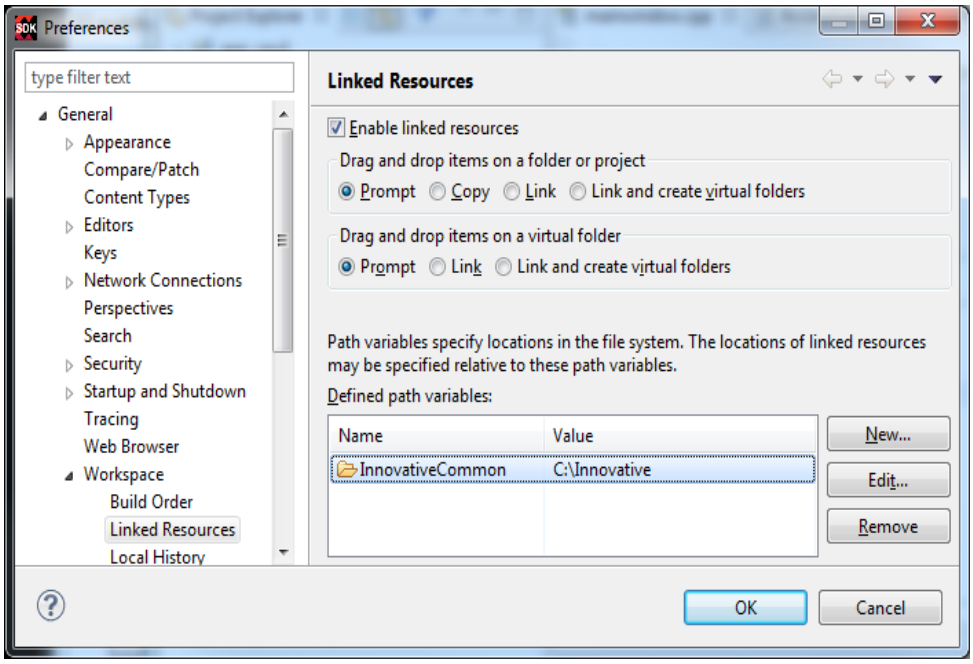

Figure 3. SDK Workspace Linked Resources

If the "InnovativeCommon" path variable does not exist, press the **New** button to create it or otherwise press the **Edit** button. In either case, ensure that its value is the path of the installed "Innovative" folder.

Next, in the left pane of the Preferences dialog, open the **C/C++** list and, in that list, open the **Build** list. From that list, select **Environment**. The following figure depicts the "SDK Preferences" dialog that appears:

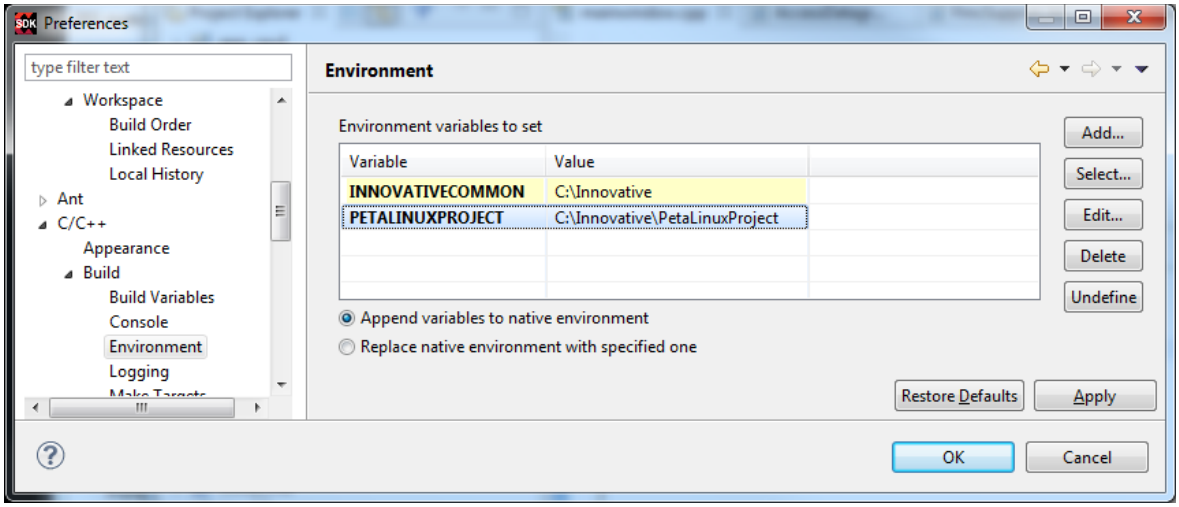

Figure 4. SDK Workspace Environment

If the "INNOVATIVECOMMON" path variable does not exist, press the **Add** button to create it or otherwise press the **Edit** button. In either case, ensure that its value is the path of the installed "Innovative" folder. The value of the "PETALINUXPROJECT" environment variable must be set to the path of the "PetaLinuxProject" folder. Press the **Apply** button.

Back in the left pane of the Preferences dialog, from the **C/C++ Build** list, select **Build Variables**. On a Windows host, the build variables must have the names, values, and types shown below:

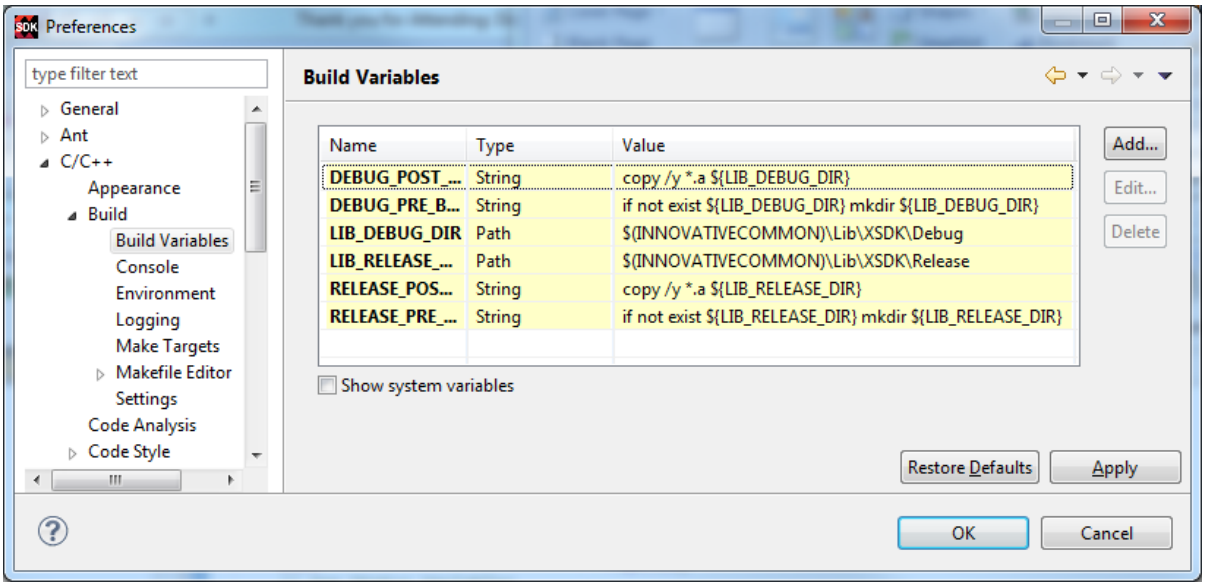

Figure 5. Windows SDK Workspace Build Variables

If the variables must be created, use the information in the table below:

Name	Type	Value
DEBUG_POST_BUILD_STEP	String	copy /y *.a ${LIB_DEBUG_DIR}
DEBUG_PRE_BUILD_STEP	String	if not exist ${LIB_DEBUG_DIR} mkdir ${LIB_DEBUG_DIR}
LIB_DEBUG_DIR	Path	$(INNOVATIVECOMMON)\Lib\XSDK\Debug
LIB_RELEASE_DIR	Path	$(INNOVATIVECOMMON)\Lib\XSDK\Release
RELEASE_POST_BUILD_STEP	String	copy /y *.a ${LIB_RELEASE_DIR}
RELEASE_PRE_BUILD_STEP	String	if not exist ${LIB_RELEASE_DIR} mkdir ${LIB_RELEASE_DIR}

Table 1. Windows SDK Workspace Build Variables

On a Linux host, the build variables must instead have these values:

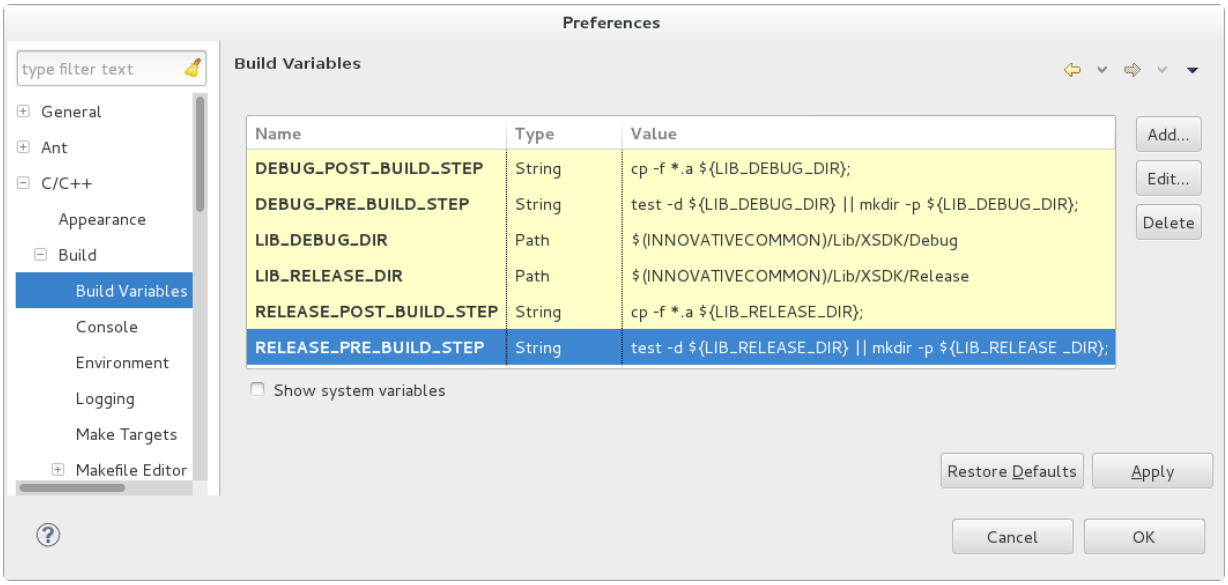

Figure 6. Linux SDK Workspace Build Variables

If the variables must be created, use the information in this table:

Name	Type	Value
DEBUG_POST_BUILD_STEP	String	cp -f *.a ${LIB_DEBUG_DIR};
DEBUG_PRE_BUILD_STEP	String	test -d ${LIB_DEBUG_DIR} || mkdir -p ${LIB_DEBUG_DIR};
LIB_DEBUG_DIR	Path	$(INNOVATIVECOMMON)/Lib/XSDK/Debug
LIB_RELEASE_DIR	Path	$(INNOVATIVECOMMON)/Lib/XSDK/Release
RELEASE_POST_BUILD_STEP	String	cp -f *.a ${LIB_RELEASE_DIR};
RELEASE_PRE_BUILD_STEP	String	test -d ${LIB_RELEASE_DIR} || mkdir -p ${LIB_RELEASE _DIR};

Table 2. Linux SDK Workspace Build Variables

When finished, press the **OK** button in the Preferences dialog.

Importing Malibu Projects

Select the **File** menu from the workbench top menu bar and then select **Import** from the File dropdown menu. In the Import wizard, open the **General** list, left-click **Existing Projects into Workspace** and then click the **Next** button.

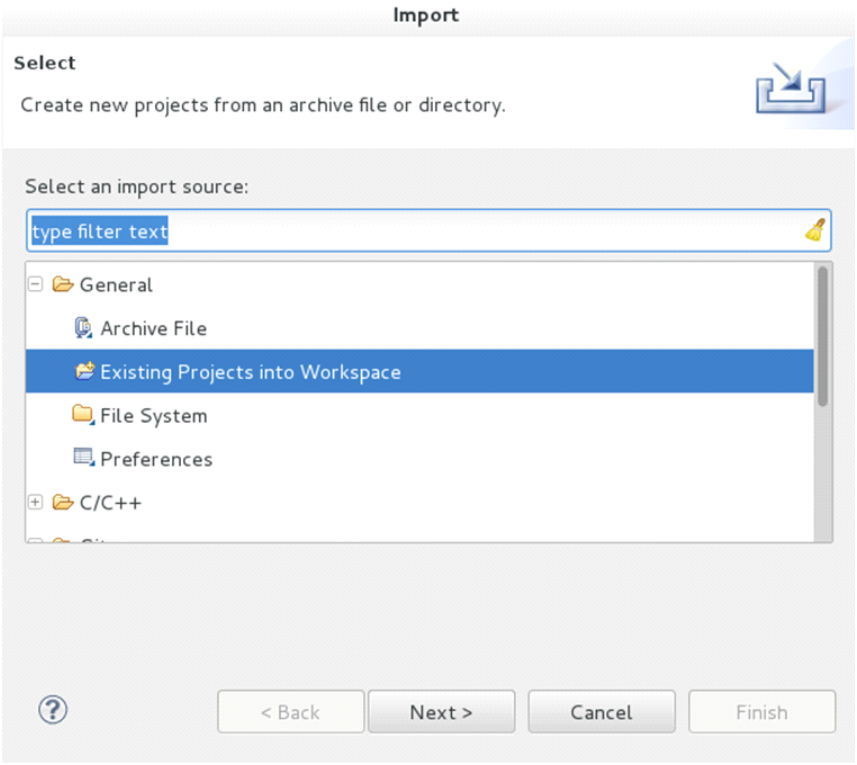

Figure 7. Importing Existing SDK Projects into Workspace

Press the **Select root directory** radio button and left-click the **Browse...** button. In the file explorer dialog that opens, navigate to the installed “Innovative” folder, then into the “Malibu” subfolder there, select the “XSDK” subfolder inside, and click the **OK** button. On a Windows host, the selection appears as below:

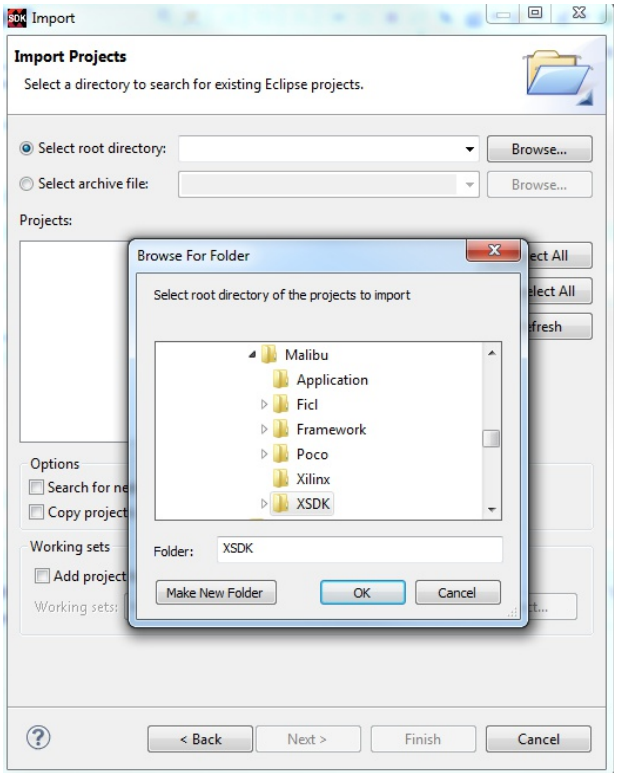

Figure 8. Selecting an Existing Windows SDK Project Folder

On a Linux host, the selection appears as below:

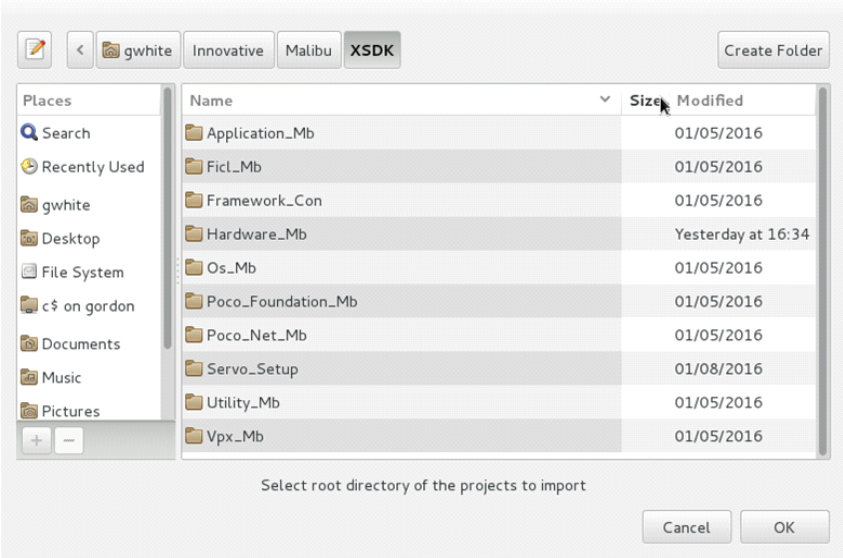

Figure 9. Selecting an Existing Linux SDK Project Folder

In either case, once the "XSDK" subfolder has been selected as shown, press OK. Back in the Import wizard, note that the presence of a "RemoteSystemsTempFiles" folder may trigger the informational message, "Some projects cannot be imported because they already exist in the workspace", but all other projects should be checked automatically. Click the **Finish** button.

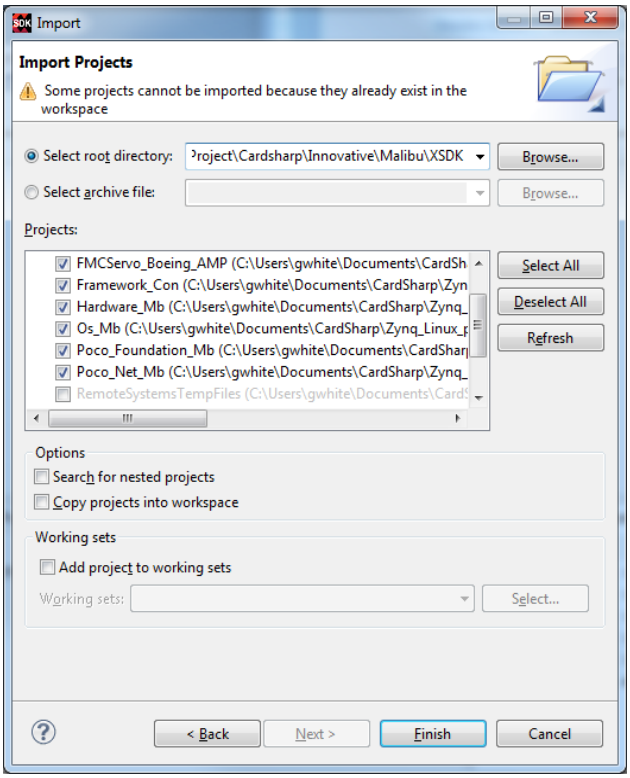

Figure 10. Selecting Existing SDK Projects

Back in the workspace, the Project Explorer view now should show the imported projects:

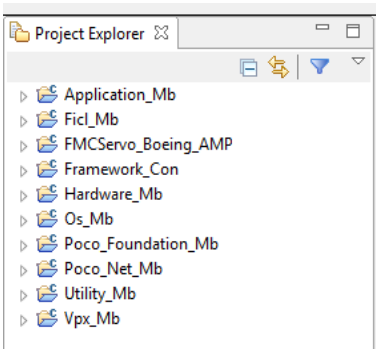

Figure 11. SDK Project Explorer View

Xilinx SDK Standalone AMP Projects

This is a procedure to install Xilinx SDK standalone asymmetric multi-processing (AMP) application projects on a development host PC running either 64-bit Windows or Linux. This procedure presumes that the Xilinx Software Development Kit (SDK) has been installed on the host.

Importing Standalone AMP Projects

The “app_cpu1_sdk_project_export.zip” file in the installed “FMCServo_AMP” example is a Xilinx SDK project archive into which a hardware platform specification, a board support package, and the standalone servo application have been exported. These support development of the standalone AMP application for CPU1 of the Zynq PS compatible with PetaLinux running on CPU0.

Select the **File** menu from the workbench top menu bar and then select **Import** from the File dropdown menu. In the Import wizard, open the **General** list, left-click **Existing Projects into Workspace** and then click the **Next** button.

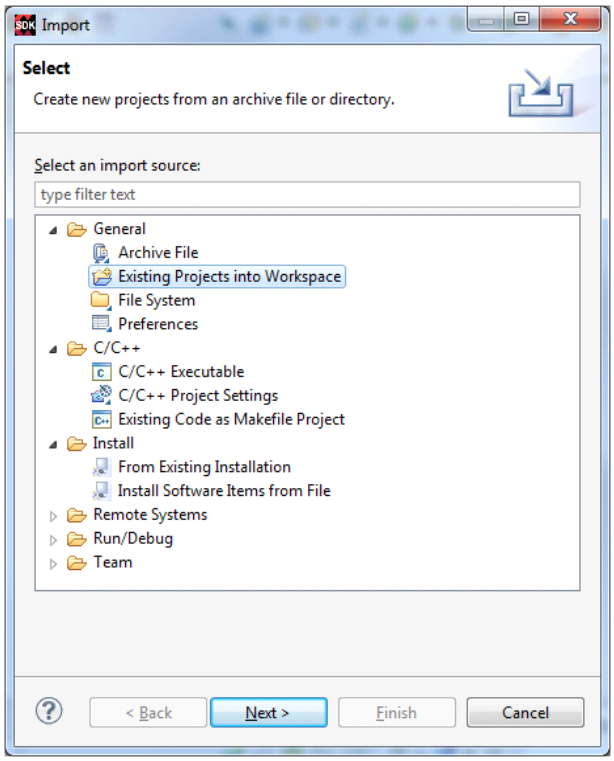

Figure 12. Importing Standalone SDK Projects into Workspace

Select the radio button for **Select archive file**. Browse to, select, and open "app_cpu1_sdk_project_export.zip". If any of the projects isn't checked on, click the **Select All** button. Click **Finish**.

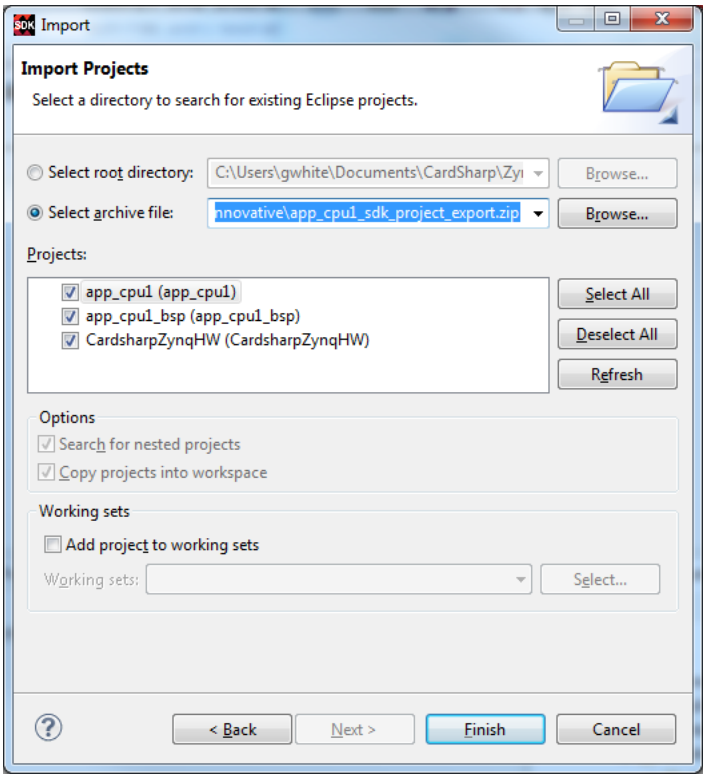

Figure 13. Selecting Standalone SDK Projects

The three projects should now be restored in Project Explorer, as shown:

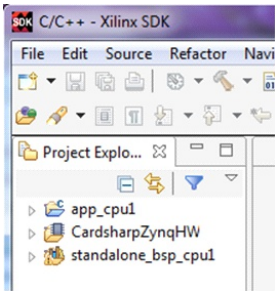

Figure 14. SDK Project Explorer View of Standalone Projects

When done, messages in the SDK console window should show that the “app_cpu1” application was built successfully.

References

http://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf
http://www.wiki.xilinx.com/XAPP1079+Latest+Information
http://www.xilinx.com/support/documentation/application_notes/xapp1078-amp-linux-bare-metal.pdf

Chapter 3: Detailed Host-Target Communication

Ethernet Communication

With the Cardsharp board powered off, ensure that the boot mode of the board has been set to QSPI boot by setting SW1 away from the near edge of the board, as shown:

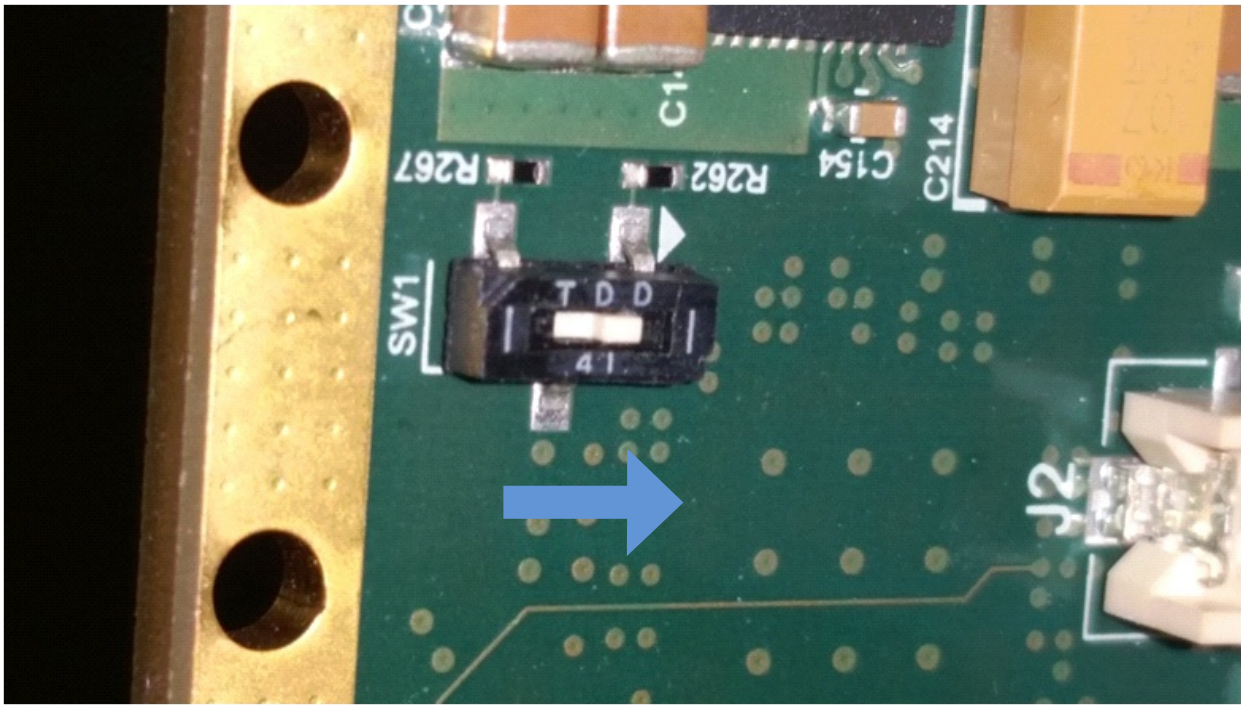

Figure 15. QSPI Boot Switch Setting

Power-on the board. After the board successfully boots PetaLinux, establish an Ethernet connection with the target Cardsharp using a SSH or Telnet client program (e.g., “PuTTY”, available from http://www.chiark.greenend.org.uk/~sgtatham/putty/) configured to connect with address 192.168.0.10. A PuTTY configuration screen is shown below:

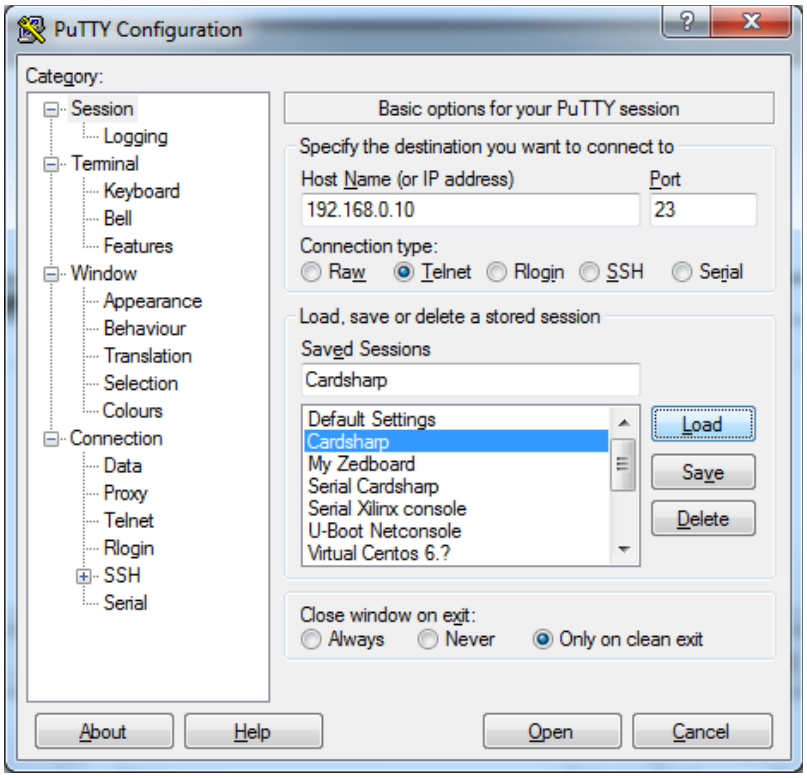

Figure 16. Putty Ethernet Configuration for Cardsharp

Then, enter user name “root” and password “root” to log into the PetaLinux system.

System Logs

Linux kernel serial console logging is disabled because it prevents booting if no UART terminal is connected. Instead, a UDP network console is enabled by a boot argument (see https://www.kernel.org/doc/Documentation/networking/netconsole.txt). The boot argument configures PetaLinux to log to port 6666 at 192.168.0.2. Besides the Linux programs mentioned in that document, on Windows the Cygwin (https://www.cygwin.com/) nc6 package can be used, with the following command line:

```
nc6 -u -l -p 6666 192.168.0.10
```

The U-boot network console has not been enabled because it prevents booting unless an Ethernet connection to the specified IP address is present. After logging in to PetaLinux, a log of device driver output can be viewed with the “dmesg” command. The startup process logs userspace messages to the “/var/log/boot” file.

File Transfers

File transfers between a development host and the Cardsharp are enabled by a SFTP server running on PetaLinux. FTP file transfers are limited to the /var/ftp directory on Cardsharp while SFTP can access the entire filesystem. “WinSCP” (available from http://winscp.net/) is an open source free SFTP/FTP/SCP client for Windows that provides file transfer capability, basic file manager functionality, and scripting. A WinSCP configuration screen is shown below:

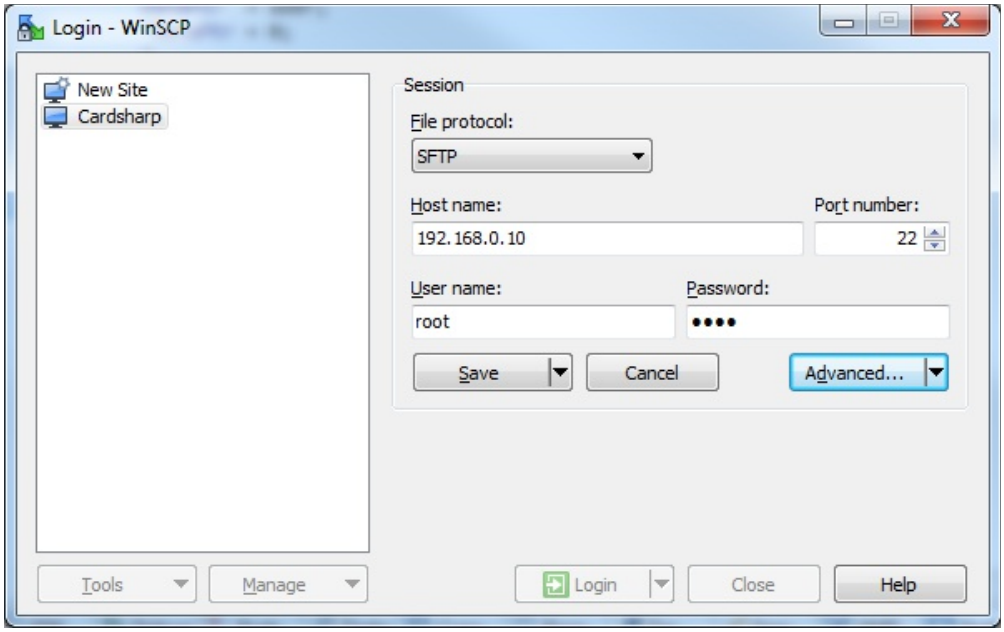

Figure 17. WinSCP Configuration for Cardsharp

User Interface Protocols

Cardsharp PetaLinux has been configured with the Ncurses library (see, e.g., http://invisible-island.net/ncurses/) and with the Busybox httpd server (see https://wiki.openwrt.org/doc/howto/http.httpd). The following figure illustrates proper PuTTY configuration for viewing Ncurses applications:

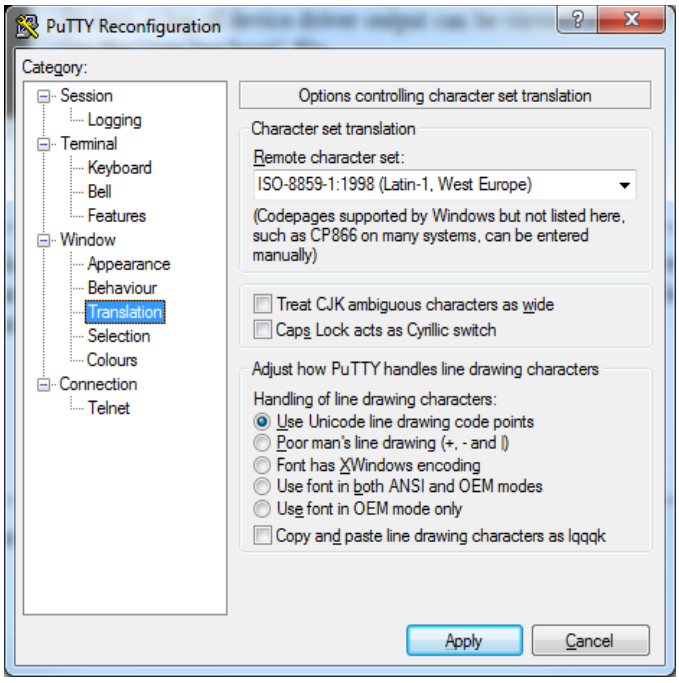

Figure 18. PuTTY Configuration for Ncurses

JTAG Communication

With the Cardsharp powered off, connect a “Xilinx Platform Cable USB II” ” (http://www.xilinx.com/products/boards-and-kits/hw-usb-ii-g.html or http://products.avnet.com/shop/en/ema/development-tools/3074457345629474182) or the compatible “Digilent XUP-USB” (https://www.digilentinc.com/Products/Detail.cfm?Prod=XUP-USB-JTAG) to the Cardsharp using the provided cable and to a USB port of the host development PC. The necessary drivers were installed during the Xilinx tool installation. When downloading standalone software to Cardsharp DDR-RAM or on-chip memory (OCM) via JTAG and it is desired to prevent PetaLinux from booting, set the boot mode of the board to JTAG boot by setting SW1 toward the near edge of the board, as shown:

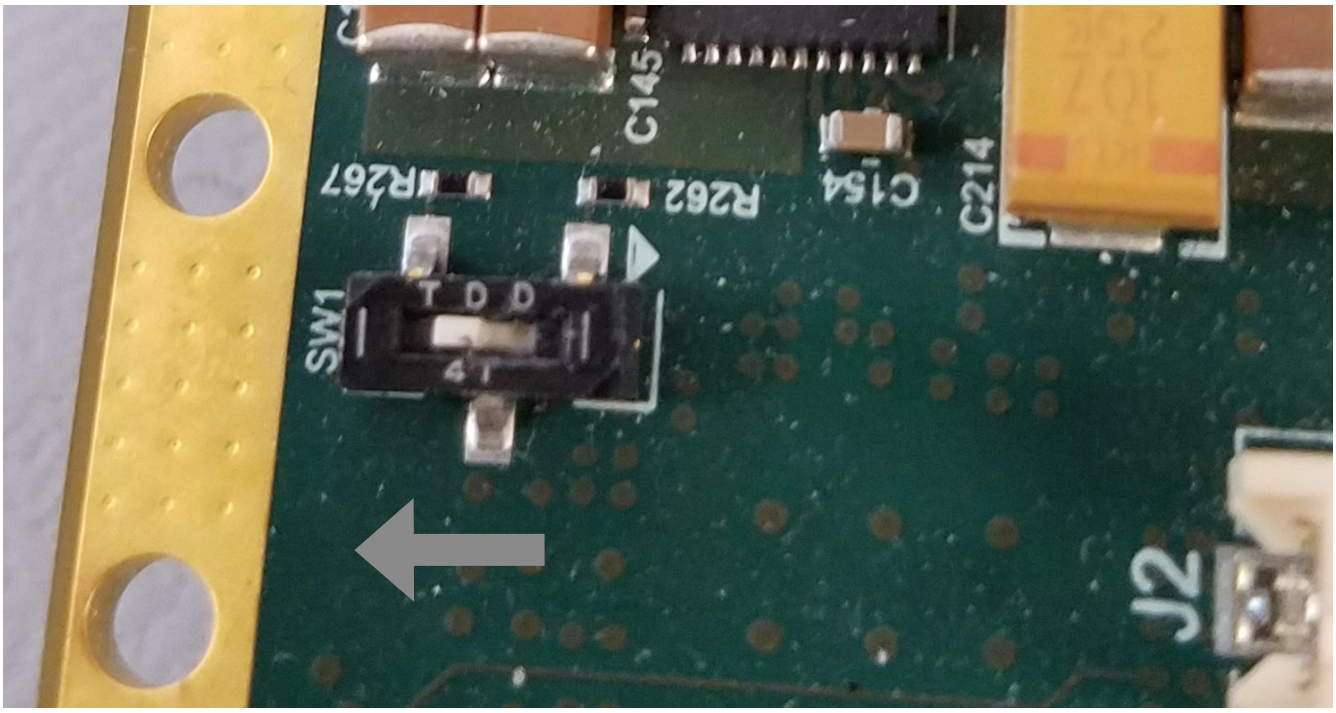

Figure 19. JTAG Boot Switch Setting

Chapter 4: An Example Debugging Session

Debugging PetaLinux Applications

This is a procedure to debug a PetaLinux application developed using the Xilinx Software Development Kit (SDK) on a development host PC running either 64-bit Windows or Linux. It is presumed that a Xilinx SDK project has been created and that a Debug configuration of the project software had been built successfully. An Innovative Integration Cardsharp target system running Xilinx PetaLinux with an Ethernet connection to the host PC is needed.

Downloading a Bitstream

If the application requires a PL bitstream to be downloaded, this can be done from the PetaLinux command line as so:

```
cat your_PL_filename.bit > /dev/xdevcfg
```

Afterwards, the “prog_done” file should indicate that the programming was successful:

```
cat /sys/devices/soc0/amba/f8007000.devcfg/prog_done
1
```

Configuring the Debug Profile

See the “Example Design: Debugging the Linux Application Using SDK” section of chapter 6, “Linux Booting and Debug in SDK” of the Xilinx UG1165 manual, Zynq-7000 All Programmable SoC: Embedded Design Tutorial (step 5, page 78 of the v2015.4, November 18, 2015, edition). See also “Linux Application Debugging with System Debugger” in the Xilinx SDK Help (UG782), http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/SDK_Doc/SDK_tasks/sdk_t_linux_application_debugging_system_debugger.html .

For the target connection's “Host” address, always use `192.168.0.10` . When the debug configuration is complete, press the **Debug** button.

Debug Perspective

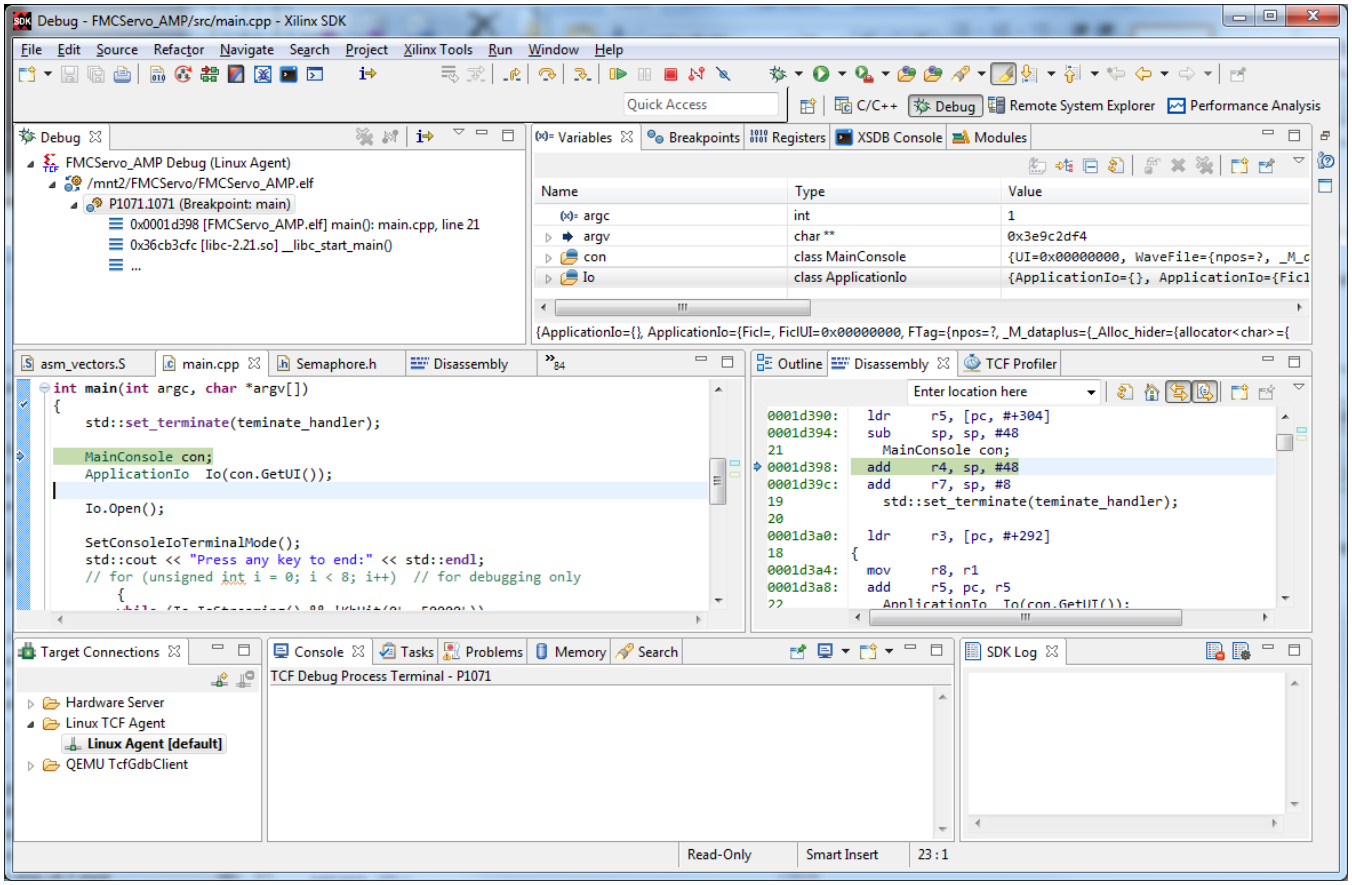

Figure 20. Debug Perspective

For help on the views in the Debug perspective, select **Search** from the SDK **Help** menu and search for “debug views”. Alternately, place the mouse cursor in any view by left-clicking and then press the **F1** key. When using a SDK hotkey, ensure that the cursor is in the appropriate view for that hotkey. For example, **F5** in the Debug view will by default single-step into a function while, in a source code window, **F5** will refresh the file view.

Debugging Standalone Applications

This is a procedure to debug a standalone “bare-metal” application developed using the Xilinx Software Development Kit (SDK) on a development host PC running either 64-bit Windows or Linux. It is presumed that a Xilinx SDK project has been

created and that a Debug configuration of the project software had been built successfully. An Innovative Integration Cardsharp target system with a JTAG connection to the host PC is needed.

Downloading a Bitstream

If the application requires a PL bitstream to be downloaded, this can be done from the SDK by selecting the **Xilinx Tools** menu from the workbench top menu bar and then selecting **Program FPGA** from the Xilinx Tools dropdown menu. In the Program FPGA dialog, either select the appropriate Hardware Platform from the dropdown list, press the **Search...** button, select the desired file from the popup list dialog, and press **OK,** or press the **Browse..** button, navigate to any directory containing a desired bitstream file, select the file, and press **Open**. Then, upon returning to the Program FPGA dialog, press the **Program** button.

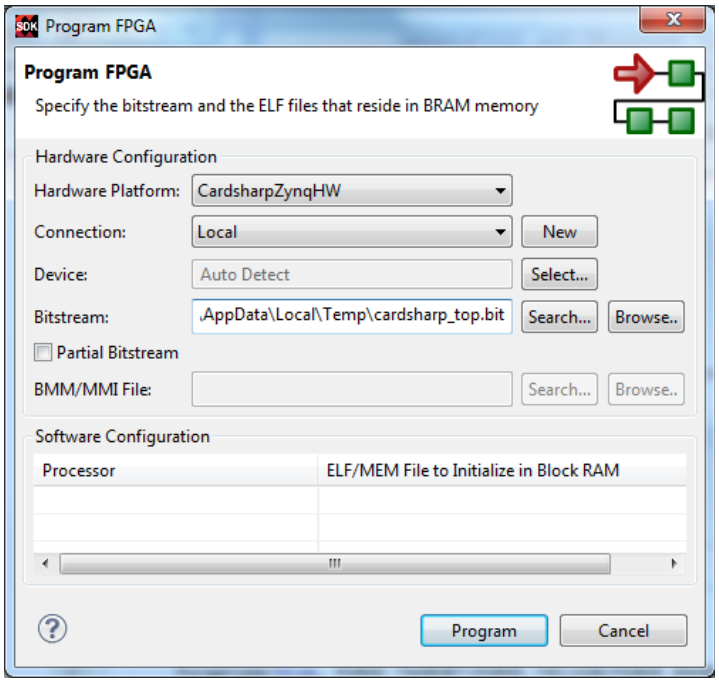

Figure 21. Program FPGA Dialog

Configuring the Debug Profile

See “Launch Configurations” under “Working with Xilinx System Debugger” in the Xilinx SDK Help, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/SDK_Doc/SDK_concepts/sdk_c_debug_run_configuration.html.

If the standalone application will be running in AMP mode, i.e., PetaLinux will be running on the other ARM CPU, then the **Reset entire system**, **Program FPGA**, **Run ps7_init**, **Run ps7_post_config**, and **Enable Cross-Triggering** check boxes of the debug configuration's **Target Setup** tab must be cleared. However, the **Reset processor** check box for the application's CPU on the **Application** tab should be checked. Also, before the standalone application can be loaded and run by the

debugger, PetaLinux must vacate CPU1, so the “zynq_remoteproc” device driver must be installed first (see usage procedure for zynq_remoteproc).

Profiling Applications

See chapter 8, “Software Profiling Using SDK” of the Xilinx UG1165 manual, Zynq-7000 All Programmable SoC: Embedded Design Tutorial. See also “TCF Profiling”, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/SDK_Doc/SDK_tasks/sdk_t_tcf_profiling_with_tcf_debugger.html , and “Profiling Linux Applications with System Debugger”, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/SDK_Doc/SDK_tasks/sdk_profiling_linux_apps_with_sysdbg.html, in the Xilinx SDK Help.

Either PetaLinux or standalone applications can be profiled. All caveats regarding debugging standalone AMP applications also apply to profiling them.

Chapter 5: Servo Mode

Cardsharp applications can be broadly divided into two categories: data acquisition/playback vs real-time processing applications. These two types of applications require very different methods of behavior by the system that cannot easily be reconciled into a single ‘one size fits all’ system. Yet the Malibu library needs to support both styles of program in a simple and natural way.

Data Acquisition Applications.
Acquisition applications need to acquire large quantities of data, but do not need to process the data immediately. In order to support higher rates, these applications use large buffers, and large chains of these buffers to allow data to be moved from the hardware into memory without CPU intervention. Thus, data can be acquired at high rates, but the system only needs to be involved at relatively rare intervals to process a block of data.

These applications are very natural for bus-mastering, and often this form of DMA is used to move this data into host memory. In addition, these buffers and the FIFOs in the hardware allow some ‘slack’ in the system so that the application can fall behind for a short time if it is busy performing some other service. After the completion of this task the system can process the buffers in the queues and in the hardware FIFOs and catch up. As long as enough slack is built into the system, no data loss will result.

Servo Applications.
In a Servo application, the requirements are polar opposites from the Data Acquisition application. In this case, each event needs to be processed at once, with no delay. This data is analyzed to produce an output update event that is output to the DACs in the system at once. It is therefore vital that data is read into the system and processed without any buffering. Hardware FIFOs are only useful at the single event level; if you fall behind by even a single event your servo is failing.

Since the amount of data moved from the hardware is so small, bus mastering or DMA is much less useful in this case than it is for the Data Acquisition case.

Servoing on Cardsharp
For Data Acquisition applications, Malibu on PetaLinux uses interrupts to provide a simple means of notifying an application to process buffers of data to and from analog hardware. This driver model is used for all of the streamed hardware provided on the baseboard. Despite the strength of this model for the continuously-streaming applications, it fails to properly servo in any efficient manner. Closed-loop servo control software typically require real-time processing of ADC samples and generate feedback signals thru the DAC. The servo application example utilizing the Malibu library is intended to demonstrate the consistent, low latency responses required by a control loop system.

Servo Timing

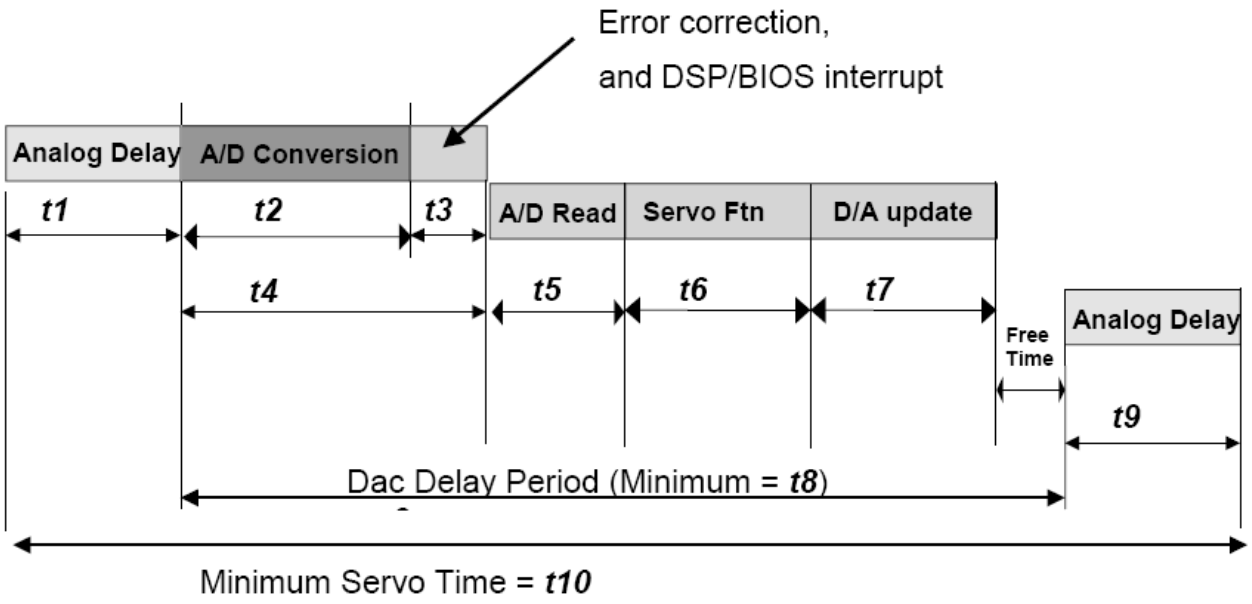

Table 3. Servo Time Measurements

Interval	Purpose	Typical FMC-Servo Timing (PetaLinux)
t1	Adc Analog Delay	8 µs
t2	Adc Conversion	4 µs
t3	Logic + Poll/Detect	2 µs
t4	*t2* + *t3*	6 µs
t5	A/D Read Time	4.8 µs (8 pairs) 2.3 µs (4 pairs)
t6	Servo Calculation Time	~24 µs
t7	D/A Write and Update	2.4 µs (8 pairs) 1.2 µs (4 pairs)
t8	Minimum Dac Delay Time	~10-13 µs.
t9	Dac Analog Delay	2.5 µs.
t10	Total Servo Time	~46 µs.

The analog delay inserts a fixed skew between input and output and does not affect the update rate directly.

Servo Example

The Servo application consists of a Malibu application on PetaLinux and a separate standalone application that runs on the second ARM processor of the Xilinx Zynq. The Malibu application sets up the hardware, reports status, and saves logged data upon program completion. Aided by firmware logic, the standalone application reads the A/D converters, performs the real-time servo calculations, and writes the results of those to the D/A converters.

Tools and Platform Requirements

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

Processor	Development Environment	Innovative Tool set	Project Directory
Host PC (Windows or Linux)	Xilinx Software Development Kit (SDK)	Cardsharp Malibu	Examples\Servo

Table 4. Development Tools

The Malibu library is source code compatible with the both environments listed in Table 4. .

Malibu Application Program Layout and Design

The example Malibu application is located in Examples\FMCServo_AMP. Table 5. list the source files.

Table 5. Malibu Application Source Files.

Examples\FMCServo_AMP\PetaLinux
main.cpp
ApplicationIo.h, .cpp
ConsoleIo.h, .cpp
MainConsole.h, .cpp
ModuleIo.h, .cpp
RemoteAppControl.h, .cpp
RemoteInterfaceImpl.h
QueueT.h
Semaphore.h

The application begin its execution with the main() function of main.cpp. Custom console functions can be added to MainConsole.h and .cpp. ConsoleIO.h and .cpp implement non-blocking terminal input. Application specific I/O functions are defined in ApplicationIo.h and .cpp. All other hardware and firmware specific module interface code is defined in ModuleIo.h and .cpp. RemoteAppControl.h, .cpp, RemoteInterfaceImpl.h, QueueT.h, and Semaphore.h interface to the standalone application through a shared uncached area of the Zynq's On-Chip Memory (OCM).

The Examples\FMCServo_AMP\XSDK subdirectory contains the Xilinx XSDK Servo project files for the Malibu application.

Standalone Application Program Layout and Design

The standalone application source files are listed in Table 6. .

Table 6. Standalone Application Source Files

Examples\FMCServo_AMP\app_cpu1_sdk_project_export.zip
main.cpp
ResourceTable.c
QueueT.h
RemoteInterfaceImpl.h
Semaphore.h
ServoTask.h, .cpp
Support.h, .cpp
ProcessPacket.cpp
lscript.ld

The application begin its execution with the main() function of main.cpp. ResourceTable.c contains information needed by PetaLinux to load the app into memory for execution by the standalone CPU. RemoteInterfaceImpl.h, QueueT.h, and Semaphore.h interface to the Malibu application on PetaLinux through the shared area of OCM. Firmware interface code is defined in ServoTask.h, .cpp, and Support.h. The lscript.ld file is the linker script, usually managed by the Xilinx SDK. The ProcessPacket.cpp file must be updated with code implementing the servo algorithm.

Usage Procedure

To program the Zynq PL with the servo firmware, enter a command like the following, referencing the binary bitstream file's name:

```
root@Cardsharp2015:~# cat /mnt/cs_servo_top.bit.bin > /dev/xdevcfg
```

The Petalinux “lscpu” command initially states: “On-line CPU(s) list: 0,1”. Executing the following command will install /lib/firmware/app_cpu1 by default:

```
root@Cardsharp2015:~# modprobe zynq_remoteproc
```

Currently, the required starting address (0x00200000) and the maximum length (0x00100000 bytes) of app_cpu1 are hard-coded in zynq_remoteproc. To see feedback that the modprobe command was successful, enter the following command:

```
root@Cardsharp2015:~# dmesg | tail
```

The result should include these messages (with perhaps a different image size):

```
Probing II zynq_remoteproc_driver
CPU1: shutdown
 remoteproc0: 200000.remoteproc-II is available
 remoteproc0: Note: remoteproc is still under development and considered experimental.
 remoteproc0: THE BINARY FORMAT IS NOT YET FINALIZED, and backward compatibility isn't yet
guaranteed.
 remoteproc0: firmware_loading_complete
 remoteproc0: powering up 200000.remoteproc-II
 remoteproc0: Booting fw image app_cpu1, size 364275
 remoteproc0: remote processor 200000.remoteproc-II is now up
 remoteproc0: stopped remote processor 200000.remoteproc-II
```

The insertion of the zynq_remoteproc module may be verified with the PetaLinux command “lsmod”. At this point, the “lscpu” command reports, “On-line CPU(s) list: 0” and “Off-line CPU(s) list: 1”, and the standalone application is running on CPU 1.

A Settings.ini file should be prepared with the desired settings for the PetaLinux servo application, FMCServo_Boeing_AMP.elf, and placed in the user's current directory. If both files are present in the same directory, /mnt/Servo, for example, then enter the following commands to execute the servo application:

```
root@Cardsharp2015:~# cd /mnt/Servo
root@Cardsharp2015:/mnt/Servo# ./FMCServo_Boeing_AMP.elf
```

For Rx.SampleRate=0.006 and Tx.SampleRate=0.006 (6 KHz.), typical output begins:

```
 Log ==> Module Device Opened Successfully...
 Log ==> Logic Version: 5, Hdw Variant: 0, Revision: 0, Subrevision: 0
 Log ==> Board Family: 0, Type: 3, Board Revision: 0, Chip: 0
 Log ==> Requested PLL Frequencies: [ADC] 6000 [DAC] 6000
 Log ==> Actual PLL Frequencies: [ADC] 6000 [DAC] 6000
 Log ==> Servo processor started
 Log ==> Analog I/O started
 Log ==> Stream Mode started
 Log ==> Duo "Clock" output data
 Log ==> Out # 0:  Divider: 0, Out Freq: 6000, Actual: 6000
Sample Count: 0
Rate (KSPS): 0
PLL Locked: +
 Log ==> Out # 1:  Divider: 0, Out Freq: 6000, Actual: 6000
 Log ==> Out # 2:  Divider: 0, Out Freq: 6000, Actual: 6000
 Log ==> Out # 3:  Divider: 0, Out Freq: 6000, Actual: 6000
 Log ==> Ad9510 PLL output data
 Log ==> Out # 0:  Divider: 0, Out Freq: 0, Actual: 0
 Log ==> Out # 1:  Divider: 0, Out Freq: 0, Actual: 0
 Log ==> Out # 2:  Divider: 0, Out Freq: 0, Actual: 0
 Log ==> Out # 3:  Divider: 0, Out Freq: 0, Actual: 0
 Log ==> Out # 4:  Divider: 0, Out Freq: 0, Actual: 0
 Log ==> Out # 5:  Divider: 20, Out Freq: 6000, Actual: 6.12e+06
 Log ==> Out # 6:  Divider: 0, Out Freq: 0, Actual: 0
 Log ==> Out # 7:  Divider: 0, Out Freq: 0, Actual: 0
```

```
 Log ==> Ad9508 Divider output data
 Log ==> Out # 0:  Divider: 1020, Out Freq: 6000, Actual: 6000
 Log ==> Out # 1:  Divider: 1020, Out Freq: 6000, Actual: 6000
 Log ==> Out # 2:  Divider: 1020, Out Freq: 6000, Actual: 6000
 Log ==> Out # 3:  Divider: 1020, Out Freq: 6000, Actual: 6000
Press any key to end:
Sample Count: 48008
Rate (KSPS): 0
PLL Locked: +
Sample Count: 96080
Rate (KSPS): 0
PLL Locked: +
Sample Count: 144160
Rate (KSPS): 0
PLL Locked: +
Sample Count: 192240
Rate (KSPS): 0
PLL Locked: +
Sample Count: 240320
Rate (KSPS): 39.9884
PLL Locked: +
Sample Count: 288400
Rate (KSPS): 47.9884
PLL Locked: +
Sample Count: 336480
Rate (KSPS): 47.9993
PLL Locked: +
```

Sampling status obtained from the standalone application will continue to be reported each second. Servo processing, status reports, and the execution of the PetaLinux servo application will end when any keystroke is sent to the Cardsharp.

To remove the standalone servo application and the zynq_remoteproc module, send the following command to the target board:

```
root@Cardsharp2015:~# modprobe -r virtio_rpmsg_bus
root@Cardsharp2015:~# modprobe -r zynq_remoteproc
```

Confirmation of the removal can be seen with the “dmesg | tail” command line:

```
zynq_remoteproc 200000.remoteproc-II: zynq_remoteproc_remove
zynq_remoteproc 200000.remoteproc-II: Deleting the irq_list
 remoteproc0: releasing 200000.remoteproc-II
```

The removal of the zynq_remoteproc module may be verified with the PetaLinux lsmod command. The lscpu command should once again indicate that both CPUs are available to PetaLinux.

If desired, a different standalone application could be copied to /lib/firmware on the target, replacing the app_cpu1 file there. This could be accomplished via FTP or SFTP over Ethernet, using a Windows application such as WinSCP, for instance. Then, the new application could be loaded and run by reinserting the zynq_remoteproc module (i.e., “modprobe

zynq_remoteproc"). The procedure for inserting and removing that module can be repeated as often as needed to change the applications running on CPU1.

The zynq_remoteproc module accepts an optional "firmware" parameter, specifying a path to a standalone application file relative to /lib/firmware. For instance, to specify a file named "adaptive.elf" in the /lib/firmware/servo directory, enter this command:

root@Cardsharp2015:~# modprobe zynq_remoteproc firmware="servo/adaptive.elf"

Configuration File

The "Settings.ini" file in the current directory when the application is started holds parameters for data acquisition including the on-board timebase, analog channel selection, range and triggering, etc. Each of these parameters is described below. Each line of the file must set a parameter by following one of the quoted names with an equals sign (=) and a string or numeric value, as appropriate.

Clock Group. The module features an on-board AD9510 PLL which may be used as a sample clock during analog acquisition. Alternately, an external sample clock may be used. If "SampleClockSource" is 0, the external clock is used as the analog sample clock; if it is 1, the internal PLL is selected and is programmed to generate the sample rate of "Rx.SampleRate" or (Tx) "SampleRate" MHz during acquisition. If an external clock source is selected, then "Rx.SampleRate" or "SampleRate" is used to inform the program of your intended (external) sample rate. In that case, you are expected to supply a clock running at the specified rate to the external clock input connector on the module. "ExtClockSrcSelection" selects an external clock from either the FMC Servo front panel (0) or the GPS 10 MHz reference clock on Cardsharp (1).

```
void ApplicationIo::StreamPreconfigure()
{
…
    //   Route ext clock source
    IX6ClockIo::IIClockSelect cks[] = { IX6ClockIo::cslFrontPanel, IX6ClockIo::cslCarrier };
       Module.Fmc().Clock().ExternalClkSelect(cks[Settings.ExtClockSrcSelection]);

    if(cks[Settings.ExtClockSrcSelection] == IX6ClockIo::cslCarrier)
        {
        Module.Fmc().FmcInfo().FmcClock2_3_Enable(true);
        Module.Fmc().FmcInfo().FmcClock2_IsSingleEnded(true);
        Module.Fmc().FmcInfo().FmcClock3_IsSingleEnded(true);
        }
    else
        {
        Module.Fmc().FmcInfo().FmcClock2_3_Enable(false);
        Module.Fmc().FmcInfo().FmcClock2_IsSingleEnded(true);
        Module.Fmc().FmcInfo().FmcClock3_IsSingleEnded(true);
        }

…
    //   Route clock
    IX6ClockIo::IIClockSource src[] = { IX6ClockIo::csExternal, IX6ClockIo::csInternal };
    Module.Fmc().Clock().Source(src[Settings.SampleClockSource]);
    Module.Fmc().Clock().Adc().Frequency(Settings.Rx.SampleRate * 1e6);
    Module.Fmc().Clock().Dac().Frequency(Settings.Tx.SampleRate * 1e6);
    {
    std::stringstream msg;
    msg << "Requested PLL Frequencies: [ADC] " << Settings.Rx.SampleRate * 1e6
            << " [DAC] " << Settings.Tx.SampleRate * 1e6;
    Log(msg.str());
    }
```

Active Channels Group. The FMC module supports simultaneous acquisition up to the maximum number of channels. Each Rx channel 0-7 is enabled if the corresponding entry "Ch*N"* in the "Rx.ActiveChannels" group is nonzero. The number of enabled Rx channels must be even, and the value of "Poller.AdcAF" must be set to half that number. Similarly, each Tx channel 0-7 is enabled if the corresponding entry "Ch*N"* in the "ActiveChannels" group is nonzero, and the number of enabled Tx channels must be even.

```
void ApplicationIo::StreamPreconfigure()
{
…
    //
    //  Set Channel Enables
    Module.Fmc().Output().ChannelDisableAll();
    for (unsigned int i = 0; i < Module.Fmc().Output().Channels(); ++i)
        {
        bool active = Settings.Tx.ActiveChannels[i] ? true : false;
        if (active==true)
            Module.Fmc().Output().ChannelEnabled(i, true);
        }
    //  Channel Enables
    Module.Fmc().Input().ChannelDisableAll();
    for (unsigned int i = 0; i < Module.Fmc().Input().Channels(); ++i)
        {
        bool active = Settings.Rx.ActiveChannels[i] ? true : false;
        if (active==true)
            Module.Fmc().Input().ChannelEnabled(i, true);
        }
…
}
…
bool ApplicationIo::StartStreaming()
{
…
    Module.Fmc().AdcFifo()->AlmostFullLevel(Settings.Poller.AdcAF);
```

Trigger Group. Acquisition may be triggered using an external signal or via software. "Rx.ExternalTrigger" and (Tx) "ExternalTrigger" provide the means of selection, hardware if nonzero, software otherwise. Triggers act as a gate on data flow - no data flows until a trigger has been received. If software, the application program issues a command following configuration to initiate data flow. If hardware, a signal applied to the external trigger connector controls data flow. In either case, the initiation of triggering is delayed by "Trigger Delay" seconds. "ExtTriggerSrcSelection" selects either the front panel (0) or FMC (1) external trigger source.

In `Unframed` mode ("Rx.Framed" or "Tx.Framed" is 0), triggers are either rising edge (Rx.EdgeTrigger or EdgeTrigger is 1) or level sensitive (Rx.EdgeTrigger or EdgeTrigger is 0). In the latter case, 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 ideal for conventional data acquisition applications. In `Framed` mode ("Rx.Framed" or "Tx.Framed" is 1), triggers are always rising edge sensitive. Upon detection of each edge, "Rx.FrameSize"/"Tx.FrameSize" samples are acquired from all active channels, then acquisition terminates until the next trigger edge is detected.

```
bool ApplicationIo::StartStreaming()
{
…
    if (Settings.Tx.Framed)
        {
        // Granularity is firmware limitation
        int framesize = Module.Fmc().Output().Info().TriggerFrameGranularity();

        if (Settings.Tx.FrameSize % framesize)
```

```
            {
            std::stringstream msg;
            msg << "Error: Ouput frame count must be a multiple of " << framesize;
            Log(msg.str());

            return false;
            }
        }
    if (Settings.Rx.Framed)
        {
        // Granularity is firmware limitation
        int framesize = Module.Fmc().Input().Info().TriggerFrameGranularity();

        if (Settings.Rx.FrameSize % framesize)
            {
            std::stringstream msg;
            msg << "Error: Input frame count must be a multiple of " << framesize;
            Log(msg.str());

            return false;
            }
        }

…
    Trig.DelayedTriggerPeriod(Settings.Tx.TriggerDelayPeriod);
    Trig.ExternalTrigger(Settings.Rx.ExternalTrigger ? true : false ||
            Settings.Tx.ExternalTrigger ? true : false);
…
    //
    // Trigger Configuration
    //  Frame Triggering
    Module.Fmc().Output().Trigger().FramedMode((Settings.Tx.Framed)? true : false);
    Module.Fmc().Output().Trigger().Edge((Settings.Tx.EdgeTrigger)? true : false);
    Module.Fmc().Output().Trigger().FrameSize(Settings.Tx.FrameSize);

    Module.Fmc().Input().Trigger().FramedMode((Settings.Rx.Framed)? true : false);
    Module.Fmc().Input().Trigger().Edge((Settings.Rx.EdgeTrigger)? true : false);
    Module.Fmc().Input().Trigger().FrameSize(Settings.Rx.FrameSize);

    //  Route External Trigger source
    IFmcIoDevice::AfeExtSyncOptions syncsel[] = {IFmcIoDevice::essFrontPanel,
       IFmcIoDevice::essP16};
    Module.Fmc().Output().Trigger().ExternalSyncSource(syncsel[Settings.ExtTriggerSrcSelection]);
    Module.Fmc().Input().Trigger().ExternalSyncSource(syncsel[Settings.ExtTriggerSrcSelection]);

    // if clock is carrier, settings are already done in preconfigure
    IFmcClockIo::IIClockSelect cks[] = { IFmcClockIo::cslFrontPanel, IFmcClockIo::cslCarrier };
    if(cks[Settings.ExtClockSrcSelection] != IFmcClockIo::cslCarrier)
        {
        if(syncsel[Settings.ExtTriggerSrcSelection] == IFmcIoDevice::essP16)
            {
            Module.Fmc().FmcInfo().FmcClock2_3_Enable(true);
            Module.Fmc().FmcInfo().FmcClock2_IsSingleEnded(true);
            Module.Fmc().FmcInfo().FmcClock3_IsSingleEnded(true);
            }
        else
            {
            Module.Fmc().FmcInfo().FmcClock2_3_Enable(false);
            Module.Fmc().FmcInfo().FmcClock2_IsSingleEnded(true);
            Module.Fmc().FmcInfo().FmcClock3_IsSingleEnded(true);
            }
        }
```

Data Logging Group. These controls govern the size of data files created by the application containing packet data received from the module during real-time streaming. The value of "Rx.SamplesToLog" sets the upper-bound on the number of stored events (total samples from all channels). If the "Rx.AutoStop" is set to 1, streaming will automatically terminate once the specified number of events have been processed.

```
bool ApplicationIo::StartStreaming()
{
…
    int  SamplesPerWord = Module.Fmc().Input().Info().SamplesPerWord();
    WordsToLog = Settings.Rx.SamplesToLog / SamplesPerWord;
…
bool  ApplicationIo::IsDataLoggingCompleted()
{
    if (!WordsToLog)
        return false;
    else
        return SampleCount >= WordsToLog;
}
…
void  ApplicationIo::CheckDataLimit()
{
    //
    //
    if (Settings.Rx.AutoStop && IsDataLoggingCompleted() && !Stopped)
        {
        // Stop counter and display it
        double elapsed = RunTimeSW.Stop();

        StopStreaming();
        Log("Stream Mode stopped automatically.");
        Log(std::string("Elapsed (sec.): ") + FloatToString(elapsed));
        }
}
```

Test Counter Group. Use these control to enable a logic-specific test mode if you are developing custom FPGA logic. If you are using the stock factory-supplied logic, a nonzero value for "Rx.TestGenEnable" enables test data to replace A/D data from each channel. In that case, "Rx.TestGenMode" selects the test mode, unpaced sawtooth (0) or paced sawtooth (1). A nonzero value for "TestGenEnable" enables test data to replace D/A data for each channel. In that case, "TestGenMode" selects the test mode, ramp (0), sine wave (1), DAC test pattern (2), zeros (3), max positive (4), max negative (5), alternating 1's and 0's (6), or alternating two 1's and two 0's (7).

```
bool ApplicationIo::StartStreaming()
{
…
    //  Output Test Generator Setup
    Module.SetOutputTestConfiguration( Settings.Tx.TestGenEnable, Settings.Tx.TestGenMode );
    Module.SetInputTestConfiguration( Settings.Rx.TestGenEnable, Settings.Rx.TestGenMode );
```

Decimation Group. These controls govern the behavior enable the decimation logic. When "Rx.DecimationEnable"is nonzero, only one of every "Rx.DecimationFactor" samples of each Rx channel's acquired data is retained within the internal ADC FIFO. When "Tx.DecimationEnable"is nonzero, only one of every "Tx.DecimationFactor" samples of each Tx channel's data read from the DAC FIFO is output to the DACs.

```
bool ApplicationIo::StartStreaming()
{
…
    // Set Decimation Factor
    int factor = Settings.Tx.DecimationEnable ? Settings.Tx.DecimationFactor : 0;
    Module.Fmc().Output().Decimation(factor);
```

```
        factor = Settings.Rx.DecimationEnable ? Settings.Rx.DecimationFactor : 0;
        Module.Fmc().Input().Decimation(factor);
```

Data Streaming. During data flow, the number of received data packets, data transfer rate, and the PLL lock status are shown in real time. Data collection terminates when any key is pressed or when the amount of data specified by "Rx.SamplesToLog" is accumulated. "Rx.OverwriteBdd" controls whether a new BinView binary data descriptor file should be created as streaming terminates. Normally, this should be enabled (1) so that a valid BDD is available for use by BinView when it is opened to view acquired data. But under some circumstances, such as when comments are added to the BDD file, it may be desirable to avoid re-creating the file each run by setting the value to 0.

```
void ApplicationIo::InitBddFile(BinView & graph)
{
    // Optionally skip update.
    PathSpec spec(graph.BinFile());
    spec.Ext(".bdd");
    if (FileExists(spec.Full()) && !Settings.Rx.OverwriteBdd)
        return;
```

PetaLinux Application Organization

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. To decouple the *ApplicationIo* from the user interface, an Interface class is used 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

For information on board access and data handling in ApplicationIo, see the Malibu Library User's Manual chapter on the Snap example. The section below details the code required to perform real-time packet polling and processing.

Packet Processing

In order to process packets via the standalone application, the Malibu application must create the following object:

```
        RemoteAppControl                *RemoteAppInterface;
```

The `RemoteAppControl` class accesses the memory region shared with the standalone application to command it, and to obtain status and sampled data from it. After setting up all of the software, firmware, and hardware, but just prior to enabling triggers, the following code informs the standalone application of the number of receive and transmit channels and signals it to begin servo processing:

```
bool ApplicationIo::StartStreaming()
{
…
  RemoteAppInterface->Start(LVDTChannels(), Module.Fmc().Output().ActiveChannels());
  Log("Servo processor started");
…
}
```

The PetaLinux application can periodically report status as show below:

```
//---------------------------------------------------------------------------
//  ApplicationIo::HandleTimer() --  Per-second status timer event
//---------------------------------------------------------------------------

void ApplicationIo::HandleTimer(OpenWire::NotifyEvent & /*Event*/)
{
    double sample_rate = Module.Fmc().Clock().Adc().FrequencyActual();

    for (int i=0; i<4; i++)
        {
        // Get Info
        Module.LVDT_Demod_GetChannelStatus(i, sample_rate, Settings.LVDT.InitDone[i],
Settings.LVDT.DeltaPhase[i],
                                           Settings.LVDT.RefFreq[i], Settings.LVDT.Position[i]);

    // Update SampleCount and display status:
    SampleReport();
    // Check SampleCount against data limit:
    CheckDataLimit();

    Trig.AtTimerTick();
}

//---------------------------------------------------------------------------
// ApplicationIo::SampleReport() -- Returns count of remotely processed samples.
//---------------------------------------------------------------------------

void  ApplicationIo::SampleReport()
{
    double  FSampleCount;
    double  FSampleRate;

    // Calculate transfer rate in SPS:
    double  Period = ReportTime.Differential();
    unsigned int  SamplesDiff = RemoteAppInterface->SampleCountDiff();
    SampleCount += SamplesDiff;
    FSampleCount = static_cast<double>(SampleCount);
    double  LastRate = (0.0 != Period) ? static_cast<double>(SamplesDiff) / Period : 0.0;
    FSampleRate = ReportRate.Process(LastRate);
    UI->PeriodicStatus(FSampleCount, FSampleRate, PllLocked());
}
```

Stop the remote servo loop when data streaming stops, after triggering is disabled.

```
void ApplicationIo::DoStopStreaming()
{
  …
    // Since triggering is stopped, the ADC FIFO should be cleared by now:
    RemoteAppInterface->Stop();
  …
}
```

The last sampled data packets are left in a circular buffer in shared memory by the standalone application and can be logged to the Data.bin file in the current directory by implementing the abstract RemoteAppControl::ILogger class using **ApplicationIo**'s member Logger:

```
class RemoteDataLogger : public RemoteAppControl::ILogger
    {
    public:
    RemoteDataLogger(Innovative::DataLogger   *Logger) : DataLogger(Logger) {};
        virtual bool  Log(const int * data, unsigned int size_in_ints)
            {
                return DataLogger->Log(data, size_in_ints);
            }
    private:
        Innovative::DataLogger   *DataLogger;
    };

//-------------------------------------------------------------------------
//  ApplicationIo::HandleAfterStop() --  Post stream termination event
//-------------------------------------------------------------------------

void ApplicationIo::HandleAfterStop()
{
  …
    // Log remotely collected data:
    RemoteDataLogger  RemoteLogger(&Logger);
    RemoteAppInterface->Log(&RemoteLogger);
  …
}
```

Inter-processor Communication

A single structure located in the Zynq On-Chip Memory (OCM) holds all of the data that is shared between the two servo applications running on the two ARM processors:

```
// Located at SHARED_OCM_MEMORY_BASE:
struct RemoteInterface
    {
    unsigned int                CommandQSem;    // CPU0 => CPU1: indicates queued command
    CommandQTemplate::Queue     CommandQ;       // CPU0 => CPU1
    unsigned int                StatusQSem;     // CPU1 => CPU0: indicates queued status
    StatusQTemplate::Queue      StatusQ;        // CPU1 => CPU0: responses to commands
    DataQTemplate::Queue        DataQ;          // CPU1 => CPU0: latest samples of current run
    unsigned int                SampleCount;    // CPU1 => CPU0: total samples taken during current
run (can overflow)
    StatusMachine               State;          // CPU1 => CPU0: current state
    };
const unsigned int  SHARED_OCM_MEMORY_BASE = 0xFFFF0000; // requires sizeof(RemoteInterface) <=
0x10000
```

The two semaphores, CommandQSem and StatusQSem, are managed by a `Semaphore` class implemented identically in the two applications. Similarly, the two queues that the semaphores guard, CommandQ and StatusQ, are implemented as circular buffers by the same template class, `QueueT`, in the two applications. The commands enqueued in CommandQ are:

```
enum StatusMachine { mWaitForWork, mStartWork, mDoWork, mComplete, mError};
struct Command
    {
    StatusMachine  NewState;
```

```
    unsigned int  RunNumber;
    unsigned int  RxChannelCount;       // number of active Rx channels
    unsigned int  TxChannelCount;       // number of active Tx channels
    };
```

After enqueueing a command and setting CommandQSem, the PetaLinux application waits for the standalone application to enqueue a `Status` in StatusQ and set StatusQSem in response. If NewState is *`mStartWork`*, the standalone application acknowledges the command, saves RxChannelCount and TxChannelCount in a singleton instance of the `ServoTask` class, and begins packet processing. While in this state, data read from the ADCs is copied to DataQ, and SampleCount is incremented for each sample. When the standalone application subsequently receives a Command with NewState set to *`mComplete`*, it ceases packet processing.

Standalone Application Organization

The standalone application, after it is loaded into memory and the standalone CPU (CPU1) begins to execute it, waits in a loop in main() for a command from the PetaLinux servo application to signal that it should begin processing servo packets. Once that command is received and acknowledged, main passes RxChannelCount and TxChannelCount to `ServoTask` and then begins alternately calling a local function, **`servo_work`**, and polling CommandQSem. The **`servo_work`** function polls for a single sample by calling `ServoTask`::**`ProcessSingleSample`**() and, if one is returned, updates DataQ and SampleCount. Access to the RemoteInterface is limited to the code in main.cpp (and the `QueueT` and `Semaphore` classes, as discussed).

When **`ProcessSingleSample`** has read RxChannelCount number of samples from the ADC FIFO, it calls **`ServoTask::ProcessPacket()`**. This code, located in the ProcessPacket.cpp file, must be customized by the user to implement the desired servo algorithm, processing `RxChCount` elements of input member `RxPacketData` and storing results into `TxChCount` elements into output member `TxPacketData`. **`ProcessSingleSample`** then writes those values to the ADC FIFO.

Chapter 6: The CardSharp Board Object

Introduction

The CardSharp Board class is used to control the CardSharp hardware. The class contains a reference to device map object which maps registers required to initialize and run the CardSharp hardware. The CardSharp Board object can be thought as a shell that contains the device map and it relays function calls to its member device map.

Cardsharp Interface

```
//  Informational Methods
std::string                 Name() const
    {  return Device.Name();  }
std::string                 Revision() const
    {  return Device.Revision();  }
//  Control Methods
int                         Target();
void                        Target(int value);
virtual void                Open();
virtual void                Close();
virtual void                Reset();
virtual void                Start();
virtual void                Stop();
void                             Preconfig();
ii32                        PeekLogic( int offset );
void                        PokeLogic( int offset, ii32 value );
ii32                        PeekPort( int offset );
void                        PokePort( int offset, ii32 value );
//  SubDevices
const IX6LogicVersionInfo & Info() const;
const IMulticharUart &      Gps() const;
IX6_GpsIo &                 GpsCtrl();
BabushkaIeee1588 &          Babushka()
{  return Device.Babushka();  }
void                                SelectGps(size_t idx)
 {     Device.Gps_Sdev.SelectGps(idx); }
// Fmc connect interface
bool ConnectTo(IFmcDaughterCard * daughter, unsigned int slot=0 );
void DisconnectFrom(unsigned int slot);
```

1. **Target(int value)**
   Selects the target in case there are more than one board plugged in.

2. **Open()**
   Opens the device driver for the CardSharp and establishes the memory mapping of the register interface for the logic.

3. **Close()**
   Closes the device driver and unmaps the memory.

4. **Reset()**
   Asserts the logic reset bit.

5. **Start()**
   "Starts" the board. This is application specific, this may mean different things for different applications.

6. **Stop()**
   "Stops" the board. This is application specific, this may mean different things for different applications.

7. **Preconfig()**
   Initializes registers and fire events that needs to be done at the Preconfig time.

8. **ii32 PeekLogic(int offset)**
   Read from logic space at a given offset.

9. **ii32 PeekLogic(int offset, ii32 value)**
   Write to given value to logic space at a given offset.

10. **ii32 PokePort(int offset)**
    Read from port space at a given offset.

11. **ii32 PokePort(int offset, ii32 value)**
    Write to given value to port space at a given offset.

12. **Info()**
    The Info method returns a reference to Version information interface.

```
class IX6LogicVersionInfo
{
public:
    //
    //  Interface Functions
    virtual short           PciLogicRevision() const = 0;
    virtual short           PciLogicFamily() const = 0;
    virtual short           FpgaLogicVersion() const = 0;
    virtual short           FpgaHardwareVariant() const = 0;
    virtual short           PciLogicPcb() const = 0;       // Board Revision
    virtual short           PciLogicType() const = 0;      // Board Type
    virtual short           FpgaChipType() const = 0;
    virtual short           FpgaLogicSubrevision() const = 0;
    virtual short           BoardRevisionBuiltFor() const = 0;
    virtual unsigned int    PciLogicRawValue() const = 0;
    virtual unsigned int    FpgaInformationRawValue() const = 0;
    virtual unsigned int    FpgaSubRevRawValue() const = 0;
    unsigned int            K7_FpgaTempGrade() const
    {  return PciLogicPcb() & 0x3;  }
```

```
        unsigned int            K7_FpgaSpeedGrade() const
        {  return (PciLogicPcb() & 0xc) >> 2;  }
        virtual ~IX6LogicVersionInfo() {}
    };
```

13. **Gps()**

    The Info method returns a reference to GPS interface interface.

```
     class IMulticharUart
{
public:
    virtual             ~IMulticharUart() {};
    virtual void           SetBaud(unsigned int rate) = 0;
    virtual unsigned int   GetBaud() = 0;
    virtual void           Port(unsigned char ch) = 0;
    virtual unsigned char  Port() = 0;
    virtual void           PutString(const std::string & data) = 0;
    virtual void           GetString(std::string & data) = 0;
    virtual void           GetFixedString(std::string & data, size_t chars) = 0;
    virtual bool           FrameError() = 0;
    virtual void           ClearFrameError() = 0;
};
```

In order to make use of the GPS device, an application needs to enable the device and register an Event handler with which to receive Epoch Events. The code below demonstrates a typical use case:

```
void ApplicationIo::GpsEnable(bool state)
{
    UI->GetSettings();

    Settings.GpsStat.EpochTally = 0;
    GpsInitTally = 0;

    Module.Ref().GpsCtrl().GpsEnabled(state);
    if (state)
        {
        Log("Initializing GPS, this might take a while...");
        }

     GpsInitTimer.Enabled(state);

}

void ApplicationIo::Open()
{
    UI->GetSettings();

    // Gps
    Module.Ref().SelectGps(Settings.WhichGps);   // Hook GPS handlers after you select which
                                                 //   will be used...
    Module.Ref().GpsCtrl().Gps().OnEpoch.SetEvent(this, &ApplicationIo::HandleOnEpoch);
    // Blah blah ...

}
```

```
void ApplicationIo::HandleOnEpoch(Innovative::GpsEpochEvent & Event)
{
    {
    stringstream msg;
    msg.precision(4);
    msg << "Epochs: " << ++Settings.GpsStat.EpochTally;
    }

    {
    stringstream msg;
    time_t currentTime = Module.Ref().GpsCtrl().Gps().Time();
    msg << "UTC: " << Innovative::Trim(asctime(localtime(&currentTime)));
    Settings.GpsStat.UtcTime = msg.str();
    }

    {
    stringstream msg;
    GpsCoordinate Lat = Module.Ref().GpsCtrl().Gps().Latitude();
    msg << "Latitude: " << Lat.Print();
    Settings.GpsStat.Latitude = msg.str();
    }

    {
    stringstream msg;
    GpsCoordinate Long = Module.Ref().GpsCtrl().Gps().Longitude();
    msg << "Longitude: " << Long.Print();
    Settings.GpsStat.Longitude = msg.str();
    }

    {
    stringstream msg;
    msg << "Quality: " << Module.Ref().GpsCtrl().Gps().Fix();
    Settings.GpsStat.Quality = msg.str();
    }

    {
    stringstream msg;
    msg << "Satellites: " << Module.Ref().GpsCtrl().Gps().Satellites();
    Settings.GpsStat.Satellites = msg.str();
    }

    {
    stringstream msg;
    msg << "Lock: " <<Module.Ref().GpsCtrl().Gps().IsLocked();
    Settings.GpsStat.Lock = msg.str();
    }

    if (!Settings.DebugGps)
        return;

    for (GpsMessageList::iterator i = Event.List.begin(); i != Event.List.end(); ++i)
        DumpGpsMessage(*i);
}
```

Initialization sequence

A typical application uses the board object as follows. The Console object contains the board object and evokes methods with the same names. The object is initialized by the constructor. The board is opened by the open call. The Start method initializes registers to appropriate values for the Start operation. The reason this sounds vague is that for a given application the Start may mean different things. For example, a Stream type of application uses Start method to initialize a Run register which start DMA operation. In a Servo application, which does not involve DMA, Start means something different.

```
int main()
{
        Console c;

        c.Open();
        c.Start();
        getchar(); // Wait
        c.Stop();
        c.Close();
}
```

Chapter 7: The FmcServo Module Object

Introduction

The Malibu Library provides interface software that allows applications to control the operation of the Cardsharp and FMCServo hardware. These two boards are managed by a combined logic, but for the purposes of the software reuse it is best to divide the control code into functions that are common to all modules and for the remainder to fall into a separate class object that is logically "attached" to the Cardsharp baseboard object and used thereafter. Thus the operations common to all boards need only be written once, and the board specific ones devised when needed.

The FMC module contains the analog I/O system required for the combined configuration. So the FMC object will contain objects and interfaces used to control the PLLs, ADCs, and DAC systems on the FMC board, if any. Most of the methods the user need concern themselves with deal with initialization of the hardware prior to data-taking. Other methods not described are the interface to the carrier board itself.

FMCServo Operations

The FMCServo has both analog in and analog out devices, which in the library means that it is a bidirectional board. The class family defined for FMC card adds internal "SubDevice" objects that control parts of the logic dedicated for a particular function. For the user, these objects are black boxes that do not need to be interacted with.

Object Attachment

Before the boards are opened, the FMC board needs to be connected to its baseboard carrier object. Once connected, the carrier will call methods in the FMC board to properly initialize the hardware at critical times like baseboard Open and Close, and during analog configuration for Servo and Streaming mode. This also allows the FMC board to access the carrier's memory spaces properly when the board is opened.

```
Innovative::FmcServo     Fmc;
Innovative::Cardsharp    CS;    // an FMC carrier

CS.ConnectTo(&Fmc);
```

Shared Base Class Operations

There are some common interfaces in the ultimate base class FmcBoard, located in FmcSupport_Mb.h. These should not be manipulated by the user. These are internally used, or debug interfaces, or in the case of the Vita interfaces only for use on other carrier boards.

```
FmcStandardRomInfo &         RomInfo();
IFmcInformation &            FmcInfo();
IVitaPacketizer &            VitaIn();
IVitaTimestamp &             VitaIn_TS();
IVitaDepacketizer &          VitaOut();
IX6Aurora &                  Aurora(unsigned int idx);
```

The Input() Device

```
class FmcServoInputDevice &    Input();
```

FMC Cards have a specific device to hold configuration methods for Analog Input. This allows the often virtually identical methods needed to configure Analog Output features to be expressed in a common language, easing the learning path for users.

Input.GainRange()

```
size_t  GainRange(int ch);
void    GainRange(int ch, size_t gain_range);
```

This method pair allows the setting and reading back of the gain range setting for each channel.

Input().Info()

```
virtual IX6AnalogIoInfo &  Info();
```

This method returns an interface that contains information about the analog on the board. This can be queried by the application to test its inputs against what the hardware can support. The following methods are in the interface:

```
virtual float   MaxRate() const;
virtual float   MinRate() const;
virtual SpanInfo  Span() const;
virtual unsigned int  Bits() const;
virtual unsigned int  SamplesPerWord() const;
virtual unsigned int  TriggerFrameGranularity() const;
virtual size_t  GainRanges() const;
virtual SpanInfo  Span(size_t /*gain_range*/) const;
```

MinRate() and MaxRate() are the ADC clock rates supported by the hardware. The Span() functions give the voltage range of the board in mV. GainRanges() gives the total number of gain ranges supported. Since the FMCServo supports multiple gain ranges, the second method should be used to give the span for that gain range. Bits() and SamplesPerWord() tell the data size provided by the ADC. TriggerFrameGranularity() is how coarse framed trigger settings are internally. If greater than 1, then widths not evenly divisible by this number will be truncated.

Input() Channel Enable Methods

```
virtual unsigned int  Devices();
virtual unsigned int  ChannelsPerDevice();
virtual unsigned int  Channels();
virtual void  DeviceDisableAll() {  ChannelDisableAll();  };
virtual void  DeviceEnabled(int dev, bool state);
virtual bool  DeviceEnabled(int dev) const;
virtual void  DevicePowered(int dev, bool state);
virtual bool  DevicePowered(int dev) const;
virtual unsigned int  ActiveDevices();
virtual void  ChannelDisableAll();
virtual void  ChannelEnabled(int ch, bool state);
virtual bool  ChannelEnabled(int ch) const;
virtual void  ChannelPowered(int ch, bool state);
virtual bool  ChannelPowered(int ch) const;
virtual unsigned int  ActiveChannels();
```

Input Devices have a number of methods to control the enabling or disabling of channels in an application. By default any inactive channel is powered down. Normally, an application only needs to do a simple loop to configure the board according to a saved Settings of channels:

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

The method ActiveChannels() counts the number of channels enabled at any time.

Input().Cal()

```
virtual GeneralRangedCalibrationSection & Cal();
```

The Cal() object controls Input calibration of the analog channels. Each gain range has its own calibration. The interface has these methods:

```
bool   Calibrated(size_t gain_range) const;
void   Calibrated(size_t gain_range, bool state)

float  Offset(size_t gain_range, int ch) const;
void   Offset(size_t gain_range, int ch, float value);

float  Gain(size_t gain_range, int ch) const;
void   Gain(size_t gain_range, int ch, float value);

void   LoadFromRom();
void   StoreToRom();
```

Offsets and Gains are set by the appropriate method. The Calibrated() function tells the library to use the loaded settings. If this bit is off, a default setting of Gain=1 and Offset=0 is used. LoadFromRom() and StoreToRom() save, and recover the data from the FMC Rom.

The following code shows using Cal() to save a new set of calibration data to ROM.

```
    for (unsigned int range = 0; range < AnalogInGainRanges(); ++range)
        {
        for (unsigned int ch = 0; ch < InputChannels(); ++ch)
            {
            Module.Fmc().Input().Cal().Gain(range, ch, Settings.Rx.Gain[range][ch]);
            Module.Fmc().Input().Cal().Offset(range, ch, Settings.Rx.Offset[range][ch]);
            }
        Module.Fmc().Input().Cal().Calibrated(range, Settings.Rx.Calibrated[range]);
        }
    Module.Fmc().Input().Cal().StoreToRom();
```

Input().Trigger()

```
virtual ITriggerCfg &  Trigger();
```

This method returns the interface class to configure the Incoming Trigger logic. These are common parameters across boards:

```
        virtual void          FramedMode(bool state);
        virtual bool          FramedMode() const;
        virtual void          Edge(bool state);
        virtual bool          Edge() const;
        virtual void          External(bool state);
        virtual bool          External() const;
        virtual void          FrameSize(unsigned int samples);
        virtual unsigned int  FrameSize() const;
        virtual void          ExternalSyncSource(AfeExtSyncOptions src);
        virtual AfeExtSyncOptions  ExternalSyncSource() const;
```

If FramedMode() is set false, triggering works normally: Data starts on first trigger, and runs forever until stopped. If FramedMode() is true, then the system will take FrameSize() samples then stop, until a new trigger arrives to start a subsequent frame.

If External() is false, the system will start only on SoftwareTrigger(). If External() is true, then an external signal can trigger in addition to SoftwareTrigger(). Edge() sets the external trigger to be edge sensitive. ExternalSyncSource() sets any logic or hardware multiplexer switching which external signal will act as a trigger.

Input().Pulse()

```
virtual IPulseRepetitionIntervalIntf &  Pulse();
```

This method returns the interface class to configure the Incoming PRI trigger logic. This mode adds an even more complicated type of framed mode, where you can program a set of start offsets and widths inside a larger frame, that will take data inside each 'pulse' and not take data. These can be repeated forever, or for a certain timeframe. The interface:

```
        virtual void   Reset();
        virtual void   AddEvent(unsigned int period,
                                unsigned int delay,
                                unsigned int width );

        virtual void  Enabled(bool state);
        virtual bool  Enabled() const;
        virtual void  FiniteFrames(bool state);
        virtual bool  FiniteFrames();
        virtual void  FiniteFrameRearm(bool state);
        virtual bool  FiniteFrameRearm() const;
```

```
        virtual void           FiniteFrameCount(unsigned short count);
        virtual unsigned short FiniteFrameCount() const;
```

The first two methods are used to define the period of the pulse cycle and each pulse's delay from the start and width in clocks. Reset() cleans the internal store to give a fresh start. If Enabled() is set true, this data will be loaded and the PRI mode used at start of data-taking.

The remaining methods define how the framing works. If FiniteFrames() is set true, the cycle will repeat FiniteFrameCount() times before stopping. If FiniteFrameRearm() is true, additional triggers will restart the PRI cycle again.

Input.SoftwareTrigger()

```
virtual void SoftwareTrigger(bool state);
virtual void SoftwareTrigger(AfeDirection which, bool state);
```

These methods allow the sending of the software trigger either to the Input trigger alone, or to both together if using the direction tInputOutput in the second method.

Input.Decimation()

```
    virtual void  Decimation(unsigned int samples);
    virtual unsigned int Decimation() const;
```

Decimation greater than 0 means to skip that many samples between samples allowed through. For example, 1 means to allow one, skip one. A factor of 2 means to allow one, then skip 2.

Miscellaneous Input() methods

```
virtual void   TestModeEnabled(bool state, unsigned int mode);
virtual void   TestFrequency(double /*frequency_hz*/);
virtual double TestFrequency();
void           AdcDeviceTestMode(bool enabled, unsigned int test_mode,
                                 std::vector<unsigned int> test_pattern);

virtual void  Spi(unsigned int device, unsigned int address, unsigned int data);
virtual unsigned int Spi(unsigned int device, unsigned int address);
```

These methods configure the FMC specific input test parameters to the logic. The Spi() method pair allows access to the analog device SPI interface, if any. These are debugging methods and are usually not used in applications.

The Output() Device

```
    class FmcServoOutputDevice &    Output();
```

FMC Cards have a specific device to hold configuration methods for Analog Output (if any). This allows the often virtually identical methods needed to configure Analog Output features to be expressed in a common language as Input(), easing the learning path for users.

Output.DacClockMode()

```
enum  DacClockModes { dcmServo, dcmAcquisition };  // change mode for how data is taken
void  DacClockMode(DacClockModes state);
```

This method pair changes the clock setup to optimize for servo operation, or data-taking through the fifo modes.

Output().Info()

```
virtual IX6AnalogIoInfo &  Info();
```

This method returns an interface that contains information about the analog on the board. This can be queried by the application to test its inputs against what the hardware can support. The following methods are in the interface:

```
virtual float   MaxRate() const;
virtual float   MinRate() const;
virtual SpanInfo  Span() const;
virtual unsigned int  Bits() const;
virtual unsigned int  SamplesPerWord() const;
virtual unsigned int  TriggerFrameGranularity() const;
virtual size_t  GainRanges() const;
virtual SpanInfo  Span(size_t /*gain_range*/) const;
```

MinRate() and MaxRate() are the DAC clock rates supported by the hardware. The Span() functions give the voltage range of the devuce in mV. GainRanges() gives the total number of gain ranges supported. Output generally only supports one gain range, so the first method should be used to give the span. Bits() and SamplesPerWord() tell the data size provided by the DAC. TriggerFrameGranularity() is how coarse framed trigger settings are internally. If greater than 1, then widths not evenly divisible by this number will be truncated.

Output() Channel Enable Methods

```
virtual unsigned int  Devices();
virtual unsigned int  ChannelsPerDevice();
virtual unsigned int  Channels();
virtual void  DeviceDisableAll() {  ChannelDisableAll();  };
virtual void  DeviceEnabled(int dev, bool state);
virtual bool  DeviceEnabled(int dev) const;
virtual void  DevicePowered(int dev, bool state);
virtual bool  DevicePowered(int dev) const;
virtual unsigned int  ActiveDevices();
virtual void  ChannelDisableAll();
virtual void  ChannelEnabled(int ch, bool state);
virtual bool  ChannelEnabled(int ch) const;
virtual void  ChannelPowered(int ch, bool state);
virtual bool  ChannelPowered(int ch) const;
virtual unsigned int  ActiveChannels();
```

Output Devices have a number of methods to control the enabling or disabling of channels in an application. By default any inactive channel is powered down. Normally, an application only needs to do a simple loop to configure the board according to a saved Settings of channels:

```
//  Channel Enables
Module.Fmc().Output().ChannelDisableAll();
for (unsigned int i = 0; i < Module.Fmc().Output().Channels(); ++i)
    {
    bool active = Settings.Tx.ActiveChannels[i] ? true : false;
```

```
        if (active==true)
            Module.Fmc().Output().ChannelEnabled(i, true);
        }
```

The method ActiveChannels() counts the number of channels enabled at any time.

Output().Cal()

```
    virtual GeneralCalibrationSection & Cal();
```

The Cal() object controls Input calibration of the analog channels. Each gain range has its own calibration. The interface has these methods:

```
    bool   Calibrated() const;
    void   Calibrated(bool state);

    float  Offset(int ch) const;
    void   Offset(int ch, float value);

    float  Gain(int ch) const;
    void   Gain(int ch, float value);

    void   LoadFromRom();
    void   StoreToRom();
```

Offsets and Gains are set by the appropriate method. The Calibrated() function tells the library to use the loaded settings. If this bit is off, a default setting of Gain=1 and Offset=0 is used. LoadFromRom() and StoreToRom() save, and recover the data from the FMC Rom.

The following code shows using Cal() to save a new set of calibration data to ROM.

```
    for (unsigned int ch = 0; ch < OutputChannels(); ++ch)
        {
        Module.Fmc().Output().Cal().Gain(ch, Settings.Tx.Gain[ch]);
        Module.Fmc().Output().Cal().Offset(ch, Settings.Tx.Offset[ch]);
        }
    Module.Fmc().Output().Cal().Calibrated(Settings.Tx.Calibrated);
    Module.Fmc().Output().Cal().StoreToRom();
```

Output().Trigger()

```
virtual ITriggerCfg &  Trigger();
```

This method returns the interface class to configure the Outgoing Trigger logic. These are common parameters across boards:

```
        virtual void          FramedMode(bool state);
        virtual bool          FramedMode() const;
        virtual void          Edge(bool state);
        virtual bool          Edge() const;
        virtual void          External(bool state);
        virtual bool          External() const;
        virtual void          FrameSize(unsigned int samples);
        virtual unsigned int  FrameSize() const;
        virtual void          ExternalSyncSource(AfeExtSyncOptions src);
        virtual AfeExtSyncOptions  ExternalSyncSource() const;
```

If FramedMode() is set false, triggering works normally: Data starts on first trigger, and runs forever until stopped. If FramedMode() is true, then the system will take FrameSize() samples then stop, until a new trigger arrives to start a subsequent frame.

If External() is false, the system will start only on SoftwareTrigger(). If External() is true, then an external signal can trigger in addition to SoftwareTrigger(). Edge() sets the external trigger to be edge sensitive. ExternalSyncSource() sets any logic or hardware multiplexer switching which external signal will act as a trigger.

Output().Pulse()

```
virtual IPulseRepetitionIntervalIntf &  Pulse();
```

This method returns the interface class to configure the Outgoing PRI trigger logic. This mode adds an even more complicated type of framed mode, where you can program a set of start offsets and widths inside a larger frame, that will take data inside each 'pulse' and not take data. These can be repeated forever, or for a certain timeframe. The interface:

```
        virtual void   Reset();
        virtual void   AddEvent(unsigned int period,
                                unsigned int delay,
                                unsigned int width );

        virtual void  Enabled(bool state);
        virtual bool  Enabled() const;
        virtual void  FiniteFrames(bool state);
        virtual bool  FiniteFrames();
        virtual void  FiniteFrameRearm(bool state);
        virtual bool  FiniteFrameRearm() const;
        virtual void           FiniteFrameCount(unsigned short count);
        virtual unsigned short FiniteFrameCount() const;
```

The first two methods are used to define the period of the pulse cycle and each pulse's delay from the start and width in clocks. Reset() cleans the internal store to give a fresh start. If Enabled() is set true, this data will be loaded and the PRI mode used at start of data-taking.

The remaining methods define how the framing works. If FiniteFrames() is set true, the cycle will repeat FiniteFrameCount() times before stopping. If FiniteFrameRearm() is true, additional triggers will restart the PRI cycle again.

Output.SoftwareTrigger()

```
virtual void SoftwareTrigger(bool state);
virtual void SoftwareTrigger(AfeDirection which, bool state);
```

These methods allow the sending of the software trigger either to the Input trigger alone, or to both together if using the direction tInputOutput in the second method.

Output.Decimation()

```
    virtual void  Decimation(unsigned int samples);
    virtual unsigned int Decimation() const;
```

Decimation greater than 0 means to skip that many samples between samples allowed through. For example, 1 means to allow one, skip one. A factor of 2 means to allow one, then skip 2.

Miscellaneous Output() methods

```
void  DacReset(bool state);

virtual void   TestModeEnabled(bool state, unsigned int mode);
virtual void   TestFrequency(double /*frequency_hz*/);
virtual double TestFrequency();

virtual void  Spi(unsigned int device, unsigned int address, unsigned int data);
virtual unsigned int Spi(unsigned int device, unsigned int address);
```

DacReset() will reset the DAC devices. The test methods configure the FMC specific output test parameters to the logic. The Spi() method pair allows access to the analog device SPI interface, if any. These are debugging methods and are usually not used in applications.

The Clock() Device

```
    IX6ClockIo &                    Clock();
    IAd9510DuoRawClockDevices &     RawClockDevice();
```

The devices used to generate clocks for the ADC/DAC devices and the logic are of great and increasing complexity. Internally, often multiple clock outputs are needed at different rates. For most purposes, knowing how to program these devices is of little interest as long as the results are correct. The Clock() interface gives a standard interface to the onboard timing system while hiding the details.

The RawClockDevices() interfaces gives low level access to the PLL and programmable VCXO objects (if any), mostly for debugging purposes.

Clock()'s interface is more useful:

```
// IX6ClockIo
    enum IIClockSource { csExternal, csInternal };
    enum IIReferenceSource { rsExternal, rsInternal };
    enum IIClockSelect { cslFrontPanel, cslP16, cslCarrier=cslP16 };

    virtual ITimebaseRate & Adc();
    virtual ITimebaseRate & Dac();

    virtual void               Source(IIClockSource src);
    virtual IIClockSource      Source() const;

    virtual void               ExternalClkSelect(IIClockSelect src);
    virtual IIClockSelect      ExternalClkSelect() const ;

    virtual void               Reference(IIReferenceSource /*src*/);
    virtual IIReferenceSource  Reference() const;

    virtual void               ReferenceFrequency(double /*value*/);
    virtual double             ReferenceFrequency() const;

    virtual bool               Locked()  const;
```

The Clock system can run in two basic modes – external clock or internal clock. The former uses an external clock input as a source, which can then be possibly divided down. The latter programs the onboard PLL device to generate a single high rate clock (the VCO clock) which is then divided down to the desired sampling rate.

Note that different rates can be set for the Adc() and Dac() clocks. But note that since there is only a single high-rate clock available, the different rates can only be achieved by dividing this clock down. In most cases this can give odd results if the two rates are not multiples of each other. The clock programming uses the faster clock to generate the VCO, so this direction will fit best.

The Source() method pair sets the mode of the clock. The value csExternal selects external clocking. The Reference() method pair controls if the PLL reference is an onboard crystal or other source, or an external clock.

ReferenceFrequency() is the frequency of the input clock. This is either the PLL reference, in internal mode, or the incoming clock rate if in external clock mode. If the output sample rate is not the same as the reference rate, dividers will be used if possible to produce the correct frequency.

ExternalClockSelect() connects to a mux that changes which output signal will provide the external clock input.

Clock().Adc() and Clock.Dac()

```
virtual ITimebaseRate & Adc();
virtual ITimebaseRate & Dac();
```

Setting the system frequency seems like a simple concept. However, there are some issues, since in general the limitations of the PLL may prevent the code from delivering the frequency you wish. In addition, some devices require a higher rate clock to be delivered to them in order to produce data at a certain rate. Other, more modern devices actually require a lower rate clock than the actual sample rate. In this case the rate might change based on the system configuration.

The decision made was for the user, in general, to be giving us the sample frequency desired. If you want data at 100 MHz, then you set the frequency to 100 MHz. The system will figure out what to put on the actual physical clock outputs. This is the ITimeBaseRate interface:

```
    virtual void    Frequency(double freq);
    virtual double  Frequency() const;
    virtual double  FrequencyActual() const;

    virtual double  EffectiveFrequency() const;
    virtual double  EffectiveFrequencyActual() const;

    virtual double  MultipliedFrequency() const;
    virtual double  MultipliedFrequencyActual() const;
```

The Frequency() method pair is used to set the desired output rate, and the readback returns your requested rate. FrequencyActual() returns the actual rate achieved, which presumably matches the request very closely, depending on the ability of the PLL to match your demand.

The MultipliedFrequency() and MultipliedFrequencyActual() readbacks show the rate on the outputs if the PLL has a defined clock multiplier – if the device needs an elevated data rate.

The EffectiveFrequency() and EffectiveFrequencyActual() readbacks show the rate on the outputs if the PLL has a defined clock multiplier – if the device requires a reduced data rate on the physical outputs.

In nearly all cases the Effective and Multiplied Frequencies are the same as the true ones. And aside from curiosity, the application doesn't need to know these rates.

FmcServo FIFO Configuration Devices

```
IPolledFifo *   AdcFifo() {  return &AdcFifo_SDev.Fifo();  }
IPolledFifo *   DacFifo() {  return &DacFifo_SDev.Fifo();  }
```

The FMCServo hardware differs from nearly all analog boards in that it does not support data streaming. Instead, there are FIFOs in the logic that provide the data, and level flags that can be used to poll for data in these FIFOs. This is the interface to each:

```
virtual unsigned int  Count();

virtual bool          AlmostEmpty();
virtual bool          Empty();
virtual bool          AlmostFull();
virtual bool          Full();

virtual void          AlmostFullLevel(unsigned int words) ;
virtual unsigned int  AlmostFullLevel() const;

virtual void          AlmostEmptyLevel(unsigned int words);
virtual unsigned int  AlmostEmptyLevel() const;

virtual void          Data(unsigned int value);
virtual unsigned int  Data() const;

virtual void          Delay(unsigned int ticks);
virtual unsigned int  Delay() const;
```

Use AlmostFullLevel() and AlmostEmptyLevel() to set the depth at which the corresponding flags will be set by the logic. The units are in two-sample words, so a servo should have at least two channels enabled.

The methods to return the flags for AlmostEmpty(), AlmostFull(), Empty() and Full() are provided. The former are the useful ones for Servos, as you can set the levels to signal only when a complete incoming event has arrived for processing.

Data() reads or writes to the FIFO. The ADC fifo only reads, the DAC fifo only writes. Again, the data is in pairs of samples per access.

Delay() only works on the DAC device, and delays the DAC update so that a conversion will wait until the data is read and processed in a servo application, to reduce latency. Count() reads the FIFO level, but the flags provide a better signal that data has arrived for processing.

Chapter 8: Aspects of Malibu for Cardsharp

Buffers and Buffer Access Classes

Introduction

Innovative Boards are required to transport chunks of data to and from the hardware, and to and from mass storage on the operating system. In this process, the data is not analyzed but just physically moved about. The Buffer classes are made to facilitate this by allowing you to create and move buffers about in our system without copying data unless 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 were the data buffers must begin for optimal performance. To insure that buffers are compatible with this library, Malibu ensures 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.

Buffer Data Access

With data movement covered, now we come to the issue of data access. In general, a buffer may hold data of any type and in any format. How, then, can we access the data without error prone casting? 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` template 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.

Buffer Classes

The Buffer class contains a data block and a header and trailer block. The header block is used to hold parametric information when a buffer is transmitted by Malibu to board hardware or stored to disk. Other processes may ignore the header, although it is always present and sized to hold at least 2 words. Buffer uses managed aligned blocks, and is reference counted for fast copying as long as the data is unchanged.

Buffer Class Types

There are several kinds of buffers, that are differentiated mostly by the size of the header and trailer blocks. This is due to the requirements of data transport over hardware. Cardsharp streaming uses McBuffer classes.

Buffer Utility Classes

Holding Template

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.

CouldHold Template
The CouldHold Template is the inverse of the Holding Template. Given a buffer size, it will calculate how many elements of a certain size could be placed into it without resizing.

```
Buffer CharStore(1000);
int bytes = CouldHold<char>(CharStore.SizeInInts()) );
```

Convert Template
In some instances, you might have a buffer of generic type that needs to be re-typed as a particular buffer type. For example you might have a buffer with data of a VeloBuffer read from a file but is now just a Buffer. This template creates a properly typed Buffer class containing all the data, header, and the trailer from the original buffer, which is now destroyed.

ConvertData Template
This template creates a properly typed Buffer class containing the data from the original buffer, which is now destroyed. The header and trailer data is that of a default buffer, and would need to be initialized afterward.

MessageDatagram
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.

Header Access Datagrams
Datagram wrappers like AccessDatagram access the Data portion of a buffer. But sometimes you need to access the Header section, particularly to format the header for transmission or find the data source on reception of a packet. Each buffer class has a predefined Header datagram:

Header Access Class Name	Buffer Type
PacketBufferHeader	**Buffer**
PmcHeaderDatagram	**PmcBuffer**
VeloHeaderDatagram	**VeloBuffer**
VitaHeaderDatagram	**VitaBuffer**
McHeaderDatagram	**McBuffer**

These classes have predefined methods to read and fill in bit fields in the header that have particular meaning, such as the packet type code fields and data size fields. In addition, as an AccessDatagram, the array access operator method works to allow manipulation of any other part of the header in addition to the special methods.

There are also similar classes for the buffer trailer found in VitaBuffers, but this is rarely used.

The example below shows an example of using the header to differentiate between kinds of packets in a data processing program. This allows the data to be properly managed based on its type.

```
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
```

Access Datagrams

The data access requirements seem to require contradictory features: Support for many different types of data quickly and easily is required, but a minimal code base is also 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 an applicaton-specific 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, and the size of the datagram avoids walking off the end of the buffer.

Template `AccessDatagram<T>`

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 if the buffer class attached to can be resized.

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`

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`
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`
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;
};
```

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:

Class Name	Data Type
IntegerDG	**int**
UIntegerDG	**unsigned int**
FloatDG	**float**
DoubleDG	**double**
ComplexDG	**Complex**
ShortDG	**short**
UShortDG	**unsigned short**
CharDG	**char**
UCharDG	**unsigned char**

Hardware Access and Bit Control

Memory Spaces and Register Classes

A major part of programming registers to the logic is register declaration and bit manipulation. Since the process of shifting, masking, and-ing and or-ing is notoriously error prone and hard to decipher afterward, we used classes to encase the methods. Once these were made and tested, they could be reused over and over.

Addressing Space Classes

```
//  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 CardSharp carrier card implements two such spaces, but only the LogicMemory space will be used outside of the data streaming engine.

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 its own offset to turn an FMC relative address into the absolute offset needed for the physical access.

```
//
//  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)

{
}
```

Malibu uses registers heavily to both implement and document the hardware definition of the registers of the carrier card and FMC card added to it. Ordinarily the user will not define his own registers unless he is developing custom logic.

BaseboardExtension Template

```
class NewRegisters :  public BaseboardExtension<X5_210M>
{
     NewRegister(X5_210M & board)
                :  inherited(board),
                   AdcSelectReg(LogicMemory, mmDemodAdcSelect)
                        {}

     Register AdcSelectReg;
};
```

To add registers to an existing class, the BaseboardExtension template is defined. If you derive a class from this template, with template argument the board type you will use, then the template links itself to the LogicMemory and PortMemory addressing spaces of the class. Thus your registers will access his memory space.

Register Class Types

Class	Description
Register	General Register – all reads and writes access the hardware directly
ReadOnlyRegister	Register with no write methods.
ShadowRegister	Register that does not read the hardware, but caches value in memory.
RefreshableShadowRegister	ShadowRegister with a Refresh() method to read the hardware on demand
CachedShadowRegister	ShadowRegister that only writes when Apply() method is called. This is useful when you need to change multiple register fields with a single write access.

Register Bits and Fields

Class	Description
RegisterBit	Defines a bit using a start bit index.
RegisterBitGroup	Defines a field of bits using a start bit and a width.
ReadOnlyRegisterBit	Same as RegisterBit, but no write methods.
ReadOnlyRegisterBitGroup	Same as RegisterBitGroup, but no write methods.

Registers can have sections defined, both a single bits and mulibit fields using these objects. Changing these objects changes the register without affecting any other bits in the register. These accesses do a read-modify-write on the register they are linked to.

Bit Manipulation Classes

In cases where defining a register and the associated bit and field classes is too overweight, a set of bit manipulation wrappers was also developed. These all share a common user interface IBitManipulator that allows changing a bit, a field, or the entire word.

```
class IBitManipulator
{
public:
    virtual ~IBitManipulator();
    bool            Bit(unsigned int bit) const;
    void            Bit(unsigned int bit, bool value);
    unsigned int    Field(unsigned int bit, unsigned int size) const;
    void            Field(unsigned int bit, unsigned int size, unsigned int value);
    void            Mask(unsigned int mask);
    unsigned int    Value() const;
    void            Value(unsigned int value);
};
```

Class	Description
MemoryBitManipulator	Manipulates an internal word.
PtrBitManipulator	Manipulates an external word via a pointer to it.
RegisterBitManipulator	Manipulates a register object's word.

MultiChannel Data Streaming Support Classes

The MultiChannel Stream interface uses several classes to pass data to the application.

Buffer Classes – McBuffer and McHeaderDatagram
The Multichannel Stream has its own dedicated buffer type and its own header datagram to configure the header for each buffer. When sending data, the header must be configured to direct the data to the appropriate channel. When data is received, the header gives the source of the data so that it can be directed to the proper analysis code.

McHeaderDatagram has the following methods. ChannelId() is the source or destination channel for the data. PacketSize() and DataSize() give the size of the packet in 32 bit words. Other buffer classes use PacketSize() to give the total size of the data including the header, while DataSize() is just the size of the data. In MultiChannel streaming, since the header is not included in the stream's busmaster region, both these function return the size of the data.

```
    unsigned short  ChannelId() const;
    void            ChannelId(unsigned short channel);
    size_t          PacketSize() const;
    void            PacketSize(size_t size);
    size_t          DataSize() const;
    void            DataSize(size_t size);
```

Data Image Classes – McImage
The Multichannel stream supports "image mode" processing, where an application can be notified directly when data movement is required without the extra copying that creating and using a McBuffer. In effect, an image is a buffer header plus a pointer to the exact region of busmaster memory needed to be filled, or processed as input.

While a much simpler class internally, McImage retains the same ability as Buffer classes to have Access Datagrams wrapped around them for typing the contained data. Header Datagrams also function, accessing the header:

```
//  The header already gives the channel, and advance distance.
McHeaderDatagram ImageH(semiImage);
size_t ring     = ImageH.ChannelId();
size_t sizeInts = ImageH.PacketSize();
```

A major difference between an image and buffer is that the buffer is permanent – its data will remain intact for as long as the object exist. A McImage is entirely different: the pointer to the data will become invalid the instant the image is sent to the stream, or when the stream's callback function returns. An Image should be thought of as a volatile entity.

If needed, an McImage can be "converted" into an equivalent buffer that contains the same information. The CopyTo() method fills a passed in McBuffer object with the header and data contained in the Image. Note that this might require an allocation and will copy the entire image. In the other direction, an McImage can be copied into from a McBuffer with the CopyFrom() method. There is a parameter to allow starting the copy from an offset into the buffer for partial copies in cases where one buffer needs to be divided into several images.

Streaming Alerts: McAlert
MultiChannel streaming produces four kinds of signals. Two are related to data streaming itself: one for data in, and one for data out. The third is a system alert, an informational packet to indicate some exceptional condition such as trigger changes, data overflow and underflow, and so on.

The McAlert has a fixed maximum size for data. This means that creating an object does not dynamically allocate memory. When an alert arrives, the stream extracts its data, packages it in an McAlert object, and notifies the user by an event. The McAlert object can also be wrapped by an AccessDatagram for data extraction.

Streaming External Interrupts: McExternalInt
MultiChannel streaming produces four kinds of signals. Two are related to data streaming itself: one for data in, and one for data out. The third is Alerts and the fourth is sent on the state-changes of external signals. Four external interrupt lines exist that can be programmed to send a notification when the signal goes from low-to-high, high-to-low, or both.

```
bool  Flag(int idx);

bool  RisingEdgeFlag(int idx);
void  RisingEdgeFlag(int idx, bool state);

bool  FallingEdgeFlag(int idx);
void  FallingEdgeFlag(int idx, bool state);

unsigned int  RisingMask();
void          RisingMask(unsigned int data);

unsigned int  FallingMask();
void          FallingMask(unsigned int data);
```

Flag() returns true if either the Rising or Falling edge is set for that bit. RisingEdgeFlag() and FallingEdgeFlag() return the appropriate flag: true if the state changed, false if not. RisingMask() and FallingMask() return the entire mask area for the edge so indicated. This class also can be accessed via an AccessDatagram, although since it is only one word long the usefulness is limited.

Image Mode Transmit Hints: McHintPack

When in image mode, the stream receives interrupts when data is processed and free for reuse. One interrupt might give back data on multiple channels, or multiple times on a single channel. When the Stream has adjusted its bookkeeping on free data in the Busmaster region, it notifies the user with an event, passing a McHintPack giving the channels that have just been acknowledged.

The McHintpack derived from the DataPack template, a simple template giving a fixed size array that can be logically resized down to smaller sizes. Its structure means that it also does not allocate memory. It has a member function that implements an array access operation for reading and writing.

Chapter 9: Cardsharp Hardware

Introduction

Cardsharp is a powerful embedded instrument which combines Xilinx Zynq system-on-the-chip (SoC) with a full HPC FMC site in a very compact standalone design. Along with Innovative wide assortment of ultimate performance FMC modules Cardsharp allows users to quickly build various customized systems with unparalleled flexibility.

Cardsharp is built in a rugged XMC form factor (149 mm x 74 mm) and can be plugged into an XMC carrier such as Innovative SBC-Nano adding more connectivity choices.

The Cardsharp system main features are:

- Xilinx XCZ045 Zynq SoC with dual floating point A9 CPU cores and programmable logic block.
- Boots Linux from on-board 32 GB eMMC
- 16 MB QSPI memory
- 256 Mb x 32 DDR3 PS (Processing System) memory
- 256 Mb x 64 DDR3 PL (Programmable Logic) memory
- Self-bootable standalone operation
- JTAG port for FPGA PS and PL programming and debugging
- Single Gigabit Ethernet port
- Single External USB 2.0 port
- 2x Internal Serial ports
- VITA 57 HPC FMC site
- Up to 6GB/s data transfer from FMC site to PS memory
- FMC Vadj in the range of 1.5V to 2.5V (factory preset to 2.5V)
- Supports Innovative full line of FMC modules and third party FMC modules
- UCD9090 power sequencer/supervisor
- System expansion support via XMC site interface

Cardsharp Hardware

- Up to 8x Gen2 PCIe lanes supported
- 7 differential/14 single-ended DIOs on XMC P16
- Up to 2x QSFP ports support
- Optional support for IEEE-1588 network or GPS-synchronized timing
- Support for external reference clock, trigger and PPS signals
- 12V +/-5% DC operation

Please note that not all features may be available and/or supported in some configurations.

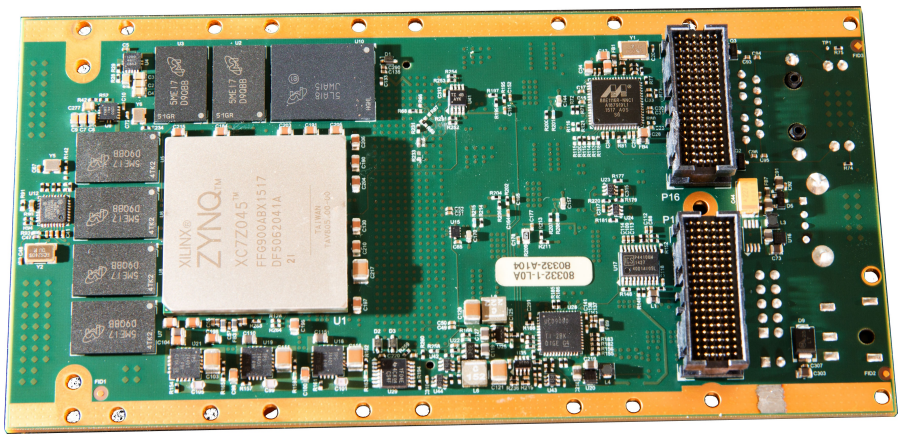

Figure 22. Cardsharp board view from the XMC connectors side

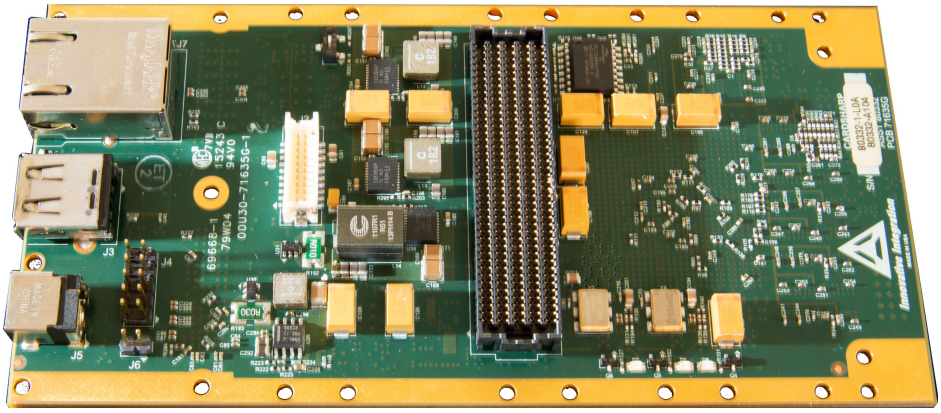

Figure 23. Cardsharp board view from the XMC connectors side

Figure 24. Cardsharp standalone system with FMC-Servo module installed

Figure 25. Cardsharp / SBC-Nano system with FMC-Servo module installed

Cardsharp Hardware

The Cardsharp system is packaged in a compact enclosure (165mm x 82mm x 35mm for standalone version). The unit is powered by external power supply (DC 12V +/-5%). Typical power consumption of the Cardsharp itself is about 12W, the total system power consumption depends on the used FMC module and the mode of operation.

Please refer to the power supply description section in this manual for further details on the Cardsharp power subsystem features and operation.

Custom application logic development for the Cardsharp is supported by the FrameWork Logic system from Innovative using VHDL and/or MATLAB Simulink. Signal processing, data analysis, and application-specific algorithms may be developed for use in the FMC Module logic and integrated with the hardware using the FrameWork Logic.

Software support for the Cardsharp includes Windows and Linux drivers for on-card peripherals, system integration and test, data logging and support applets. The Malibu Toolkit provides C++ development tools and examples for peripheral configuration and use, module interfacing examples and data logging.

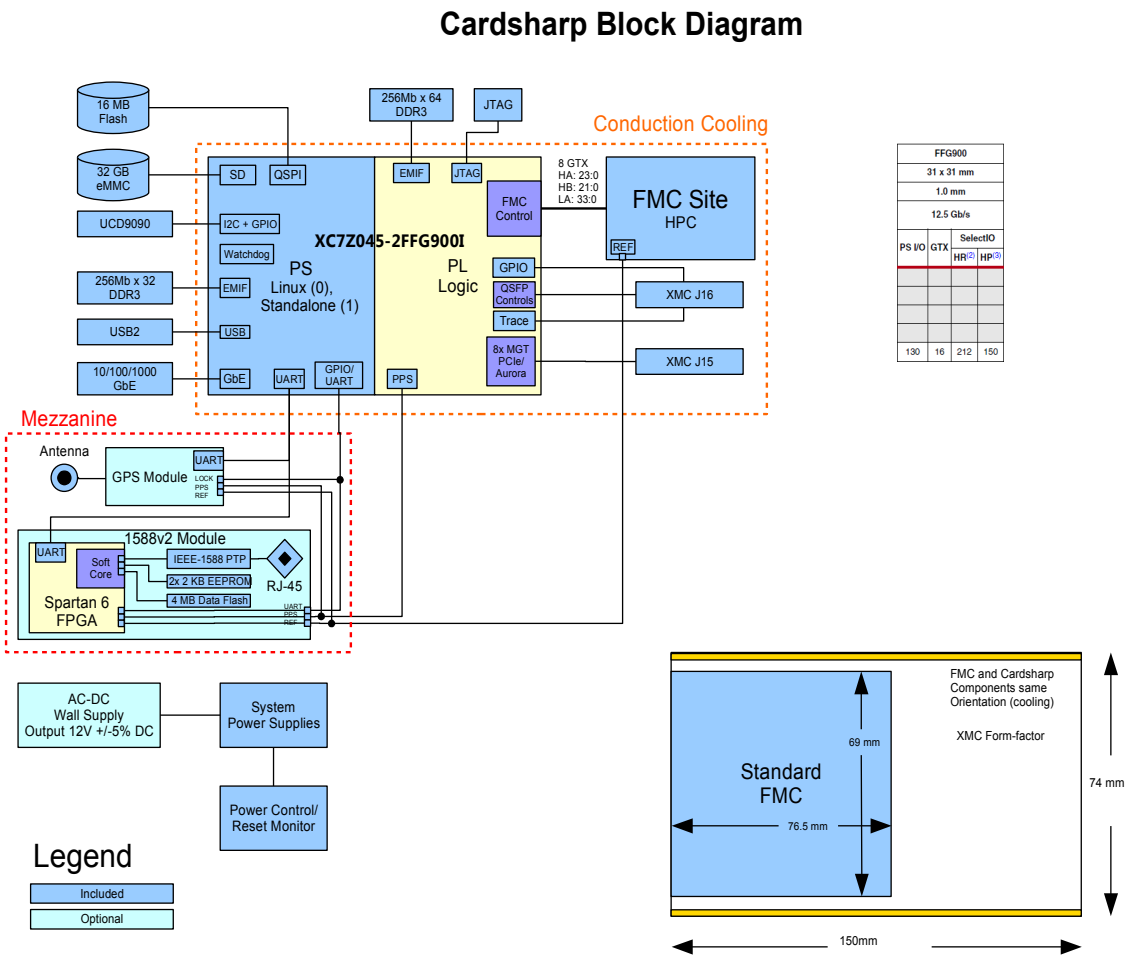

Figure 26. Simplified Block Diagram of the Cardsharp System

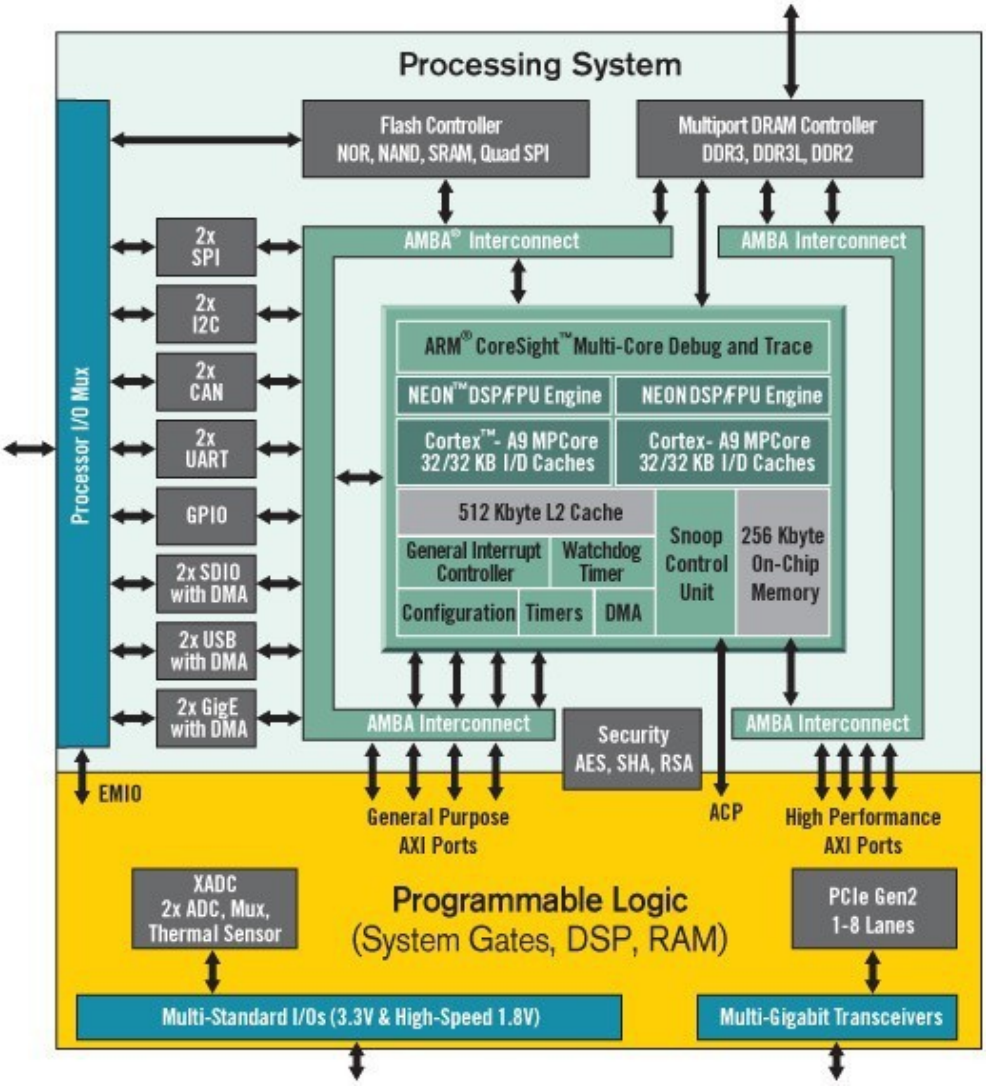

Figure 27. Zynq SoC block diagram

Zynq SoC

The Cardsharp system is built around Xilinx Zynq-7000 System-on-the-chip (SoC) with main features summarized in Table 1 below.

Cardsharp Zynq®-7000 All Programmable SoC	
Device Name	Z-7045
Part Number	XC7Z045
Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™
Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor
Maximum Frequency	Up to 1GHz
L1 Cache	32KB Instruction, 32KB Data per processor
L2 Cache	512KB
On-Chip Memory	256KB
External Memory Support	DDR3, DDR3L, DDR2, LPDDR2
External Static Memory Support	2x Quad-SPI, NAND, NOR
DMA Channels	8 (4 dedicated to Programmable Logic)
Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
Peripherals w/ built-in DMA	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
Security (shared with Programmable Logic)	RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot
Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts
7 Series Programmable Logic Equivalent	Kintex-7 FPGA
Logic Cells (Approximate ASIC Gates)	350K (~5.2M)
Look-Up Tables (LUTs)	218,600
Flip-Flops	437,200
Total Block RAM (# 36Kb Blocks)	19.1Mb (545)
Programmable DSP Slices (18x25 MACCs)	900
Peak DSP Performance (Symmetric FIR)	1,334 GMACs
PCI Express® (Root Complex or Endpoint)	Gen2 x8
Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs

Security (shared with Processing System)	AES and SHA 256b Decryption and Authentication for Secure Programmable Logic Configuration
HR I/O	212
HP I/O	150
PS I/O	128
GTX Transceivers	16
Unique Footprint	FFG900
PCB Footprint Dimensions (mm)	31 x 31

Table 7. Zynq SoC Main Features

Note: not all Zynq SoC features are available and/or supported in the Cardsharp system

QSPI

Cypress Semiconductor (formerly Spansion) S25FL128S type 128 Mbit (16 Mbyte) SPI Flash Memory (Quad Serial Peripheral Interface or QSPI) used for storing the system bootloader and PL configuration files.

eMMC

Micron Technology MTFC32G4M type embedded MultiMediaCard (eMMC) controller/32GB NAND Flash Memory used for storing PS core OS and other software files required for Cardsharp operation. eMMC function is similar to traditional SSD-s or HDDs operation in common PC-s. The eMMC on Cardsharp is used in a 4-bit wide data mode.

PS DDR3 Memory

A single 256MB x 32 DDR3L SDRAM type memory bank is attached to Zynq PS cores. The PS memory operates at 800MHz clock speed (DDR3-1600) and consists of two 4Gb (256MB x 16) memory chips (Micron Technology MT41K256M16HA-125).

PL DDR3 Memory

A single 256MB x 64 DDR3L SDRAM type memory bank is attached to Zynq Programmable Logic. The PL memory operates at 800MHz clock speed (DDR3-1600) and consists of four 4Gb (256MB x 16) memory chips (Micron Technology MT41K256M16HA-125).

Reset

If a remote reset is needed, Reset header J6 can be used along with a momentary push-button. Shorting J6 contacts will reset the Zynq ZoC. Reset signal can be also applied from the XMC connector P15 pin C2 (XMC_TRST_N signal). Reset is an active low signal.

Reset can also be tripped by the watchdog timer and by the UCD9090 system monitor if one or more of the monitored system voltages fall outside of the allowed range.

Connector Type:	2 Positions Header, Unshrouded Connector 0.100" T/H STR
Number of Connections:	2
Connector Part Number	Amphenol FCI 77311-118-02LF
Mating Connector	Amphenol FCI 65039-035LF or similar

Table 8. Reset Header J6 Information

Pin	Type	Signal	Note
1	I	SW_PS_SRST_B	Zynq SoC Reset, Active Low
2	P	GND	Ground

Table 9. Reset Header J6 Pinout

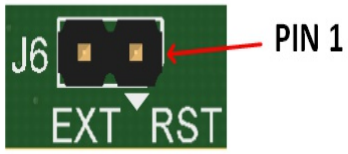

Figure 28. Reset Header J6 Pin Arrangement

Watchdog Timer

The Cardsharp has support for the watchdog timer to prevent runaway PS operation. A system reset is issued if the PS fails to periodically re-trigger the watchdog timer. Please note that the watchdog functionality must be enabled at the OS/software level to be operational.

USB 2.0 Port

The Cardsharp board has a single USB 2.0 Host Port which utilizes Microchip USB3320 type transceiver. The 5V power at the USB connector is protected from overcurrent condition (load current > 500mA) by Micrel MIC2025-1 USB Power switch. This port is accessible via standard USB 2.0 type A connector J3 on the rear panel of the Cardsharp system.

Connector Type:	Type A USB 2.0 Shielded I/O Receptacle R/A
Number of Connections:	4
Connector Part Number	Molex 67643-0910
Mating Connector	USB Type A

Table 10. USB Connector J3 Information

Pin	Type	Signal
1	P	+5V
2	I/O	USB D-
3	I/O	USB D+
4	P	Ground

Table 11. USB Connector J3 Pinout

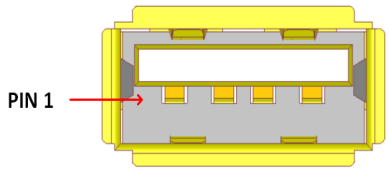

Figure 29. USB Connector J3 Pin Arrangement

Gigabit Ethernet Port

The Cardsharp board has a single Gigabit Ethernet Port utilizing Marvell Alaska 88E1518 Integrated 10/100/1000 Mbps Energy Efficient Ethernet Transceiver. This port is accessible via standard RJ45 Ethernet jack J7 on the rear panel of the Cardsharp system.

Gigabit Ethernet Connector J7 is located on the back panel of the Cardsharp board. It is an industry standard RJ45 jack with integrated magnetics and two build-in LED indicators.

LED indicators on the Ethernet port show port current status and configuration.

GbE LED	Function
Yellow	Gigabit Link Present
Green	Link Activity

Table 12. Gigabit Ethernet LED Functions

Connector Type	CONN, ETHERNET RJ45, INTEGRATED MAGNETICS FOR 1000 BASE T
Number of Connections	8
Connector Part Number	Bel L869-1A1T-32
Mating Cables	Modular RJ-45 CAT5/6 cables

Table 13. Gigabit Ethernet Connector J7 Information

Pin	Type	Signal

1	I/O	TP0+
2	I/O	TP0-
3	I/O	TP1+
4	I/O	TP2+
5	I/O	TP2-
6	I/O	TP1-
7	I/O	TP3+
8	I/O	TP3-

Table 14. Gigabit Ethernet Jack J7 Pinout

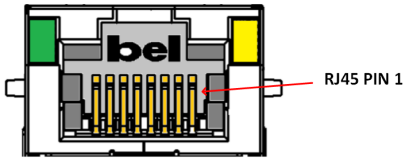

Figure 30. Gigabit Ethernet Connector J7 Pin Arrangement

Power Subsystem

The Cardsharp power subsystem block diagram is shown on figure 7.

The Cardsharp uses 12V DC as an input voltage to all on-board power supplies. J5 is the external power connector. The 12V to 3.3V DC-DC converter turns on immediately after the 12V input is applied and provides power to the Texas Instrument UCD9090 Power Sequencer/Voltage Supervisor IC which controls the Cardsharp power supplies operation. After the initial pre-programmed delay, the power sequencer starts turning on power supplies according to the Xilinx power sequence recommendations for the Zynq chip. After the power is fully on, the sequencer continues to monitor a few essential for the operation power rails. If at any time one or more of the monitored power supply voltages crosses the predetermined safety threshold, the whole Cardsharp power will be shut down by the UCD9090.

The Texas Instruments USB-TO-GPIO USB Interface Adapter used for the initial UCD9090 programming via Cardsharp 10-pin header J4. The J4 header pin functions are provided in table 9 below. Please note that few of the J4 header pins are reserved for the Cardsharp on-board JTAG chain access.

Note	Input / Output	Signal Name	J4	J4	Signal Name	Input / Output	Note

			pin #	pin #			
ON-BOARD JTAG TDO	O	CON_JTDO	**1**	**2**	CON_JTMS	I	ON-BOARD JTAG TMS
ON-BOARD JTAG TCK	I	CON_JTCK	**3**	**4**	CON_JTDI	I	ON-BOARD JTAG TDI
3.3V POWER	O	3P3V_UCD	**5**	**6**	GROUND	N/A	GROUND
UCD9090 CONTROL	I	UCD9090_CNTRL	**7**	**8**	UCD9090_ALERT	O	UCD9090 ALERT
UCD9090 CLOCK	I	UCD9090_CLK	**9**	**10**	UCD9090_DATA	I/O	UCD9090 DATA

Table 15. J4 Header Pinout

Connector Type	CONN HEADER 10POS 0.100” (2.54MM) VERT T/H
Number of Connections	2 x 5
Connector Part Number	Amphenol FCI 67997-410HLF
Mating Connector	Amphenol FCI 87606-305LF or similar

Table 16. J4 Header Information

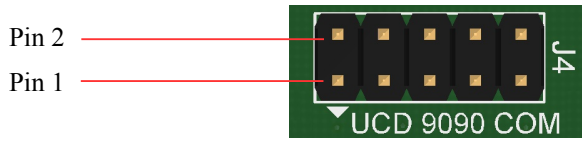

Figure 31. J4 Header Pin Arrangement

Most power supply circuitry on the board have DC-DC buck topology for the high efficiency operation along with a few LDO regulators. All on-board power supplies have overcurrent, overvoltage and overheating protection.

Please note that the Cardsharp itself can operate with a wide input voltage range - from 6V to 14V. The 12V +/-5% requirement for the system input voltage comes from the VITA 57.1 standard for the FMC modules since the Cardsharp simply passes the input 12V to the FMC module. However, most Innovative FMC modules can operate in in much wider range on the 12V input and hence allow usage of the wider system input voltage; please consult factory for further information.

The Cardsharp generates on-board 3.3V and Vadj voltage rails required for the FMC module operation. Vadj is preset at the factory to 2.5V, but it can be set in the range of 1.5V to 2.5V. Please contact factory if a Vadj option other than 2.5V is needed.

The Cardsharp has a circuitry to monitor Zynq's 1.0V rail's current and also the entire board current from the the 12V external power supply. This makes it possible to measure power consumption of the Cardsharp system in real time, which can be very useful in the battery powered systems.

Table 11 provides additional information on the Cardsharp power supplies.

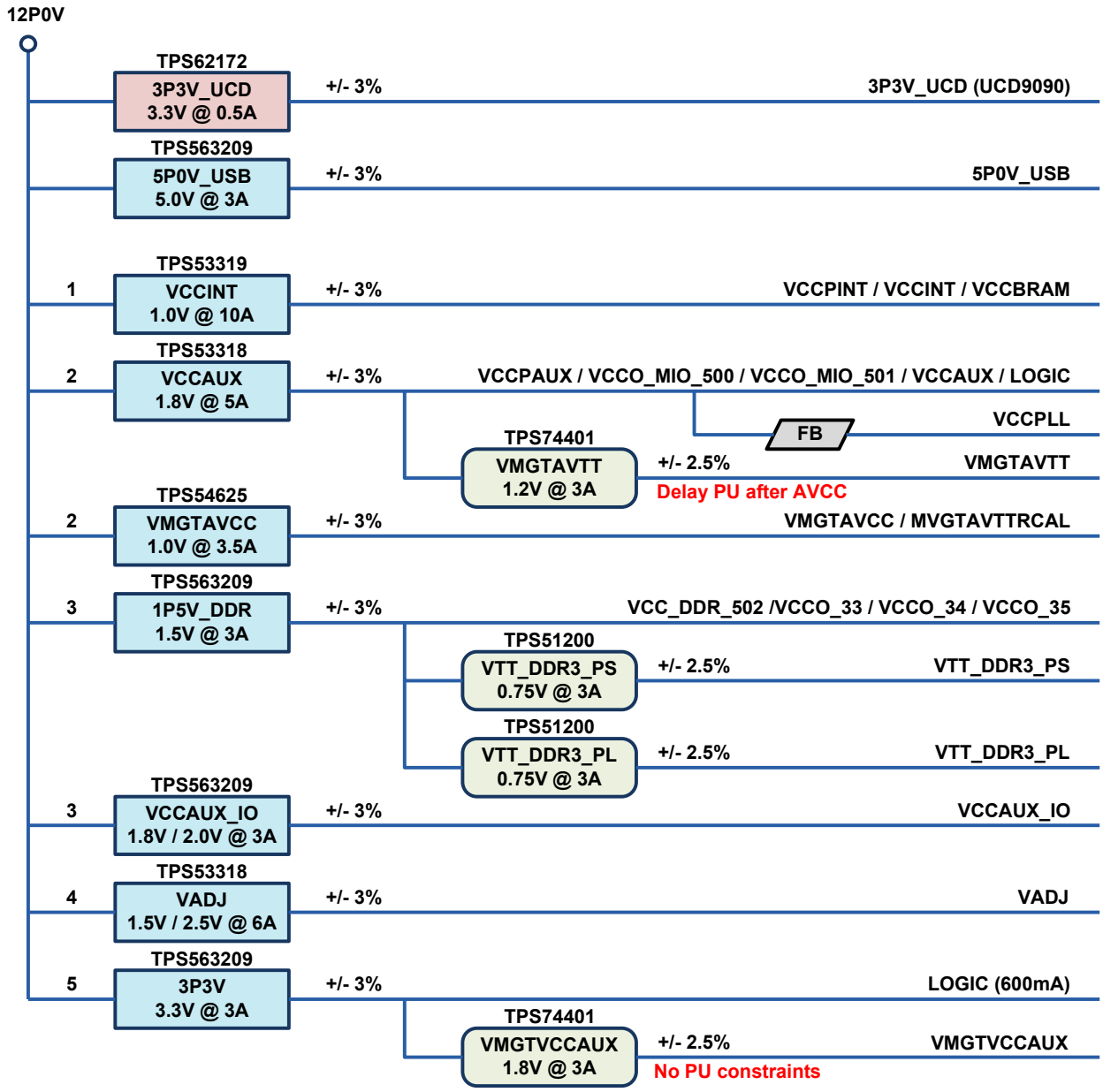

Figure 32. Cardsharp Power Block Diagram

Power Rail	Nominal Voltage (V)	Maximum Available Current (A)	Purpose	Type	Rank in Power sequence	UCD9090 Monitored
12P0V	12.0	8*	Input Power	Varies	N/A	yes
3P3V_UCD	3.3	1.0	UCD9090 power	DC-DC	Always on	no
5P0V_USB	5.0	1.0 (limited to 0.5A at the USB port connector)	USB 2.0 Port Power	DC-DC	Controlled by USB3320	no
1P0V	1.0	10.0	Zynq VCCINT / VCCPINT	DC-DC	1	yes
VCCAUX	1.8	5.0	Zynq VCCAUX / LOGIC	DC-DC	2	no
VMGTAVCC	1.0	3.5	Zynq VMGTAVCC	DC-DC	2	yes
VMGTAVTT	1.2	3.0	Zynq VMGTAVTT	LDO	2	yes
1P5V_DDR	1.5	3.0	Zynq DDR Inteface / DDR3 Memory	DC-DC	3	yes
VTT_DDR3_PS	0.75	3.0	PS DDR3 Memory Termination Voltage	LDO	3	no
VTT_DDR3_PS	0.75	3.0	PS DDR3 Memory Termination Voltage	LDO	3	no
VCCAUX_IO	2.0	3.0	Zynq VCCAUX_IO	DC-DC	3	yes
VADJ	2.5*	6.0	Zynq FMC Interface / FMC Vadj	DC-DC	4	yes
3P3V	3.3	3.0	Zynq Bank 0 / Logic	DC-DC	5	no
VMGTVCCAUX	1.8	3.0	Zynq VMGTVCCAUX	LDO	5	yes

Table 17. Cardsharp Power Supplies

Notes:

1. The maximum available current value for the 12P0V rail is shown for the Innovative supplied external power supply P/N 80200-9.

2. Vadj is preset at the factory to 2.5V, but can be set in the range of 1.5V to 2.5V; consult factory for details.

External Power Supply

Innovative offers a 12V DC 100W laptop style power supply (Innovative P/N 80200-9) as the external power source.

Specifications	
Input Range	AC 90 to 264V, 50-60 Hz
Output Voltage	12V DC
Maximum Current	8.33A
Output Power	100W
Safety	CE/FCC/IEC/UL
Dimensions	136mm x 58.5mm x 33.7mm 5.35” x 2.30” x 1.37”
Weight (with Power Cord)	Approx. 0.375 kg (0.83 lb)
Power Cord *	IEC 320 C13

Table 18. External Power Supply 80200-9 Specifications

Note: Power cord specified is for US/Japan orders only. Contact Innovative for other power cord options.

IMPORTANT! Power is NEVER completely off unless the 12V input is removed. To avoid board damage make sure the external power is disconnected before inserting or removing an FMC Module and/or inserting the Cardsharp board into an XMC Carrier.

When working with Cardsharp on an XMC Carrier (such as Innovative SBC-Nano) the external power must be disconnected from the Cardsharp input to avoid electrical damage since in this case power to the Cardsharp is being supplied by the XMC Carrier via the P15 connector.

The external power to the Cardsharp standalone system is provided via the J5connector on the rear panel. This connector is mounted on the Cardsharp board.

Connector Type:	DC Power Jack Connector 2.5 mm Center Pin, 5 A, Right Angle, Through Hole, Shielded
Number of Pins:	2
Connector Part Number	CUI PJ-051BH
Mating Connector	CUI PP3-002B or similar 2.5mm Power Plug

Table 19. External Power Connector J5 Information

Pin	Function
1 (center)	+ POWER
2 (case)	- POWER (GROUND)

Table 20. External Power Connectors J5 Pinout

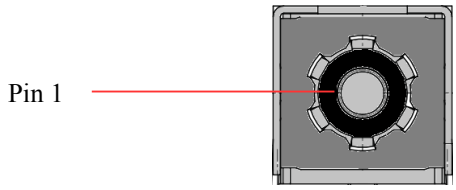

Figure 33. External Power Connectors J5 Pin Arrangement

Power Consumption

The Cardsharp power consumption varies with the system configuration, type of the used FMC module and the application software.

Voltage	Typical Current Required (A)	Typical Power (W)
12V	0.85 (Mixed Activity, no FMC module) 2.5 (Streaming with FMC module)	10 30

Table 21. Cardsharp System Typical Power Consumption

System Thermal Design

The Cardsharp system has been designed to use conduction cooling to facilitate effective heat dissipation from installed FMC module, Zynq SoC and other heat generating parts in the system. The Cardsharp chassis can dissipate 10W to 30W depending on the FMC module installed and mode of operation. In typical usage case the system will be attached to a large cold plate to ensure adequate cooling. If no cold plate is available, or if it is not sufficient for the adequate system cooling, an external fan option is available.

External Fan Connector J9

The two-pin header J9 provides power for an external cooling fan if needed.

Connector Type	CONN HDR 2POS 1.25MM STR TIN
Number of Connections	2
Connector Part Number	Hirose DF13-2P-1.25DSA(20)
Mating Connector	Hirose DF13-2S-1.25C 2POS HOUSING; Hirose DF13G-2630SCF CRIMP CONTACT (x 2)

Table 22. External Fan Connector J9 Information

Pin	Type	Signal	Note
1	I/O	XMC_FAN_PWR	12V Power for External Fan
2	I/O	GND	Ground

Table 23. External Fan Connector J9 Pinout

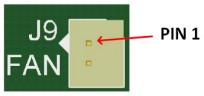

Figure 34. External Fan Connector J9 Pin Arrangement

Environmental Limits

Condition	Limits
Operating Ambient Temperature	0 to 50 C
Humidity	5 to 95 %, non condensing
Storage Temperature	0 to 100 C
Forced Air Cooling	Dependent on application
Vibration	2g, 9-200 Hz, Class 3.3 per ETSI EN 300 019-1-3 V2.1.2 (2003-04)
Shock	4g peak, Class 3.3 per ETSI EN 300 019-1-3 V2.1.2 (2003-04)

Table 24. Cardsharp Environmental Limits

GPS/IEEE-1588 Interface

The Cardsharp has built-in support for Innovative GPS and IEEE-1588 timing and synchronization modules. Only one of the two optional units can be used – it is either GPS or IEEE-1588. GPS/IEEE-1588 interface signals are available on the internal connector J2 (Hirose DF13-20DP-1.25V(55)). Connector pinout and the pin functions are given in the table 4 below.

The GPS unit provides GPS disciplined 10 MHz reference clock and also 1 PPS (pulse per second) signal for the Cardsharp system. These signals are typically used for the FMC module synchronization and data time-stamping.

Similarly, the IEEE-1588 unit, when plugged into a IEEE-1588 network, generates 10 MHz reference clock and 1 PPS signal synchronized with network's Grandmaster signals.

Note	Input / Output	Signal Name	J2 pin #	J2 pin #	Signal Name	Input / Output	Note
LVDS (P)	I	GPS_10MHZ_P	**1**	**2**	GPS_10MHZ_N	I	LVDS (N)
GROUND	N/A	GROUND	**3**	**4**	GROUND	N/A	GROUND
LVDS (P)	I	GPS_PPS_P	**5**	**6**	GPS_PPS_N	I	LVDS (N)
GROUND	N/A	GROUND	**7**	**8**	GROUND	N/A	GROUND
SERIAL TRANSMIT, 3.3V LVCMOS	O	GPS_TXD	**9**	**10**	GPS_RXD	I	SERIAL RECEIVE 3.3V LVCMOS
SERIAL 1 RECEIVE 3.3V LVCMOS	I	GPS_RX1	**11**	**12**	GPS_RESET_N	O	GPS RESET 3.3V LVCMOS
3.3V	O	3P3V	**13**	**14**	3P3V	O	3.3V
12V	O	12P0V	**15**	**16**	12P0V	O	12V
N/A	N/A	N/C	**17**	**18**	N/C	N/A	N/A
N/A	N/A	N/C	**19**	**20**	N/C	N/A	N/A

Table 25. GPS / IEEE-1588 Interface Connector J2 Signals

Connector Type	CONN HDR 20POS 1.25MM STR TIN
Number of Connections	20
Connector Part Number	Hirose DF13-20DP-1.25V(55)
Mating Connector	Hirose DF13-20S-1.25C 20POS HOUSING; Hirose DF13G-2630SCF CRIMP CONTACT (x 20)

Table 26. GPS / IEEE-1588 Interface Connector J2 Information

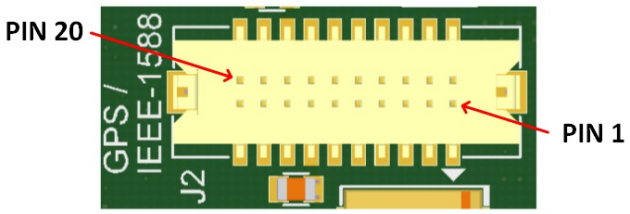

Figure 35. GPS / IEEE-1588 Interface Connector J2 Pin Arrangement

JTAG

The Cardsharp on-board JTAG chain (Zynq + FMC) can be accessed for programming and debugging purposes via connector J8. JTAG signals are also available on the P15 XMC connector allowing Cardsharp to be programmed from the XMC carrier board with built-in JTAG interface (such as Innovative SBC-Nano) without an external JTAG programmer. Figure 8 shows the Cardsharp board JTAG chain diagram. As mentioned above, JTAG signals can also be accessed on the header J4. If no FMC module is present in the system, the FMC module JTAG chain is simply bypassed. This is done without any user's interaction – the bypass mux uses the "FMC Present" signal to determine the mode of operation.

In the Cardsharp system JTAG signals are accessible via the 14-pin JTAG connector on the rear side of the case. This connector has a standard Xilinx Platform Cable USB II compatible pinout.

Table 20 below shows signal pinouts for the J8, J4 and P15 Cardsharp board connectors and also the 14-pin JTAG connector on the rear side of the Cardsharp system case.

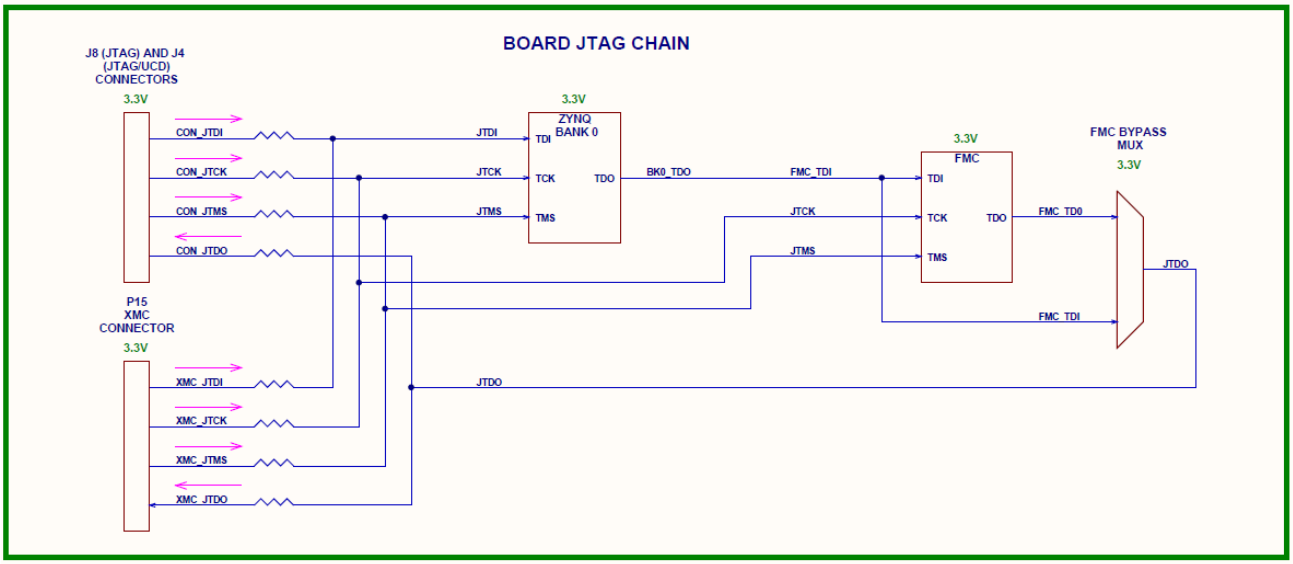

Figure 36. Cardsharp Board JTAG Chain

Signal Name	*Purpose*	*J8 pin #*	*J4 pin #*	*P15*	*Backplane JTAG Connector*	*Input / Output*
3.3V	3.3V for JTAG Programmer	**1**	**5**	**N/A**	**2**	O
TMS	JTAG TMS Signal	**2**	**2**	**C6**	**4**	I
TCK	JTAG TCK Signal	**3**	**3**	**C4**	**6**	I
TDO	JTAG TDO Signal	**4**	**1**	**C10**	**8**	O
TDI	JTAG TDI Signal	**5**	**4**	**C8**	**10**	I
GROUND	Board Ground	**6**	**6**	**Multiple pins (A2, A4, A6 etc.)**	**3, 5, 7, 9, 11**	N/A
TRST_N*	JTAG RESET/HALT	**N/A**	**N/A**	**C2**	**14**	I

Table 27. Cardsharp JTAG Signals.

Note: JTAG RESET/HALT signal is not used by Xilinx for FPGA programming. However, in Cardsharp system it can be used to reset the Zynq SoC.

Connector Type	CONN HDR 6POS 1.25MM STR TIN
Number of Connections	6
Connector Part Number	Hirose DF13-6P-1.25DSA(20)
Mating Connector	Hirose DF13-6S-1.25C 6POS HOUSING; Hirose DF13G-2630SCF CRIMP CONTACT (x 6)

Table 28. JTAG Connector J8 Information

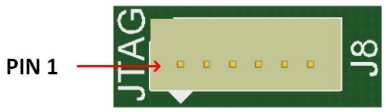

Figure 37. JTAG Connector J8 Pin Arrangement

On-Board LED Indicators

There are three green LED indicators located on the edge of the on the Cardsharp board.

LED Function	Reference Designator	Note
DONE	D7	Indicates that Zynq SoC PL configuration is completed
LED 0	D8	Function is defined in PL firmware
LED 1	D6	Function is defined in PL firmware

Table 29. Cardsharp LED Indicator Functions

BOOT MODE SWITCH

The miniature slide switch SW1 on the board used to set the board's boot mode – either JTAG or QSPI.

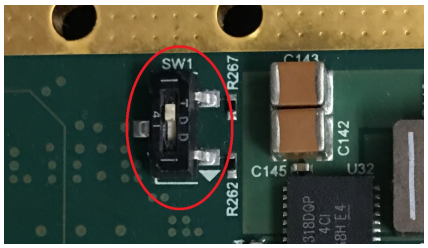

Figure 38. Cardsharp Boot Mode Switch Sw1 with the slider in JTAG Boot Mode position

Ground Loops

Four ground loops (W1..W4; two on each side of the board) are provided for debugging convenience. They can be used for attaching scope probe or multimeter ground. Note that the gold plated strips on the sides of the board are not electrically connected to the board ground, so they can't be used as a reference for making measurements on the board; their purpose is to conduct heat from the board to the system chassis ground.

FMC Module Site

The Cardsharp has a single industry-standard VITA 57 FMC (FPGA Mezzanine Card) module site. A large variety of FMC modules are available for signal processing, analog and digital IO, and communications from Innovative and other vendors. Please refer to VITA 57 standard for general FMC electrical and mechanical specifications and the FMC module documentation for information about specific modules.

The FMC module site provides high performance IO expansion for the Cardsharp. FMC site is VITA 57 High Pin Count (HPC) site.

Make sure that the Zynq SoC has the appropriate configuration file for the used FMC module before inserting the FMC module and powering up the system! Using the wrong Zynq configuration file may cause serious damage to the Cardsharp and the FMC module!

FMC Module Site	
Site	Single
Specification Compliance	VITA 57 FMC, HPC (High Pin Count)
High Speed Pairs	8 lanes (Tx/Rx pairs) connected to Zynq SoC PL; up to 6.5 Gbps max rate
Signal Pairs	80 diff pairs total LA: 34 diff pairs HA: 24 diff pairs HB: 22 diff pairs
IO Standards	LA, HA, HB; all Zynq-7 PL HR (High Range) IO standards supported
Power	3.3V @ 3A 12V @ 1A 3.3V AUX @ 0.5A Vadj = 2.5V @ 4A Note: Vadj voltage is preset at the factory to 2.5V, but optionally can be set any voltage in 1.5V to 2.5.V range. Contact factory for additional information.

Table 30. Cardsharp FMC Site Features

Note: Features listed are supported by the Cardsharp carrier board. System features are FMC module dependent.

FMC Module Site Connectivity

The FMC Module site (High Pin Count or HPC per VITA 57) is connected to the Zynq ZoC; that includes 8 High-Speed Data Pairs (DP; RX/TX). Vadj for the FMC Site is factory preset to 2.5V. Consult factory if other Vadj voltage value is required by the OEM application. J1 is the FMC Site connector.

Tables below provide FMC site connectivity details. All high-speed signal routing to FMC module is done with differential 100 Ohm pairs. The trace lengths are tightly matched in each signal group. The Zynq SoC package internal signal delays are also taken into account for the best possible delay matching in signal groups. Typically all high-speed signals are LVDS type.

The Cardsharp JTAG signals are provided to the FMC site. However, since there is usually no need for JTAG programmability on most FMC modules, typically TDI and TDO signals simply shorted on these FMC modules.

For Ground and Power pin connections refer to the VITA 57 signal table provided elsewhere in this document. Please note that the Cardsharp FMC signal nomenclature used in this chapter may not exactly match the VITA 57 standard signal naming.

Most FMC high-speed signals are connected directly to the Zynq SoC, so for safe operation it is very important to ensure that no signal ever exceeds the maximum allowed by the Zynq SoC specification for input voltage levels.

Signal Name	J1 pin	Zynq SoC pin	Signal Name	J1 pin	Zynq SoC pin

FMC_LA_0_N	G7	AH21	FMC_LA_P0	G6	AG21
FMC_LA_1_N	D9	AG20	FMC_LA_P1	D8	AF20
FMC_LA_2_N	H8	AK20	FMC_LA_P2	H7	AJ20
FMC_LA_3_N	G10	AK18	FMC_LA_P3	G9	AK17
FMC_LA_4_N	H11	AJ19	FMC_LA_P4	H10	AH19
FMC_LA_5_N	D12	AG19	FMC_LA_P5	D11	AF19
FMC_LA_6_N	C11	Y21	FMC_LA_P6	C10	W21
FMC_LA_7_N	H14	Y23	FMC_LA_P7	H13	Y22
FMC_LA_8_N	G13	AB24	FMC_LA_P8	G12	AA24
FMC_LA_9_N	D15	AA23	FMC_LA_P9	D14	AA22
FMC_LA_10_N	C15	AC23	FMC_LA_P10	C14	AC22
FMC_LA_11_N	H17	AE21	FMC_LA_P11	H16	AD21
FMC_LA_12_N	G16	AF24	FMC_LA_P12	G15	AF23
FMC_LA_13_N	D18	AG25	FMC_LA_P13	D17	AG24
FMC_LA_14_N	C19	AD24	FMC_LA_P14	C18	AC24
FMC_LA_15_N	H20	AH24	FMC_LA_P15	H19	AH23
FMC_LA_16_N	G19	AJ24	FMC_LA_P16	G18	AJ23
FMC_LA_17_N	D21	AF22	FMC_LA_P17	D20	AE22
FMC_LA_18_N	C23	AE23	FMC_LA_P18	C22	AD23
FMC_LA_19_N	H23	AK21	FMC_LA_P19	H22	AJ21
FMC_LA_20_N	G22	AK23	FMC_LA_P20	G21	AK22
FMC_LA_21_N	H26	AK25	FMC_LA_P21	H25	AJ25
FMC_LA_22_N	G25	AK26	FMC_LA_P22	G24	AJ26
FMC_LA_23_N	D24	AH27	FMC_LA_P23	D23	AH26
FMC_LA_24_N	H29	AK28	FMC_LA_P24	H28	AK27
FMC_LA_25_N	G28	AJ29	FMC_LA_P25	G27	AJ28
FMC_LA_26_N	D27	AK30	FMC_LA_P26	D26	AJ30
FMC_LA_27_N	C27	AF25	FMC_LA_P27	C26	AE25
FMC_LA_28_N	H32	AG27	FMC_LA_P28	H31	AG26
FMC_LA_29_N	G31	AG30	FMC_LA_P29	G30	AF30
FMC_LA_30_N	H35	AG29	FMC_LA_P30	H34	AF29
FMC_LA_31_N	G34	AF27	FMC_LA_P31	G33	AE27
FMC_LA_32_N	H38	AC27	FMC_LA_P32	H37	AB27
FMC_LA_33_N	G37	AE26	FMC_LA_P33	G36	AD25

Table 31. FMC Site LA Signal Group Connectivity Details

Note: Each differential signal pair matched to +/-1 psec; signals in the group matched to +/-10 psec.

Signal Name	J1 pin	Zynq SoC pin	Signal Name	J1 pin	Zynq SoC pin
FMC_HA_N_0	F5	AF14	FMC_HA_P_0	F4	AG14
FMC_HA_N_1	E3	AE13	FMC_HA_P_1	E2	AF13
FMC_HA_N_2	K8	AG12	FMC_HA_P_2	Zynq SoC	AH12
FMC_HA_N_3	J7	AD14	FMC_HA_P_3	J6	AD13
FMC_HA_N_4	F8	AH14	FMC_HA_P_4	F7	AH13
FMC_HA_N_5	E7	AE12	FMC_HA_P_5	E6	AF12
FMC_HA_N_6	K11	AJ15	FMC_HA_P_6	K10	AK15
FMC_HA_N_7	J10	AJ16	FMC_HA_P_7	J9	AK16
FMC_HA_N_8	F11	AJ14	FMC_HA_P_8	F10	AJ13
FMC_HA_N_9	E10	AH18	FMC_HA_P_9	E9	AJ18
FMC_HA_N_10	K14	AK13	FMC_HA_P_10	K13	AK12
FMC_HA_N_11	J13	AF15	FMC_HA_P_11	J12	AG15
FMC_HA_N_12	F14	AF18	FMC_HA_P_12	F13	AF17
FMC_HA_N_13	E13	AE16	FMC_HA_P_13	E12	AE15
FMC_HA_N_14	J16	AE18	FMC_HA_P_14	J15	AE17
FMC_HA_N_15	F17	AD16	FMC_HA_P_15	F16	AD15
FMC_HA_N_16	E16	AA15	FMC_HA_P_16	E15	AA14
FMC_HA_N_17	K17	AG17	FMC_HA_P_17	K16	AG16
FMC_HA_N_18	J19	AB12	FMC_HA_P_18	J18	AC12
FMC_HA_N_19	F20	AB15	FMC_HA_P_19	F19	AB14
FMC_HA_N_20	E19	AC17	FMC_HA_P_20	E18	AC16
FMC_HA_N_21	K20	AB17	FMC_HA_P_21	K19	AB16
FMC_HA_N_22	J22	AC18	FMC_HA_P_22	J21	AC19
FMC_HA_N_23	K23	AB19	FMC_HA_P_23	K22	AB20

Table 32. FMC Site HA Signal Group Connectivity Details

Note: Each differential signal pair matched to +/-1 psec; signals in the group matched to +/-10 psec.

Signal Name	J1 pin	Zynq SoC pin	Signal Name	J1 pin	Zynq SoC pin
FMC_HB_N_0	K26	R26	FMC_HB_P_0	K25	R25
FMC_HB_N_1	J25	W24	FMC_HB_P_1	J24	V23
FMC_HB_N_2	F23	V24	FMC_HB_P_2	F22	U24
FMC_HB_N_3	E22	V22	FMC_HB_P_3	E21	U22
FMC_HB_N_4	F26	R23	FMC_HB_P_4	F25	R22
FMC_HB_N_5	E25	T23	FMC_HB_P_5	E24	T22
FMC_HB_N_6	K29	V26	FMC_HB_P_6	K28	U25
FMC_HB_N_7	J28	P24	FMC_HB_P_7	J27	P23
FMC_HB_N_8	F29	T25	FMC_HB_P_8	F28	T24
FMC_HB_N_9	E28	P26	FMC_HB_P_9	E27	P25
FMC_HB_N_10	K32	N27	FMC_HB_P_10	K31	N26
FMC_HB_N_11	J31	T27	FMC_HB_P_11	J30	R27
FMC_HB_N_12	F32	W26	FMC_HB_P_12	F31	W25
FMC_HB_N_13	E31	W28	FMC_HB_P_13	E30	V27
FMC_HB_N_14	K35	W30	FMC_HB_P_14	K34	W29
FMC_HB_N_15	J34	V29	FMC_HB_P_15	J33	V28
FMC_HB_N_16	F35	U29	FMC_HB_P_16	F34	T29
FMC_HB_N_17	K38	U27	FMC_HB_P_17	K37	U26
FMC_HB_N_18	J37	P29	FMC_HB_P_18	J36	N29
FMC_HB_N_19	E34	P28	FMC_HB_P_19	E33	N28
FMC_HB_N_20	F38	U30	FMC_HB_P_20	F37	T30
FMC_HB_N_21	E37	R30	FMC_HB_P_21	E36	P30

Table 33. FMC Site HB Signal Group Connectivity Details

Note: Each differential signal pair matched to +/-1 psec; signals in the group matched to +/-10 psec.

Signal Name	J1 pin	Zynq SoC pin	Signal Name	J1 pin	Zynq SoC pin
FMC_RX_N_0	C7	AE7	FMC_RX_P_0	C6	AE8
FMC_RX_N_1	A3	AG7	FMC_RX_P_1	A2	AG8
FMC_RX_N_2	A7	AJ7	FMC_RX_P_2	A6	AJ8
FMC_RX_N_3	A11	AH9	FMC_RX_P_3	A10	AH10
FMC_RX_N_4	A15	AD5	FMC_RX_P_4	A14	AD6
FMC_RX_N_5	A19	AF5	FMC_RX_P_5	A18	AF6
FMC_RX_N_6	B17	AG3	FMC_RX_P_6	B16	AG4
FMC_RX_N_7	B13	AH5	FMC_RX_P_7	B12	AH6

Table 34. FMC Site High Speed Data Pair (DP) RX Signal Group Connectivity Details

Note: These are multi-gigabit transceiver data pairs (DP). Each differential signal pair matched to +/-1 psec; no group matching is required.

Signal Name	J1 pin	Zynq SoC pin	Signal Name	J1 pin	Zynq SoC pin
FMC_TX_N_0	C3	AK1	FMC_TX_P_0	C2	AK2
FMC_TX_N_1	A23	AJ3	FMC_TX_P_1	A22	AJ4
FMC_TX_N_2	A27	AK5	FMC_TX_P_2	A26	AK6
FMC_TX_N_3	A31	AK9	FMC_TX_P_3	A30	AK10
FMC_TX_N_4	A35	AD1	FMC_TX_P_4	A34	AD2
FMC_TX_N_5	A39	AE3	FMC_TX_P_5	A38	AE4
FMC_TX_N_6	B37	AF1	FMC_TX_P_6	B36	AF2
FMC_TX_N_7	B33	AH1	FMC_TX_P_7	B32	AH2

Table 35. FMC Site High Speed Data Pair (DP) TX Signal Group Connectivity Details

Note: These are multi-gigabit transceiver data pairs (DP). Each differential signal pair matched to +/-1 psec; no group matching is required.

Signal Name	J1 pin	Zynq SoC pin	Comment
FMC_GBTCLK0_P	D4	AD10*	FMC module generated high speed group data pair (DP) reference clock 0. Provides clock for the FMC_RX_N/P 0...3 and FMC_TX_N/P 0...3 signal pairs. A blocking 0.1uF capacitor used in the signal path on the Cardsharp carrier board.
FMC_GBTCLK0_N	D5	AD9*	
FMC_GBTCLK1_P	B20	AA8*	FMC module generated high speed group data pair (DP) reference clock 1. Provides clock for the FMC_RX_N/P 4...7 and FMC_TX_N/P 4...7 signal pairs. A blocking 0.1uF capacitor used in the signal path on the Cardsharp carrier board.
FMC_GBTCLK1_N	B21	AA7*	
FMC_CLK0_M2C_P	H4	AD18	FMC module generated clock 0.
FMC_CLK0_M2C_N	H5	AD19	
FMC_CLK1_M2C_P	G2	AA18	FMC module generated clock 1.
FMC_CLK1_M2C_N	G3	AA19	
FMC_CLK2_BIDIR_P	K4	R28*	Connected to the Zynq SoC when the FMC_CLK_DIR signal is High; connected to the Cardsharp external Reference Clock (usually 10MHz) otherwise. In typical application the reference clock is provided by the Cardsharp board to the FMC module.
FMC_CLK2_BIDIR_N	K5	T28*	
FMC_CLK3_BIDIR_P	J2	T25	FMC module or Cardsharp generated clock. Direction is dependent on the FMC module used.
FMC_CLK3_BIDIR_N	J3	U25	

Table 36. FMC Site Clock Signal Connectivity Details.

Notes: Each clock differential pair matched to +/-1 psec.

* Not a direct connection.

Signal Name	J1 pin	Zynq SoC Pin/ Direction	Comment
FMC_PRSNT_M2C_L	H2	Y20 / In	FMC module present (active low). Must be connected to ground on the FMC module.
FMC_CLK_DIR	B1	AD20* / In	FMC module signal; on the Cardsharp carrier board controls direction of the FMC_CLK2_BIDIR clock. 3.3V CMOS levels. High level sends the reference clock from Cardsharp to the FMC module. Low level sends the FMC module originated clock signal from the FMC module to the Zynq SoC.
FMC_SCL	C30	AC14* / Out	FMC System Management I2C serial clock. 3.3V CMOS levels.
FMC_CDA	C31	AH17* / Out	FMC System Management I2C serial data. 3.3V CMOS levels.
FMC_GA0	C34	N/A / In	FMC Geographical Address 0. Connected to ground on the Cardsharp carrier board.
FMC_GA1	D35	N/A / In	FMC Geographical Address 1. Connected to ground on the Cardsharp carrier board.
FMC_PG_C2M	D1	AA13* / Out	Power Good from the Cardsharp carrier board. 3.3V CMOS levels. High when 12V, 3.3V and Vadj are within tolerance.
FMC_PG_M2C	F1	AA17* / In	Power Good from the FMC module. 3.3V CMOS levels. High when when the FMC power supplies, including VIO_B_M2C, VREF_A_M2C, VREF_B_M2C, are within tolerance.
TCK	D29	N/A / Out	JTAG Clock. 3.3V CMOS levels.
TDI	D30	N/A / Out	JTAG Data In. 3.3V CMOS levels.
FMC_TDO	D31	N/A / In	FMC module JTAG Data Out. 3.3V CMOS levels.
TMS	D33	N/A / Out	JTAG Mode Select. 3.3V CMOS levels.

TRST_N	D34	N/A / Out	JTAG Reset (active low). 3.3V CMOS levels. Not used in Cadsharp system.
FMC_VREF_A_M2C	H1	Zynq SoC Banks 9, 10, 11, 12 Reference Voltage / In	FMC reference voltage associated with the signaling standard used by the Bank A (Zynq SoC Banks 9, 10, 11 and 12) data pins. Note that Zynq SoC Banks 9, 10, 11 and 12 are powered by the Vadj power supply.
FMC_VREF_B_M2C	K1	Zynq SoC Bank 13 Reference Voltage / In	FMC reference voltage associated with the signaling standard used by the Bank B (Zynq SoC bank 13) data pins. Note that Zynq SoC Bank 13 is powered by the FMC_VIO_B_M2C power supply.
FMC_VIO_B_M2C	J39, K40	Zynq SoC Bank 13 Power / In	Voltage generated by the FMC module to power bank B (Zynq SoC Bank 13) data pins.

Table 37. FMC Site Miscellaneous Signal Connectivity Details.

* Not a direct connection. A signal level shifter used on the Cardsharp carrier board.

Connector Type	VITA 57 FMC CC-HPC-10 0.050 PITCH SOCKET ARRAY ASSEMBLY
Number of Connections	400, arranged in a 40 x 10 configuration
Connector Part Number	Samtec ASP-134486-01
Mating Connector	Samtec ASP-134488-01

Table 38. FMC Connector J1 Information

Figure 39. FMC Connector J1 Pin Arrangement

Operation with an XMC Carrier

While the primary function of Cardsharp is to work as an FMC carrierwith variety of FMC modules, it can also be plugged into an XMC carrier (such as Innovative SBC-Nano or SBC-Duo) and can be used as an XMC module itself.

Cardsharp XMC Carrier Connectivity Details	
Specification Compliance*	VITA 42.3 - XMC Modules for PCI Express VITA 20 – Conduction Cooled PMC PCISIG PCI Express Gen2
Size	74 mm x 150 mm
Mounting Height (with FMC module)	25 mm
VPWR	12V +/-5%
3.3V; 3.3V AUX; +12V; -12V	Not Used
Max Power	Up to 40 W
Cooling	Conduction per VITA 20
Processor Interface	PCI Express Gen2 (5.0 Gbps); 8 lanes max, 1 lane minimum
Secondary Interfaces	7 differential or 14 single-ended digital IO lines; JTAG for in-system programmability 4-pin Trace interface for debugging QSFP controls (up to 2 cages supported; a special carrier is required)

Table 39. Cardsharp XMC Carrier Connectivity Details

Note: Mechanically Cardsharp is not fully compatible with VITA 42.3 specification.

XMC Connectors P15 and P16

	Column					
Row	**A**	**B**	**C**	**D**	**E**	**F**
1	PET0p0	PET0n0	3.3V (NC)	PET0p1	PET0n1	VPWR
2	DGND	DGND	TRST#	DGND	DGND	MRSTI#
3	PET0p2	PET0n2	3.3V (NC)	PET0p3	PET0n3	VPWR
4	DGND	DGND	TCK	DGND	DGND	MRSTO# (NC)
5	PET0p4	PET0n4	3.3V (NC)	PET0p5	PET0n5	VPWR
6	DGND	DGND	TMS	DGND	DGND	+12V (NC)
7	PET0p6	PET0n6	3.3V (NC)	PET0p7	PET0n7	VPWR
8	DGND	DGND	TDI	DGND	DGND	-12V (NC)
9	RFU (NC)	RFU (NC)	RFU (NC)	RFU (NC)	RFU (NC)	VPWR
10	DGND	DGND	TDO	DGND	DGND	GA0 (NC)
11	PER0p0	PER0n0	MBIST# (NC)	PER0p1	PER0n1	VPWR
12	DGND	DGND	GA1 (NC)	DGND	DGND	MPRESENT#
13	PER0p2	PER0n2	3.3VAUX (NC)	PER0p3	PER0n3	VPWR
14	DGND	DGND	GA2 (NC)	DGND	DGND	MSDA (NC)
15	PER0p4	PER0n4	RFU (NC)	PER0p5	PER0n5	VPWR
16	DGND	DGND	MVMRO (NC)	DGND	DGND	MSCL (NC)
17	PER0p6	PER0n6	RFU (NC)	PER0p7	PER0n7	XMC_RFU12
18	DGND	DGND	RFU (NC)	DGND	DGND	RFU (NC)
19	PEX REFCLK+	PEX REFCLK-	RFU (NC)	WAKE# (NC)	ROOT# (NC)	RFU (NC)

Table 40. Cardsharp XMC P15 Connector Pinout

Notes:

1. PET and PER groups (8 differential signal pairs each) are the PCIe Transceivers and PCIe Receivers respectively (signal direction is from the Cardsharp board). Along with the PEX REFCLK+/- pair they comprise the PCIe interface for the Cardsharp system when it is plugged into an XMC carrier; it can accommodate up to 8 lanes at PCIe Gen2 (5 Gbps) speed. Alternatively these signals can be used for a high speed QSFP interface (up to 2 cages); a special carrier, like the Innovative Cardsharp QSFP Extension board is required.

2. VPWR for Cardsharp must be 12V +/-5%. **When Cardsharp is plugged into an XMC carrier, the power is provided from the P15 connector and the external Cardsharp power must be disconnected from the Cardsharp J5 power connector to avoid system damage.**

3. 3.3V; 3.3V_AUX; +12V and -12V are not used in the Cardsharp system. All on-board power is generated from the input VPWR (12V).

4. All RFU pins are reserved per VITA42 and VITA42.3 specifications. The only exception is the RFU12 pin (F17), which can be used as an optional reset signal with some XMC carriers.

5. DGND is the system ground.

XMC P15 Signal	Direction (relative to Cardsharp)	Description	Cardsharp Usage
PET0px/PET0nx	O	PCI Express Tx +/-	PCIe or QSFP Transmitters
PER0px/PER0nx	I	PCI Express Rx +/-	PCIe or QSFP receivers
PEX REFCLK+/-	I	PCI Express reference clock, 100 MHz +/-	PCIe or QSFP reference clock
MRSTI#	I	Master Reset Input, active low	PCIe Reset
MRSTO#	O	Master Reset Output, active low	Not Used
GA0	I	Geographic Address 0	Not Used
GA1	I	Geographic Address 1	Not Used
GA2	I	Geographic Address 2	Not Used
MBIST#	O	Built-in Self Test in progress, active low	Not Used
MPRESENT#	O	Module Present, active low when present	Tied to ground
MSDA	I/O	PCI Express Serial ROM data	Not Used
MSCL	I	PCI Express Serial ROM clock	Not Used
MVMRO	I	PCI Express Serial ROM Read Only	Not Used
WAKE#	O	Wake indicator to upstream device, active low	Not Used
ROOT#	I	Root device, active low	Not Used

Table 41. P15 Connector Signal Descriptions

	Column					
Row	A	B	C	D	E	F
1	(NC)	(NC)	DIO_P0	(NC)	(NC)	DIO_P1
2	DGND	DGND	DIO_N0	DGND	DGND	DIO_N1
3	(NC)	(NC)	DIO_P2	(NC)	(NC)	DIO_P3
4	DGND	DGND	DIO_N2	DGND	DGND	DIO_N3
5	(NC)	(NC)	DIO_P4	(NC)	(NC)	DIO_P5
6	DGND	DGND	DIO_N4	DGND	DGND	DIO_N5
7	(NC)	(NC)	DIO_P6	(NC)	(NC)	QSFP_XL_SCL
8	DGND	DGND	DIO_N6	DGND	DGND	QSFP_XL_SDA
9	(NC)	(NC)	TRACE_CLK	(NC)	(NC)	TRACE_D_0
10	DGND	DGND	TRACE_CTL	DGND	DGND	TRACE_D_1
11	(NC)	(NC)	QSFP0_CLK_P	(NC)	(NC)	QSFP1_CLK_P
12	DGND	DGND	QSFP0_CLK_N	DGND	DGND	QSFP1_CLK_N
13	(NC)	(NC)	QSFP0_MODPRES_N	(NC)	(NC)	QSFP1_MODPRES_N
14	DGND	DGND	QSFP0_INT_N	DGND	DGND	QSFP1_INT_N
15	(NC)	(NC)	QSFP0_LPMODE	(NC)	(NC)	QSFP1_LPMODE
16	DGND	DGND	QSFP0_MODESEL_N	DGND	DGND	QSFP1_MODESEL_N
17	(NC)	(NC)	QSFP0_RESET_N	(NC)	(NC)	QSFP1_RESET_N
18	DGND	DGND	QSFP0_SCL	DGND	DGND	QSFP1_SCL
19	(NC)	(NC)	QSFP0_SDA	(NC)	(NC)	QSFP1_SDA

Table 42. Cardsharp XMC Secondary Connector P16 Pinout

XMC P16 Signals Used in Cardsharp System	Direction (relative to Cardsharp)	Description
DIO_P0..6 / DIO_N0..6	I/O	Digital IO, 7 differential pairs or 14 single ended. These signals originated in Zynq SoC bank 12 which is powered by Vadj. Output signal levels are determied by Vadj value and the type of logic cell used as an input and/or output. For example, if Vadj =2.5V and DIOs are configured as 14 single-ended signals, then the output cell type would be LVCMOS25 with corresponding logic levels. When driven from the outside, the signals must be within the LVCMOS25 signal levels spec. **Exceeding maximum allowed input voltage for the given Vadj and the used logic type may permanently damage the Zynq SoC and render the system non-operational.**
QSFP_XL_SCL	O	QSFP Oscillator I2C Bus Serial Clock
QSFP_XL_SDA	I/O	QSFP Oscillator I2C Bus Serial Data
TRACE_CLK	I/O	Trace Debug Bus Clock
TRACE_CTL	I/O	Trace Debug Bus Control
TRACE_D_0	I/O	Trace Debug Bus DATA 0
TRACE_D_1	I/O	Trace Debug Bus DATA 1
QSFP0_CLK_P	O	QSFP0 Clock +
QSFP0_CLK_N	O	QSFP0 Clock -
QSFP0_MODPRES_N	I	QSFP0 Module Present (Active Low)
QSFP0_INT_N	I	QSFP0 Interrupt (Active Low)
QSFP0_LPMODE	O	QSFP0 Low Power Mode
QSFP0_MODESEL_N	O	QSFP0 Mode Select (Active Low)
QSFP0_RESET_N	O	QSFP0 Reset (Active Low)
QSFP0_SCL	O	QSFP0 Serial Interface Clock
QSFP0_SDA	I/O	QSFP0 Serial Interface Data
QSFP1_CLK_P	O	QSFP1 Clock +
QSFP1_CLK_N	O	QSFP1 Clock -
QSFP1_MODPRES_N	I	QSFP1 Module Present (Active Low)
QSFP1_INT_N	I	QSFP1 Interrupt (Active Low)
QSFP1_LPMODE	O	QSFP1 Low Power Mode
QSFP1_MODESEL_N	O	QSFP1 Mode Select (Active Low)
QSFP1_RESET_N	O	QSFP1 Reset (Active Low)

XMC P16 Signals Used in Cardsharp System	Direction (relative to Cardsharp)	Description
QSFP1_SCL	O	QSFP1 Serial Interface Clock
QSFP1_SDA	I/O	QSFP1 Serial Interface Data

Table 43. P16 Connector Signal Descriptions

Connector Type	Xmc Pin Header, 0.05 In Pin Spacing, Vertical Mount
Number of Connections	114, Arranged As 6 Rows Of 19 Pins Each
Connector Part Number	Samtec ASP-103614-04
Mating Connector	Samtec ASP-103612-04

Table 44. XMC Connectors P15 and P16 Information

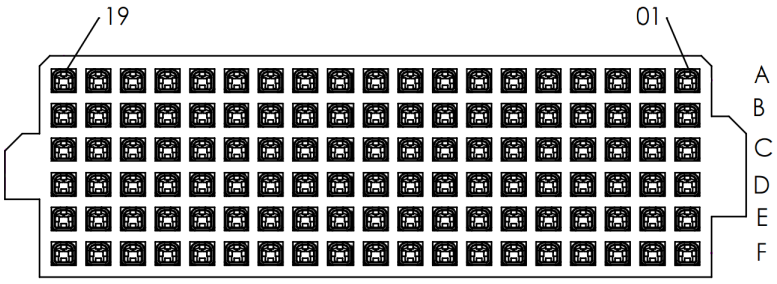

Figure 40. XMC Connectors P15 and P16 Pin Arrangement

Mechanicals

Solidworks 3D models of the Cardsharp and related products are available upon request.

Detailed drawings for mechanical design work are available through technical support.

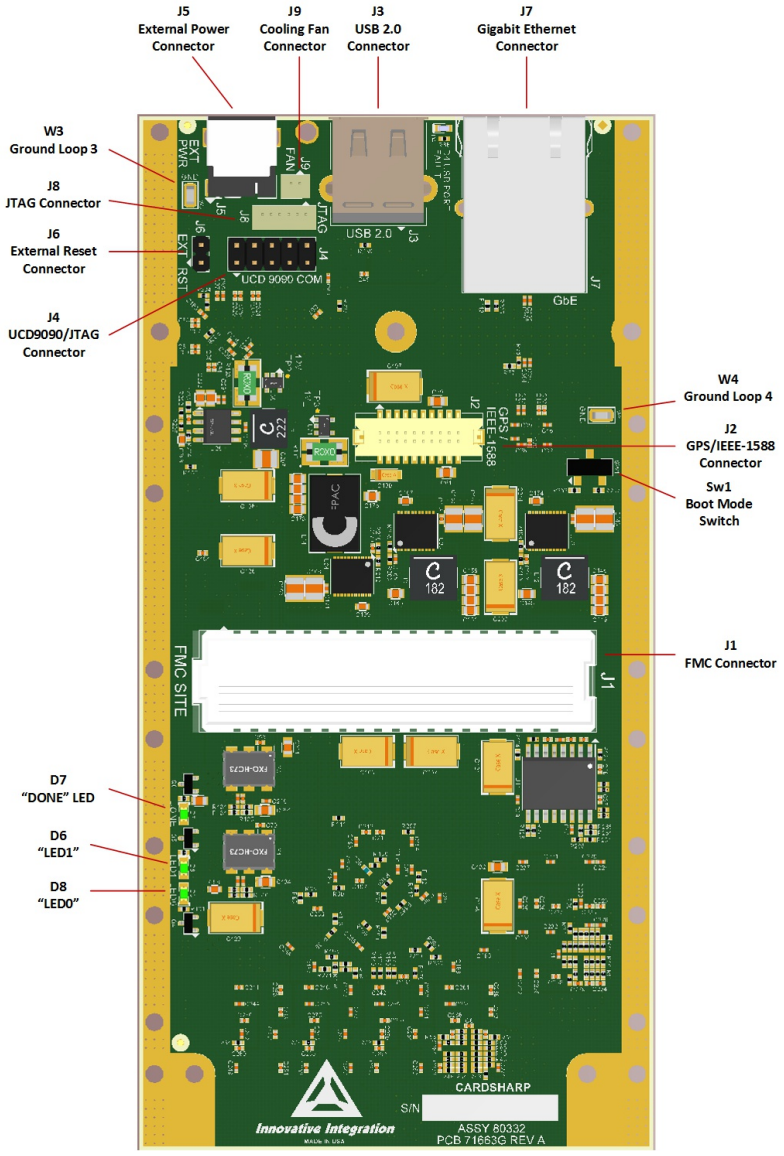

Figure 41. Cardsharp Board – FMC Connector Side View

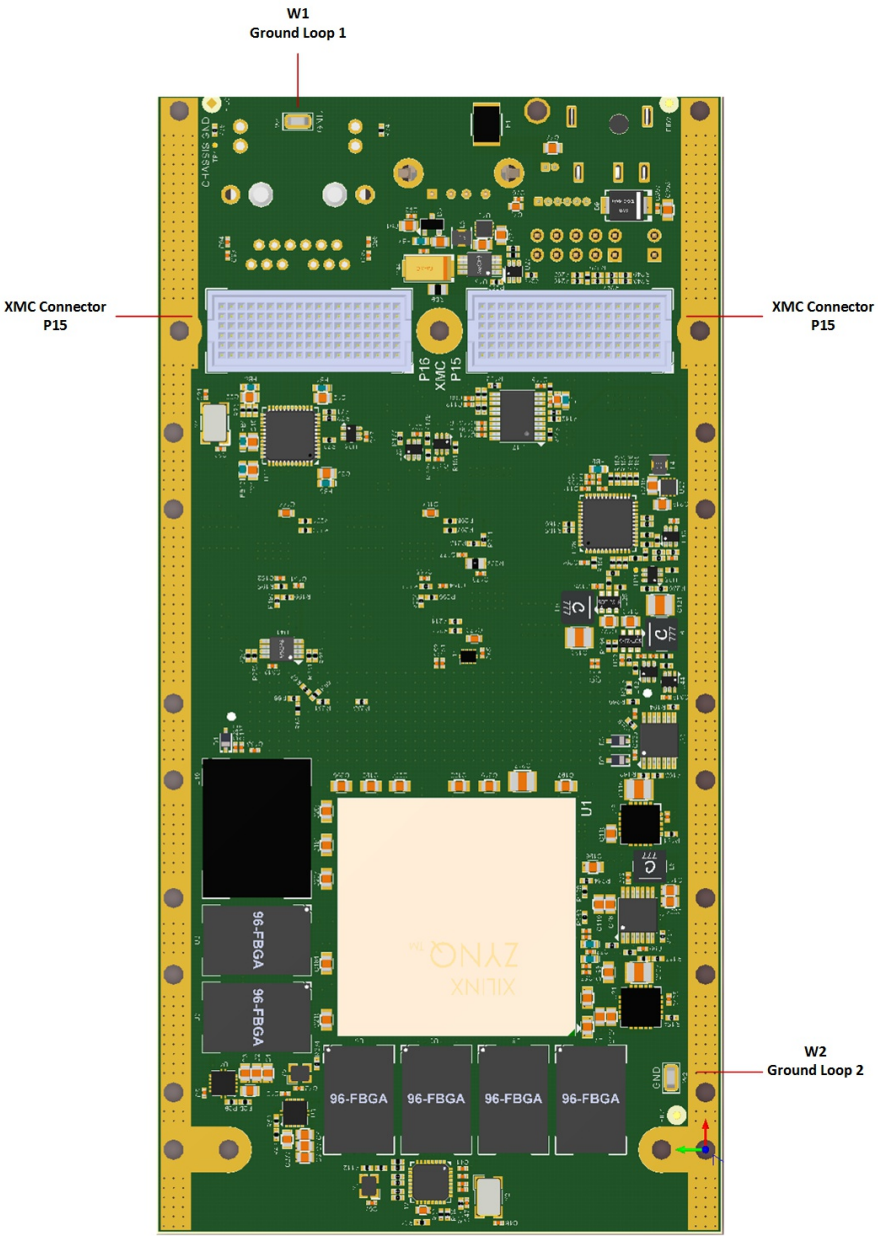

Figure 42. Cardsharp Board – XMC Connectors Side View

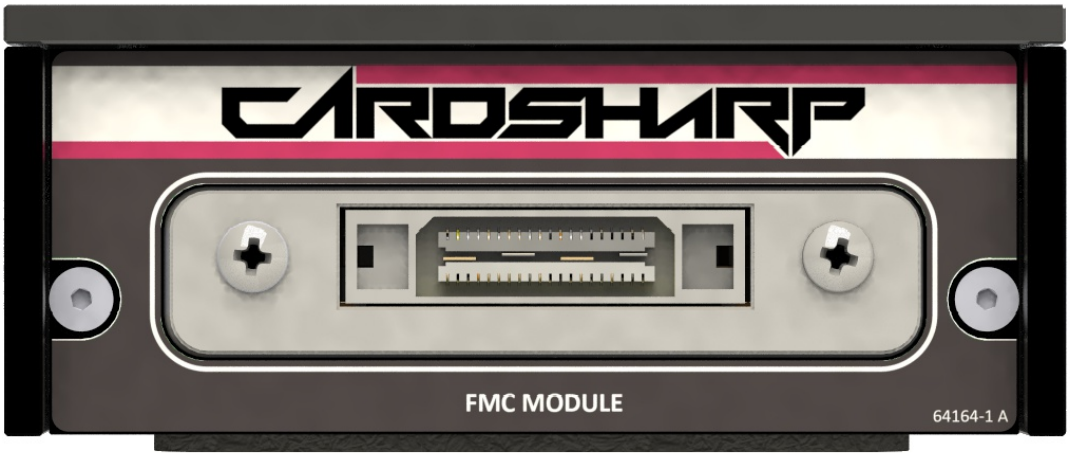

Figure 43. Standalone Cardsharp System with FMC-Servo Module installed – Front View

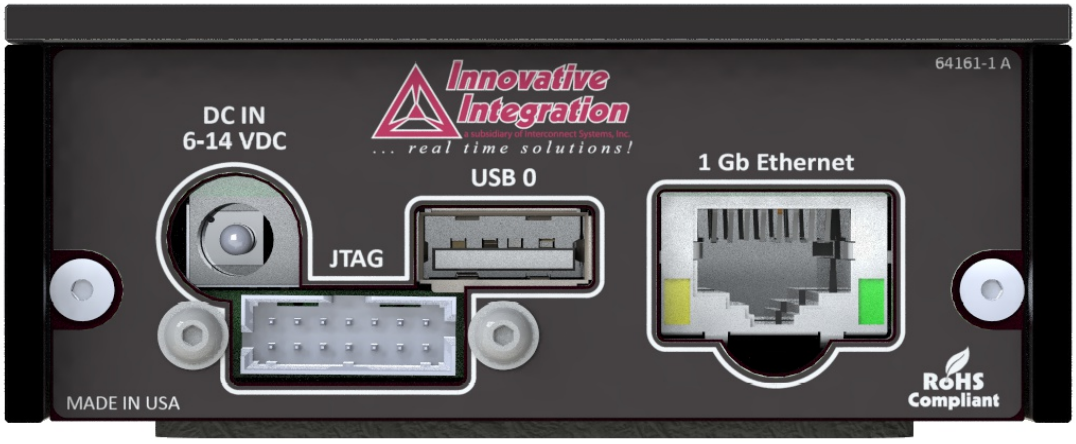

Figure 44. Standalone Cardsharp System – Rear View

Figure 45. Cardsharp / SBC-Nano system with FMC-1000 module installed – Front View

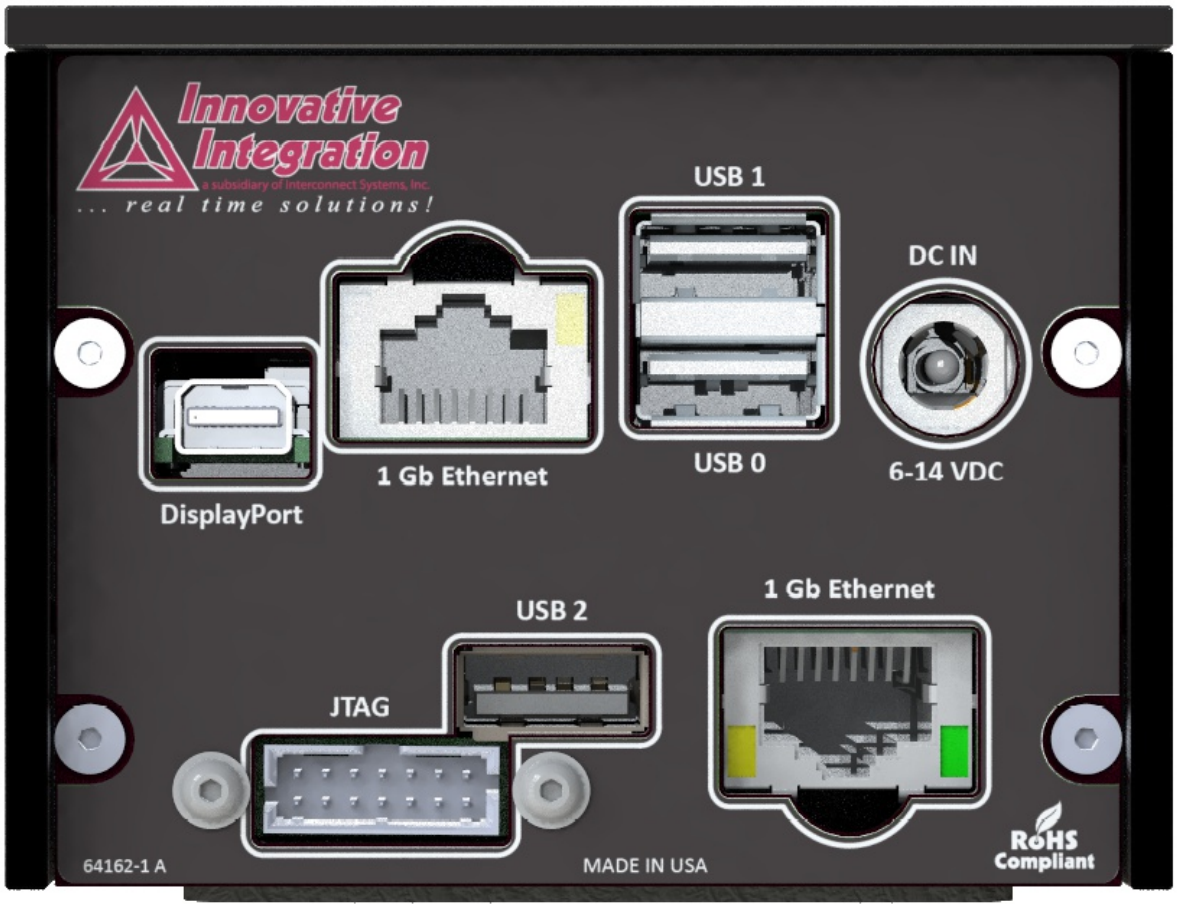

Figure 46. Cardsharp / SBC-Nano system – Rear View

System	Dimensions in mm	Dimensions in inches
Standalone Cardsharp System, no Fan option	165 x 82 x 32	6.5” x 3.3” x 1.26”
Standalone Cardsharp System, with Fan option	165 x 82 x 55	6.5” x 3.3” x 2.2”
Cardsharp/SBC-Nano System, no Fan option	165 x 82 x 52	6.5” x 3.3” x 2.05”
Cardsharp/SBC-Nano System, with Fan option	165 x 82 x 72	6.5” x 3.3” x 2.85”

Table 45. Dimensional Information for some common Cardsharp system configurations.