PAVE Framework User's Guide

V1.0

September 27, 2001

XILINX®

PAVE Framework User’s Guide

PAVE Framework User’s Guide

The following table shows the revision history for this document.

	Version	Revision
09/17/01	1.0_001	Initial Xilinx release.
09/27/01	1.0_004	Minor updates and cleanup.

Contents

Preface: About This Manual

Additional Resources 14
Typographical Conventions 17

Chapter 1: Introduction

PAVE Framework 19
Why PAVE? 20
Scalability 20
Reliability 21
Low Cost 21
Developer Productivity 22

Chapter 2: Getting Started

System Requirements 23
Hardware 23
Software 23
Installation 24
PAVE SIF Tutorial 25
Generating a Framework 25
Using the framework with Tornado 26
PAVE VxSim Tutorial 30
Using the Standard Tornado-II VxSim with PAVE 30
Running the Durango VxSim Tutorial 31
Installing to an alternate drive 33

Chapter 3: Using the PAVE SIF

IRL Register model 35
SIF Flow 37
Initial Steps 37
System Simulation 37
System Upgrade 37
System Generator 38
generatesystem.bat 40
INP File format 40
Building a Custom Framework 41
Payload Generator 42
generatepayload.bat 42
Formats for SelectMAP and JTAG 44

Chapter 4: Using the PAVE API

How the PAVE API Operates 49

Object Oriented Nature of the PAVE API 51
API Structure 53
Functional Description 55
PAVE API Classes 56
Payloads 58
Payload Header Structure 58
Payload Configuration Segment Structure 58

Chapter 5: Using PAVE for JTAG Configuration

How it operates 61
Design considerations 62
Code example 62
Rewiring the XIRL Interface Register 63

Chapter 6: Using PAVE for SelectMAP Configuration

How it operates 65
Design considerations 66
Code example 66
Rewiring the XIRL Interface Register 67

Chapter 7: Network Configuration

Network Domains 69
Definitions 70
Host, Client and Server Domains 70
Client and Server Relationships 70
Configuration Across a TCP/IP Network 71
Pull Configuration 71
Push Configuration 71

Appendix A: Architecting Systems for Upgradability with IRL

Summary 73
Introduction 73
IRL Concepts 73
What is IRL? 73
Elements of an IRL system 74
Host, Upgrade Portal, and Network Concepts 76
Target Software Concepts 77
Target Hardware concepts 77
IRL Examples 78
Basic IRL-enabled System 78
IRL in a Bridge System 80
Memory usage for storing bitstreams 81
Use of PAVE in IRL Systems 82
Available Development Platforms 82
Summary 82
Revision History 83

Appendix B: Using Durango with the MCP750 and PAVE

Durango Board 85
Durango Block Diagram 86
Durango MCP750 PAVE Implementation 86
Durango Framework 87
Code Examples 88
Resources 89
XIRL Interface Register Tables 90
Signal Name Descriptions 91

Appendix C: Durango Reference Design

Hardware Features 93
Availability 96
Additional Information 96
Durango PCB Layout 96

Appendix D: Using ADM-XRC with the MCP750 and PAVE

ADM-XRC Board 99
ADM-XRC MCP750 PAVE Implementation 100
Setup 100
INP Files for ADM-XRC 101
Applications 101
Push and Pull 103
FFT 105
ADM-XRC Payloads 109
Resources 109

Appendix E: PAVE API Summary

API Reference Manual 111

Appendix F: Glossary

Index 177

About This Manual

This manual describes the operation and use of the PAVE Framework and API . The chapters cover the following topics:

- Introduction - Details of the PAVE Framework
- Getting Started - Installation of PAVE and how to generate a framework for your hardware
- Using the PAVE API - Use of the PAVE API in your embedded system
- Using PAVE for JTAG Configuration - Reconfiguration of your hardware with JTAG
- Using PAVE for SelectMAP Configuration - Reconfiguration of your hardware with SelectMAP
- Network Configuration - Details on using the PAVE client and server for upgrading hardware and software
- Using PAVE with VxSim - Simulation of your software and hardware

In addition, this guide includes six appendices provide reference information:

- Appendix A - XAPP412 - Architecting Systems for Upgradability with IRL (Internet Reconfigurable Logic)
- Appendix B - Using Durango with the MCP750 and PAVE
- Appendix C - Durango Reference Design
- Appendix D - Using ADM-XRC with the MCP750 and PAVE
- Appendix E - PAVE API Summary
- Appendix F - Glossary

Users should have the following skills prior to using PAVE:

- Understanding of C++ programming language
- Familiarity with use of the Wind River Systems (WRS) Tornado IDE tools, including VxWorks and VxSim
- Digital Design techniques and FPGA design skills

Training on C++ is widely available through college courses, self-help books and on-the-job training. WRS offers training on their tools; links to this are listed below. Xilinx offers courses in FPGA and digital design and tutorials on using the Xilinx tools. See below for links to Xilinx training and tutorials.

Additional Resources

For additional information on PAVE and IRL, go to http://www.xilinx.com/xilinxonline. The following table lists resources you can access from this page. You can also directly access some of these resources using the provided URLs.

Resource	Description/URL
C++ Resources	
C++ on Google	http://directory.google.com/Top/Computers/Programming/Languages/C%2B%2B/
Learning C++ Today	http://cyberdiem.com/vin/learn.html
Wind River Systems Resources	
Training	WRS offers a series of regularly-scheduled classes in the use of their tools: http://www.windriver.com/training/index.html
Support	WRS offers a variety of support services for their tools: http://www.windriver.com/corporate/html/tsmain.html
Xilinx Resources	
IRL Training	The IRL team is offering the following free training, Introduction to IRL Architectures: http://www.xilinx.com/support/training/freelearning.htm
Training	Training covering Xilinx design flows, from design entry to verification and debugging along with the use of VHDL and Verilog and additional IRL training can be found at: http://www.xilinx.com/support/education-home.htm
Tutorial	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at http://support.xilinx.com/support/searchtd.htm
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which describe device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://support.xilinx.com/partinfo/databook.htm
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://support.xilinx.com/xcell/xcell.htm

Resource	Description/URL
Tech Tips	Latest news, design tips, and patch information on the Xilinx design environment http://support.xilinx.com/support/techsup/journals/index.htm
Self-Support and Problem Solvers	Self-Supportability tools and Problem Solvers as follows: How to Find Answers: How to use support.xilinx.com to solve your problem Search our Knowledge Base: Fill out this keyword search form to find the answers to your questions Answer Browser: Browse through the Answers Knowledge base by part type Configuration Problem Solver: This problem solver will automatically fix your configuration issues Install Problem Solver: This problem solver will automatically troubleshoot software installation issues Programmer Solutions: Device support list, software, and HW-130 information Virtex Power Estimator: Estimate Virtex power consumptions with our web form or download the Excel spreadsheet Technical Tips: This is *the* resource for hot issues and tips to get up and running quickly http://support.xilinx.com/support/troubleshoot

Typographical Conventions

The following typographical conventions are used in this manual:

- Red text indicates a cross-reference to information within this document. Click red text to open the specified cross-reference.
- Blue-underlined text indicates a link to a Web page. Click blue-underlined text to browse the specified Web site.
- `Courier font` indicates C++ code

  ```
  printf("%s\n", CNSTApplicationNAME);
  /* As in real C code, C code comments in this guide are enclosed in
  slash-asterisks as shown here */
  // Alternatively, comments might be on a single line like this
  ```
- Courier indicates C++ classes and API Calls:

  ```
  classIRLDevice::GetPayloadChecksum
  this->GetPayloadSize(szPayloadPath,&dwBitstreamSize,&dwModuleSize)
  ```
- Courier indicates drive letters, file names and paths, and contents of files:

  ```
  C:\
  D:\_platform_systemgenerator\_builds\_\durango.wsp
  generatedevice   Durango   VirtexIIEngine.inp  classIRLDevice
  ```
- **`Courier bold`** indicates executables, scripts, and literal commands that you enter in a syntactical statement. However, braces "{ }" in Courier bold are not literal and square brackets "[ ]" in Courier bold are literal only in the case of bus specifications, such as bus [7:0].
  - The **`generateserver.exe`** and **`generatedevice.exe`** tools create the required Tornado workspace and project files.
  - Run the **`generatesystem.bat`** batch file;
- Courier bold also indicates menu commands:

  `File` $\rightarrow$ **`Open`**
- *Italic font* denotes the following items.
  - Variables that are substituted with user-defined values

    `edif2ngd` *`design_name`*
  - References to other documents.

    See the *Virtex-E Data Sheet* for more information.
  - Emphasis in text

    If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
- The standard, IEEE 1149.1 JTAG, will be referred to as "JTAG" for simplicity.
- Angle brackets "<>" indicate something you fill in based on context, such as a path or a drive letter:

  `<path>\_build\client\simulator\default\vxWorks.exe`
- Square brackets "[ ]" denote the following items
  - A generated entry or parameter

    `_builds\server\[servername]\devices`
  - C code syntax

    `strPayloadHeader.theDeviceConfiguration[0].dwBitstreamSize,`
  - Bus specifications

    SMAP_D[2:0]

Chapter 1

Introduction

PAVE Framework

The PAVE Framework is an embedded applications development environment that can be used in the design and deployment of upgradable systems applications. It consists of two parts as shown in Figure 1-1:

- PAVE SIF (System Integration Framework) - This is a software environment that ties leverages the Wind River System Tornado II Integrated Development Environment (IDE), the Xilinx Foundation tools, and Microsoft Visual Studio tools.
- PAVE API - A collection of C++ classes and object models that abstract an implementation of a Xilinx FPGA, called the IRL-enabled Device implementation.

The PAVE Framework treats the programmable hardware as an object within the system, similar to software objects used in C++. As a result, applications that are written using PAVE tend to be highly object oriented, modular, and extremely upgradable. A single module can be changed without replacing the entire framework.

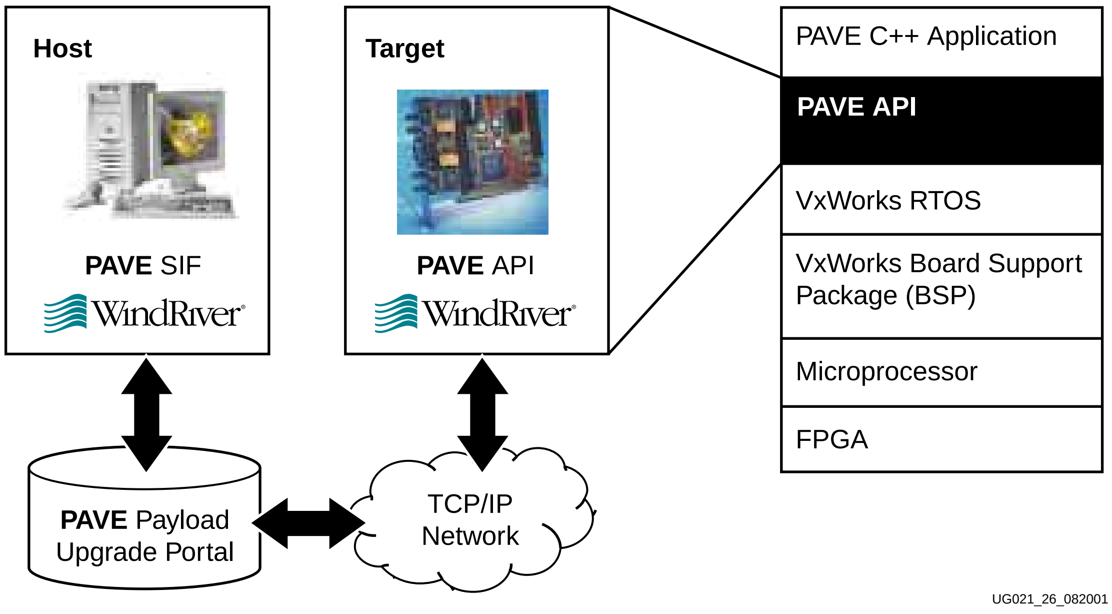


Figure 1-1: **Xilinx PAVE in IRL-enabled System**

PAVE has been written in C++, an object oriented programming language that abstracts the various underlying software components. This abstraction allows you to look at the larger picture without spending time revisiting the minor underlying details whenever your code changes. This allows a modular approach to fixing or upgrading code. This approach can be likened to repairing a car; if your alternator fails, you replace just the alternator, and not the whole engine.

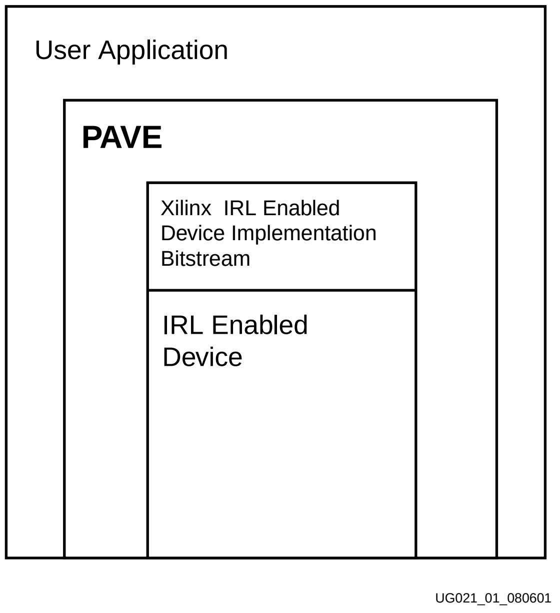


Figure 1-2: **PAVE API, Application and IRL-Enabled Device**

With PAVE, you can update any single module without replacing the complete system software, as in our example above. As seen in Figure 1-2, every part of the system is a module that can be changed. Since the relationship between the software and hardware are clearly defined on a register-level basis, your FPGA designs and the software drivers for them can be easily upgraded without having to change your entire system.

Why PAVE?

Xilinx created PAVE so that customers could more easily create field upgradable products. When we say 'customers', we mean not only the traditional FPGA designer who normally uses our devices but also the software application developer and system architect who are also responsible for designing the whole system. We understand that design of the upgradable product not only involves the FPGA based target (the traditional responsibility of the hardware design and firmware developer) but also the network-based delivery of the payload modules to the target plus the creation / integration of the software on the host development tools. Xilinx has taken a broad view of the problems to be solved in designing field upgradable systems and is creating solutions that address different requirements of the upgradable system specification.

Customers have told us that these requirements include:

Scalability

Customers want a tool that they can use for many different uses. This lowers support costs and training time.

- Upgrade application design: the ability to use a single design methodology that can span the spectrum of different memory systems as well as microprocessors is an obvious benefit.

The PAVE API enables a customer's C++ upgrade application to operate with any type of memory and 32 bit microprocessor which is supported by a Wind River VxWorks Board Support Package (BSP). This C++ application therefore does not need to be re-written when a new hardware platform is targeted. This includes flash, RAM and hard disk memory as well as all of the popular 32 bit microprocessor architectures, making the upgrade design extensible across a wide range of end products.

Also, the PAVE API is provided in source form so that a user can modify it even further to more closely match their needs.

- FPGA configuration: a single design methodology that can program the FPGA either via SelectMAP or JTAG.

  The PAVE API abstracts the SelectMAP and JTAG operation through software based state machines and objects. The engineer does not have to worry about the specific functional and timing issues involved with using these configuration methods.
- Updating both hardware and software: customers tell us they update software in their systems as frequently, if not more often, than the hardware. A single methodology for both would be the most efficient.

  The payload created by the PAVE System Integration Framework (SIF) is defined to contain both software modules in addition to FPGA bitstreams. Either or both can be present in a payload.
- Performance: customers want to adjust the speed of the upgrade process to match their system requirements. The PAVE API performance can scale with the processor speed and it works over any type of local or system bus.
- Upgrade architecture: some applications require all targets to be updated on the initiative of a central controller (a 'push' update) or for each unit to be updated when it chooses (a 'pull').

  The PAVE Framework is applicable to both equally well. The PAVE SIF and API can be used in either scenario.

Reliability

All customers have different ways to guarantee that their products are reliable and of high quality. An upgrade tool needs to be flexible and powerful enough that it can enable highly reliable upgrades.

The PAVE API enables applications that can control which version of either an FPGA upgrade or a software upgrade is to be used. Applications can also be written monitoring the success of the upgrade and what to do thereafter. These customer written applications naturally are tuned to the very specific needs of the product. For instance, a 'rollback' application can be written that monitors the status of the upgrade operation, and based upon its results, rolls back to the previous FPGA configuration. The API supplies the methods to do this.

Low Cost

Gaining the benefits of field upgradability without adding to the production cost of the end product is a key goal. The ability to easily leverage the resources which are already in the system - the microprocessor, memory, operating system - to create an upgradable design can accomplish this goal of low cost.

The PAVE API is software that is run on the Target system processor. It's only hardware requirement is a 32 bit register between the microprocessor and FPGA, that the API talks to.

Engineering development costs are also lowered.

Code described in the PAVE framework can be reused across products because the PAVE SIF can easily import and export project files. Also, the PAVE API customer application can quickly be modified for new hardware in a new product. Both of these combine to give lowered project development costs.

Developer Productivity

The upgrade process is not just about manipulating the target system, it is also about the ease of creating the upgrade payload and managing upgrades in the future. For an application developer, they want simple ways for their software application to access the devices in the upgrade process. Also, how many companies have experienced the agony of losing a designer and then having to re-create the work so that the product can be upgraded?

The PAVE SIF contains developer utilities as well as an easy to use directory structure to help the developer quickly create applications and manage them for years to come, as the product evolves.

Chapter 2

Getting Started

This chapter describes installing the PAVE SIF and generating a sample framework. The Durango example design is used for this sample to both insure your tools are correctly installed and to give you a short tutorial in generating the framework. Details on how to customize the framework for your design can be found in Chapter 3, **Using the PAVE SIF**.

System Requirements

The following list details the requirements for the PAVE Framework development platform

Hardware

Host System

- 64 MB RAM (128 MB recommended).
- 300 MB disk space for typical installation.
- A CD-ROM for installation.
- Intel Pentium II or better; Intel Pentium III recommended.
- A network interface card with an Ethernet TCP/IP connection.
- Windows NT or Windows 2000
- Netscape 4.7 or Internet Explorer 4.0 or later web browser.

Target System

- VxWorks 5.4 or higher
- VxWorks compatible Board Support Package (BSP) supporting a 32-bit architecture.

Software

- WRS Tornado II
- Microsoft Visual Studio v6.0
- Xilinx Foundation Software, 3.1i or higher

Installation

The Xilinx PAVE Framework is only available thought web distribution. The PAVE Framework can be found at:

http://www.xilinx.com/xilinxonline/pave_dl/finaldwnldpage.htm

From this page you can download the latest release. Release notes for the release can be found at this location.

The IRL home page is at:

http://www.xilinx.com/irl

You will need to be registered with the Xilinx web site and agree to the PAVE Software License to gain access to the files. If you are not currently registered for the XIlinx website, follow this link:

http://www.xilinx.com/xlnx/xil_reg_profile.jsp

and click on the "New customers please register" link.

After downloading the PAVE files from the aforementioned web address, complete the following steps to install a new PAVE Framework.

1. Ensure that you have enough disk space to install the new PAVE Framework.
2. Backup any existing PAVE Framework that you are currently developing in prior to installing newer versions. The new PAVE Framework can be installed directly over your existing framework after you have backed up your previous version.
3. Review the online release notes for the version of PAVE you are using. The release notes can be found on the web page where you downloaded the PAVE Framework.
4. Unzip the PAVE zip file to your `D:\` drive. PAVE must be installed in the root of your drive (e.g. `D:\`). PAVE has some very long path names; making them much longer will cause the `.bat` files to fail. Spaces in the pathnames are *not* allowed.

   If you are must use a drive letter beside `D:\`, you will need to modify the WRS workspace files. See **Installing to an alternate drive** for details on this procedure.
5. After completing the previous step, you will now have three new directories in `D:\`
   - `D:\_platform_systemgenerator`
   - `D:\_platform_mcp750durango`
   - `D:\_platform_mcp750admxrc`

   The first directory, `D:\_platform_systemgenerator`, is a template that you will use in the next two sections of this chapter to generate a framework and perform a system verification. Under `D:\_platform_systemgenerator`, you should see directory tree that resembles the tree illustrated in Figure 2-1.

   The second and third directories are pre-generated frameworks that target specific hardware configurations.

   The `D:\_platform_mcp750durango` targets the Motorola MCP750 and Durango IRL reference design. More details on this can be found in Appendix B, **Using Durango with the MCP750 and PAVE**.

   The `D:\_platform_mcp750admxrc` targets the Motorola MCP750 and Alpha Data ADM-XRC design. More details on this can be found in Appendix D, **Using ADM-XRC with the MCP750 and PAVE**.

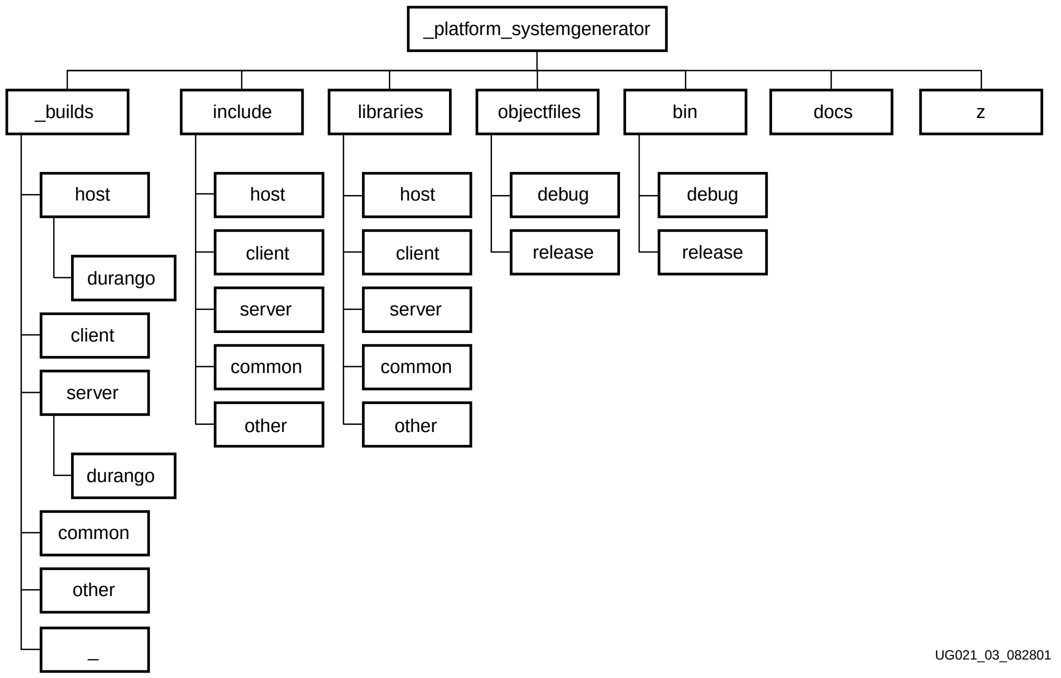


Figure 2-1: **System Generator Tree Diagram**

PAVE SIF Tutorial

The following sections present a brief tutorial on using the PAVE SIF, using the Durango example. In addition to showing you the basics of how to use the SIF, this validates your setup and tool installations. The following screenshots appear best in Acrobat or Acroread when scaled to 134%

Generating a Framework

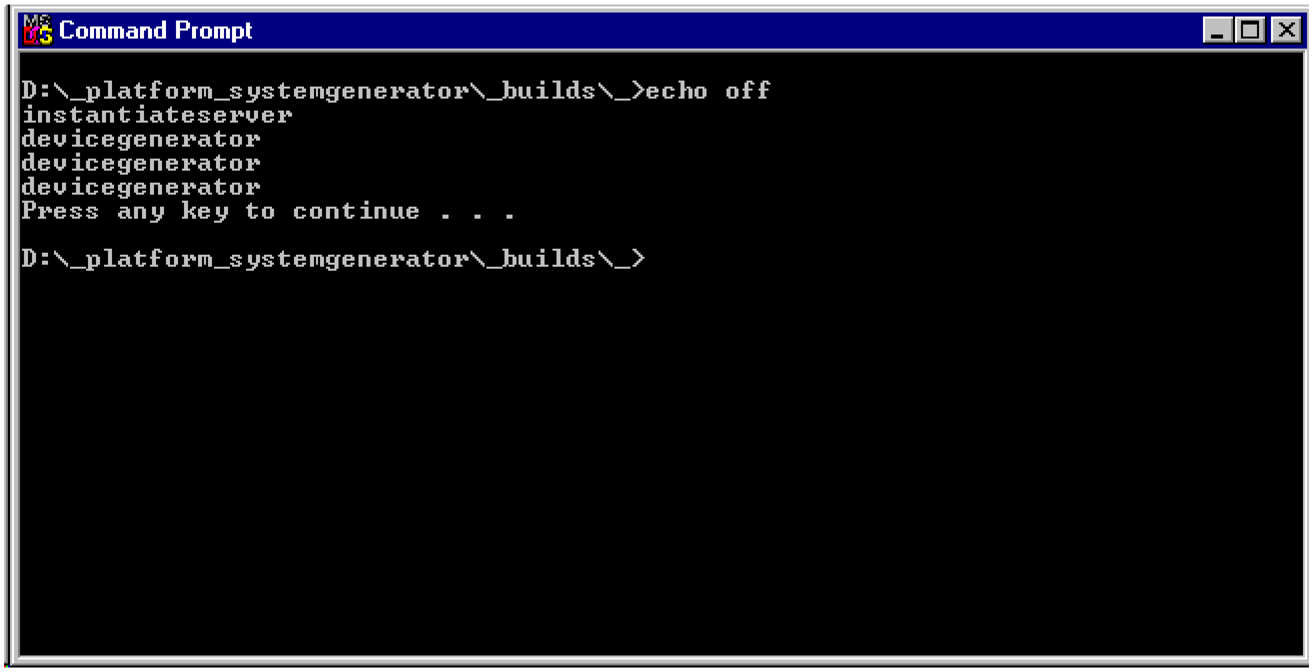
1. From your command prompt, change directory to:

   `<drive letter>:\_platform_systemgenerator\_builds\_`

   In this directory you will find several executable and `.bat` files along with several `.inp` files. These four `.inp` files are input files to the code generation tools. Several things you should note about this:

   - The system description file, `durango.inp`, specifies the programmable devices in the system, referencing the other three `.inp` files.
   - The device specification `.inp` files (`VirtexIIEngine.inp`, `VirtexIIBridge.inp`, and `ControlCPLD.inp`) each have a specification of the registers in the corresponding device. You can specify the register programming model for a device in these files. Details on the .inp file format can be found in Chapter 3, **Using the PAVE SIF**.

2. Run the **`generatesystem.bat`** batch file; your screen output should look like Figure 2-2. If you run it a second time it will complain about overwriting the `durango.wsp` file. This is to prevent you from unintentionally overwriting the `durango.wsp` file as you may have made some changes to it in the Tornado II tool. To solve this manually delete the file, `durango.wsp` prior to running the **`generatesystem.bat`** file.



```
D:\_platform_systemgenerator\_builds\_>echo off
instantiateserver
devicegenerator
devicegenerator
devicegenerator
Press any key to continue . . .

D:\_platform_systemgenerator\_builds\_>
```

Figure 2-2: **Running the System Generator**

WRS Tornado workspace files use absolute paths instead of relative paths. Once you generate the system, the generated workspace files (.wsp) will be "hard-coded" to that directory. If you need to change the path, you must regenerate the system.

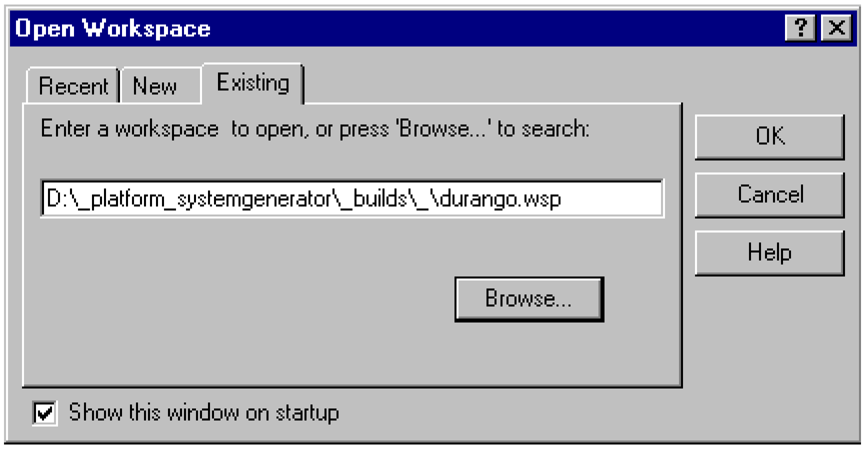
3. Go back to the `_builds\_` directory. You will now find that both a tornado and a VisualStudio workspace files have been generated, respectively named `durango.wsp` and `durango.dsw`.

Using the framework with Tornado

Once you have created the system components that are resident in your system framework, you can build the PAVE System Integration Framework. The steps below will guide you through the build process.

1. Start the Wind River Systems Tornado II Integrated Development Environment (IDE).

2. Select **`File->Open Workspace`**. You will be presented with a dialog box that resembles the dialog box shown in the Figure 2-3 below.



Enter a workspace to open, or press 'Browse...' to search:

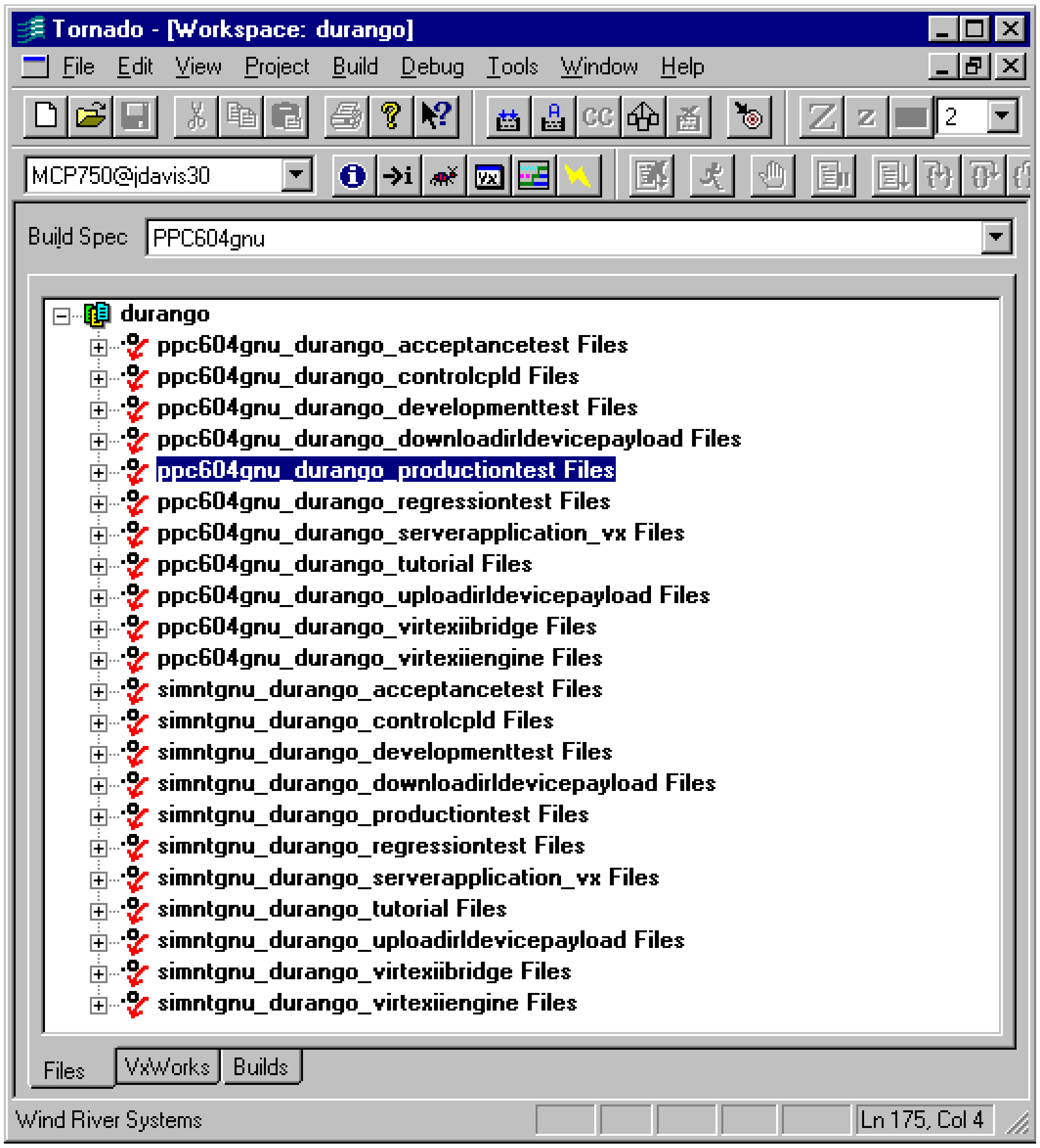
D:_platform_systemgenerator_builds_\durango.wsp

☑ Show this window on startup

Figure 2-3: **Opening the durango.wsp file**

3. Select the `durango.wsp` file in the `_builds\_` directory. This is the VxWorks workspace file for the framework that was generated in the previous section. Click the OK button and the workspace will open as shown in Figure 2-4. This window is the Tornado II IDE workspace window. If not already selected, select the Files tab in the workspace window.
4. The **`generateserver.exe`** and **generatedevice.exe** tools create the required Tornado workspace and project files.There are two sets of project files shown in the workspace, the `ppc604gnu` and the `simntgnu` workspace files. These represent the downloadable and simulatable build specifications respectively.

   Within these sets of project files you will see project trees for the `VirtexIIEngine`, `VirtexIIBridge`, `ControlCPLD`, and the several test application project files. The `VirtexIIEngine`, `VirtexIIBridge`, and `ControlCPLD` projects build the software objectfiles for the IRL-enabled devices that are resident on the Durango system component.
5. Generate the dependencies by right clicking on `ppc604gnu_durango_controlcpld` and selecting the **`Dependencies`** choice. Ensure that all of the checkboxes are selected. Click on the **`Advanced`** button and select the **`Quick Scan`** option as seen in Figure 2-6. Click **`OK`** for the Advanced dependencies and again for the original dependencies dialog. The dialog box in Figure 2-7 shows the dependencies are being recalculated for the elements of the selected build.



```
durango
  ppc604gnu_durango_acceptancetest Files
  ppc604gnu_durango_controlcpld Files
  ppc604gnu_durango_developmenttest Files
  ppc604gnu_durango_downloadirldevicepayload Files
  ppc604gnu_durango_productiontest Files
  ppc604gnu_durango_regressiontest Files
  ppc604gnu_durango_serverapplication_vx Files
  ppc604gnu_durango_tutorial Files
  ppc604gnu_durango_uploadirldevicepayload Files
  ppc604gnu_durango_virtexiibridge Files
  ppc604gnu_durango_virtexiiengine Files
  simntgnu_durango_acceptancetest Files
  simntgnu_durango_controlcpld Files
  simntgnu_durango_developmenttest Files
  simntgnu_durango_downloadirldevicepayload Files
  simntgnu_durango_productiontest Files
  simntgnu_durango_regressiontest Files
  simntgnu_durango_serverapplication_vx Files
  simntgnu_durango_tutorial Files
  simntgnu_durango_uploadirldevicepayload Files
  simntgnu_durango_virtexiibridge Files
  simntgnu_durango_virtexiiengine Files
```

Figure 2-4: **Durango Workspace in Tornado II**

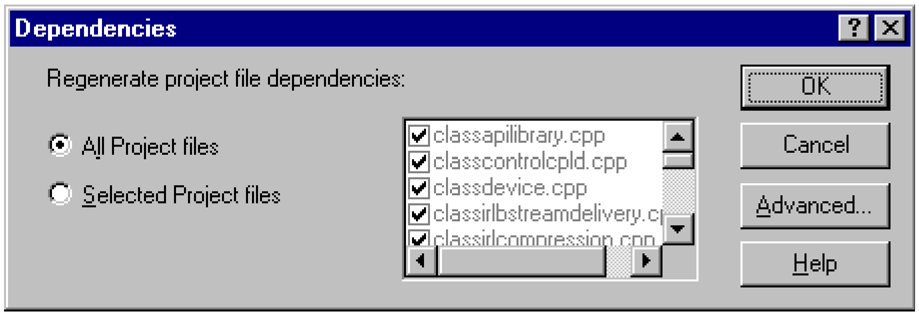
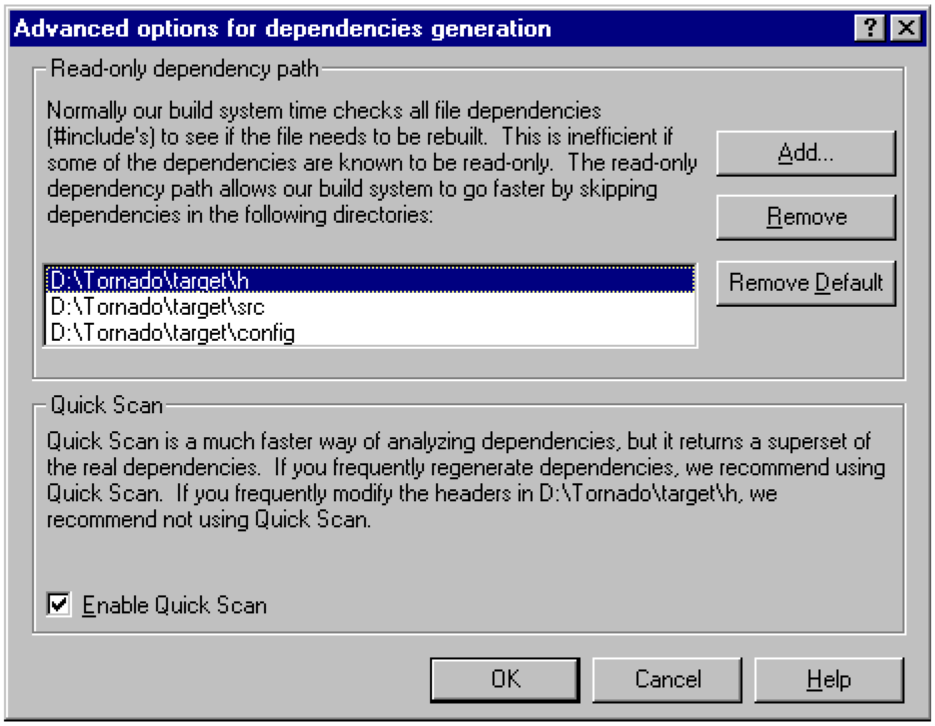


Figure 2-5: **Dependencies Dialog**



Advanced options for dependencies generation

Read-only dependency path

Normally our build system time checks all file dependencies (#include's) to see if the file needs to be rebuilt. This is inefficient if some of the dependencies are known to be read-only. The read-only dependency path allows our build system to go faster by skipping dependencies in the following directories:

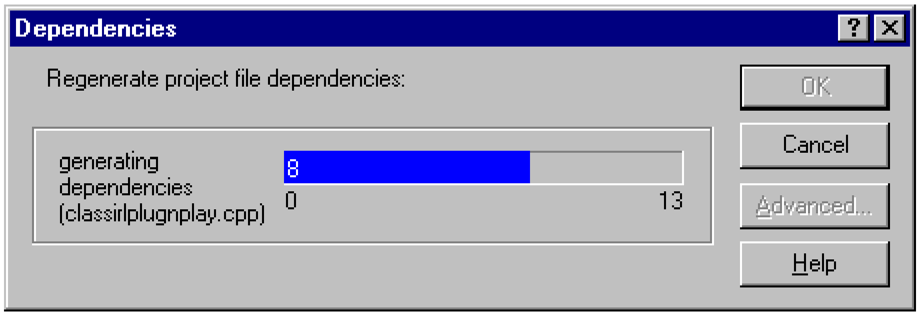
D:\Tornado\target\h
D:\Tornado\target\src
D:\Tornado\target\config

Quick Scan

Quick Scan is a much faster way of analyzing dependencies, but it returns a superset of the real dependencies. If you frequently regenerate dependencies, we recommend using Quick Scan. If you frequently modify the headers in D:\Tornado\target\h, we recommend not using Quick Scan.

[✓] Enable Quick Scan

Figure 2-6: **Advanced Dependencies Dialog with Quick Scan enabled**



Dependencies

Regenerate project file dependencies:

generating dependencies (classirlplugnplay.cpp)

Figure 2-7: **Updating Dependencies**

6. Once the Dependencies have been generated, you can build the `.out` file. Right click on `ppc604gnu_durango_controlcpld` and select the **`Rebuild All`**. A build window will pop up in Tornado, if it completes successfully, it will say “Done” at the end.

After you have successfully completed this process, you will be able to follow the normal Tornado procedures to load and run the object modules for the PAVE Framework that you have generated.

PAVE VxSim Tutorial

The `_platform_mcp750durango` framework provided with PAVE contains project files for simulating a solution using the Tornado-II VxSim simulator. The simulator is an excellent way to become familiar with the PAVE API and the development tools in the Wind River Systems Tornado-II Integrated Development Environment. The Durango workspace shown in Figure 2-4 lists projects prefixed with `ppc604gnu` and also `simntgnu`. The `simntgnu` prefixed projects can be built and downloaded to VxSim just as if one were targeting physical hardware. Using PAVE and VxSim one can define the register set of a device under development and begin to exercise the software and hardware interfaces before physical devices are available.

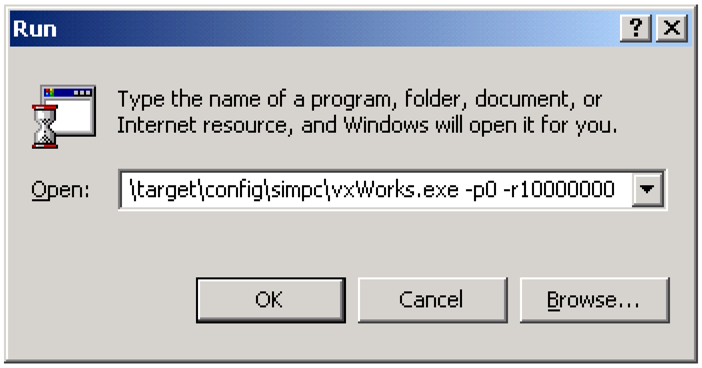
Tutorial code within the Durango workspace presents a simple but generalizable register based data flow example as well as a JTAG configuration flow example.

Using the Standard Tornado-II VxSim with PAVE

The standard VxSim simulator shipped with Tornado-II does not provide networking support, however, together with the Tornado-II debugger it can be used to trace the flow of code within the simntgnu_durango_tutorial project of the Durango framework.

1. Bring up the Tornado-II IDE by double clicking on the workspace file `\_builds\_\durango.wsp.`
2. Build the following projects shown in the Durango workspace by right-clicking on them in the Files view and selecting the **`ReBuild All`** option:
   - `simntgnu_durango_contolcpld`
   - `simntgnu_durango_virtexiibridge`
   - `simntgnu_durango_virtexiiengine`
   - `simntgnu_durango_tutorial`
3. Launch the standard VxSim simulator from the Tornado install directory by selecting **Start->Run** from the Windows task bar and entering the following command line for execution

```
<Drive Letter>:\Tornado\target\config\simpc\vxWorks.exe -p0 -r10000000
```

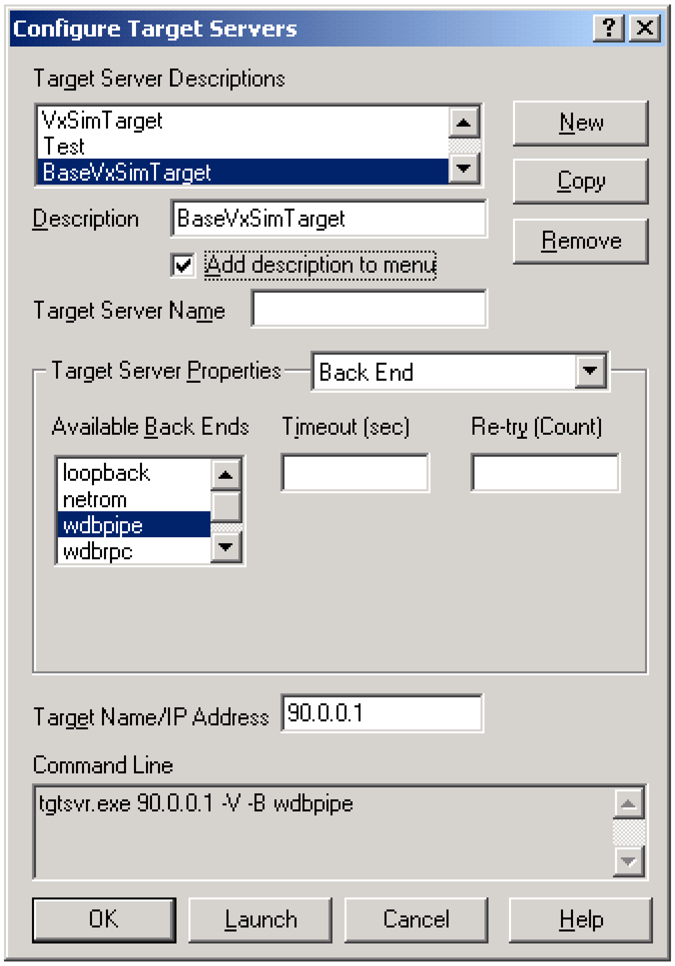

Run

Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: \target\config\simpc\vxWorks.exe -p0 -r10000000

Figure 2-8: **Running VxWorks for the Simulator**

4. Configure the target server by selecting **Tools -> Target Server->Configure** and configuring as shown in Figure 2-9. The description you put in this box will appear on your launch dialog box when you use your Target Server in the future. Choose OK instead of Launch.
5. Download each of the files listed in step 2 to the simulator using the right click and selecting the **`Download "<filename>"`** option.


Configure Target Servers

Target Server Descriptions

VxSimTarget
Test
BaseVxSimTarget

Description: BaseVxSimTarget

[✓] Add description to menu

Target Server Name:

Target Server Properties: Back End

Available Back Ends: loopback, netrom, wdbpipe, wdbrpc (wdbpipe selected)

Timeout (sec):

Re-try (Count):

Target Name/IP Address: 90.0.0.1

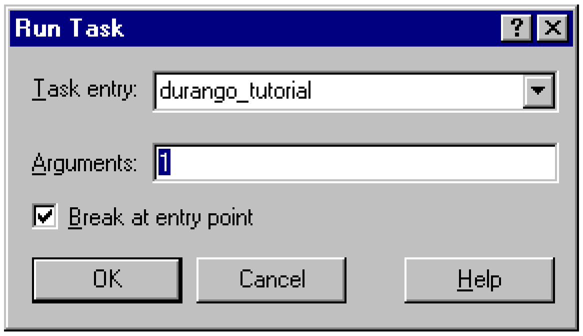
Command Line:

```
tgtsvr.exe 90.0.0.1 -V -B wdbpipe
```

Figure 2-9: **Target Server Configuration for VxSim**

Running the Durango VxSim Tutorial

You are now ready to run the `simntgnu_durango_tutorial`. Launch the Debugger by first clicking on the “bug” icon in Tornado, and then clicking on the “running man” icon”. This brings up the Run Task dialog shown below. We will step through the tutorial code beginning at the function "durango_tutorial".


Run Task

Task entry: durango_tutorial

Arguments: 1

[✓] Break at entry point

Figure 2-10: **Run Task dialog**

The dialog box input is the equivalent to a executing an

```
-> sp (durango_tutorial, 1)
```

command from the Tornado Shell to spawn a new task with the specified entry point. However, by launching a task through the debugger, one can trace through the code using all of the Tornado Debugger functions such as step into, step over, run until a breakpoint, and observe the system through watch variables and dumping memory contents.

The `durango_tutorial` code shows how to access a `classRegister` object, how to trap accesses to the register, and also how a JTAG configuration is performed. Upon entry, the durango_tutorial simulation code is steered by the `ATTRclassDurangoSIMULATIONMODE` compiler flag defined in all the Durango `simntgnu` projects. When simulation is enabled, memory is allocated to represent the address space of a board instead of discovering and memory mapping any physical hardware. Along with the flag mentioned above the `ATTRclassSimulationENABLESIMULATION` compiler flag allows register operations to be trapped and user code placed in the appropriate device `usersim` file to be executed. Sample code in

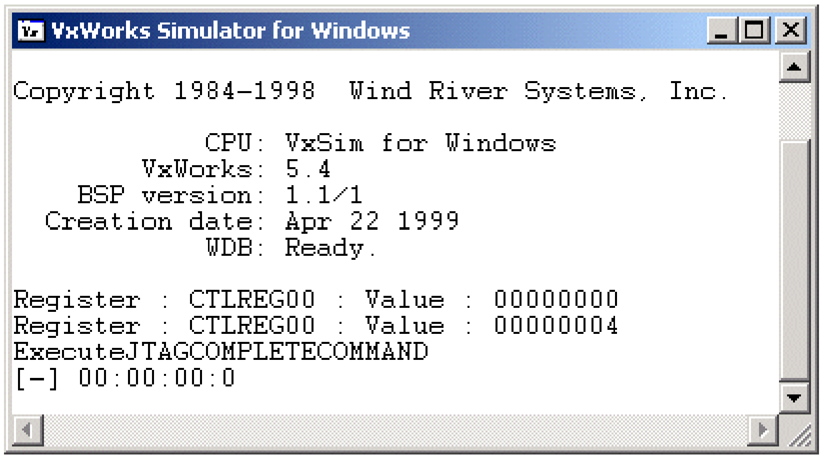
`\_builds\server\durango\devices\virtexiiengine\virtexiiengineusersim.cpp`

is delivered with the Durango framework to multiply any uploads to the `CTLREG00` register (defined in the inp file) by two and write the result back to `CTLREG00`. The following code in `\_builds\server\durango\tests\tutorial\tutorial.cpp` exercises the register level accesses and trapping:

```
ptrServer->ptrDesignatedDevice->DisplayRegisters();
ptrServer->ptrDesignatedDevice->ptrCTLREG00->Upload( 2 );
```

Output from the register level operations can be viewed in the VxWorks Simulator window as shown in Figure 2-11. Using the `classRegister` objects with VxSim enables interface testing without physical hardware.

Along with the simple register example, durango_tutorial is set up to perform a JTAG configuration. An IRL payload file, `mcp750durango_tutorial_jt.irl,` is read in and split into it's constituent loadable software module and bitstream. In the JTAG case the bitstream is in the form of an XSVF file which is parsed and sequenced through the XIRL interface register by the tutorial code. Using the debugger and setting breakpoints, one can trace through the JTAG configuration process. The PAVE application developer needs only call the API methods exposed in the tutorial to initiate the underlying software state machines. Upon successful completion of the JTAG process the `JTAGCOMPLETECOMMAND` is printed to the Simulator window (Figure 2-11).



```
Copyright 1984-1998  Wind River Systems, Inc.

            CPU: VxSim for Windows
        VxWorks: 5.4
    BSP version: 1.1/1
  Creation date: Apr 22 1999
            WDB: Ready.

Register : CTLREG00 : Value : 00000000
Register : CTLREG00 : Value : 00000004
ExecuteJTAGCOMPLETECOMMAND
[-] 00:00:00:0
```

Figure 2-11: **VxSIm Window**

Installing to an alternate drive

The WRS Tornado tools depend on absolute paths to locate files. The directories, `_mcp750durango` and `_mcp750admxrc` were generated with the `D:\` as their location and they must be run from the `D:\` or Tornado will not recognize them.

If you are unable to run them from the `D:\` drive here is the procedure to fix this.

- Durango - The `_platform_systemgenerator` directory is an ungenerated version of the `_platform_mcp750durango` directory. Copy the `_platform_systemgenerator` to the desired drive and run the system generator to create the desired framework.
- ADM-XRC - To move the _platform_mcp750admxrc to another drive you must edit all the `.wsp` and `.wpj` files, changing every instance of “`D:`” to “`C:`” (case is not important). This must be done with an editor that *does not* change the carriage returns to a DOS format. Do not use Notepad or Wordpad as these will change the files to DOS format. Suggested editors that are capable of changing the files correctly are:

  Ultraedit has search and replace capability that can recursively search the directories for strings in certain files. Ultraedit is shareware with a 30 day demo period. Ultraedit can be obtained at:

  http://www.ultraedit.com

  XEmacs is a free open source text editor, under the GNU license. XEmacs can be obtained at:

  http://www.xemacs.org

Chapter 3

Using the PAVE SIF

PAVE is distributed in a development framework called the PAVE Systems Integration Framework, or SIF. This framework consists of a directory structure that is populated with the requisite software source code and header files, Wind River Systems Tornado II IDE workspace and project files, utility applications, scripts, and documentation.

In Chapter 2 the basic process of generating a framework was presented. This chapter expands on this by describing how to use and customize the PAVE System Integration Framework to create a framework for your IRL-enabled device This chapter goes further into the underlying concepts and the file formats used in the PAVE SIF.

While developers are not restricted to work within the framework, they are highly encouraged to do so. First, the SIF is a turnkey solution that enables systems architects to immediately focus on developing their application. The bulk of the setup has been completed. Secondly, the framework enables the developer to take advantage of future enhancements to the PAVE API with very little effort. Third, and most importantly, using the framework will enhance the support that you will get from Xilinx IRL Solutions.

IRL Register model

The PAVE Framework views hardware from a register level, thus all hardware can be described to the software in terms of registers. This allows hardware to be viewed as a component that can be replaced and upgraded as needed, just as software is upgraded. With this viewpoint, Xilinx has developed the IRL register model to allow a simple interface that supports reconfiguration of an IRL-enabled devices across any memory mapped bus (e.g. PCI). Figure 3-1 show the IRL register model.

The IRL-Enabled design is the TargetFPGA with the appropriate registers that PAVE can communicate with. Since these registers are viewed by the software as objects, this register model can be expanded by Xilinx or the user as needed through updates of the PAVE Framework or use of the SIF. Any series of additional registers can be added quickly and the corresponding software objects to use these registers can be added to the C++ code.

The IRL Register model in PAVE v1.0 supports configuration via JTAG and Xilinx SelectMAP. The XIRL Interface Register is a 32-bit memory mapped register that handles the configuration of the IRL-enabled device. It is shown outside the TargetFPGA as it has to be available at configuration time. This register could be in another FPGA, a CPLD, or even be a port on the processor, provided the software can directly address it. It is not considered to be part of the IRL-enabled device since it must reside in another device. The mapping of this register is shown in Figure 3-2. More detail on JTAG and SelectMAP configuration via this register can be found in Chapter 5, **Using PAVE for JTAG Configuration** and Chapter 6, **Using PAVE for SelectMAP Configuration**.

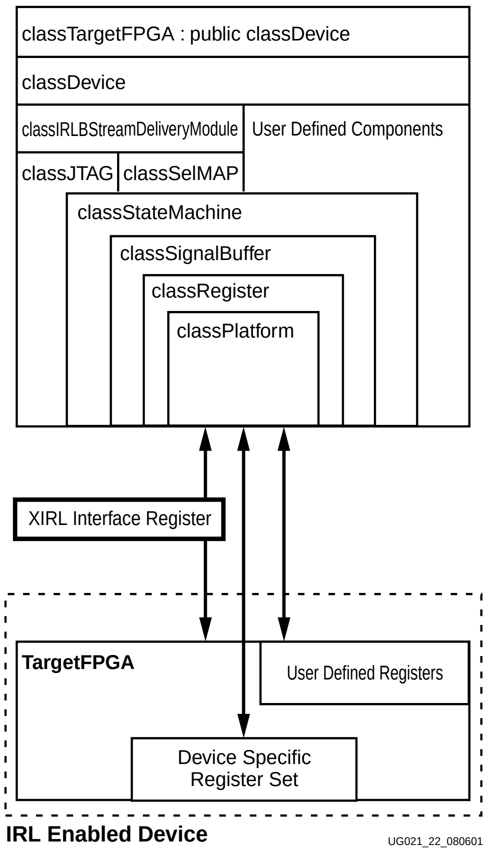


Figure 3-1: **IRL Register Model**

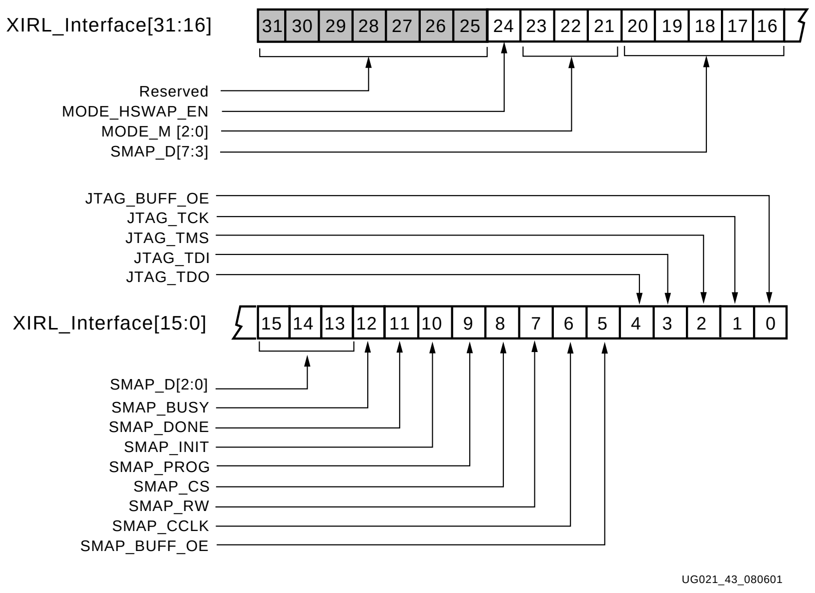


Figure 3-2: **XIRL Interface Register Map**

SIF Flow

The flow of the SIF can be broken down in two main goals, upgrading your system and simulating your upgradable system. This flow is represented in Figure 3-3.

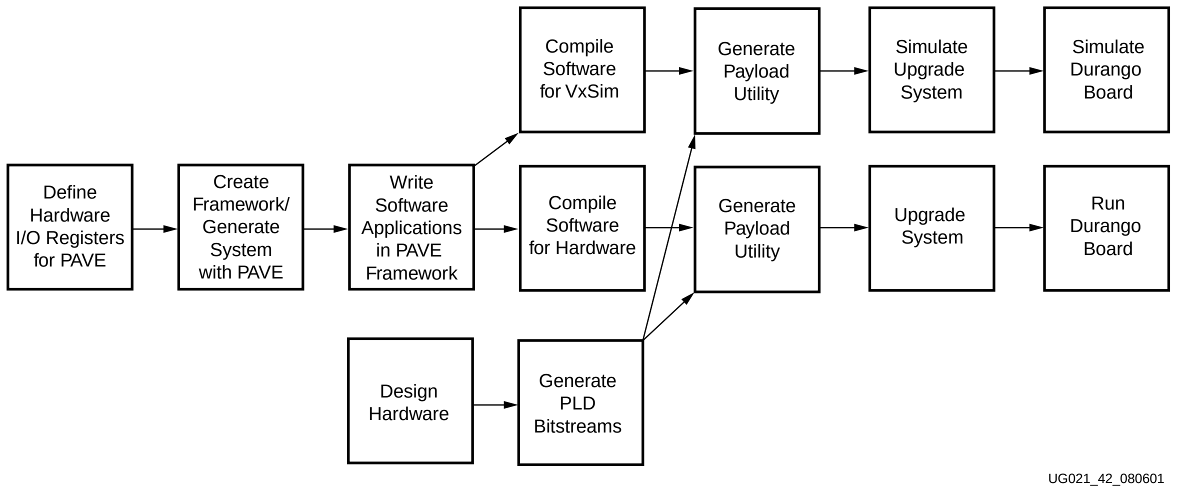

Define Hardware I/O Registers for PAVE
Create Framework/ Generate System with PAVE
Write Software Applications in PAVE Framework
Compile Software for VxSim
Compile Software for Hardware
Design Hardware
Generate PLD Bitstreams
Generate Payload Utility
Generate Payload Utility
Simulate Upgrade System
Upgrade System
Simulate Durango Board
Run Durango Board
UG021_42_080601

Figure 3-3: **PAVE Flow Chart**

Initial Steps

- Define the hardware registers - Using the `.inp` file format, described later in this chapter, the designer defines his register set for the SIF.
- Run the SIF System Generator to create a device framework.
- Develop your embedded application in Tornado, using the Tornado Workspace generated by the SIF.
- Design your physical hardware, including the FPGA designs.
- Generate the FPGA bitstreams.

System Simulation

- Compile the simulation projects in Tornado. In the Durango workspace these are listed under the build specification `SIMNTgnu`.
- Generate the payload.
- Simulate upgrading your system.
- Simulate the system. This involves reading and writing the registers defined in the initial steps. Further simulation of your FPGA and board designs can be performed but this is beyond the scope of this manual.

System Upgrade

- Compile the hardware projects in Tornado. In the Durango workspace these are listed under the build specification `PPC604gnu`.
- Generate the Payload.
- Upgrade your system. Examples of the upgrade process for the ADM-XRC can be found in Appendix D. The ADM-XRC code shipped with PAVE v1.0 includes push and pull code segments.
- Verify your hardware. In PAVE v1.0, Durango includes a “hello world” bitstream that blinks some LEDs on the board; the ADM-XRC include a series of applications,

including an FFT, and some graphics routines. Details on using the Durango board can be found in Appendix B.

Figure 3-4 shows the tool flow of the SIF and where the Xilinx and WRS tools are used. There are two major tools in the SIF, the System Generator and the Payload Generator.

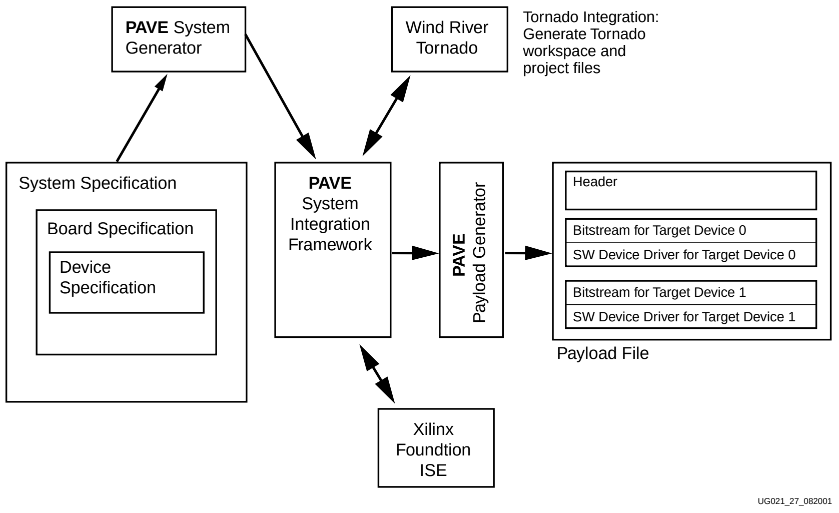

PAVE System Generator
Wind River Tornado
Tornado Integration: Generate Tornado workspace and project files
System Specification
Board Specification
Device Specification
PAVE System Integration Framework
PAVE Payload Generator
Header
Bitstream for Target Device 0
SW Device Driver for Target Device 0
Bitstream for Target Device 1
SW Device Driver for Target Device 1
Payload File
Xilinx Foundtion ISE
UG021_27_082001

Figure 3-4: **System Integration Framework Tool Flow Diagram**

System Generator

The PAVE SIF includes several executables that process the user's text descriptions of his hardware.The **`generatedevice.exe`** command in the **`generatesystem.bat`** script are the executables that create C++ source and header files, and related project files for the IRL-enabled devices that comprise the server system component.

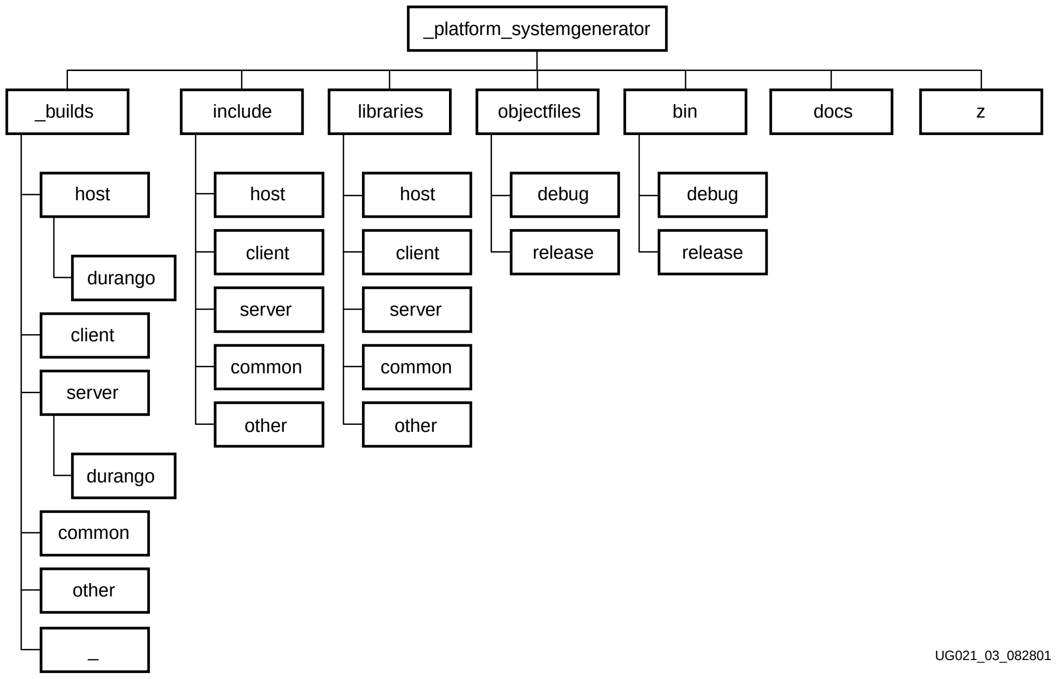


Figure 3-5: **System Generator Tree Diagram**

The most important components of the PAVE Systems Integration Framework directory tree structure are illustrated above in Figure 3-5

The major source code components of the PAVE API are contained in the `libraries\server` and the `include\server` directories. Additional source code components that are required by the build can be found in the `libraries\common` and `include\common` directories. The other major element of the build tree can be found under the `_builds\server` and `_builds\host` directory structure. Note that under this directory there is a directory called the `durango` directory. This directory is provided as a tutorial directory for illustrative purposes. This subdirectory tree is called the device component framework tree. It contains the files that are specific to a particular class of system component within a system. Figure 3-6 below depicts the `durango` directory and its subdirectories.

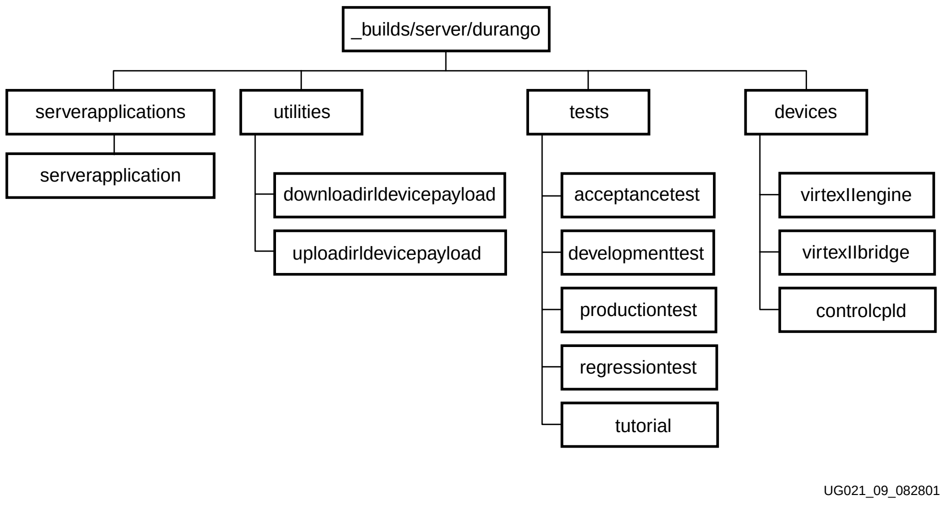


Figure 3-6: **SIF Framework Tree Diagram**

Note that the `durango` directory consists of four major subdirectories. These subdirectories are listed below.

```
_builds\server\durango\clientserverapplications
_builds\server\durango\devices
_builds\server\durango\tests
_builds\server\durango\utilities
```

Each subdirectory consists of a set of project files (Tornado `.wpj` IDE project files) that build specific test, development, and utility applications that are useful in the systems development process. The developer is not required to implement these applications. However, the PAVE framework and its code generation utilities automatically setup the directories and generate the source code and header files thereby making it easier to implement the functionality. The `_builds\server\durango\devices` subdirectory contains the source code and header files for the IRL-enabled Devices that comprise a specific system component. For example, the devices directory is comprised of three subdirectories. These are the `VirtexIIEngine`, `VirtexIIBridge` and `controlCPLD` directories. These subdirectories contain the source code, header, and project files that are required to build an object module for each of these devices.

As can be inferred from the directory structure, the PAVE Systems Integration Framework is setup to facilitate systems level development in a uniform context. Multiple software development efforts for multiple system components (and the constituent parts) can occur in parallel within the given software development framework. An additional benefit of this approach is that components of the framework can be readily leveraged for future product development efforts. Finally, the Systems Integration Framework is designed to enable developers to bifurcate derivative development efforts using a base framework. In essence, the framework is in itself an object oriented construct.

generatesystem.bat

This file specifies how to build the framework. From the **generatesystem.bat** file, a step by step review of the commands:

1. Generate a server for the IRL-enabled device. The **generateserver.exe** program is called to generate the code and directory structure for the board:

   ```
   generateserver   Durango.inp
   ```

2. Generate the individual devices The **generatedevice.exe** program is called to generate the code and framework for the devices on the board. In the case of Durango the devices are `ControlCPLD`, `VirtexIIEngine`, and `VirtexIIBridge`. The **generatedevice.exe** executable command takes three arguments.

   - The first argument specifies the name of the system component for which you are generating devices.
   - The second argument specifies the Device Specification file that contains information related to the programming interface of the device.
   - The third argument specifies the C++ class assigned to the device.

   The first line listed below is interpreted as "generate for the Durango board a device called `VirtexIIEngine` with baseclass of `classIRLDevice`".

   ```
   generatedevice   Durango     VirtexIIEngine.inp   classIRLDevice
   generatedevice   Durango     VirtexIIBridge.inp   classPCIDevice
   generatedevice   Durango     ControlCPLD.inp      classDevice
   ```

   The VirtexIIEngine is `classIRLDevice` because it can be reconfigured directly under software control. The VirtexIIBridge performs a PCI function so it is assigned to `classPCIDevice`. The CPLD is a generic device and is assigned to `classDevice`. More detail on classes can be found in Chapter 4, **Using the PAVE API**.

3. Rename the template Tornado workspace file.

   ```
   move /y          vxw_.wsp      durango.wsp
   ```

   The `vwx_.wsp` is a temporary file. This command won't overwrite the `durango.wsp` to prevent accidentally wiping out any changes you might have made in the workspace.

Note that any number of differing board setups could be put in this script, thus the name **generatesystem**.

INP File format

The **generatesystem.bat** requires a `.inp` file that lists the devices in the system. There are two inp file formats. The first type describes the system; an example is `durango.inp`. The Durango board is composed of several reconfigurable objects as described in the `durango.inp` file:

```
VirtexIIEngine
VirtexIIBridge
ControlCPLD
```

The second type describes the actual register specification. An example of this is the `VirtexIIEngine.inp` device specification file. In this file you will find the following line:

```
CTLREG00      0x00   0      32     true   true   true   0x00   apiDWORD
```

This line represent the register specification for the VirtexIIEngine IRL-enabled device of the Durango system component. Invoking **generatedevice.exe** causes the creation of a

C++ class library that encapsulates the device-programming interface comprised of the registers specified above. Both types of `.inp` files can have multiple entries.

Table 3-1: **Breakdown of devicegenerator .inp format**

Register Field Name	Device Offset	Start Bit	Number of Bits	Readable	Writable	Initialize	Initial Value	Access Width
`CTLREG00`	`0x00`	0	32	`true`	`true`	`true`	`0x00`	`apiDWORD`

Table 3-1 list the fields for the `Device Generator .inp` format. The definition of each field is provided below:

- Register Field Name: Specifies the name of the register.
- Device Offset: Specifies the offset of this register within the device.
- Start Bit: Specifies the starting bit of the register field within a register.
- Number of Bits: Specifies the register field width in number of bits.
- Readable: Specifies whether or not the register is readable.
- Writable: Specifies whether or not the register is writable.
- Initialize: Specifies whether or not the register should be initialized when it its device object is instantiated.
- Initial Value: Specifies the initial value that is written to the register field when that register object is instantiated. This field is ignored if the Initialize parameter is false.
- Access Width: Specifies the access width of the register field. The valid values are `apiDWORD`, `apiWORD`, or `apiBYTE`.

Building a Custom Framework

The Durango example is compliant with the IRL register model and can be used as a starting point for your hardware description.You may use the `VirtexIIEngine.inp`, `VirtexIIBridge.inp`, `ControlCPLD.inp` and the top level `Durango.inp` files as templates to create your own system components within the framework. Here are some key things to remember.

- The specification files must have a `.inp` extension, and be ordered hierarchically as shown in the Durango example.
- The **`generateserver.exe`** and **`generatedevice.exe`** utilities will use the name of the specification files to create the source, header, and associated project files. For example, if one of your device specification files is named `MyDevice.inp`, then the corresponding C++ class that will be generated by **`generatedevice`** is `classMyDevice`. Note that these names are case sensitive. For example, the file `mydevice.inp` will yield a class named `classmydevice`.
- The system component framework will be generated under the `_builds\server` directory. An example of this generated directory can be seen in Figure 3-6 above.
- The device framework will be generated under `_builds\server\[servername]\devices`
- You must generate the system component framework before the device frameworks as indicated in the **`generatesystem.bat`** script file.

Note that durango directories are generated under the `_builds\host` and `_builds\server` directories. Again, these directories encapsulate code for the generated board, Durango. For example, the `_builds\host\durango` directory has all of the code for host side applications related to the generated board. The `_builds\server\durango` directory is a tree that contains the target side files related to the generated board.

This structure was chosen because it enables the developer to swap out whole components of the application framework. For example, if the developer wishes to develop a new `VirtexIIEngine` codebase, he would swap out the `_builds\server\durango\devices\virtexIIengine` directory with a new framework. Likewise, the entire `_builds\server\durango` directory could be replaced. Over time a base of applications could be built up and swapped in and out this way.

Payload Generator

The PAVE API expects to receive the upgrade information in the form of a payload. The payload consists of a header and at least one software module and bitstream. This structure can be seen in Figure 3-7.

After generating the framework, compiling your code, and creating the bitstreams, you must generate a payload for the PAVE API to download. The SIF includes a utility program to handle this for you. In the `_builds\_` directory, you will find two files, **`generatepayload.bat`** and **`generatepayload.exe`**. The former copies your software modules (Tornado .out files) and bitstreams (Xilinx `.bit` and `.xsvf` files) to the local directory, generates the payload by calling the latter, then finally cleans up the directory.

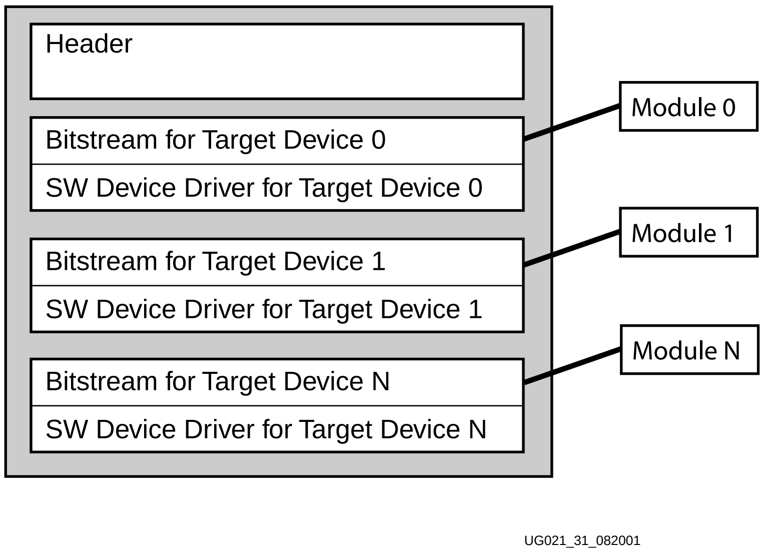

Header
Bitstream for Target Device 0
SW Device Driver for Target Device 0
Bitstream for Target Device 1
SW Device Driver for Target Device 1
Bitstream for Target Device N
SW Device Driver for Target Device N
Module 0
Module 1
Module N
UG021_31_082001

Figure 3-7: **Payload Diagram**

generatepayload.bat

The majority of the commands in the **`generatepayload.bat`** file are self-explanatory commands such as **`copy`** and **`del`**, but the **`generatepayload.exe`** command bears further examination. Here are some sample commands and the usage:

```
generatepayload 0 target.irl smap target.out target.bit FAFAFAFA AFAFAFAF
1 1 1
```

```
generatepayload 0 target.irl jtag target.out target.xsvf FAFAFAFA AFAFAFAF
1 1 1
```

```
generatepayload <Module Index> <Configuration Type> <Software Device
Driver> <Bitstream> <Vendor Code> <Device ID> <Device Type> <Revision
Code>
```

Table 3-2 list the arguments for the **`generatepayload.exe`** `command`. The definition of each argument is provided below:

Table 3-2: **Breakdown of generatepayload.exe command usage**

Argument	Sample Value	Comment
Module Index	`0`	`0-63`
Output File	`target.irl`	Also an input file if it exists
Configuration Type	`smap`	Either `smap` or `jtag`
Software Device Driver	`target.out`	Generated by Tornado
Bitstream	`target.bit`	SelectMAP uses `.bit`; JTAG uses `.xsvf`.
Vendor Code	FAFAFAFA	User defined, 32-bits maximum.
Device ID	AFAFAFAF	
Device Type	`1`	
Revision Code	`1`	
IRL Version	`1`	

1. Module Index - Location in the payload from 0 to 63. To instantiate multiple devices in one payload, rerun the command with the appropriate data for the next device and increment the Module Index by one.Always start at 0 and increment by one. A graphic representation of the Module Index can be seen in Figure 3-7 above.
2. Output File - Name of the output file. The recommended extension is "`.irl`". This argument is also an input file when multiple modules are instantiated in the payload.
3. Configuration Type - Valid types in PAVE v1.0 are "`smap`" (SelectMAP) and "`jtag`".
4. Software Device Driver - File name of the Tornado `.out` file.
5. Bitstream - Name of the file containing the bitstream. This cam be either a `.bit` or a `.xsvf` for SelectMAP and JTAG respectively.
6. Vendor Code - User-defined. Maximum size is 32-bits.
7. Device ID - User-defined. Maximum size is 32-bits.
8. Device Type - User-defined. Maximum size is 32-bits.
9. Revision Code - User-defined. Maximum size is 32-bits.
10. IRL Version - User-defined. Maximum size is 32-bits.

Formats for SelectMAP and JTAG

SelectMAP requires that the module be created with a `.bit` file. JTAG configuration requires use of the `.xsvf` format. Prior to generating the `.xsvf` format, if the target is a Xilinx PROM you must also convert it to a `.mcs` format. The overall flow is shown in Figure 3-8.

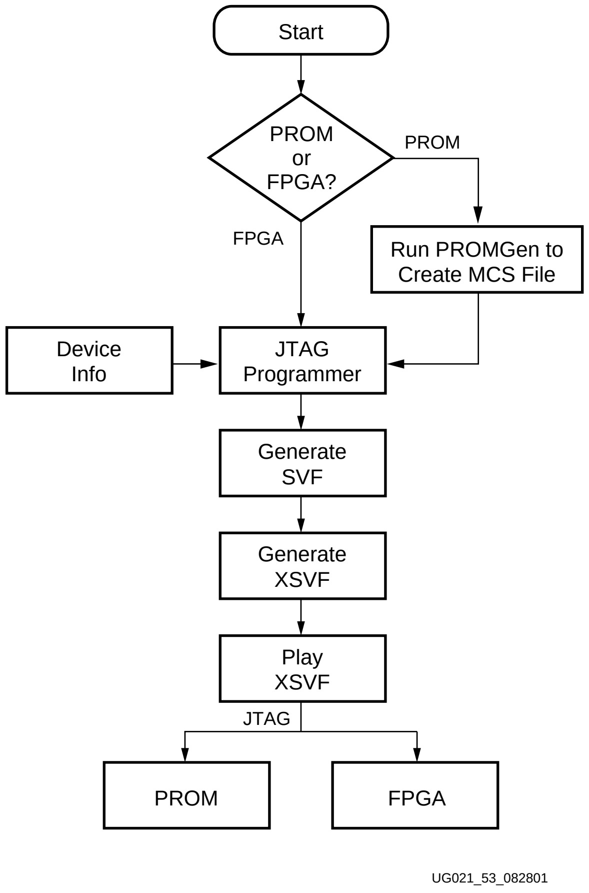
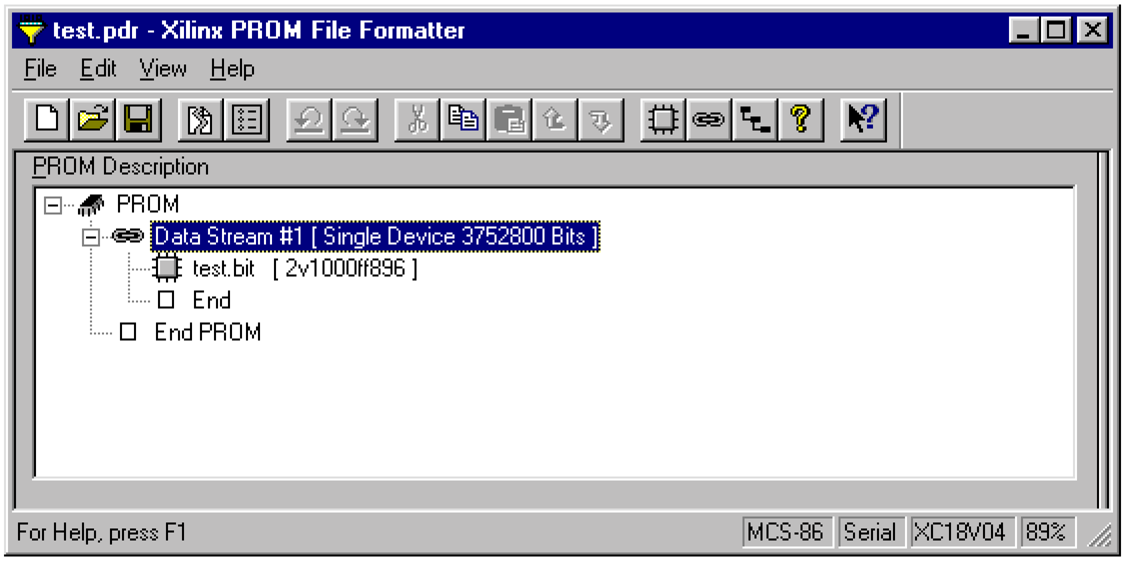


Figure 3-8: **XSVF Flow**

The "Device Info" is any information the JTAG Programmer GUI requests, such as Package type. Generating the XSVF file requires use of several Xilinx tools, including the Xilinx JTAG Programmer and the SVF2XSVF conversion utility. Here are the steps for 3.1i.

1. If the target is a PROM, you must generate a `.mcs` file first. Otherwise, skip to step 2. Open the PROM File Formatter (seen in Figure 3-9) and create a PROM file. You must target a device that supports JTAG (XC18VXX series PROMs).



PROM
Data Stream #1 [Single Device 3752800 Bits]
test.bit [2v1000ff896]
End
End PROM

For Help, press F1 | MCS-86 | Serial | XC18V04 | 89%

Figure 3-9: **PROM File Formatter**

2. Open the JTAG Programmer and select **`Edit -> Add Device`**. In the selection dialog, select your `.bit` or `.mcs` file.

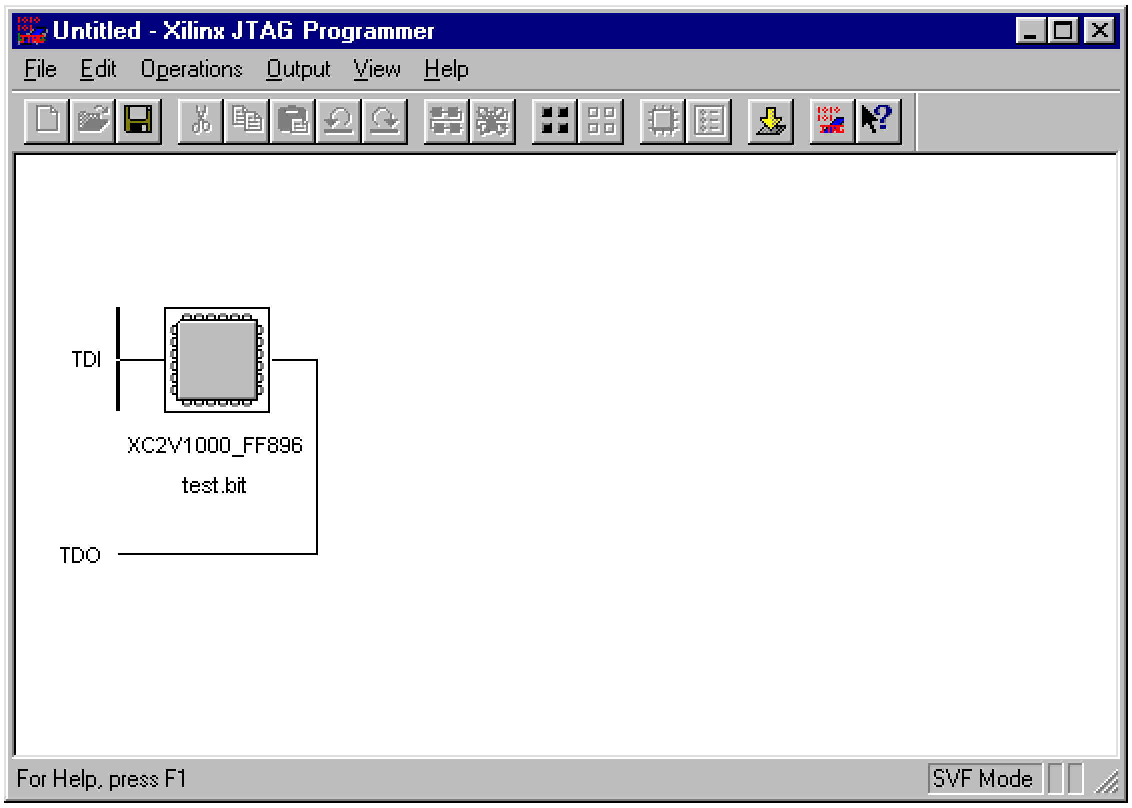
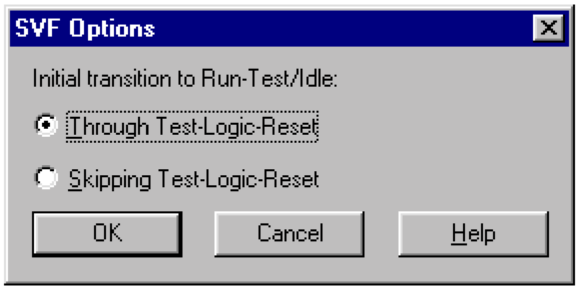

Untitled - Xilinx JTAG Programmer
File Edit Operations Output View Help
TDI
XC2V1000_FF896
test.bit
TDO
For Help, press F1
SVF Mode

Figure 3-10: **JTAG Programmer**

3. Choose the **`Output -> Create SVF File`** option. You should see a dialog as seen in Figure 3-11; click OK and then set the name of the SVF file. .



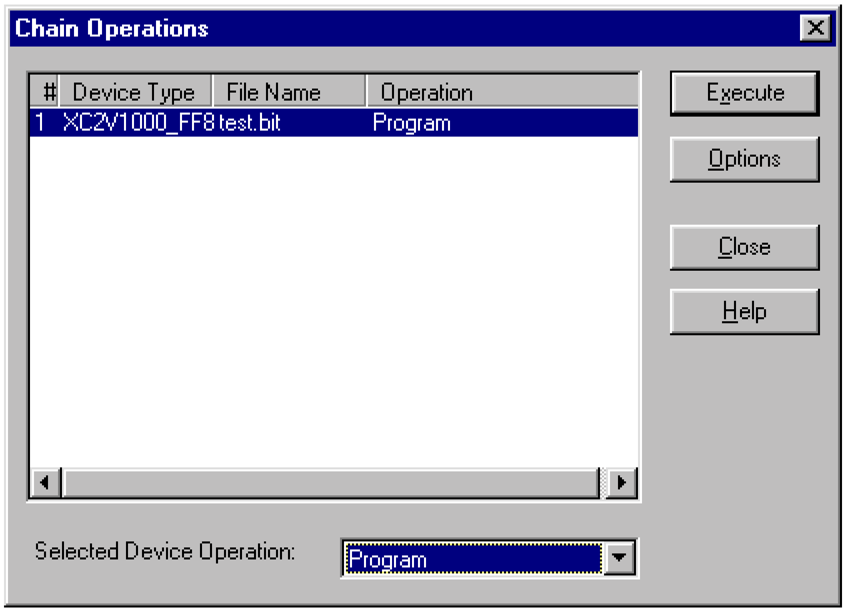
Initial transition to Run-Test/Idle:

[● Through Test-Logic-Reset]

○ Skipping Test-Logic-Reset

Figure 3-11: **SVF Options**

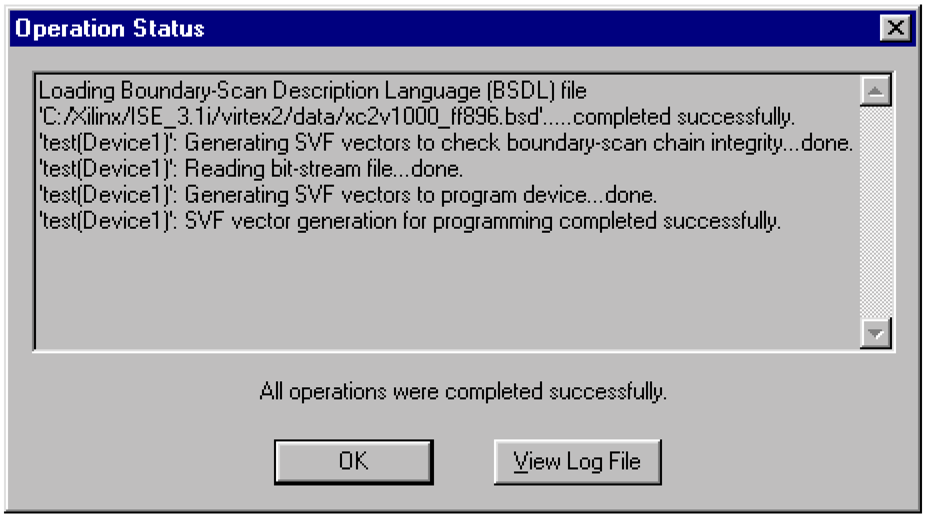
4. Select **`Operations -> Chain Operations...`** and set the device you just added to "Program"Then click Execute. and it will generate the SVF file as seen in Figure 3-13.



#	Device Type	File Name	Operation
1	XC2V1000_FF8	test.bit	Program

Selected Device Operation: Program

Figure 3-12: **Chain Operations**

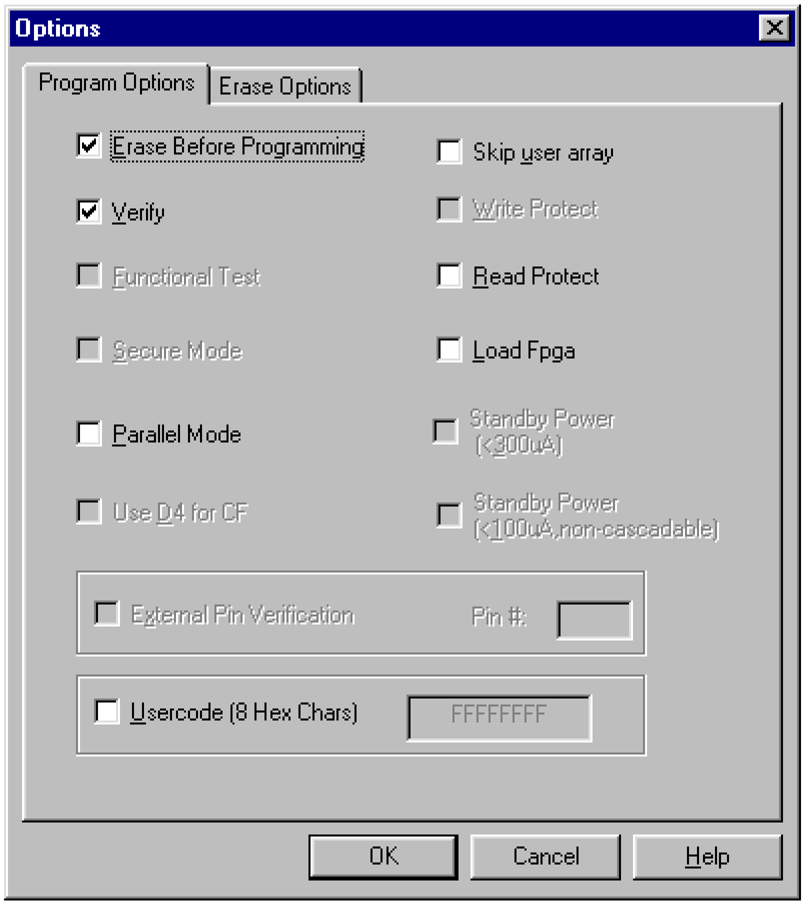


```
Loading Boundary-Scan Description Language (BSDL) file
'C:/Xilinx/ISE_3.1i/virtex2/data/xc2v1000_ff896.bsd'.....completed successfully.
'test(Device1)': Generating SVF vectors to check boundary-scan chain integrity...done.
'test(Device1)': Reading bit-stream file...done.
'test(Device1)': Generating SVF vectors to program device...done.
'test(Device1)': SVF vector generation for programming completed successfully.
```

All operations were completed successfully.

Figure 3-13: **Status of SVF Generation**

It you are using a `.mcs` file you have the further option of verifying the device with the SVF vectors. While in the Chain Operations, select the Options button and check the Verify Program option, as seen below. This is the only readback operation supported in PAVE v1.0.



[✓] Erase Before Programming
[ ] Skip user array
[✓] Verify
[ ] Write Protect
[ ] Functional Test
[ ] Read Protect
[ ] Secure Mode
[ ] Load Fpga
[ ] Parallel Mode
[ ] Standby Power (<300uA)
[ ] Use D4 for CF
[ ] Standby Power (<100uA,non-cascadable)
[ ] External Pin Verification Pin #:
[ ] Usercode (8 Hex Chars) FFFFFFFF

Figure 3-14: **Options to Allow PROM Readback**

5. The SVF2XSVF utility must be run on the resulting `.svf` files.

   For FPGA programing use this command line:

   `svf2xsvf -d -fpga -i <input file name>.svf -o <output file name>.xsvf`

   For PROM programming use this command line:

   `svf2xsvf -d -i <input file name>.svf -o <output file name>.xsvf`

   The SVF2XSVF utility can be found at:

   ftp://ftp.xilinx.com/pub/swhelp/cpld/eisp_pc.zip

Chapter 4

Using the PAVE API

The PAVE API is the runtime portion of the PAVE Framework. The PAVE v1.0 API, which is principally focused on device configuration, operates on the XIRL Interface Register described in the previous chapter. This chapter will review details about the API, it's classes, functions, and code, and discuss how to use it.

How the PAVE API Operates

Figure 4-1 below illustrates PAVE API interaction with the XIRL Interface Register. The basic idea that is portrayed in the diagram is that the PAVE API uses its methods to construct a `classSignalBuffer` object, which is then sequenced to the hardware via the methods of the `classRegister` and `classPlatform` objects. Additionally, all methods of the `classRegister`, `classSignalBuffer`, and `classPlatform` objects are exposed for device driver writers to potentially use in their development efforts. In particular, the combination of these classes with the `classStateMachine` object forms a powerful set of utilities that can be used to construct very complex embedded applications.

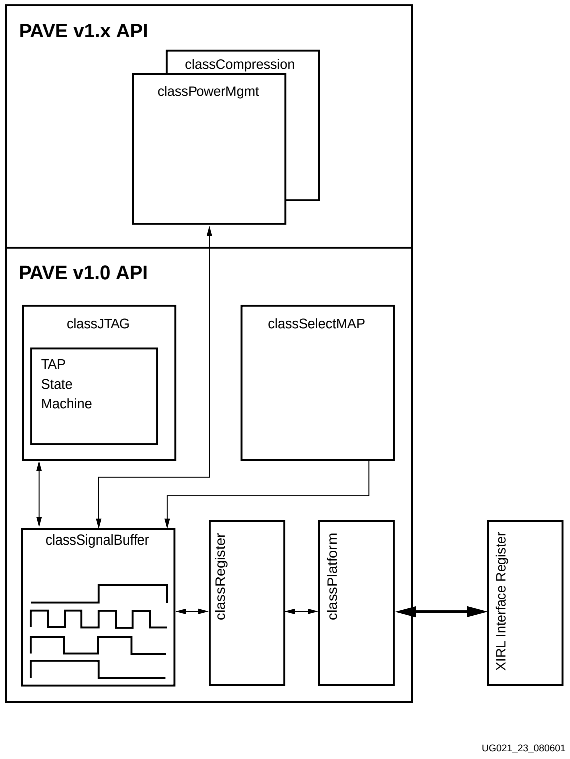


Figure 4-1: **PAVE API Example**

PAVE v1.0 implements JTAG and SelectMAP as shown in Figure 4-1. In a future version of PAVE, additional features, such as compression of the payload and power management, could be added using the same underlying structure.

To illustrate this further, you could define additional user registers (via the PAVE SIF) and then use existing classes or write new classes to manipulate these registers as required. One example of this is the need to do Endian swapping. The `classIRLPlatform` contains a method to endian swap a dword:

```
enAPIReturnCodes         classIRLPlatform::DWORDEndianSwap(
apiDWORD                 *ptrSourceDWord,
apiDWORD                 *ptrDestinationDWord
);
```

Instead of reordering the register or software drivers for processors you want to support, a simple bit of code could return the register data in the same format regardless of the underlying processor:

```
if (this->bSwapAccess == TRUE)
    {
        dwTemp2 = dwTemp;
        this->ptrPlatform->DWORDEndianSwap(&dwTemp2, &dwTemp);
        *ptrDWORDRegister = dwTemp;
    }
    else
    {
        *ptrDWORDRegister = dwTemp;
    }
```

Object Oriented Nature of the PAVE API

The PAVE API and its components are a collection of C++ classes and object models that abstract an implementation of a Xilinx Field Programmable Gate Array, FPGA, called the IRL-enabled Device implementation. PAVE encapsulates a hierarchical set of hardware and software specifications that define various levels of functionality in the IRL-enabled Device implementation. The minimum set of specifications for the IRL-enabled Device implementation requires a baseline set of features that are implemented in an FPGA which enables it to be upgraded via software configuration using either JTAG or the Xilinx SelectMAP protocols.

PAVE imposes a view of an FPGA as a standalone device object within a system component. As a result, applications that are written using the PAVE API tend to be highly object oriented, modular, and extremely upgradable. The device implementation, as expressed in a `.bit` or `.xsvf` configuration bitstream file, and its requisite controlling software module, the device driver, are component entities that uniquely define the functionality of an FPGA within a particular system component. User, vendor, and third party application content can be built on top of the PAVE API, as shown in Figure 4-2.

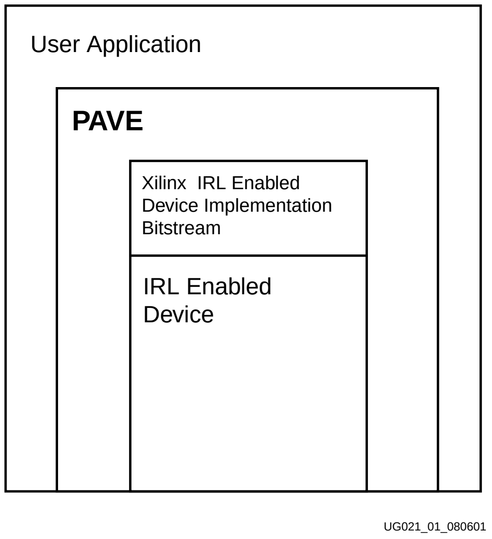


Figure 4-2: **PAVE Relationship to other Software and Hardware Objects**

It is important to understand that PAVE views these two elements, bitstream and software driver, as component pieces of a single entity. This entity is called the Payload. The payload is essentially a concatenated bitstream and binary object module that are prepended with a header. PAVE enables the application developer to deliver payloads to target devices while simultaneously upgrading the embedded application program that hosts the embedded device with a new device API. For example, consider the block diagram in Figure 4-3 below, which is similar to the Durango board.

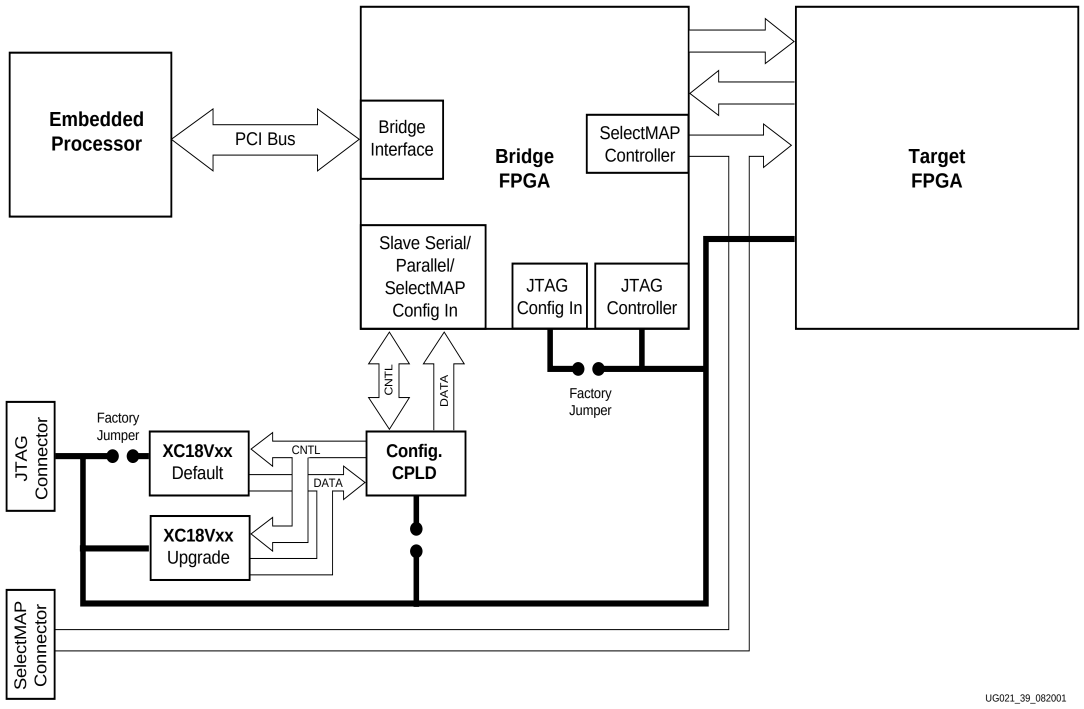


Figure 4-3: **Bridge and Target Block Diagram**

In this example, the reconfigurable target component consists of two FPGA devices, the Target and the Bridge FPGAs. Each device has associated with it a specific payload that is comprised of the configuration bitstream and software object module for the device. Figure 4-4 below provides a logical depiction of a payload for this hardware.

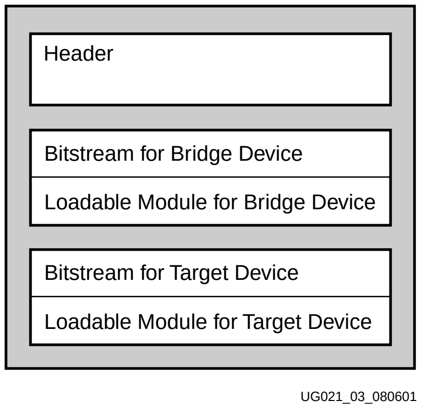

Header
Bitstream for Bridge Device
Loadable Module for Bridge Device
Bitstream for Target Device
Loadable Module for Target Device
UG021_03_080601

Figure 4-4: **Bridge and Target Sample Payload**

Either device in the system component can be individually upgraded with a new configuration bitstream, software, or both. The traditional monolithic view of the application has been replaced with a component object view. Component object oriented development is a prevalent technique in other types of application software development where it is necessary for those applications to be upgraded once they have been deployed. PAVE provides the embedded application developer with the same functionality via dynamic linking and full and (in the future ...partial) device reconfiguration. Additionally, the PAVE development model supports fully software controllable, dynamic and adaptive system component configuration. This opens the door for the development of a tremendously broad value proposition to product developers and end-users by way of extended product life cycles and lower support costs.

API Structure

The PAVE v1.0 software architecture is comprised of nine major software components that support either JTAG or SelectMAP device configuration. Figure 4-5 shows the architecture of the PAVE components.

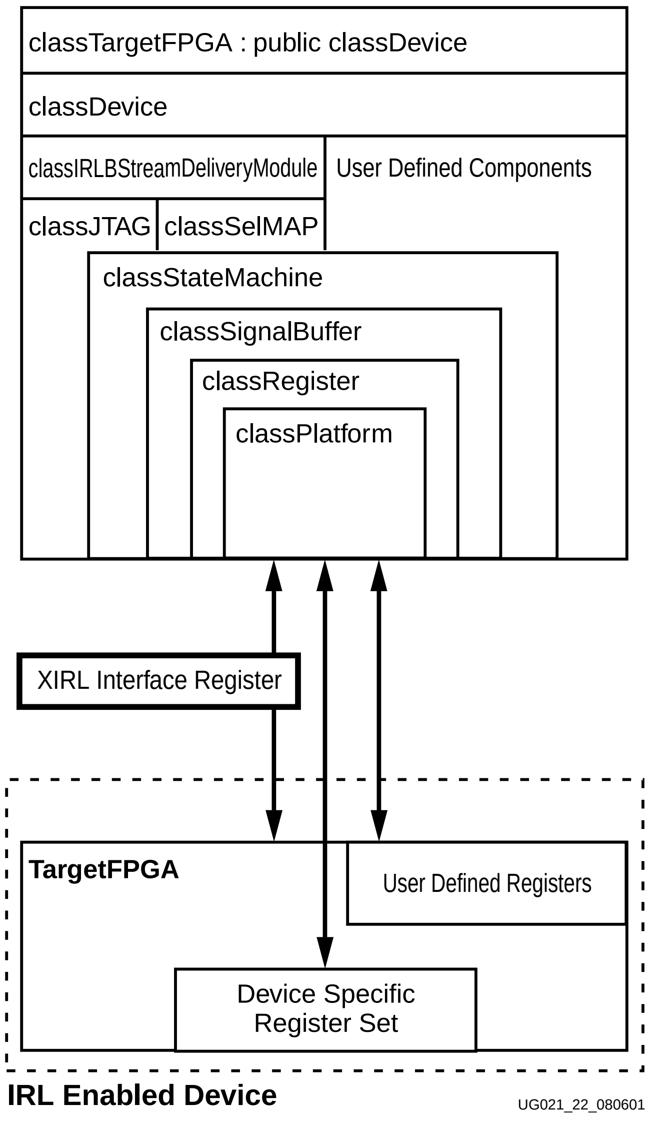


Figure 4-5: **PAVE v1.0 API Structure**

This diagram depicts the PAVE software components as an interdependent set of C++ objects. All PAVE API interaction with the target device occurs via a set of interface registers. The primary register for device configuration is the XIRL Interface Register, seen in Figure 4-6. The optional user-defined Registers are a set of registers that are components of the optional PAVE API components. Additionally, the device specific registers set completes the programming model for an IRL-enabled Device.

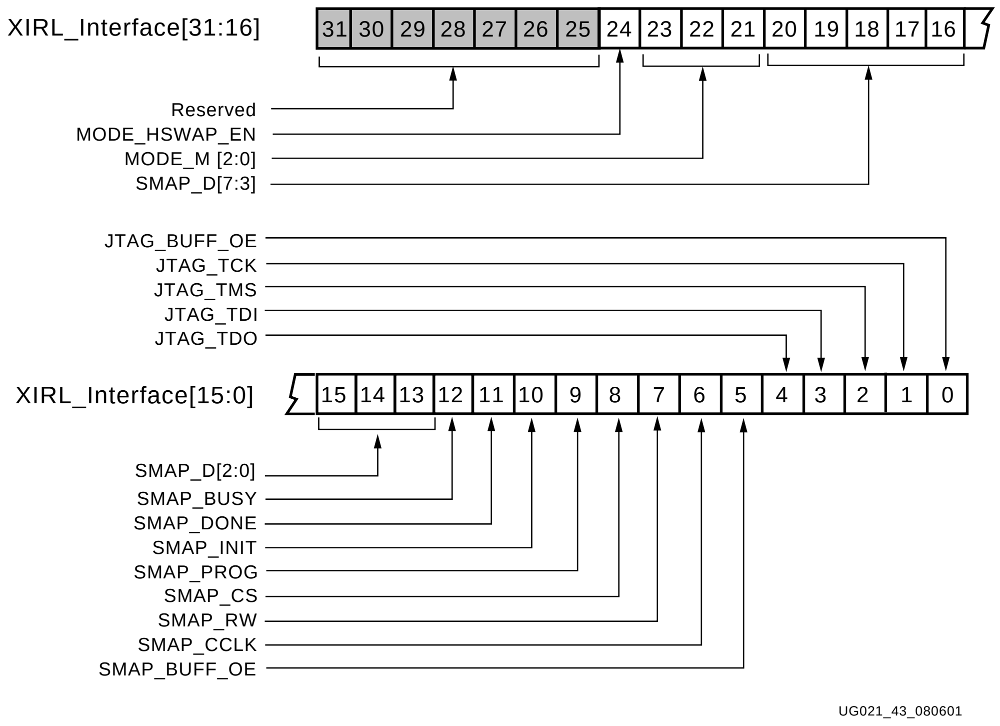


Figure 4-6: **XIRL Interface Register**

Functional Description

The various elements of the PAVE API communicate with IRL-enabled Devices through memory mapped registers. In particular, the `classIRLBStreamDeliveryModule` object contains a `classRegister` object called the `ptrXIRLInterface` register. This register object is an abstraction of a physical device port through which configuration bitstreams are uploaded to the targeted device. Additionally, the IRL-enabled Device specification defines a set of optional registers that provide facilities for additional extensions. Figure 4-7 below illustrates the relationship between the PAVE API and the physical device hardware. The arrows in the diagram illustrate that software is communicating to hardware .

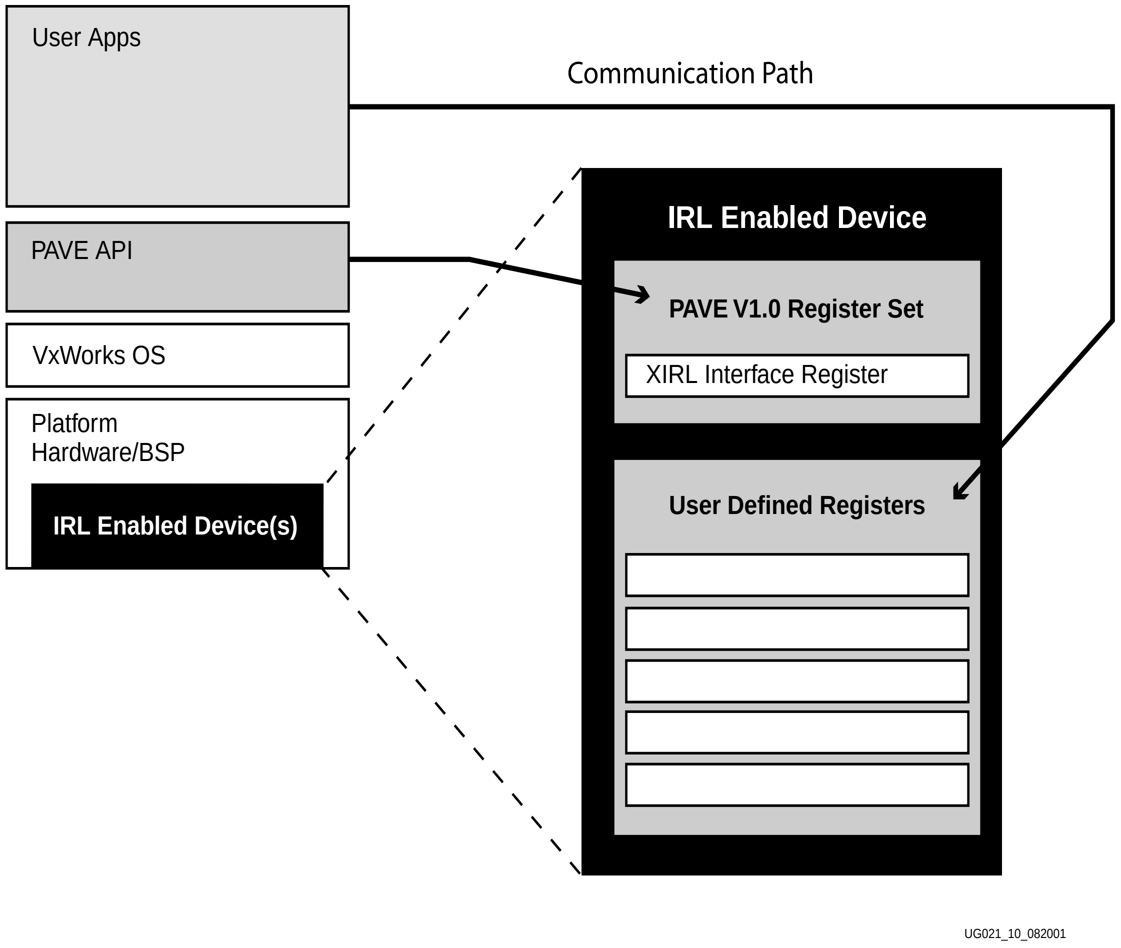


Figure 4-7: **Communication Paths**

PAVE API Classes

The PAVE API consists of several C++ classes that form an object model called the PAVE object model. These classes include

- The `classRegister` object: This class is a C++ class that abstracts a device register. It provides functionality to read from and write to the control registers (and fields thereof) of an IRL-enabled device implementation. The `classRegister` class is a key component of the PAVE API. It greatly facilitates the development of embedded device drivers by handling the details of the masking and shifting operations that normally accompany read, write, and modify operations on a register.
- The `classDevice` object: This class a C++ base class from which all PAVE devices are derived. The PAVE `devicegenerator` tool that is distributed in the PAVE framework is a C++ code generator tool that creates IRL-enabled device software models that are derived from the `classDevice` object. The `classDevice` object is a container class for multiple instances of `classRegister` objects.
- The `classIRLBStreamDeliveryModule` object: This class is a C++ object that encapsulates the JTAG or SelectMAP upgrade functionality of the IRL-enabled device. This class is a base class from which other bitstream delivery modules can be derived for specific hardware implementations. The `classIRLBStreamDeliveryModule` is contained in the `classDevice` object. The `classIRLBStreamDeliveryModule` is comprised of separate JTAG and SelectMAP component objects. These are the `classJTAGComponent` and `classSelectMAPComponent` objects respectively. The `classIRLBStreamDeliveryModule` is a required component in the IRL-enabled Device implementation and therefore it defines PAVE v 1.0 functionality.
- The `classStateMachine` object: The `classStateMachine` object is a C++ object that abstracts the functionality of a finite state machine. This object is contained in the `classJTAGComponent` object and is used to setup the TAP controller state machine used in JTAG configuration mode. The `classStateMachine` object is very generic and can also be used to facilitate the implementation of very complex real-time state machines in embedded applications.
- The `classSignalBuffer` object: The `classSignalBuffer` object is a C++ class that encapsulates the functionality required to setup and sequence a set of control signals to a hardware registers via software control. This class also contains a number of utility methods that allow the developer to generate arbitrary test vectors that can be used in device test benches.

The PAVE API framework is additionally composed of a number of supporting classes and object models. These classes include

- The `classIRLPlatform` object: The PAVE API can be used in a wide variety of hardware platform and embedded operating system environments. The `classPlatform` object facilitates this porting effort by forming a thin abstraction layer between the platform specific code and application code. Figure 4-8 and Figure 4-9 below illustrate how the Objects and the RTOS are related to the upper layer application code.
- The `classPCIDevice` code is used to interface to the PCI interface on the board. In this revision of the PAVE code the Durango board is a simple PCI target, with the XIRL configuration register. This PCI interface only has one register beyond the regular PCI configuration space. The `virtexIIBridge.inp` describes this register. The PCI configuration space is accessed through methods that the `VirtexIIBridge` object inherits by being instantiated as a `classPCIDevice` object.

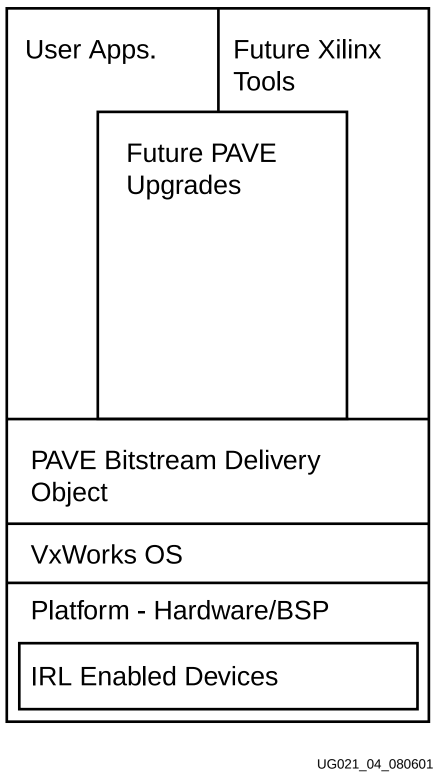


Figure 4-8: **System Relationships**

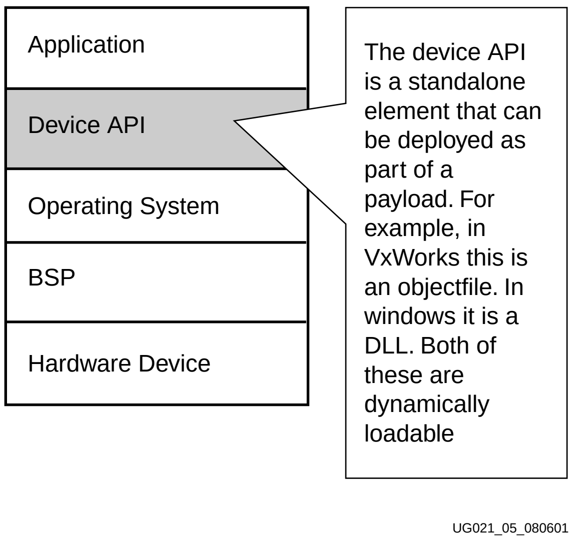

Application
Device API
Operating System
BSP
Hardware Device
The device API is a standalone element that can be deployed as part of a payload. For example, in VxWorks this is an objectfile. In windows it is a DLL. Both of these are dynamically loadable
UG021_05_080601

Figure 4-9: **Device API**

Payloads

The principle element handled by the PAVE v1.0 API is called the payload. A payload consists of the three sets of elements illustrated below in Figure 4-10.

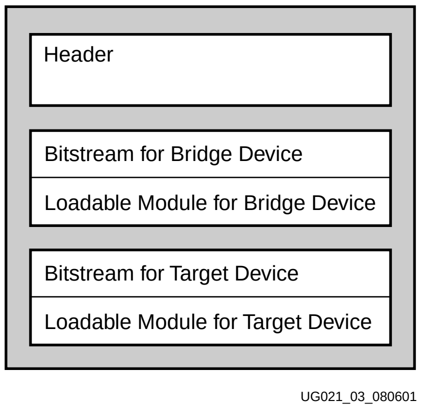

Header
Bitstream for Bridge Device
Loadable Module for Bridge Device
Bitstream for Target Device
Loadable Module for Target Device
UG021_03_080601

Figure 4-10: **Generic Payload structure**

This diagram shows the structure of a PAVE payload. The header component consists of size and classification information for up to 64 device elements in the payload. These are called device configuration segments. Note that each device element has associated with it a bitstream and loadable module. Each instance of a PAVE device object has methods that understand how to parse the payload and find the correct payload components that are required for upgrading. The structures found in a payload are defined below.

Payload Header Structure

```
typedef      struct
{
    size_t                    dwNumberDevicesInPayload;
    structDeviceConfiguration  theDeviceConfiguration[CNSTMAXNUMDEVICES];

} structPayloadHeader;

typedef   structPayloadHeader *pntr_structPayloadHeader;
```

The constant `CNSTMAXNUMDEVICES` is user definable. The payload header consists of `CNSTMAXNUMDEVICES` device configuration segments.

Payload Configuration Segment Structure

```
typedef      struct
{
/* dwXIRLVendorCode is the user defined vendor code for the targeted
device. */

    apiDWORD        dwXIRLVendorCode;

/* dwXIRLDeviceID is the user defined device ID for this device. The PAVE
v1.0 API uses this field to verify that the Device ID in the payload
matches the targeted device. */
```

```
   apiDWORD        dwXIRLDeviceID;

/* dwIRLRevisionCode is a user defined revision code that the PAVE v1.0
API's use to verify that the payload is the correct version for the tar-
geted device. */

   apiDWORD        dwXIRLRevisionCode;

/* dwIRLVersion is a user defined version code that the PAVE v1.0 API's
use to verify that the payload object module is built using a correct ver-
sion for the targeted device. */

   apiDWORD        dwXIRLVersion;

/* ecDeviceType indicates the type of the device. Valid values are ecIRL-
SpartanDevice, ecIRLSpartanIIDevice, ecIRLVirtexDevice, ecIRLVirtexEDe-
vice, and ecIRLVirtexIIDevice. */

   enIRLDeviceType ecDeviceType;

/* ecProgrammingMode indicates the hardware device programming interface
implementation for this device. Valid values include ecIRLSelectMapPro-
grammingMode, and ecIRLJTAGBoundaryScanProgrammingMode */

   enIRLDeviceProgrammingMode ecProgrammingMode;

/* dwBitstreamSize indicates the device bitstream or XSVF buffer size in
bytes. */

   size_t                 dwBitstreamSize;

/* dwModuleSize indicates the loadable module size in bytes. */

   size_t                 dwModuleSize;

/* dwBitstreamSize indicates the device bitstream or XSVF buffer size in
bytes. */

   size_t                 dwBitstreamOffset;

/* dwModuleSize indicates the loadable module size in bytes. */

   size_t                 dwModuleOffset;

/* The following two parameters are checksums for the components. */

   apiDWORD        dwBitstreamCheckSum;
   apiDWORD        dwModuleCheckSum;
}  structDeviceConfiguration;

typedefstructDeviceConfiguration *pntr_structDeviceConfiguration;
```

Using PAVE for JTAG Configuration

How it operates

A key component of the PAVE v1.0 API is the JTAG TAP controller state machine. This object governs the sequence of signals that are generated in the signal buffer object. The JTAG component controls state machine sequencing of the TAP controller as required to shift data into or out of the `TDI` and `TDO` signals respectively. Figure 5-1 illustrates the program flow.

The XIRL Interface Register, Figure 5-2, contains a series of bits that constitute the JTAG interface in PAVE v1.0. The PAVE software performs the function of the state machine that generates the appropriate signals for the JTAG interface.By successive writes to the XIRL Interface Register, the bus is clocked with the correct values.

From a software standpoint the use of this interface is relatively simple; a single high level method invokes the JTAG interface and handles the entire configuration, returning the status of the configuration once done. No knowledge of JTAG is required, nor are any hardware state machines required.

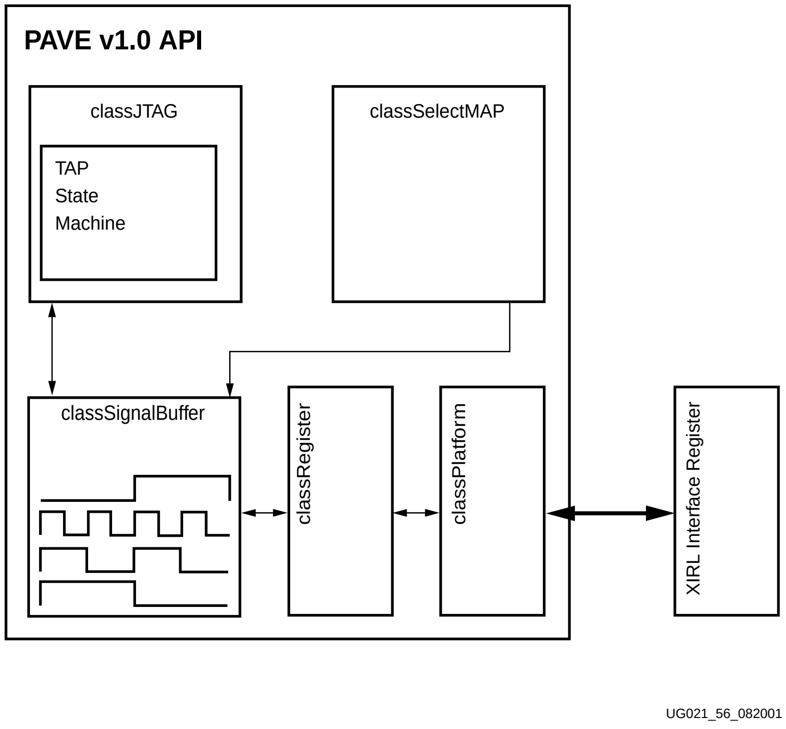


Figure 5-1: **PAVE Configuration Data Flow for JTAG and SelectMAP**

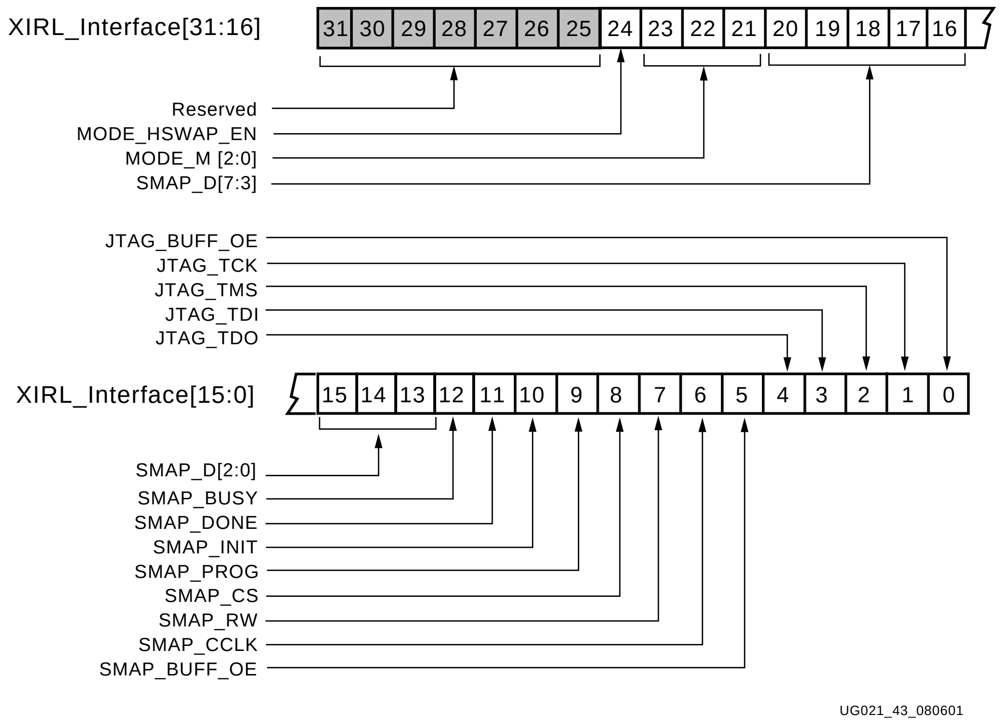


Figure 5-2: **XIRL Interface Register Map**

Design considerations

Several things should be considered in implementing a JTAG interface with PAVE:

- The payload must contain a `.xsvf` file for each module using JTAG.
- Since the clock is toggled on successive writes, the JTAG interface will run at about half of the write frequency.

Code example

This example of using the JTAG interface involves several steps. Additional code for this example can be found in the `tutorial.cpp` file.

```
// Get the size of the payload elements for the targeted device.

ptrServer->ptrDesignatedDevice->GetPayloadSize(
   "payloads/mcp750durango_tutorial_jt.irl",
   &dwBitstreamBufferSize,
   &dwModuleBufferSize);

//Cache the payload elements into a local buffer.

ptrServer->ptrDesignatedDevice->CachePayload(
   "payloads/mcp750durango_tutorial_jt.irl",
   &ptrBitstreamBuffer,
   &ptrModuleBuffer);
```

```
/* Now upload the payload from these buffers. First the JTAG mode is set,
followed by the UploadPayloadFromBuffer method. */

ptrServer->ptrDesignatedDevice->ecIRLDeviceProgrammingMode =
    ecIRLJTAGBoundaryScanProgrammingMode;
ptrServer->ptrDesignatedDevice->UploadPayloadFromBuffer(
    dwBitstreamBufferSize,
    ptrBitstreamBuffer,
    dwModuleBufferSize,
    ptrModuleBuffer);

/* Uncache the payload elements after the update has been completed. Your
error checking should occur here, if desired. */

ptrServer->ptrDesignatedDevice->UnCachePayload(
    ptrBitstreamBuffer,
    ptrModuleBuffer);

// Clean up so we don't get memory leaks or leave dangling pointers

delete ptrFrameCounter;
delete ptrServer; }
```

Rewiring the XIRL Interface Register

The connectivity of the XIRL Register is defined in the `classIRLBStreamDelivery.h` file. If you have need to quickly rewire the register to match your physical board wiring, this can be done by modifying the definition. For example to swap the `TDI` and `TDO` pins, the following section of code would be changed:

Old version:

```
#define    CNSTBSDMSignalJTAG_TDIOFFSET      0x00000000
#define    CNSTBSDMSignalJTAG_TDISTARTBIT    3
#define    CNSTBSDMSignalJTAG_TDINUMBITS     1
#define    CNSTBSDMSignalJTAG_TDOOFFSET      0x00000000
#define    CNSTBSDMSignalJTAG_TDOSTARTBIT    4
#define    CNSTBSDMSignalJTAG_TDONUMBITS     1
```

Swapped version:

```
#define    CNSTBSDMSignalJTAG_TDIOFFSET      0x00000000
#define    CNSTBSDMSignalJTAG_TDISTARTBIT    4
#define    CNSTBSDMSignalJTAG_TDINUMBITS     1
#define    CNSTBSDMSignalJTAG_TDOOFFSET      0x00000000
#define    CNSTBSDMSignalJTAG_TDOSTARTBIT    3
#define    CNSTBSDMSignalJTAG_TDONUMBITS     1
```

Chapter 6

Using PAVE for SelectMAP Configuration

How it operates

A key component of the PAVE v1.0 API is the SelectMAP object. This object governs the sequence of signals that are generated in the `classSignalBuffer` object when a SelectMAP configuration method is invoked. Figure 6-1 shows the data flow from the SelectMAP object to the XIRL Interface Register

The XIRL Interface Register, mapped in Figure 6-2, contains a series of bits that constitute the SelectMAP interface in PAVE v1.0. By writing to the XIRL Interface Register, the bus is clocked with the correct values.

From a software standpoint the use of this interface is relatively simple; a single high level method invokes the SelectMAP interface and handles the entire configuration, returning the status of the configuration once done.

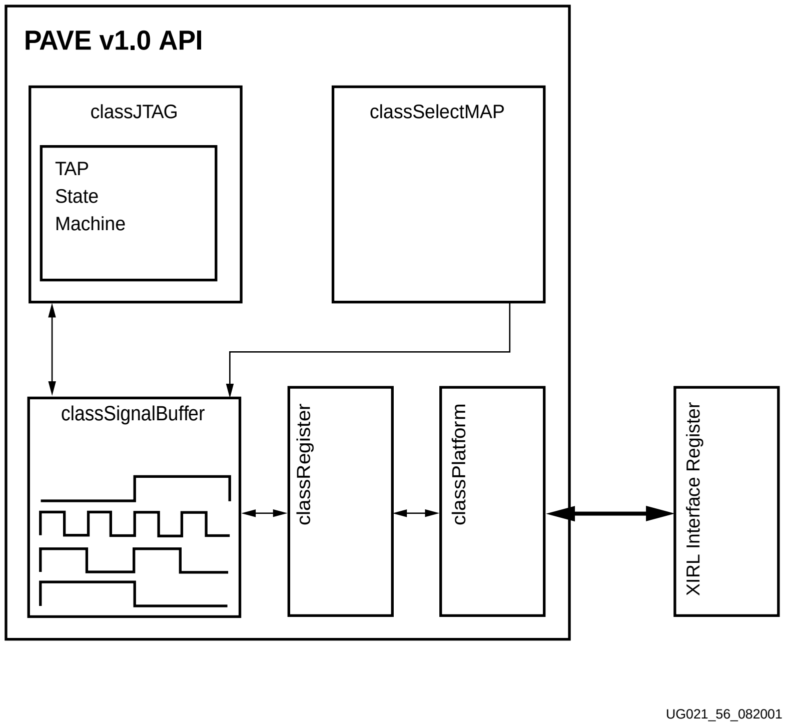


Figure 6-1: **PAVE Configuration Data Flow for JTAG and SelectMAP**

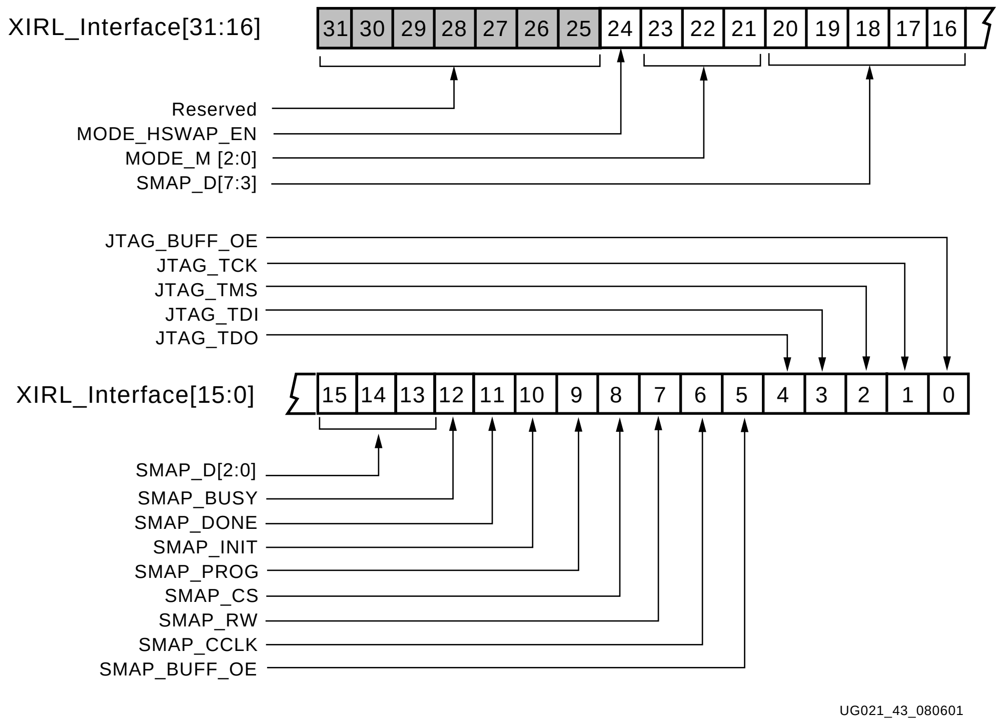


Figure 6-2: **XIRL Interface Register Map**

Design considerations

Several things should be considered in implementing a SelectMAP interface with PAVE:

- The payload must contain a `.bit` file for each module using SelectMAP.
- Due to the sequence of updates to the XIRL Register, the SelectMAP interface will run at about half of the write frequency.
- The `SMAP_BUSY` signal is not monitored in PAVE v1.0; if you intend to run this at a very high speed, make sure you do not exceed the maximum allowable speed of the device being configured.

Code example

This example of using the SelectMAP interface involves several steps. Additional code for this example can be found in the tutorial.cpp file.

```
// Get the size of the payload elements for the targeted device.

ptrServer->ptrDesignatedDevice->GetPayloadSize(
    "payloads/mcp750durango_tutorial_sm.irl",
    &dwBitstreamBufferSize,
    &dwModuleBufferSize);

//Cache the payload elements into a local buffer.

ptrServer->ptrDesignatedDevice->CachePayload(
    "payloads/mcp750durango_tutorial_sm.irl",
   &ptrBitstreamBuffer,
   &ptrModuleBuffer);

/* Upload the payload from these buffers using SelectMAP. First the
SelectMAP mode is set, then the upload is performed. */

ptrServer->ptrDesignatedDevice->ecIRLDeviceProgrammingMode =
   ecIRLSelectMapProgrammingMode;
ptrServer->ptrDesignatedDevice->UploadPayloadFromBuffer(
   dwBitstreamBufferSize,
   ptrBitstreamBuffer,
   dwModuleBufferSize,
   ptrModuleBuffer);

/* Uncache the payload elements after the update has been completed. Your
error checking should occur here, if desired. */

ptrServer->ptrDesignatedDevice->UnCachePayload(
   ptrBitstreamBuffer,
   ptrModuleBuffer);

// Clean up so we don't get memory leaks or leave dangling pointers

delete ptrFrameCounter;
delete ptrServer; }
```

Rewiring the XIRL Interface Register

The connectivity of the XIRL Register is defined in the `classIRLBStreamDelivery.h` file. If you have need to quickly rewire the register, this can be done by modifying the definition. For example to swap the CCLK and PROG pins, the following section of code would be changed:

Old version:

```
#define   CNSTBSDMSignalSMAP_CCLKOFFSET      0x00000000
#define   CNSTBSDMSignalSMAP_CCLKSTARTBIT    6
#define   CNSTBSDMSignalSMAP_CCLKNUMBITS     1
#define   CNSTBSDMSignalSMAP_PROGOFFSET      0x00000000
#define   CNSTBSDMSignalSMAP_PROGSTARTBIT    9
#define   CNSTBSDMSignalSMAP_PROGNUMBITS     1
```

Swapped version:

```
#define   CNSTBSDMSignalSMAP_CCLKOFFSET      0x00000000
#define   CNSTBSDMSignalSMAP_CCLKSTARTBIT    9
#define   CNSTBSDMSignalSMAP_CCLKNUMBITS     1
#define   CNSTBSDMSignalSMAP_PROGOFFSET      0x00000000
#define   CNSTBSDMSignalSMAP_PROGSTARTBIT    6
#define   CNSTBSDMSignalSMAP_PROGNUMBITS     1
```

Network Configuration

The PAVE API provides both Host and Target side classes to implement a basic TCP/IP sockets connection for you to test your IRL-enabled design across an ethernet cable. The sockets code in PAVE v1.0 has no security and has only been tested in a simple lab environment. Prior to deployment of any gear to the field, you should test your IRL-enabled designs in all target environments. PAVE's `classTCPIP` channel provides methods to open and close socket connections as well as to receive and send data. The `serverapplication.vx` project within the `_project_mcp750admxrc` framework provides the `classTCPIP` channel to perform reconfiguration over a network.

Network Domains

The PAVE Framework implements a system architecture that includes several domains. These domains interact with each other with predefined relationships.

The PAVE Framework is logically partitioned into five sections or domains that embody PAVE's Client/Server view of the embedded application. These are the Host, Client, Server, Common, and Other domains. In PAVE v1.0 only the client, host and server domains are being used. Figure 7-1 below illustrates the various domains and how they fit in the context of an embedded systems application.

This figure shows a typical partitioning for an embedded system and how the payload component moves through the system.

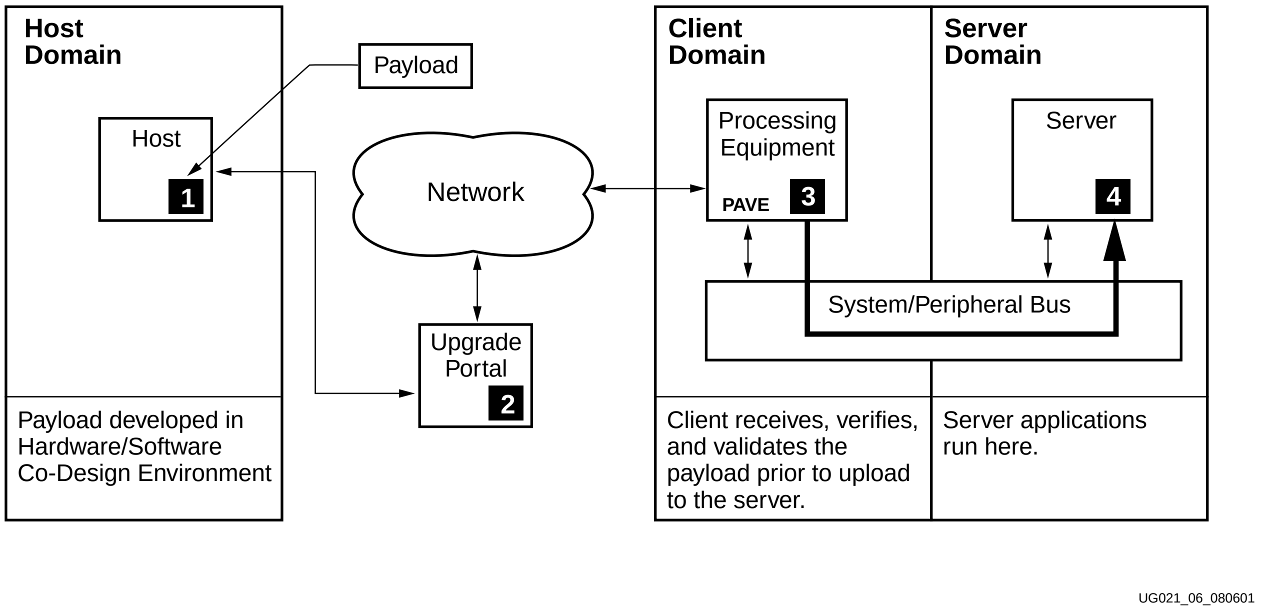

Host Domain
Host
1
Payload developed in Hardware/Software Co-Design Environment
Payload
Network
Upgrade Portal
2
Client Domain
Processing Equipment
PAVE
3
Client receives, verifies, and validates the payload prior to upload to the server.
Server Domain
Server
4
System/Peripheral Bus
Server applications run here.
UG021_06_080601

Figure 7-1: **Host, Client, and Server Domain Partitioning**

Definitions

Some of the terms being used here might be used in a different way than the common understanding. To prevent any confusion on these relationships, let us start with a few definitions:

- Object-centric - A self-centered viewpoint. Similar to being in an object and looking out at other objects. An example of this is being in a car and watching other cars.

  Object-centric terms:

  - Upload - In the PAVE Framework, objects upload information. For example, the `classRegister::Upload` method results in the register being written.
  - Download - In the PAVE framework, objects download information. For example, the `classRegister::Download` method results in the contents of a register being read.
- System-centric - Something that is evaluated in the context of the overall system. This is like viewing a whole set of objects from outside. An example of this is a bird's eye view.
  - Server - An object that provides some functionality to other objects.
  - Client - An object that requests another object to perform some function for it.
- Additional Definitions:
  - Client Domain - The client domain is a component of the PAVE Logical System Partitioning. It is comprised of those elements that are involved in the control of embedded system components.
  - Server Domain - The server domain is a component of the PAVE Logical System Partitioning. It is comprised of those elements that provide specialized functionality within an embedded system.

Host, Client and Server Domains

The Host domain encapsulates those systems and software elements that are typically associated with system development. For example, the developer workstation and associated development tools (compilers, debuggers, Xilinx Foundation and Alliance Series tools, Wind River Tornado, PAVE utilities, etc.) reside in the Host domain. The upgrade portal is considered to be part of the host domain.

The system controller (e.g. a cPCI system slot board) and the software that is targeted for it are Client Domain elements. This is referred to as the Client Domain because this component and the codes that run there function as clients within the system. The third major domain is the Server domain. This domain consists of the reconfigurable logic and associated components.

Client and Server Relationships

There are two sets of client and server relationships in an IRL-enabled system using PAVE.

- Between reconfigurable logic (server) and the processor (client). In this case, the FPGA / server provide a function to the processor / client, e.g. an FFT computation.
- Between the Target (server) and Upgrade Portal / Host (client). The host / client requests an upgrade and the Target / server fulfills this request. This relationship is not to be confused with the client and server domains.

The programs that are run on the Host and Target to handle the upgrades are named respectively. The Host / Upgrade portal runs `clientapplication_nt` and the Target runs `serverapplication_vx`.

Configuration Across a TCP/IP Network

The Tornado tools support a basic socket connection through their `sockLib` - the WRS generic socket library. The NT Host-side applications use the standard `winsock` library. Since all the API and host-side applications source code is available, additional networking code can be added as needed by the system application.

Once the socket connection is established, the transfer of the configuration data can begin. There are two primary means of doing this transfer, push and pull. More detailed instructions on running these applications can be found in Appendix D.

Pull Configuration

Pull is similar to downloading files from the internet. Pull configuration uses a knowledge of where the file is on the network and downloads it for processing.

The code is called with a fixed location of the file. This file could reside on the fixed storage for the processor or elsewhere on the network. The file “`pull`” contains this spawn process call which successively retrieves payloads stored on disk and reconfigures the FPGA:

```
sp(admxrc_alldemo, 10)
```

The argument 10 is the number of repetitions of admxrc_alldemo, a application program for the ADM-XRC board. From the this method in `serverapplication_vx.cpp`:

```
ptrServer->SampleLoad("fpga/fastlife");
```

`Sampleload` is a method in `classADMXRC` and which downloads a file locally (target processor) and then uploads it to the FPGA.

Push Configuration

Push can be compared to sending attachments on email; at the other end, the recipient opens and uses (processes) the file. This configuration method sets up a server / client connection with the Upgrade Portal and then will perform the upgrades as requested. The host / upgrade portal uploads the file to the server, which uploads it to the FPGA.

The file “push” runs on the target and sets the IP address and port, then spawns the `admxrc_serverapplication_vx` process to handle this:

```
szIP = "127.0.0.1"
wPort = 4000
sp(admxrc_serverapplication_vx,szIP,wPort)
```

The IP address and The IP address should be set to match the address of the Target. You can set the port as desired.

- `admxrc_serverapplication_vx` - handles semaphores, spawns the configuration server,
- `admxrc_configurationserver` - Opens and listens on a given port for configuration packets. Handles reconfiguration, application shutdown, and restart.

XAPP412 (v1.0) June 29, 2001

Architecting Systems for Upgradability with IRL (Internet Reconfigurable Logic)

Summary

Internet Reconfigurable Logic (IRL™) is a system design methodology to enable the remote upgrade of hardware, while insuring the reliability of the upgrade. FPGAs, which are “Field Programmable” are inherently capable of changing their functionality with a new bitstream. IRL takes advantage of this capability by delivering new bitstreams and software drivers to the remote hardware.

This application note will describe the basic concepts of an IRL-enabled system, detail design considerations for building an IRL system and give a high level description of the PAVE Framework, the Xilinx API and development framework that enables embedded systems to be upgraded.

Introduction

The advent of Xilinx FPGAs, Flash Memory devices and ubiquitous networks provide the means to store bitstreams and then upgrade them once the hardware has been shipped to the final customer. Architecting your system for IRL will allow you to upgrade software, drivers, firmware, and hardware remotely.

Reasons for enabling your system for field upgradability include:

- Interoperability - Products frequently have to interoperate with other vendor’s products, but there is no reasonable way to test all the possible interactions prior to shipping the product. If the system is IRL-enabled, interoperability issues can be resolved at a minimal cost.
- Time To Market - The hardware can be shipped sooner with a subset of the full functionality. Features that would have taken too long to add prior to the initial release can be added after shipment.
- Design Corrections - In the event a flaw in the product appears after it ships to the final customer, it can be corrected without the need for returns, recalls, field service, and the accompanying customer dissatisfaction
- Performance Upgrades - The performance of the system can be upgraded as the engineering team has time to tune the algorithms and data paths.

IRL Concepts

What is IRL?

Internet Reconfigurable Logic is a system design methodology that enables modification and upgrading of hardware and software in a target system across a network without the need for a service technician or user to directly perform the change. This methodology, when applied to the design process, creates products that are IRL-enabled. IRL can enable upgrades of multiple systems simultaneously, and the ability to go back to a previous configuration if necessary.

A typical IRL-enabled system might include a

- A 32-bit processor based design with TCP/IP networking connectivity. An industry standard example of this is the **Single Board Computer** (SBC), as typically seen in CompactPCI and VME implementations.
- **Real Time Operating System** (RTOS) such as the WindRiver® Systems’ VxWorks®
- Xilinx PAVE (**P**LD **A**PI **V**xWorks **E**mbedded) Framework

When an upgrade is available, it would be sent to the target, where the PAVE API would perform the upgrade. For example, a system, when IRL-enabled, might be able to autonomously upgrade itself and recover from a power failure during this upgrade.

Elements of an IRL system

Creating an IRL-enabled system requires certain hardware and infrastructure components that will allow the remote modifications to occur. As shown in Figure 1 below, there are several elements to an IRL System.

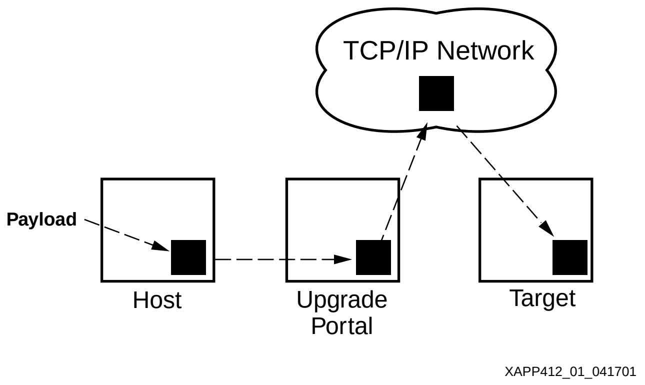


Figure 1: **Block Diagram of Internet Reconfigurable Logic System**

The **Host** is where hardware/software design environment resides and where the FPGA bitstreams and application software are created. This would include the Xilinx design tools, the RTOS build environment (such as WindRiver Systems' Tornado®) where your software applications are developed, and the PAVE System Integration Framework (SIF), which ties all of these efforts together.

Once the upgrade is created, it is assembled into a **Payload** that is sent to the system to be upgraded. The PAVE Framework includes utilities that allow generation of the payload for the build environment on the Host.

The **Upgrade Portal** is the computer your Target communicates with to obtain the payload. This could reside in your domain, or your end customers could operate it.

The **Network** shown in Figure 1 can be any TCP/IP based network: an Intranet, a local network, a Virtual Private Network (VPN) or even the public Internet. The type of network used will depend on the security requirements and the connectivity available at the location of the final product. PAVE can perform a basic TCP/IP socket connection; any additional protocols for security or other purposes would need to be added by the developer.

The **Target** system is the system that needs the hardware and/or software upgrade. This is the product shipped to your customers and which resides remotely. This IRL-enabled target system will, at a minimum, have a processor running the user's application, the PAVE API (part of the PAVE Framework), the RTOS runtime client (such as WindRiver Systems' VxWorks), and an FPGA. The processor handles communication with the network and has connectivity to the FPGA. The PAVE API is called to perform the upgrade by the user embedded application.

A typical payload structure is shown in Figure 2. Since changes in hardware usually imply new software drivers, these are included in the payload structure, so the drivers can be upgraded concurrently with the hardware. The applications that run on the target can be upgraded as well.

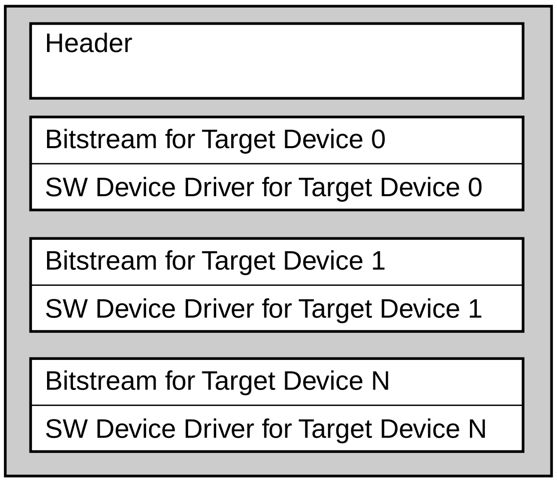

Header
Bitstream for Target Device 0
SW Device Driver for Target Device 0
Bitstream for Target Device 1
SW Device Driver for Target Device 1
Bitstream for Target Device N
SW Device Driver for Target Device N
X412_02_041701

Figure 2: **Payload Diagram**

Expanding on the block diagram in Figure 1, an IRL system in the field could look similar to Figure 3. Here we have a target processor, a system or peripheral bus, and the FPGA(s). The processor is running the user's application, PAVE API, and the WindRiver RTOS. The Upgrade portal is running a PAVE client that communicates with the PAVE Server running on the target. The payload passes from the host to the target, via the upgrade portal and the Internet. Once it arrives at the target, the PAVE Server and API perform the required functions to upgrade the system.

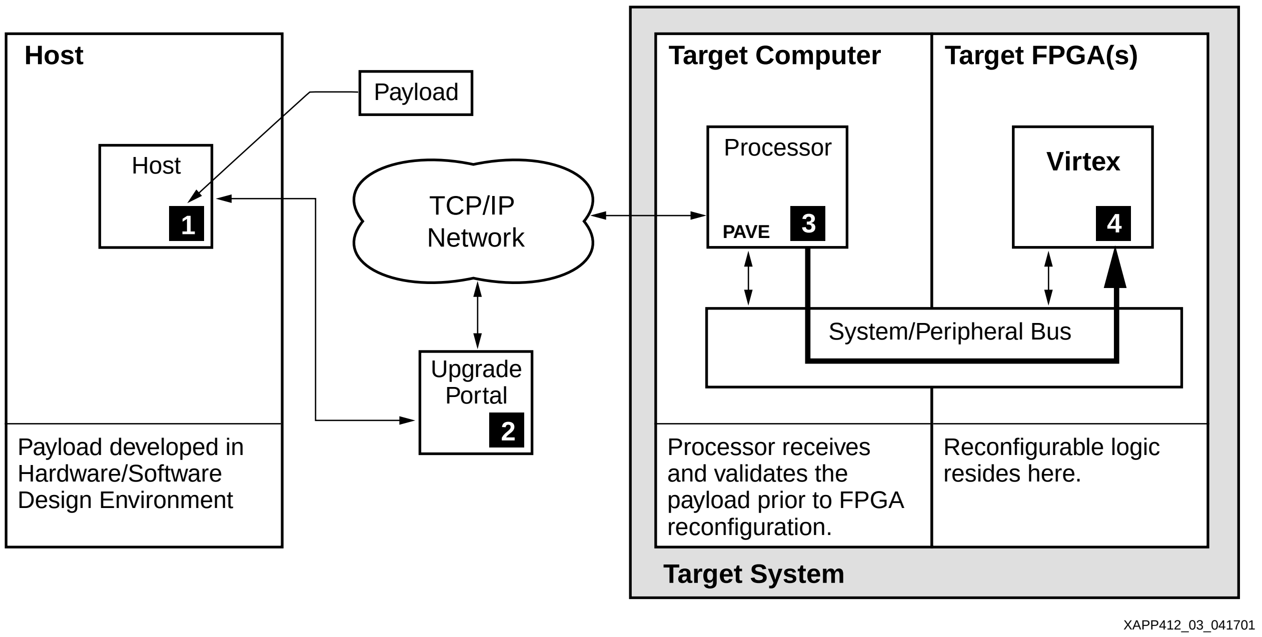

Host
Payload
Host
1
TCP/IP Network
Upgrade Portal
2
Payload developed in Hardware/Software Design Environment
Target Computer
Target FPGA(s)
Processor
PAVE
3
Virtex
4
System/Peripheral Bus
Processor receives and validates the payload prior to FPGA reconfiguration.
Reconfigurable logic resides here.
Target System
XAPP412_03_041701

Figure 3: **Fielded IRL System**

Host, Upgrade Portal, and Network Concepts

The beginning of the upgrade process is the creation of new FPGA designs and accompanying software drivers, followed by testing in an appropriate environment. Once the upgrade is ready to be sent to the field, the developer uses the utilities supplied with PAVE to create the payload.

After the payload has been assembled, the developer would publish it out to the Upgrade portal, similar to how files are published for internet delivery. Once the payload has been published to the upgrade portal, there are two main means to deliver the payload to the target system.

Push (see Figure 4) is similar to broadcasting; the payloads are sent by the upgrade portal to each target system. This allows the Upgrade portal to control the upgrade process and ensure all systems have been upgraded.

Pull (see Figure 5) is similar to FTP; the target system contacts the upgrade portal to see if new upgrades are available. If so, the payload is pulled off the portal by the target.

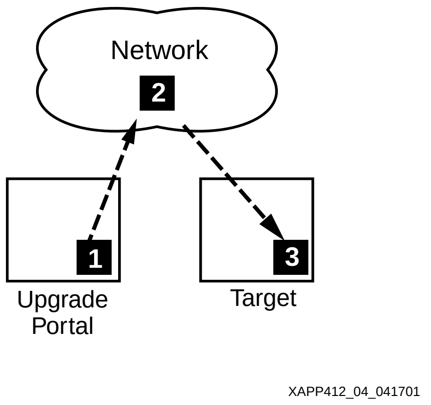


Figure 4: **Pushing a payload to the target**

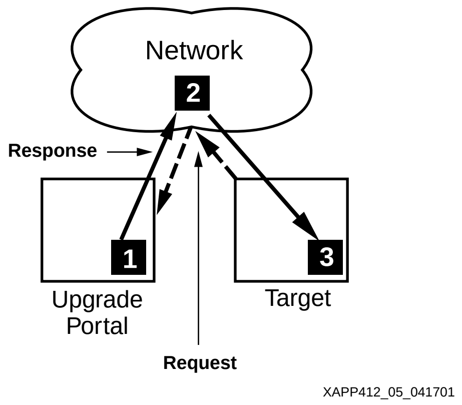


Figure 5: **Pulling a payload from the upgrade portal**

Careful consideration of using push vs. pull should be done to ensure that upgrades do not interfere with the end user's operation of the system. The operator of a high-availability system, such as the telecommunication services, might run the upgrade portal; in this case push would offer complete control over the process. A user of a low-cost consumer product would not have control of the upgrade portal. This user might prefer to have the option of upgrading or not; in this case pull would be the best choice. If the upgrades are not free, the upgrade portal may need to authenticate the user to ensure the upgrade was purchased.

Target Software Concepts

Figure 6 is a model of the software stack that runs on the target. At the highest level is the user applications. Running concurrently with the application is the PAVE API and server that caches the payload, and then performs the upgrade.

On the second level, the PAVE API provides system calls for the customer C++ applications to perform the reconfiguration process. The customer applications and API both interface directly with the RTOS.

The third level is the WindRiver RTOS. VxWorks is the run-time component of the Tornado II embedded development platform and acts as the operating system "kernel" on your target system. PAVE works directly with the VxWorks RTOS.

The **Board Support Package** (BSP) in level four in the stack is required to interface the desired processor to the RTOS. Each different SBC running an RTOS will need a Board Support Package to abstract the processor from the RTOS. The BSP used must match the RTOS and the embedded processor combination used in your system. PAVE assumes the existence of the BSP.

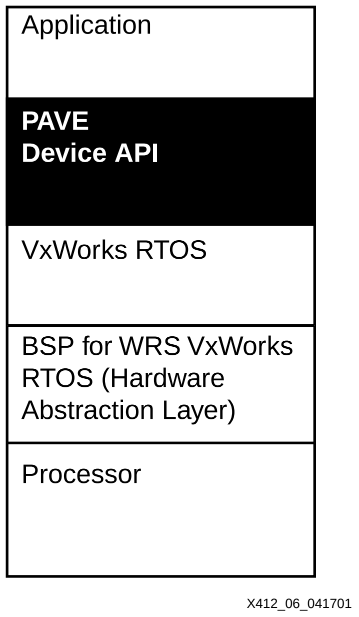


Figure 6: **Target software stack**

Target Hardware concepts

Processor Coupling

In the embedded market, processors have a bus known as the **Processor Local Bus** (PLB) that is directly fed from the processor and an **Embedded System Bus** (ESB), such as PCI, that usually requires a bridge or host chip to interface from the system bus to this secondary bus. The PLB varies depending on the processor and is not a standardized bus like PCI. The Embedded System Bus is not to be confused with the term "system bus", widely used in PC architectures to refer to the PLB. Connecting to the processor through an ESB is considered to be **Loosely coupled** and connecting through the PLB is considered to be **Tightly coupled**. In Figure 7 we see an example of these two different processor couplings.

Until recently, advanced processors (32-bit) could only be accessed through bridge chips supplied by the processor vendor. This would lead to a multi-chip connection, which added performance bottlenecks, consumed board space and power, and added cost to the design. Now, with programmable logic, it's possible to directly access the processor local bus, eliminating this series of chips, which is enhancing the importance of tight coupling to the PLB in newer designs.

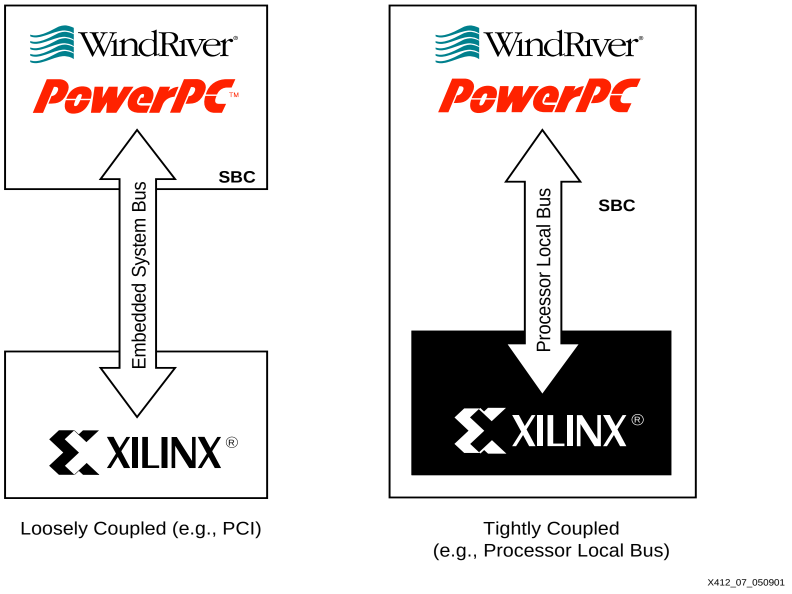


Figure 7: **Processor coupling**

Double Buffering

FPGA bitstreams are frequently stored on flash devices (including Xilinx XC1800 series devices), which can experience problems if the power fails while being written. IRL involves designing your hardware so that it is impervious to power failures during the upgrade process. The goal is to never have a piece of hardware that fails to operate.

For the IRL hardware to meet this requirement, it should have a **Double Buffer** design. One example method could consisting of a **Default** configuration that is always available and a second configuration that can store the upgrade.This Default configuration is never upgraded or changed except at the factory. Addition of the second storage location allows upgrades to occur, since the Default can not be changed. Double buffering ensures the hardware can be reliably upgraded.

Rollback is the ability to revert to a previous upgrade (possibly the Default). In a system that has space for more than two configurations, (e.g. using a commodity flash chip), it could rollback to a known good upgrade that was previously installed.

IRL Examples

Having examined the concepts that make up the IRL design methodology, let's examine a few practical examples of how to implement an IRL-enabled target system using PAVE.

Basic IRL-enabled System

Figure 8 shows an IRL-enabled system with a processor, an FPGA, and multiple FPGA configuration storage areas. The Processor communicates with the FPGA and, after configuration, can perform an update of the upgrade PROM. A register in the bridge address space receives the new bitstream and writes it out to the PROM via the JTAG controller.

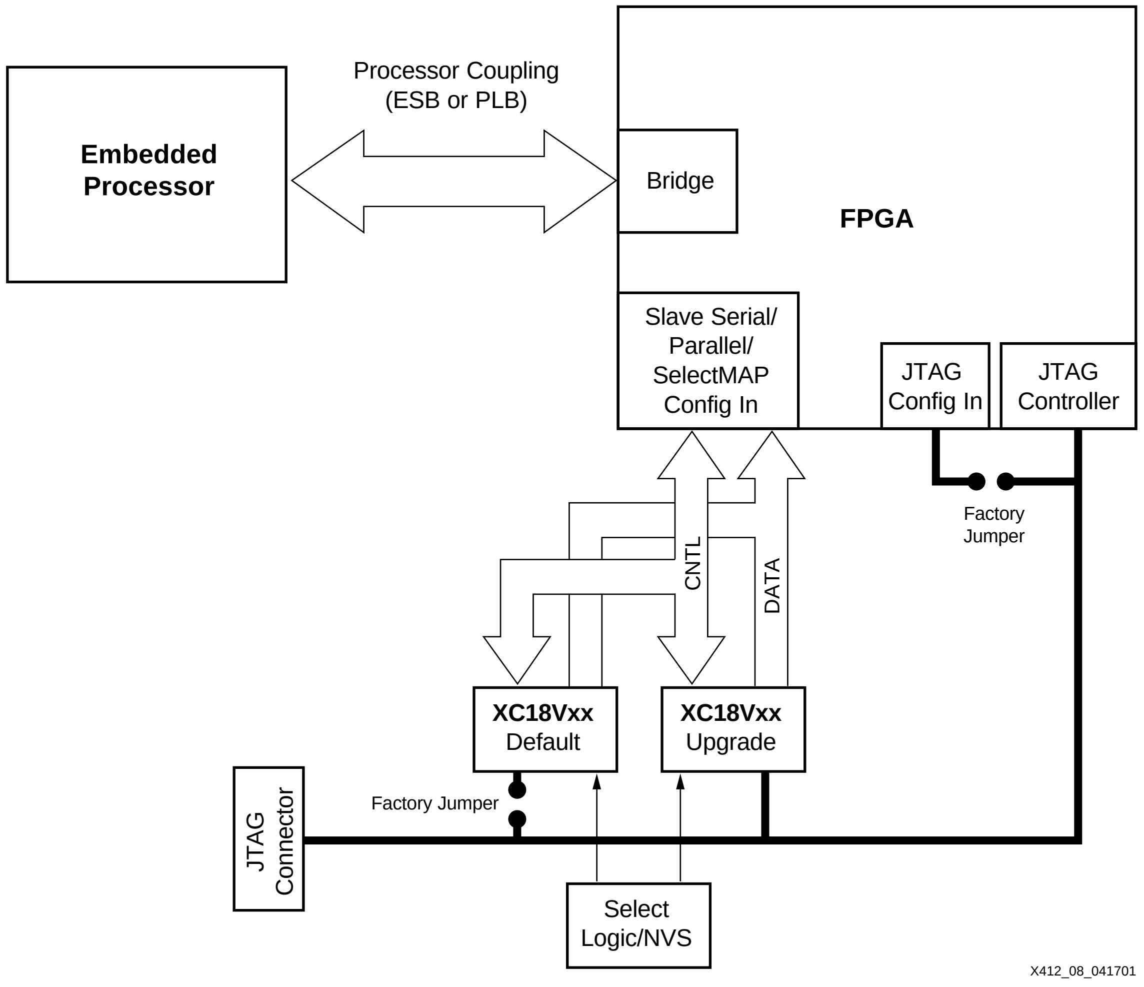


Figure 8: **Example two PROM system**

The PROM marked "Default" is the known good configuration from the factory. The default should never be upgraded in the field as it provides a baseline configuration that the hardware can revert to in case of failure of the upgrade process. This protects the hardware against power failures, customer or technician mistakes, and any other failure mode that would render the hardware inoperable (and non-upgradable). By preventing the end user from updating this PROM, he will always have a fallback position in the event the upgrade fails. The factory jumpers on the Default PROM's JTAG lines physically prevent the changing of this PROM, except during the manufacturing process. The upgrade PROM can be changed through the JTAG controller in the FPGA. With only two storage locations, the new upgrade always overwrites the old upgrade.

The Select Logic and Non-volatile storage (NVS) is to determine which PROM should be used and use the default if a configuration error occurs during the loading of the upgrade. In it's simplest form, it would attempt to load the upgrade PROM, monitor the DONE line of the FPGA, and if it failed, automatically revert to the default PROM. Adding a small NVS device, such as a Dallas Semiconductor DS2430A (scratchpad EEPROM) would allow specifying which PROM to boot from initially. This NVS could allow a more sophisticated approach of choosing among multiple upgrades. The select logic could be a CPLD or even something simpler, but, like the default, it should not be modifiable outside the factory (unless there is a double buffer for the CPLD configuration).

In the event of a configuration fault, the select logic should be able to detect this and attempt to configure the FPGA with the Default bitstream. If the bitstream in the upgrade buffer is corrupted or non-existent, the FPGA DONE signal will not go high. In this case the select logic should attempt to load the default bitstream.

IRL in a Bridge System

Figure 9 shows an IRL system with a bridge and the two PROM model discussed in the last example. The bridge FPGA initializes off the PROMs; subsequently the target FPGA can be configured from the processor through the bridge. In the previous example the FPGA was both the bridge and the target. The interface in this case could be with either the ESB or PLB. A register in the Bridge interface would accept the configuration data sent from the processor and pass it on to the target via either the SelectMAP or JTAG controllers.

Use of a bridge in your system is not an IRL requirement; this example may or may not apply to your design. This figure is an example of how you could perform double buffering, but not the only way.

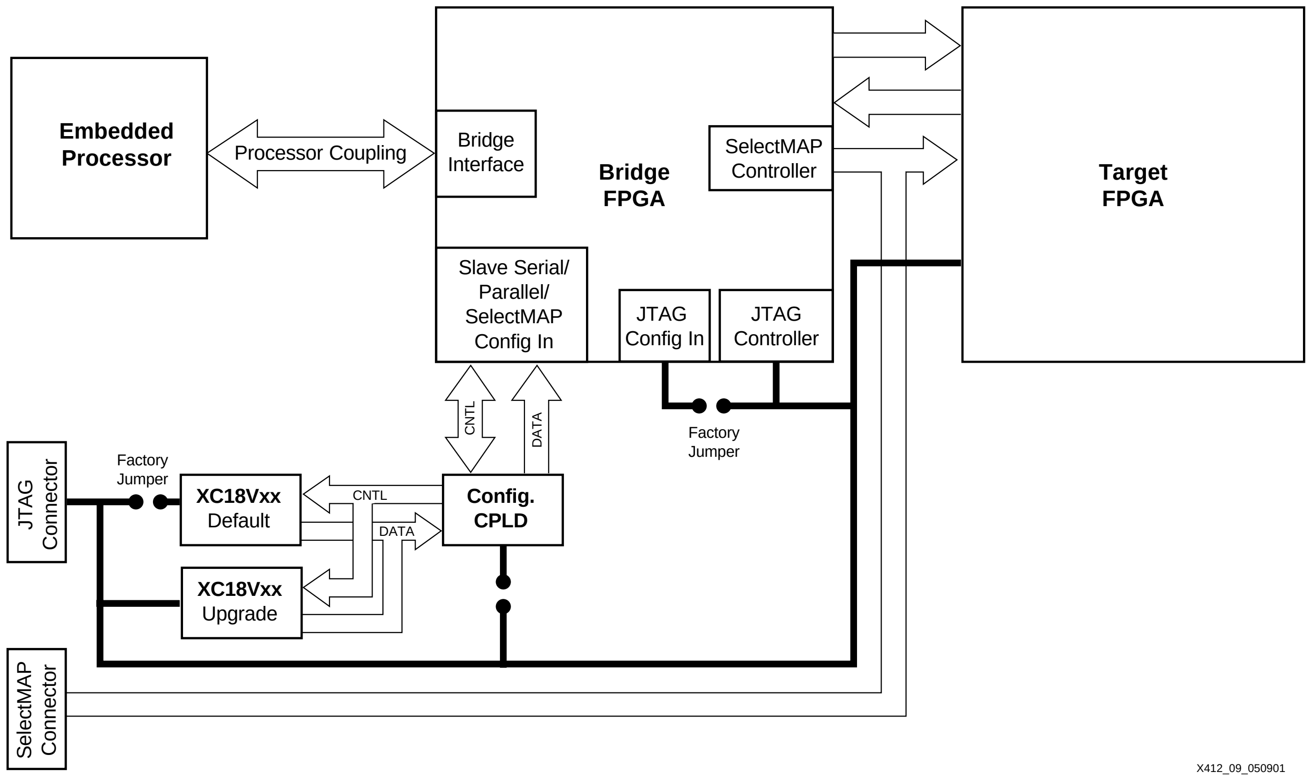


Figure 9: **System with Bridge and Target FPGAs**

In a programmable bridge system the processor cannot directly send configuration data prior to the initial configuration of the bridge FPGA. All of the aforementioned details on insuring a known good configuration still applies to this bridge. For the target FPGA in this diagram, the processor is able to send configurations directly to it from the processor's data storage. In this case, two means of configuration supported under PAVE are shown, SelectMAP and JTAG. The select logic used by the bridge is a CPLD that is acting as a mux for the two PROMs.

General IRL System Considerations for Bridges

Communication between the Processor and the target FPGA occurs through a bridge. The bridge facilitates the interface to the processor through the specified interface (e.g. ESB, PLB). Most processors require a separate chip (a Bridge) to support an ESB. When using a bridge chip, the processor is not directly mastering the bus to the FPGA. A few processors do have direct ESB support on chip. These are considered to have the bridge built-in; this bridge would be non-upgradable.

Most SBCs do not provide direct access to the PLB via a plug-in form factor. In the case of a CompactPCI system, a form factor known as PCI Mezzanine Card (PMC) is typically used. A PMC card loosely coupled to the processor could be on the SBC board, or a PMC carrier in the same chassis. A tight coupling would be the processor local bus (PLB), such as the PowerPC 405GP peripheral bus that is fed directly from the processor. The upgrade to the FPGA passes through this coupling and into the FPGA; this data is then updated into the appropriate storage area.

Memory usage for storing bitstreams

Building on the models in last two examples, this next example adds additional memory space for bitstream storage. Figure 10 shows a loosely coupled PMC system with configuration flash in addition to the two PROMs. This flash chip is a standard commodity flash, which are available in varying sizes. Depending on the design, the flash chip could store additional Bridge bitstreams, while depending on the Processor to supply the configuration to the target FPGA, or target FPGA configurations could be stored there as well. Flash chips are able to store much larger amounts of configuration data, and this could translate to multiple upgrades or support for the largest FPGAs.

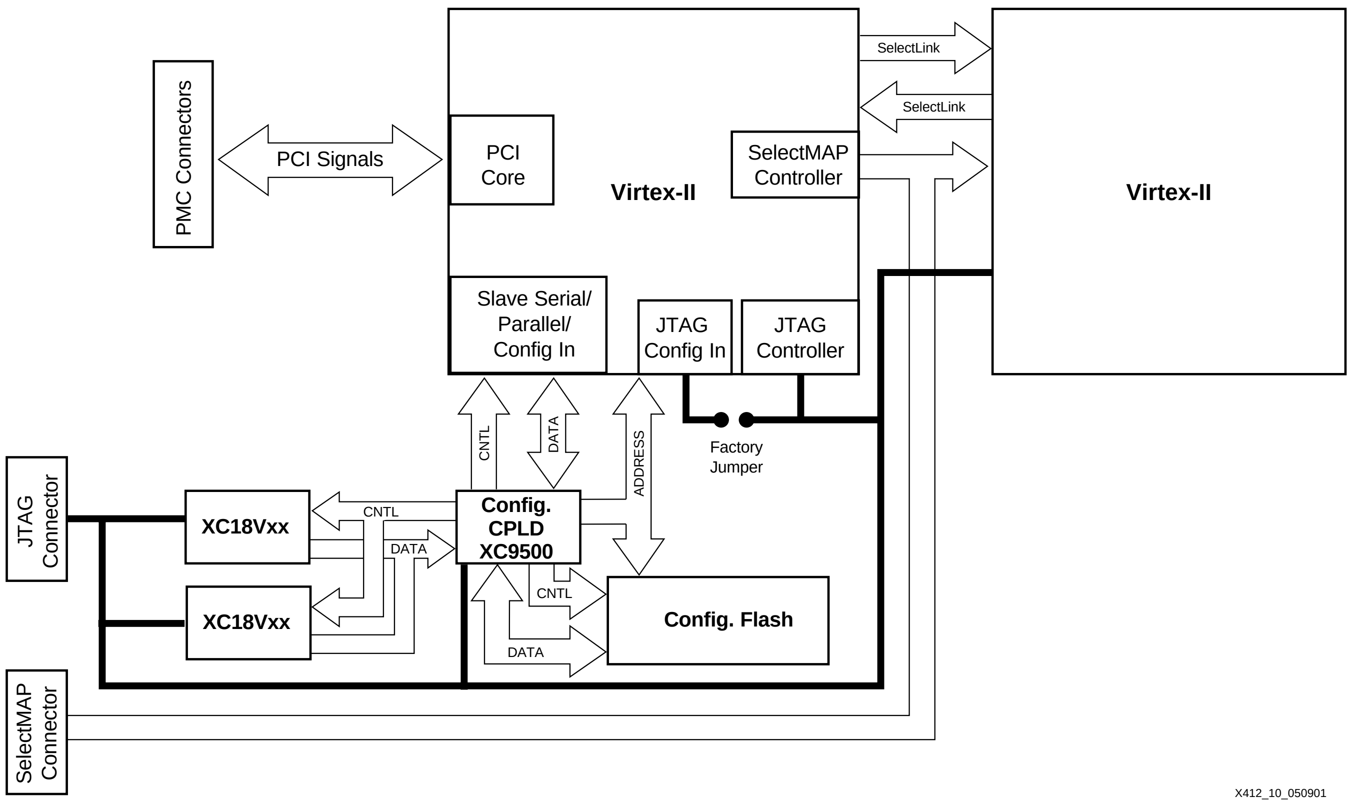


Figure 10: **PMC example with Bridge, PROMs, and Flash**

In this case, the CPLD is considered to be a thin device, basically a data mux with the majority of the logic in the FPGA. The address lines feeding from both the FPGA or CPLD to the flash chip would allow it be controlled from either chip.

In this example the default could reside in the Configuration flash or in a PROM; thus no jumpers are shown on the PROMs. If this is the case, it's the responsibility of the system designer to ensure fail-safe operation.

Use of PAVE in IRL Systems

The PAVE Framework is an embedded applications software development framework that can be employed to facilitate the development of reconfigurable embedded applications.

Object Oriented Hardware

The PAVE Framework and its components are a collection of C++ classes and object models that abstract an implementation of a Xilinx FPGA, called the IRL-enabled Device implementation. PAVE treats the programmable hardware as an object within the system, similar to software objects used in C++. As a result, applications that are written using PAVE tend to be highly object oriented, modular, and extremely upgradable. You can change a single module without replacing the whole framework.

SelectMAP and JTAG support

For PAVE 1.0, the programming interfaces supported are SelectMAP and JTAG, via the configuration register contained in your design, typically in the bridge. When compiling the design under the PAVE, you define the location of this and any other user registers in the device memory map. PAVE will encapsulate this programming interface and generate C++ source and header files and associated project files based on your design definition.

Available Development Platforms

Several development platforms that can be used for IRL are available today:

Motorola

The Motorola MCP750 SBC has the following features:

- MPC750 Power PC processor
- A PMC slot
- Ethernet connection
- Compact Flash

Motorola Computer Group can be contacted at:

http://www.mcg.mot.com

Alpha Data

The Alpha Data ADM-XRC is a PMC card that allows reconfiguration of the FPGA across a bridge. Details can be found at:

http://www.alphadata.co.uk/dsheet/adm-xrc.html

WInd River Systems

Wind River Systems makes the Tornado-II RTOS development platform.

http://www.windriver.com

Xilinx

Xilinx offers IRL training and the PAVE Framework.

http://www.xilinx.com/xilinxonline

Summary

With minor hardware and software changes, you can enable your systems for IRL and add much value for both you and your customers. The addition of IRL to your product will extend it's life and simplify support and distribution models. With IRL, you could manufacture a single physical version of your hardware and ship multiple different hardware versions. And your customers will appreciate the speedy, hassle-free upgradability of your products.

The Xilinx PAVE Framework provides a powerful software framework that allows designers to easily integrate IRL into their designs. The object oriented nature of PAVE eliminates the need to handle low level issues with JTAG or SelectMAP programming, allowing the designer to focus on the end-user's application.

Future revisions of the PAVE Framework will bring additional functionality to your IRL-enabled design. The modular nature of PAVE will allow you to add new features without disturbing your current application framework.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
6/29/01	1.0	Initial Xilinx release.

Using Durango with the MCP750 and PAVE

Durango Board

Durango, shown in Figure B-1, is a platform for demonstrating key IRL concepts and is a model for the IRL architecture. It is a PMC card that includes a Virtex-II PCI bridge and a second Virtex-II FPGA as the target of the IRL reconfiguration in PAVE v1.0.

Durango was designed to implement a target architecture for a Xilinx-based reconfigurable system. The block diagram in Figure B-2 represents the portion of the design utilized in the application and integration of the PAVE v1.0 functionality.

The architecture and design of Durango was intended to provide a platform to develop feature beyond the initial PAVE v1.0 release. Additional information on the Durango board / reference design can be found in Appendix C.

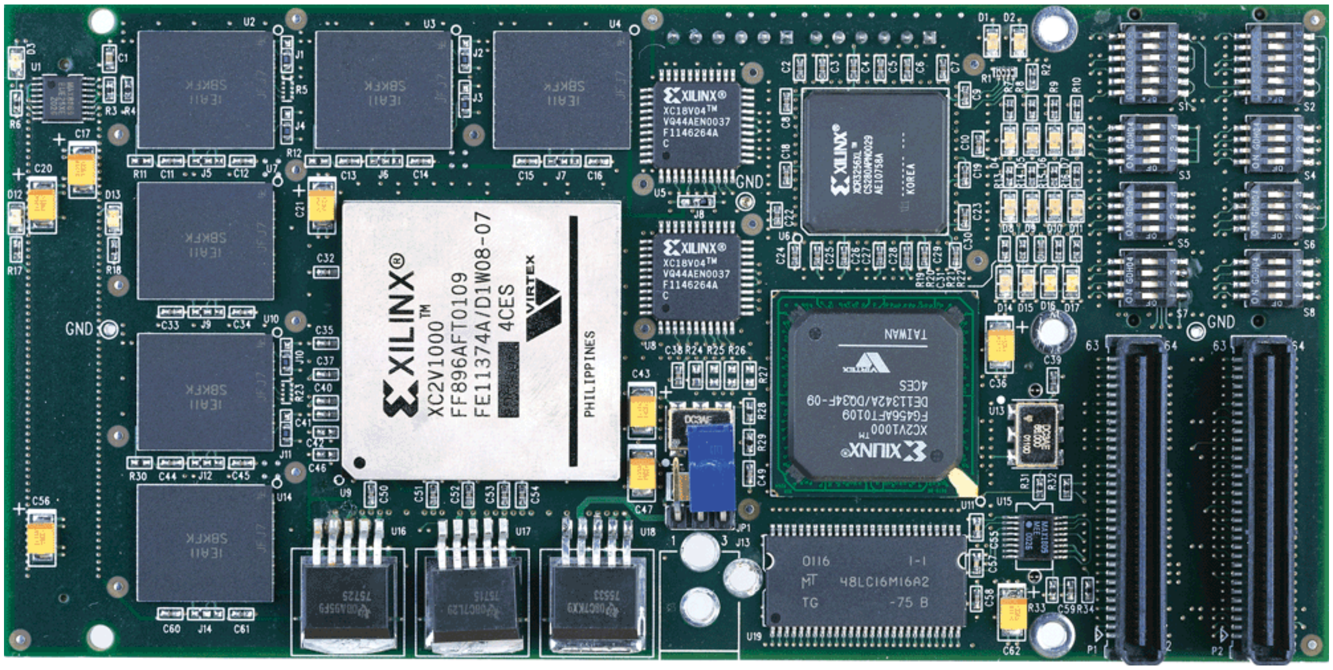


Figure B-1: **Durango IRL Reference Design (PMC side)**

Note: When viewed in Adobe Acrobat, the picture above appears best at ~200% magnification.

Durango Block Diagram

Features of the Virtex-II-based Durango supporting PAVE v1.0:

- SelectMAP Configuration PAVE v1.0 (XC2V1000)
- JTAG Configuration, PAVE v1.0 (XC2V1000, XC18V04)
- Virtex-II PCI Bridge
- Virtex-II Target
- XC18V04 Non-volatile Storage
- 32-bit, 3.3 V PCI compliant Interface
- PMC PCI
- XIRL Register
- Direct SelectMAP connector for debugging.
- Direct JTAG connector for debugging.

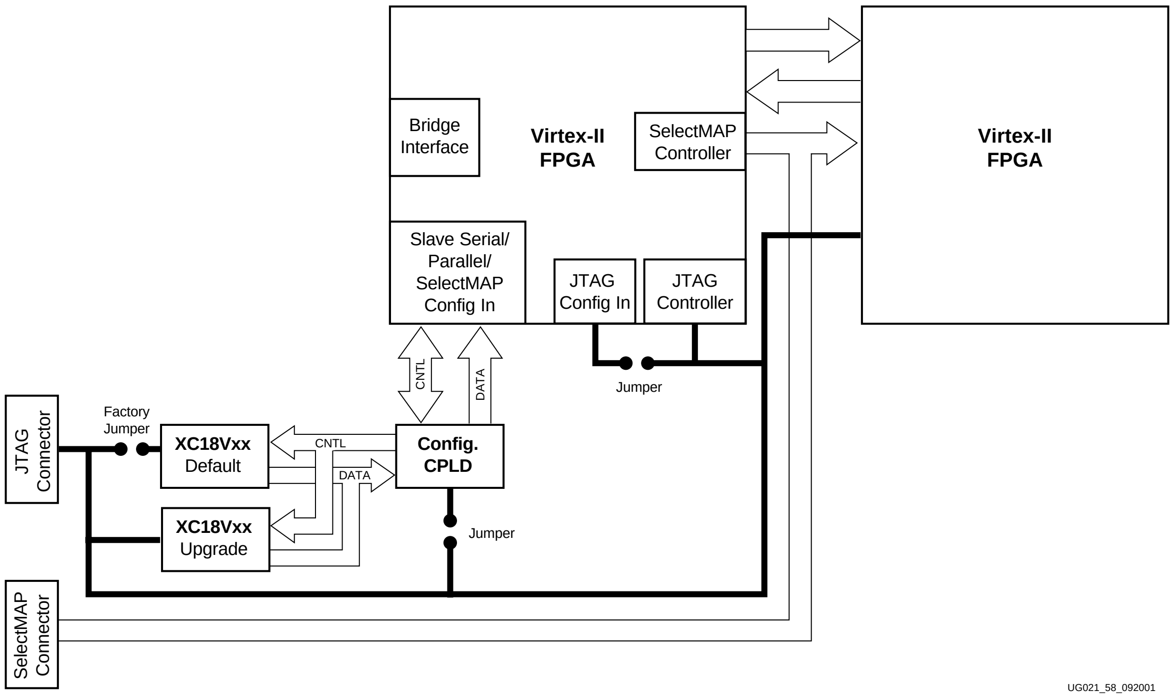


Figure B-2: **Block Diagram of Durango Features Supported in PAVE v1.0**

Durango MCP750 PAVE Implementation

The PAVE Framework was tested in two hardware platforms. One of these platforms was a combination of the Motorola MCP750 and the Avnet / Xilinx Durango board. The IRL host used an Ethernet TCP / IP connection to download the VxWorks images and payloads to the MCP750 board. The other platform was the Alpha Data ADM-XRC / MCP750, covered in Appendix D.

The MCP750 is a Single Board Computer (SBC) based on the PowerPC processor. It comes in a 6U CompactPCI form factor. Figure B-3 shows the block diagram of the MCP750 and the Durango board as the hardware was tested.

The Durango is a Virtex-II-based design (XC2V1000) and has been integrated with the MCP750 over the CompactPCI backplane. The ADM-XRC is a Virtex-E-based design (XCV1000E), housed on the PMC slot of the MCP750. The communication between the MCP750 and the ADM-XRC (see Appendix D) was performed over the onboard PCI bus. PAVE provided scalability of existing code, not only for evolution from a Virtex-E to a

Virtex-II product, but also across the unique board designs. Even with the layering difference of PCI bridging and respective latency, building this solution with the Wind River RTOS, VxWorks, enabled seamless scaling across the partitioning of the Target FPGA within the system.

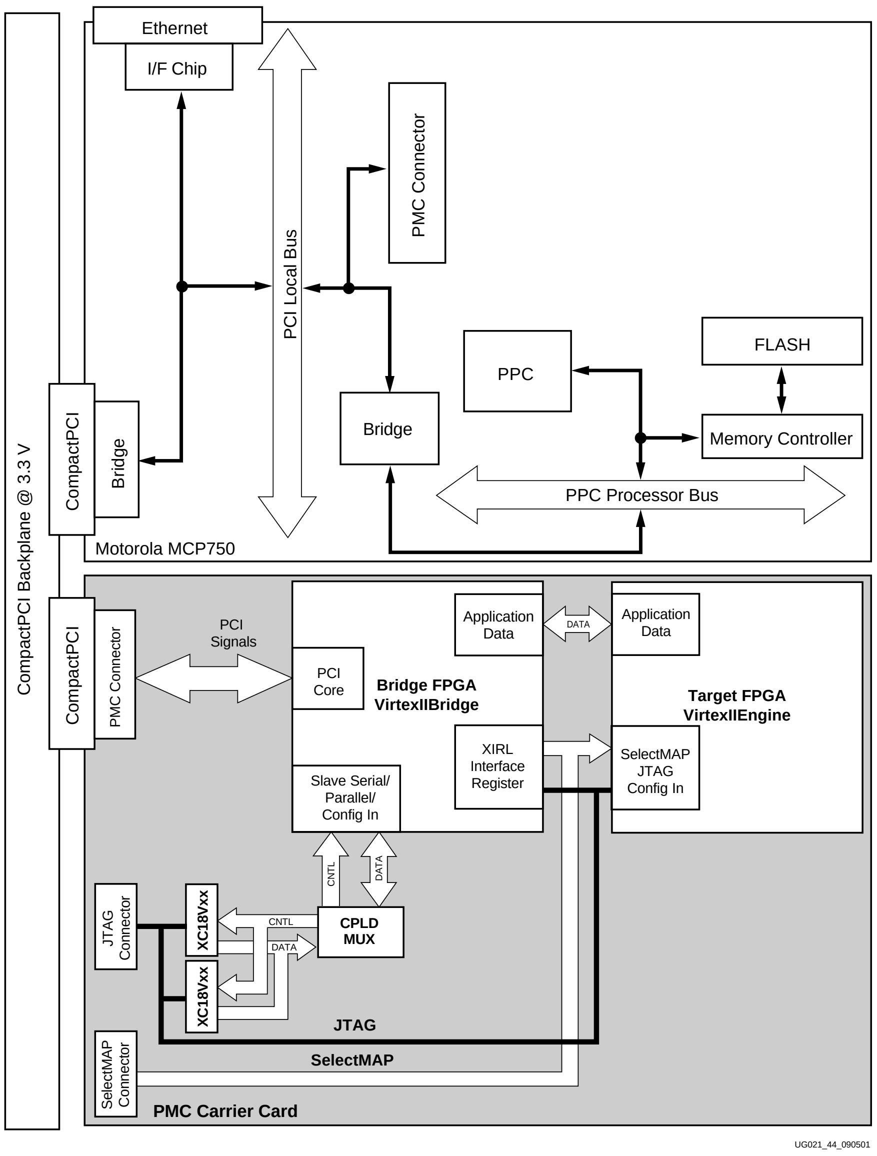


Figure B-3: **MCP750 with Durango Block Diagram**

Durango Framework

The Durango software framework can be found under `_platform_mcp750durango`. The `_platform_systemgenerator` directory is essentially an un-generated version of the Durango framework. Running the system generator on `_platform_systemgenerator`, as outlined in the getting Started chapter, will create a directory identical to `_platform_mcp750durango`.

The `_platform_mcp750durango` framework is initially ready to open in the Tornado tools, when installed in `D:\`.

Code Examples

The following illustrates a typical program for reconfiguring the Durango server component. The program is very basic in that it simply loads a payload into memory, writes it to the targeted device, and then frees the memory when the operation is completed.

```
Void durango_tutorial(
void *ptrArguments)
{   char                *ptrBaseAddress;
    void                *ptrVirtexIIEngineBitstream,
    void                *ptrVirtexIIEngineModule;
    structPayloadHeader     strPayloadHeader;
    pntr_classDurango       ptrServer;

// Initialize local variables.

ptrBitstream = NULL;
ptrModule    = NULL;

// Instantiate a dummy address space for this test.

ptrBaseAddress = (char *)calloc(4096, sizeof(char));

// Instantiate a server component.

ptrServer= new classDurango(ptrBaseAddress);

// If the server was successfully instantiated,

if (ptrServer != NULL)
{
    /* Pull the payload of the local file system into allocated memory for
       all three target devices. */

    ptrServer->ptrVirtexIIEngine->CachePayload("testpayload.bin",
       &ptrInputFPGABitstream,
       &ptrInputFPGAModule);

    // Load the payload into its targeted device.

    ptrServer->ptrVirtexIIEngine->UploadPayloadFromBuffer(
       strPayloadHeader.theDeviceConfiguration[0].dwBitstreamSize,
       ptrInputFPGABitstream,
       strPayloadHeader.theDeviceConfiguration[0].dwModuleSize,
       ptrInputFPGAModule);

// Free allocated buffers.

ptrServer->ptrVirtexIIEngine->UnCachePayload(ptrVirtexIIEngineBitstream,
            ptrVirtexIIEngineModule);
        delete ptrServer;}
else
{
// Add error code here.
}
free(ptrBaseAddress);}
```

Resources

- Motorola produces the MCP750 Single Board Computer. Additional details on the Motorola MCP750 can be found at:

  http://www.mcg.mot.com/cfm/templates/product.cfm?PageID=895&ProductID=22&PageTypeID=1
- Avnet sells the Durango board as part of a IRL Reference Design kit. Avnet also sells many of the system components used in development of PAVE.

  http://www.ads.avnet.com
  - Xilinx PMC IRL Reference Design Kit (Durango) can be ordered from Avnet Design Services. The part number is ADS-XLX-PMC-IRL
  - Durango Data sheet

    http://www.xilinx.com/partinfo/ds084.pdf
  - Additional Motorola components for this reference design are available from Avnet:
    - MCP750-1352(cPCI PPC Single Board Computer)
    - CPX2408-k (cPCI Enclosure)
    - CFLASH-001(10MB cFlash Memory Card)
- Tracewell Systems offers the T-Frame for Compact PCI which allows easy access to boards without the use of extender cards.

  www.tracewellsystems.com
  - 580-6001-F00-00 (cPCI Enclosure)
- ACT Technico sells PMC carrier cards for CompactPCI.

  www.acttechnico.com
  - 7000-3 cPCI / PMC Carrier card

XIRL Interface Register Tables

For your convenience we have included this table to assist in debugging the XIRL configuration register as you write new code and test it.

Register Name:	XIRL_Interface
BAR Space:	BAR1
Address Offset:	0x00000000
Width:	32 bits
Power Up Value:	0x01A0000

	R/W	
0	R/W	JTAG_BUFF_OE[2]
1	R/W	JTAG_TCK
2	R/W	JTAG_TMS
3	R/W	JTAG_TDI
4	R	JTAG_TDO
5	R/W	SMAP_BUFF_OE[2]
6	R/W	SMAP_CCLK
7	R/W	SMAP_RW[1]
8	R/W	SMAP_CS[1]
9	R/W	SMAP_PROG
10	R	SMAP_INIT[1]
11	R	SMAP_DONE
12	R	SMAP_BUSY[1]
15 14 13	R/W	SMAP_D[2:0][1]

Table B-1: **XIRL Interface Register Debugging Table (Lower Bits)**

	R/W	
20 19 18 17 16	R/W	SMAP_D[7:3][1]
23 22 21	R/W	MODE_M[2:0]
24	R/W	MODE_HSWAP_EN
31 30 29 28 27 26 25	R/W	Reserved

Table B-2: **XIRL Interface Register Debugging Table (Upper Bits)**

Notes:

1. Dual mode pins on Target device. Need to tri-state once configuration is complete.
2. Internal registered control signal that does not go out to Pad.

Signal Name Descriptions

- JTAG_BUFF_OE - Tri-state control for the output buffers on the JTAG signals, TCK, TMS, TDI. It is active LOW Tri-state enable and is LOW after power up.
- JTAG_TCK, TMS,TDI, TDO - JTAG signals used to program the Target FPGA.
- SMAP_BUFF_OE - Tri-state control for the output buffers on the SMAP signals. Active LOW tri-state control and is LOW after powerup
- SMAP_CCLK, SMAP_RW, SMAP_CS, SMAP_PROG, SMAP_INIT, SMAP_BUSY, SMAP_D[7:0] - SelectMAP signals used to program the Target FPGA via SelectMAP port.
- M[2:0] - FPGA mode pins. JTAG mode is the default mode after power up.
- HSWAP_EN - Controls the pull-ups during configuration while M[2:0] selects the desired configuration mode. The signal HSWAP_EN is active LOW. A logic 0 commands the IOBs to employ weak pull-ups during configuration.

Durango Reference Design

The Durango board is joint project of Xilinx and Avnet to provide an IRL reference design for a Xilinx-based embedded system. The Durango board is not only a reference design for our customers; it has been used in the development, integration, and validation of the PAVE API and SIF.

A reference design package is available from Xilinx supporting the PAVE v1.0 release. Appendix B identifies how Durango has been integrated in an embedded development platform to support PAVE v1.0 integration and validation. Durango is available for purchase from Avnet Design Services. Details on how to obtain the Durango reference package are provided later in this appendix. All other components of the development platform are commercially available. A system bill of material covering the components of the development platform is available; see **Additional Information**, page 96. Combining the PAVE v1.0 SIF, API and template applications along with the Xilinx and Wind River Systems tools enable the customer to create an upgradable platform.

Hardware Features

While Durango supports the required feature set for the PAVE 1.0 release, the architecture and design are also intended to provide a platform to develop features for future releases. Appendix B along with PAVE v1.0 support Durango for this release. The block diagram in Figure C-2 and the following list includes both components and features of the board to that will enable a flexible platform for continued development:

- Xilinx components:
  - XC2V1000-4FG456C
  - XC2V1000-4FF896C (upgradable to XC2V1500 or XC2V2000)
  - XC18V04VQ44C
  - XC95288XL-7TQ144C
  - XCR3256XL-7CS280C
- Memory:
  - Two: 16 MByte SDRAMs
  - Six: 512Kbx18 QDR SRAMs
  - One: 2 MByte Parallel Flash memory
  - Two: 4Mbit Flash memories
- Board I/O Connectors and Interfaces
  - 32-Bit, PCI Bus Interface Connector
  - IBM PPC405GP External Peripheral Bus Connector
  - 2 JTAG Connectors
  - SelectMAP Connector
  - AvBus™ I/O Connector

- Mictor Test Connectors
- Miscellaneous:
  - 50 and 60 MHz oscillator
  - LEDs
  - Dip Switches (JTAG chain)
  - Push Buttons
  - Battery backup for V-II bitstream encryption

Additional details on Durango are provided in the reference design zip file.

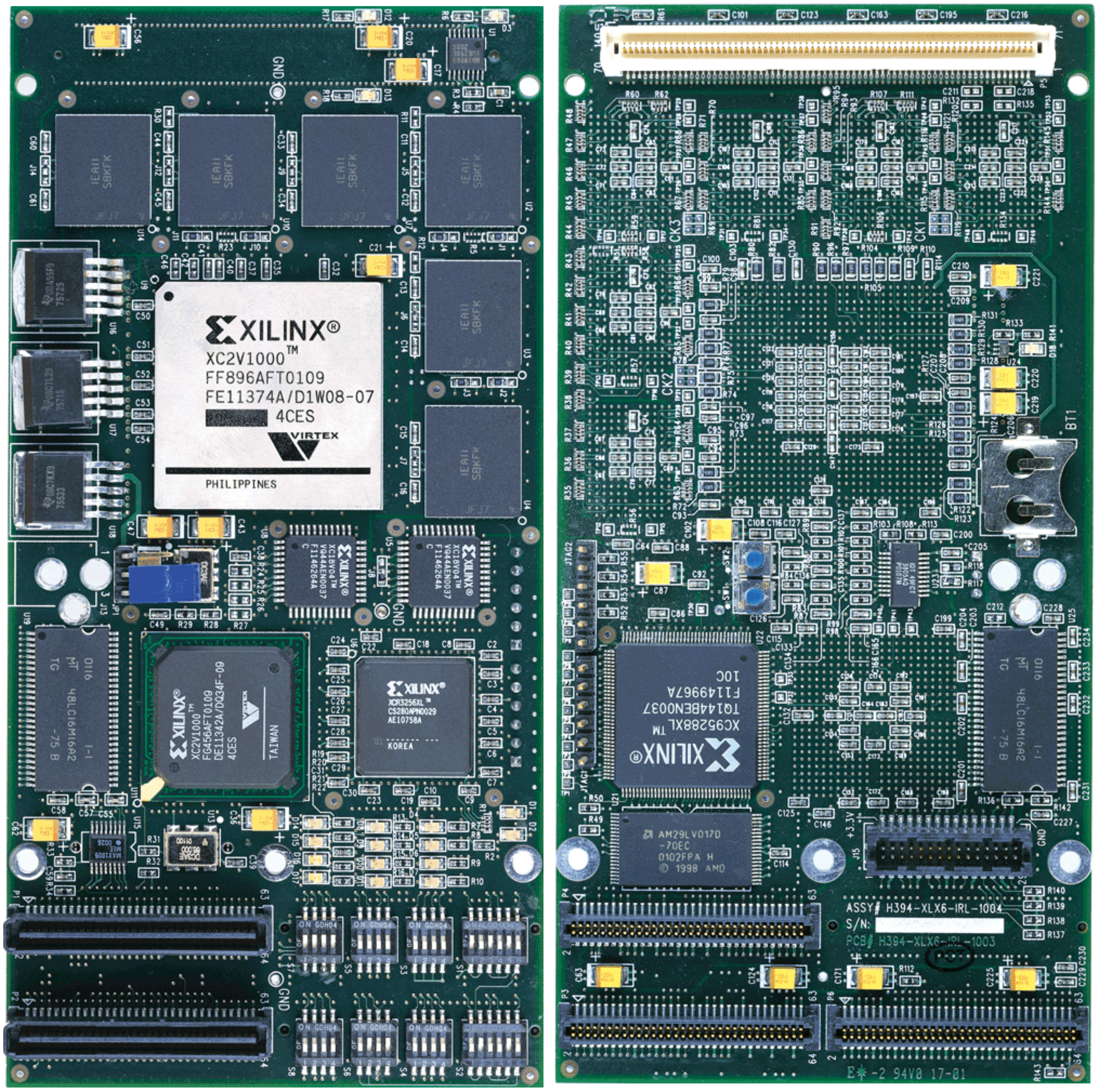


Figure C-1: **Durango IRL Reference Design Front and Rear**

Note: When viewed in Adobe Acrobat, the picture above appears best at ~200% magnification.

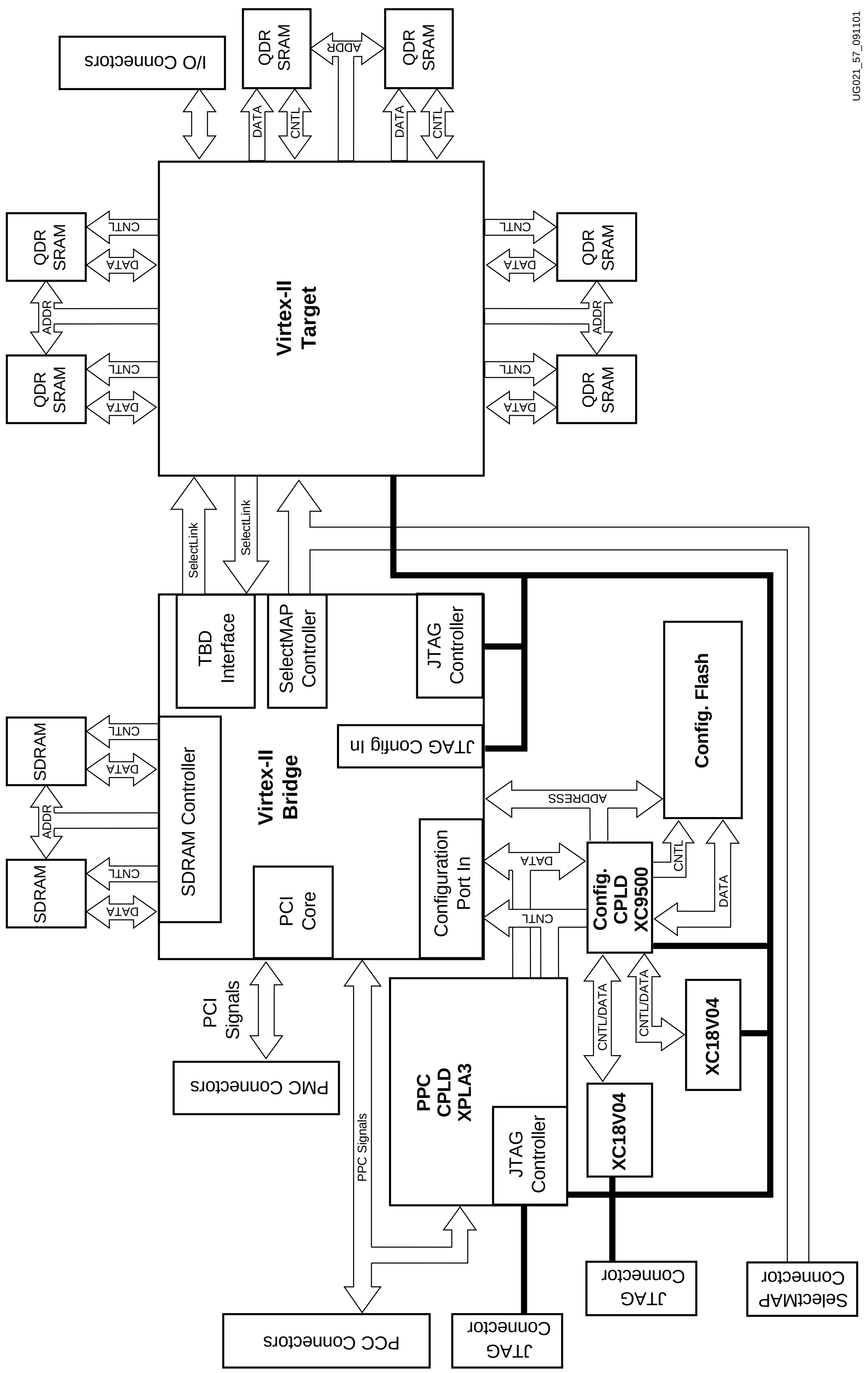


Figure C-2: **Block Diagram of Durango**

Note: To better view this drawing in Adobe Acrobat, click on the “Rotate View 90° CW” Icon in the Acrobat Command Bar.

Availability

The Durango board is available as part of the IRL Reference Design Kit, part number ADS-XLX-PMC-IRL which can be ordered via Avnet Design Services. Price, availability, and bundled options as described in detail at the Avnet Design Services web site at

www.ads.avnet.com

Additional Information

Durango has been tested with PAVE and the Motorola MCP 750 platform as detailed in Appendix B. As a Virtex-II based design, Durango does not support 5 V PCI.

In the PAVE Release zip file, Xilinx provides detailed information about the Durango hardware. This files included are:

- Schematics
- PCB layout (PADS)
- Bill of material, both for the Durango board and the development system.
- Bitstreams
- Source files for the FPGA and CPLD designs. *The Xilinx PCI core is not included as part of these source files.*

The latest version of this zip file can be obtained at:

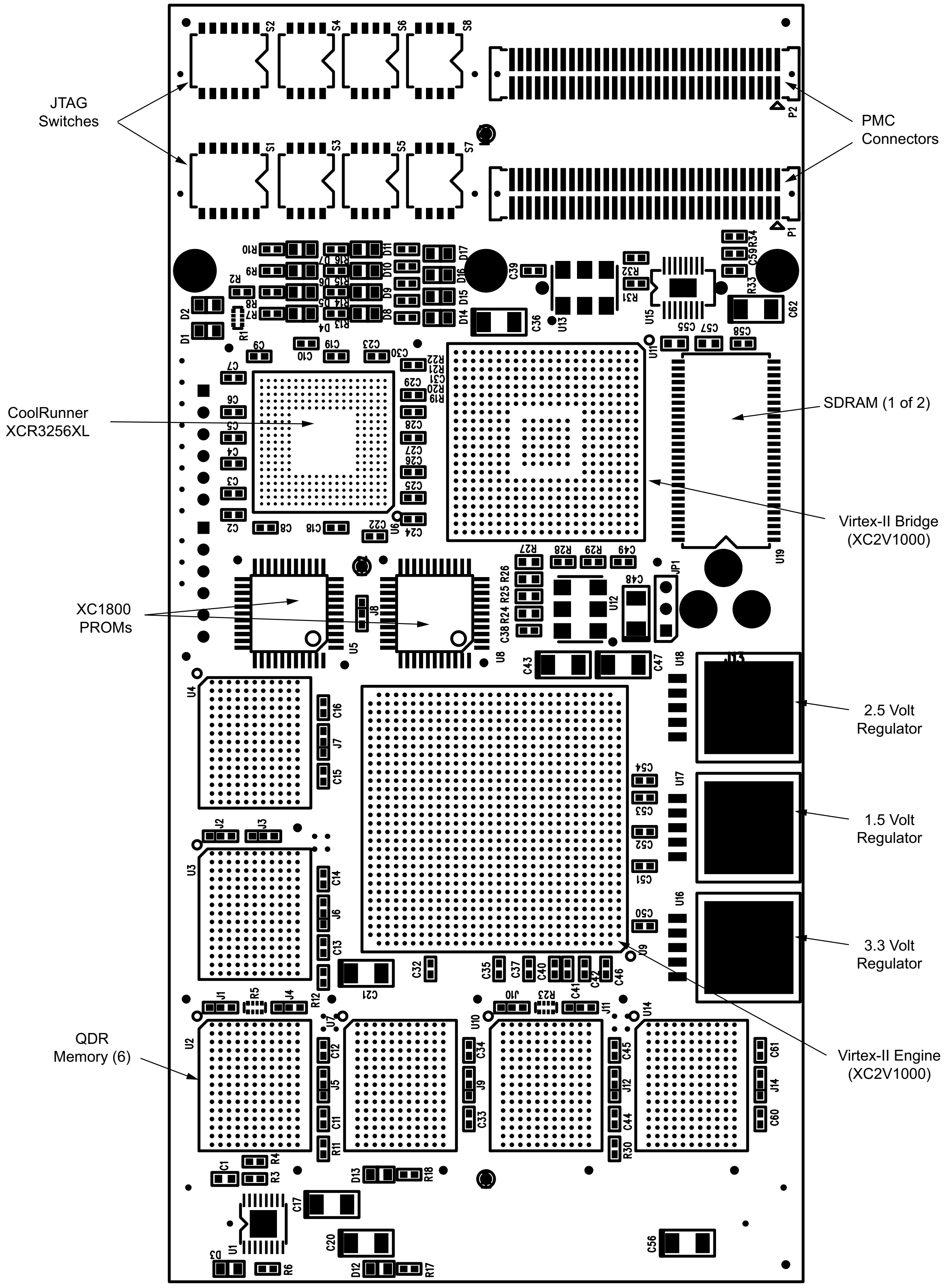
http://www.xilinx.com/irl

To compile the PCI Bridge design you must own or purchase a license for the Xilinx PCI core. Details and documentation on the Xilinx PCI core can be found at:

http://www.xilinx.com/pci

Durango PCB Layout

Examples of the Durango layout with details of the major components can be seen in Figure C-3 and Figure C-4.



UG021_46_090701

Figure C-3: **Durango Layout Diagram (front)**

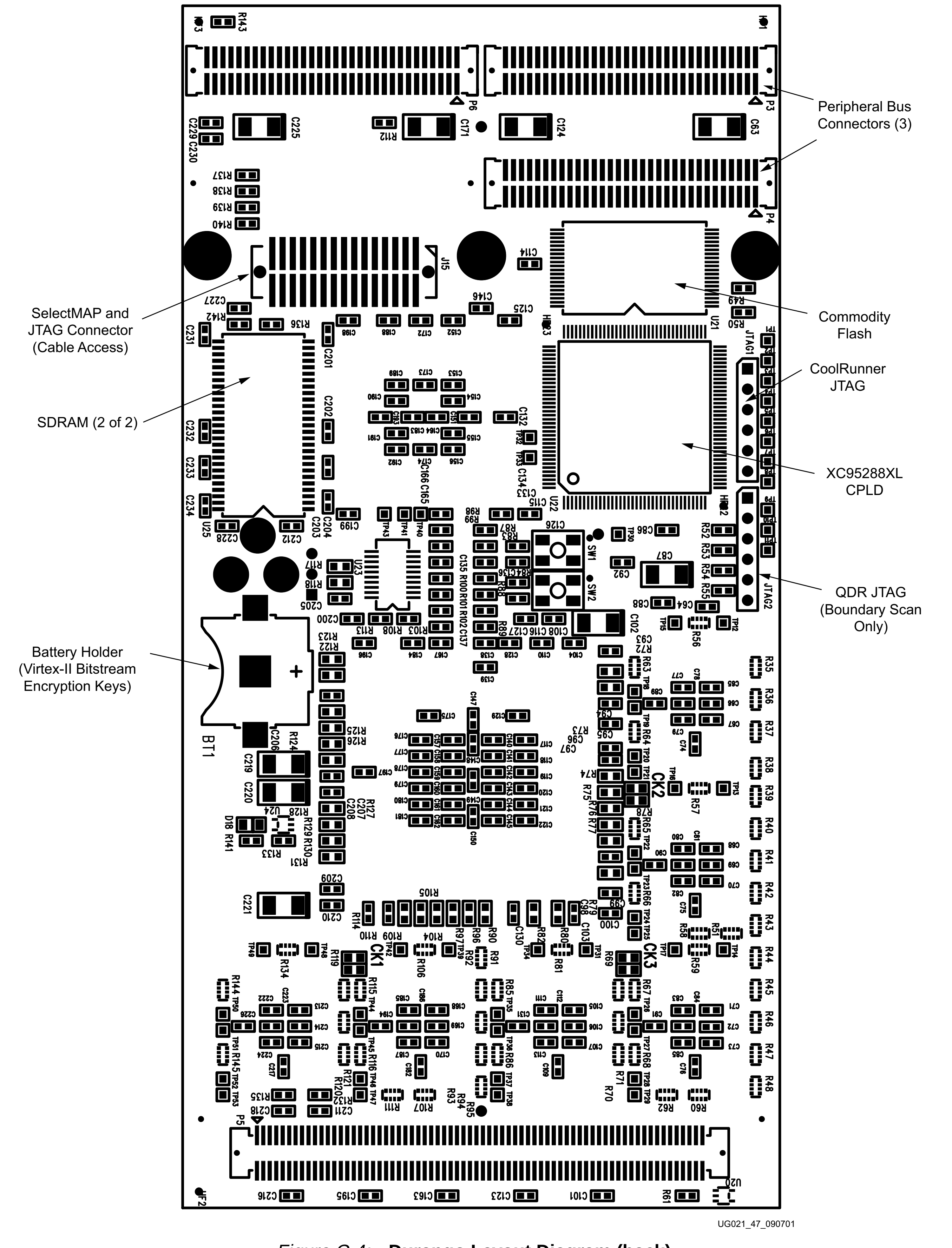

Peripheral Bus Connectors (3)
SelectMAP and JTAG Connector (Cable Access)
Commodity Flash
CoolRunner JTAG
SDRAM (2 of 2)
XC95288XL CPLD
QDR JTAG (Boundary Scan Only)
Battery Holder (Virtex-II Bitstream Encryption Keys)
UG021_47_090701

Figure C-4: **Durango Layout Diagram (back)**

Using ADM-XRC with the MCP750 and PAVE

ADM-XRC Board

The ADM-XRC, produced by Alpha Data Ltd, shown in Figure D-1, is a platform for demonstrating many of the key IRL concepts. It is a PMC card that includes a PLX 9080 PCI bridge and a Virtex-E FPGA as the target of the IRL reconfiguration in PAVE v1.0. The XCV1000E has a BG560 foot print which enabled the ADM-XRC to accommodate Virtex devices from XCV405E to XCV2000E

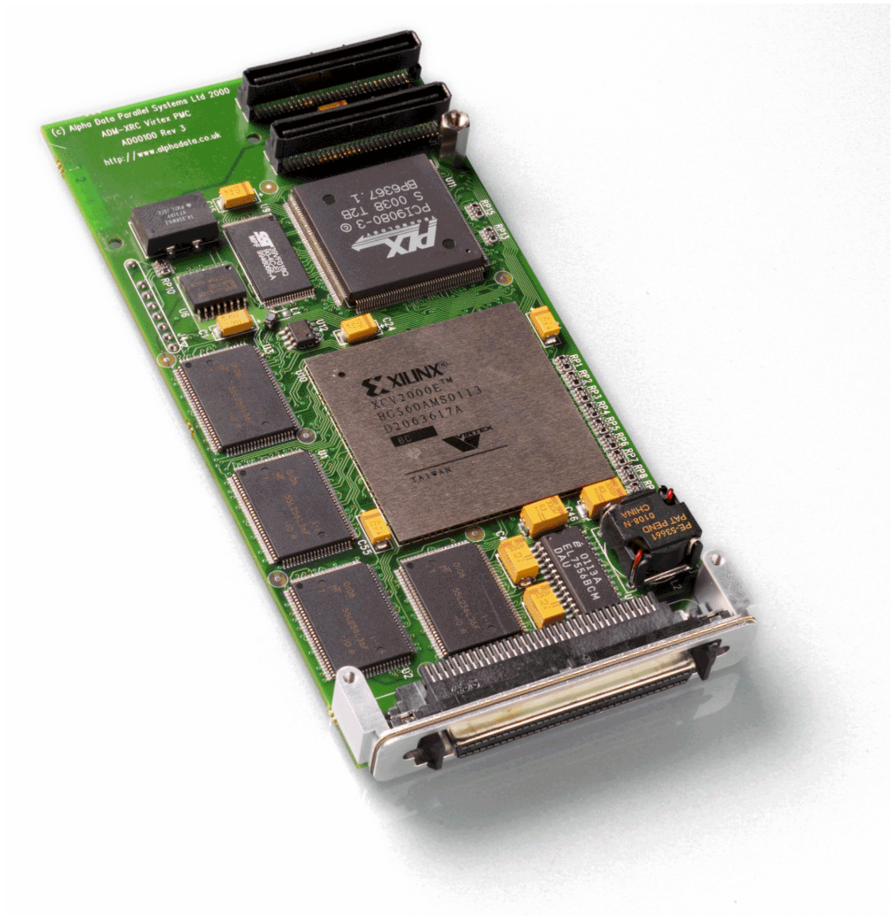


Figure D-1: **Alpha Data ADM-XRC Board**

Note: When viewed in Adobe Acrobat, the picture above appears best at ~300% magnification.

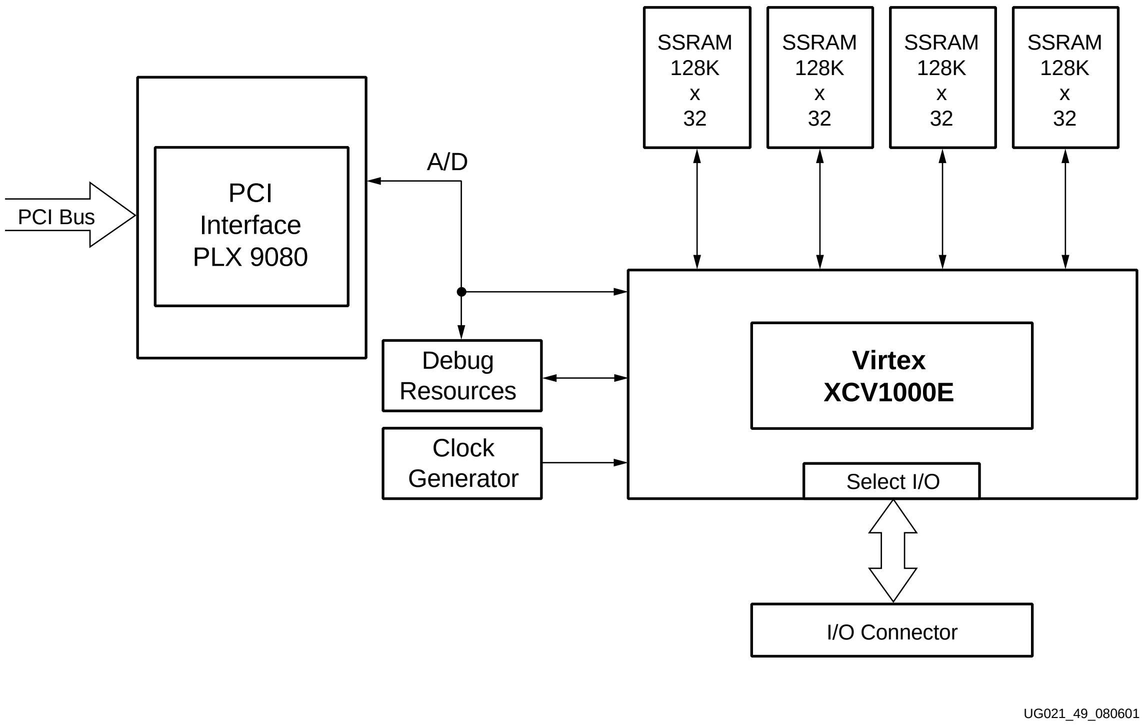


Figure D-2: **ADM-XRC Block Diagram**

ADM-XRC MCP750 PAVE Implementation

The PAVE Framework was tested in two hardware platforms. One of these platforms was a combination of the Motorola MCP750 and the Alpha Data Ltd. ADM-XRC board. The IRL host, used an Ethernet TCP/IP connection to download the VxWorks images and payloads to the MCP750 board. The other platform was the Avnet/Xilinx Durango, covered in Appendix B.

The ADM-XRC software framework is a fixed (pre-generated) framework that must be installed on `D:\` drive. The `_platform_mcp750admxrc` framework is initially ready to open in the Tornado tools, when installed in `D:\`.

Setup

The MCP750 is a Single Board Computer (SBC) based on the PowerPC processor. It comes in a 6U CompactPCI form factor. The MCP750 has a single PMC slot at 5V. The ADM-XRC board uses a PLX 9080 as the PCI interface which supports 5 V PCI. Figure D-3 show the block diagram of the MCP750 and the ADM-XRC board as the hardware was tested.

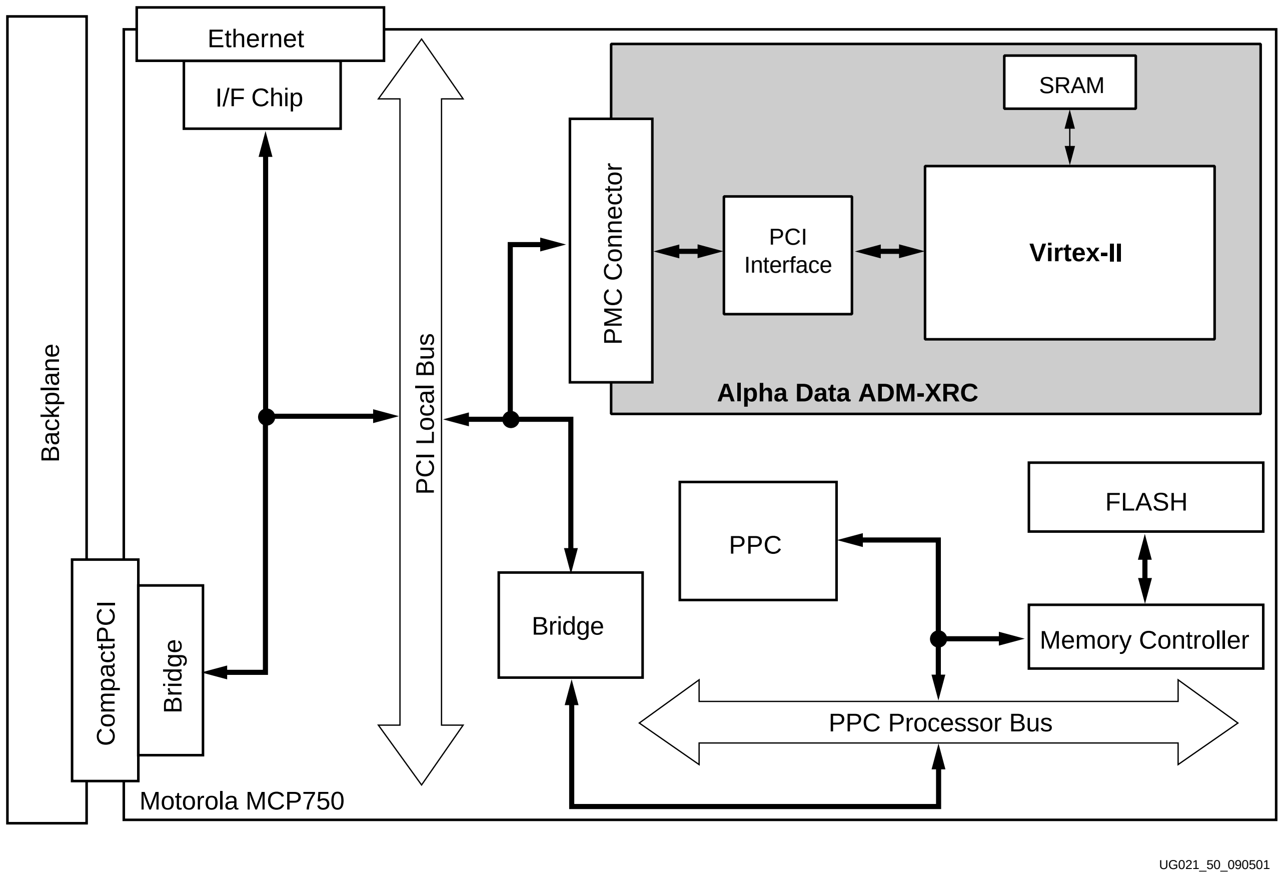


Figure D-3: **MCP750 with ADM-XRC Block Diagram**

INP Files for ADM-XRC

Similar to the Durango framework, the ADM-XRC framework was generated from a series of .inp files that describe the hardware to PAVE. The ADM-XRC contains three devices:

- PLX9080
- VirtexEngine
- ControlCPLD

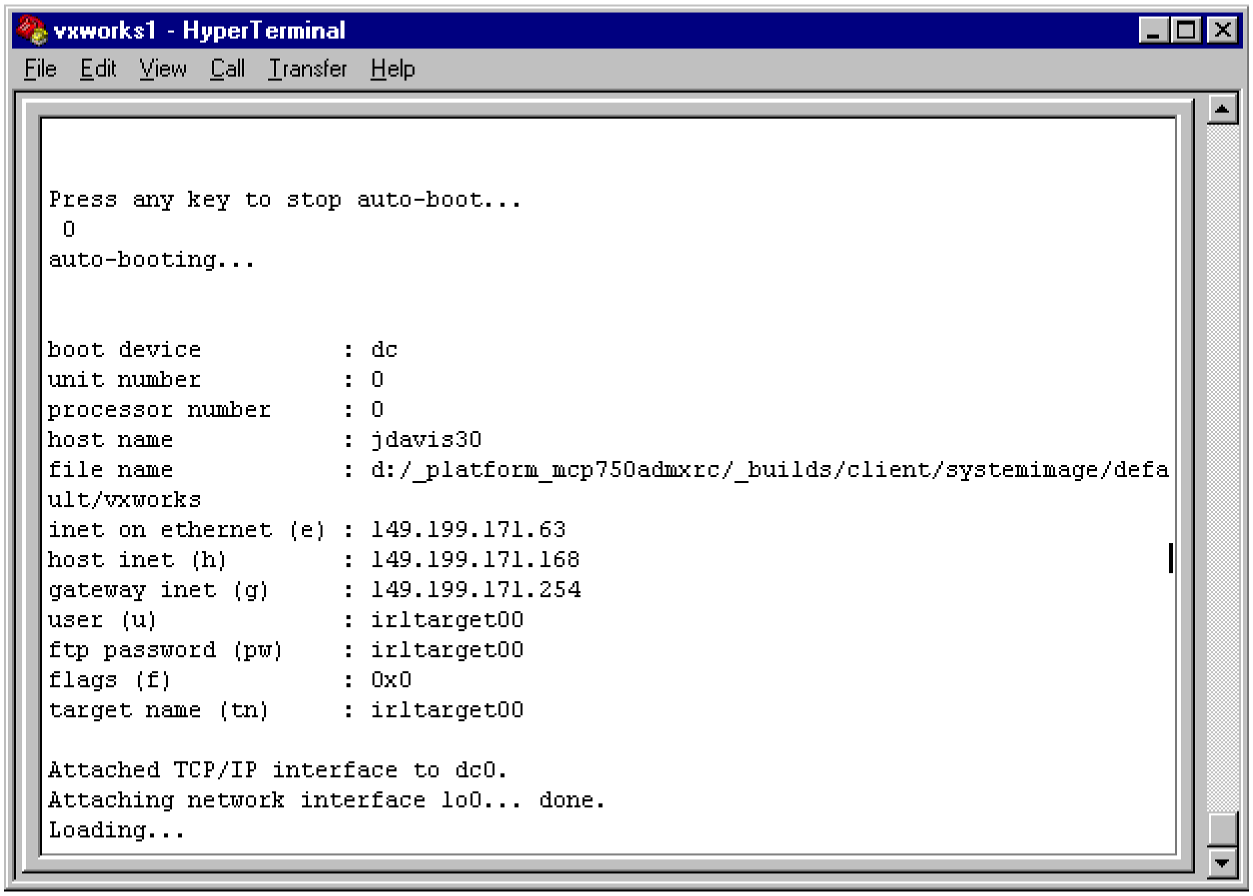
The .inp file for the PLX device maps all the registers for the PLX device, except the PCI registers. PCI registers are abstracted by `classPCIDevice`.

Applications

Included with the PAVE v1.0 release are several applications for the ADM-XRC. The push and pull applications are similar to the concepts outlined in XAPP412. The FFT application passes data to three nodes for processing: Host CPU, MCP750, and ADM-XRC V1000E FPGA.

These application descriptions presume some familiarity with how to configure and use the MCP750 and use of the Tornado tools. Steps that you must perform that are common to all applications:

- The serial port on the MCP750 should be connected to a computer running the VxWorks hyperterminal (COM1 or COM2 depending on your system.) as seen below during the boot process. Use the prompt in the terminal to configure the MCP750's settings.
- You must start the Tornado FTP server for the MCP750 board to log into and fetch the



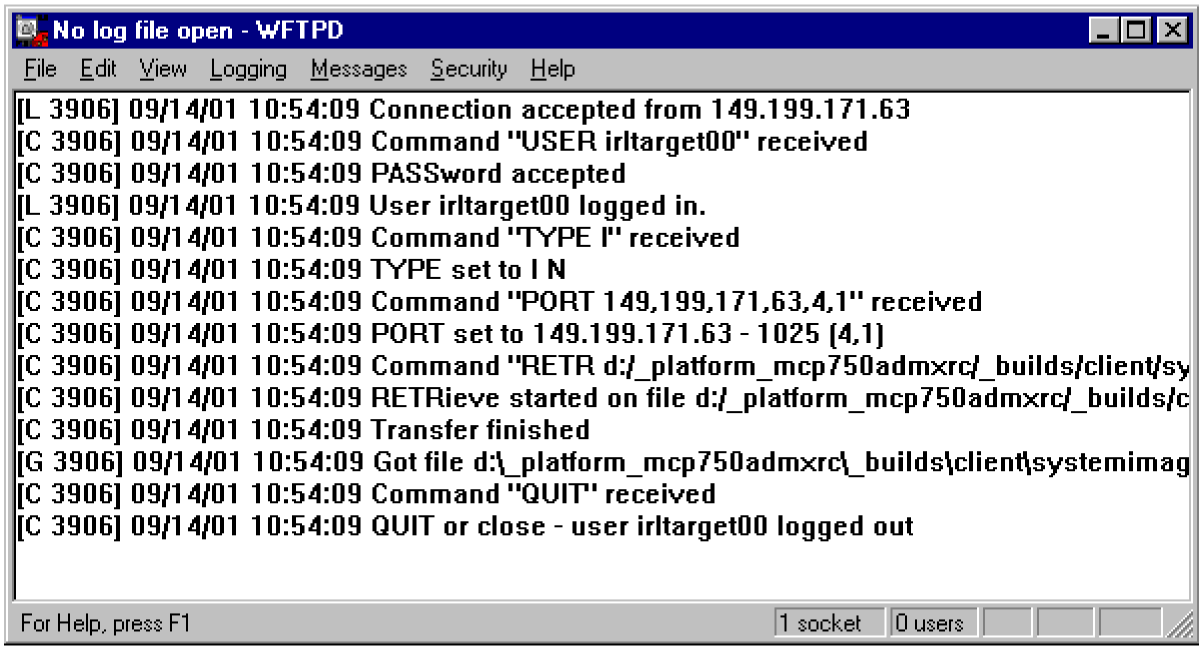
```
Press any key to stop auto-boot...
 0
auto-booting...


boot device          : dc
unit number          : 0
processor number     : 0
host name            : jdavis30
file name            : d:/_platform_mcp750admxrc/_builds/client/systemimage/defa
ult/vxworks
inet on ethernet (e) : 149.199.171.63
host inet (h)        : 149.199.171.168
gateway inet (g)     : 149.199.171.254
user (u)             : irltarget00
ftp password (pw)    : irltarget00
flags (f)            : 0x0
target name (tn)     : irltarget00

Attached TCP/IP interface to dc0.
Attaching network interface lo0... done.
Loading...
```

Figure D-4: **Tornado VxWorks Hyperterminal**

VxWorks image it needs to boot. In **`Security -> User/Rights...`** you will need to enter the name of the MCP750 board as a new user.



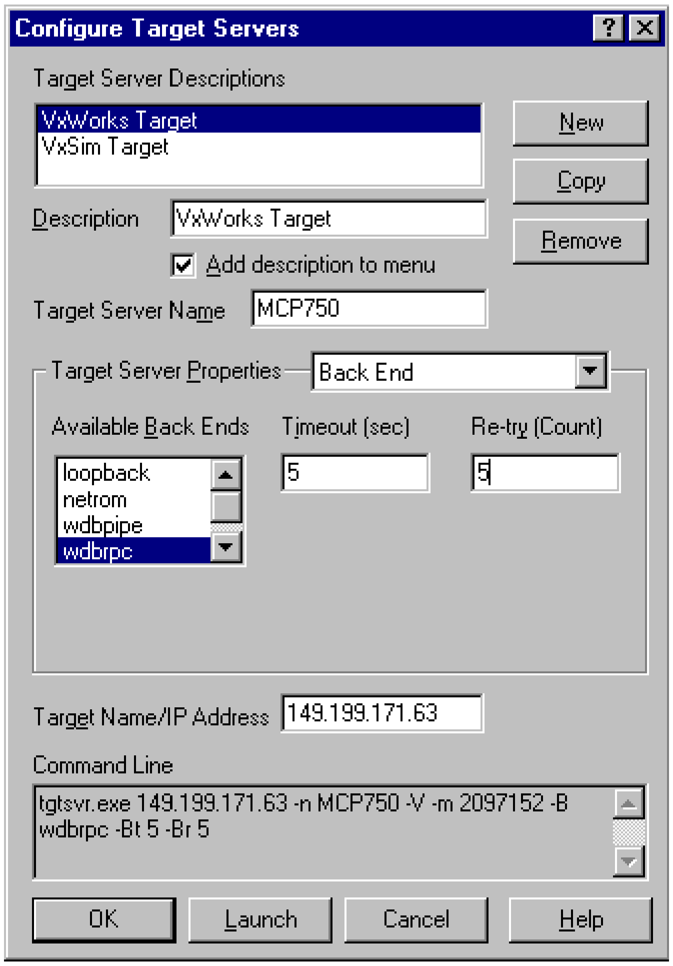
```
[L 3906] 09/14/01 10:54:09 Connection accepted from 149.199.171.63
[C 3906] 09/14/01 10:54:09 Command "USER irltarget00" received
[C 3906] 09/14/01 10:54:09 PASSword accepted
[L 3906] 09/14/01 10:54:09 User irltarget00 logged in.
[C 3906] 09/14/01 10:54:09 Command "TYPE I" received
[C 3906] 09/14/01 10:54:09 TYPE set to I N
[C 3906] 09/14/01 10:54:09 Command "PORT 149,199,171,63,4,1" received
[C 3906] 09/14/01 10:54:09 PORT set to 149.199.171.63 - 1025 (4,1)
[C 3906] 09/14/01 10:54:09 Command "RETR d:/_platform_mcp750admxrc/_builds/client/sy
[C 3906] 09/14/01 10:54:09 RETRieve started on file d:/_platform_mcp750admxrc/_builds/c
[C 3906] 09/14/01 10:54:09 Transfer finished
[G 3906] 09/14/01 10:54:09 Got file d:\_platform_mcp750admxrc\_builds\client\systemimag
[C 3906] 09/14/01 10:54:09 Command "QUIT" received
[C 3906] 09/14/01 10:54:09 QUIT or close - user irltarget00 logged out
```

Figure D-5: **Tornado FTP server**

- You have to start the target server to run the applications **-** You can run it from a command line with these settings:

  ```
  tgtsvr.exe 127.0.0.1 -n mcp750 -V -m 2097152 -B wdbrpc -Bt 5 -Br 5
  ```

  Alternately, you can run the Target Server from Tornado. **`Tools -> Target Server -> Configure...`** as shown below .Click **Launch** when done. In either case, set the IP address to match the IP address of the target.



Target Server Descriptions: VxWorks Target, VxSim Target
Description: VxWorks Target
[✓] Add description to menu
Target Server Name: MCP750
Target Server Properties: Back End
Available Back Ends: loopback, netrom, wdbpipe, wdbrpc
Timeout (sec): 5
Re-try (Count): 5
Target Name/IP Address: 149.199.171.63
Command Line:

```
tgtsvr.exe 149.199.171.63 -n MCP750 -V -m 2097152 -B
wdbrpc -Bt 5 -Br 5
```

Figure D-6: **Tornado Target Server Configuration**

- For all applications run the server side applications first. Specifically these scripts:
  - `pull`
  - `push`
  - `fft`

Push and Pull

The push and pull applications show different means of updating the FPGA and system software. Pull simply consists of a series of fetches to known locations on the network. Push is a bit more sophisticated, requiring the software to set up a socket and perform a series of processes to complete the update of the FPGA and associated driver.

This set of steps assumes you have the ADM-XRC hardware installed in a MCP750. For simplicity, the following paths will be under `D:\_platform_mcp750admxrc\` unless listed as part of a file.

Pull Application

1. Edit `\_builds\client\systemimage\default\pull`. Set the IP address to match the IP address of the MCP750; set the host name and path appropriately.

   ```
   wStatus = ioDefPathSet("irlhost01:
   d:/_platform_mcp750admxrc/_builds/client/systemimage/default");
   szIP = "127.0.0.1"
   ```

2. Open the tornado shell. You must be in `\_builds\client\systemimage\default` directory; you can verify this by the **pwd** command.

   ```
   -> pwd
   D:/_platform_mcp750admxrc/_builds/client/systemimage/default
   ```

   Invoke the pull script (use the "<" symbol before the filename)

   ```
   -> <pull
   ```

3. To stop, wait for the default 10 iterations to complete.

Push Application

1. Edit `\_builds\client\systemimage\default\push`. The Hostname and path in the `ioDefPathSet` command and the Target IP address should be changed to reflect your Host and Target systems.

   ```
   ioDefPathSet("irlhost01:d:/_platform_mcp750admxrc/_builds/client/
   systemimage/default");
   szIP = "127.0.0.1"
   ```

   Edit the following files so their IP addresses match the target's IP address and, in the case of the `configurationclient.bat` file, also set the path:

   - `\bin\release\bin\startclient.bat`
   - `\bin\release\bin\configurationclient.bat`
   - `\bin\release\bin\shutdown.bat`

2. Open `\_builds\_\admxrc.wsp` in Tornado. Build the following projects:

   - `ppc405gnu_admxrc_controlcpld`
   - `ppc405gnu_admxrc_plx9080`
   - `ppc405gnu_admxrc_serverapplication_vx`
   - `ppc405gnu_admxrc_virtexengine`

3. Open the tornado shell. You must be in `\_builds\client\systemimage\default` directory; you can verify this by the **pwd** command.

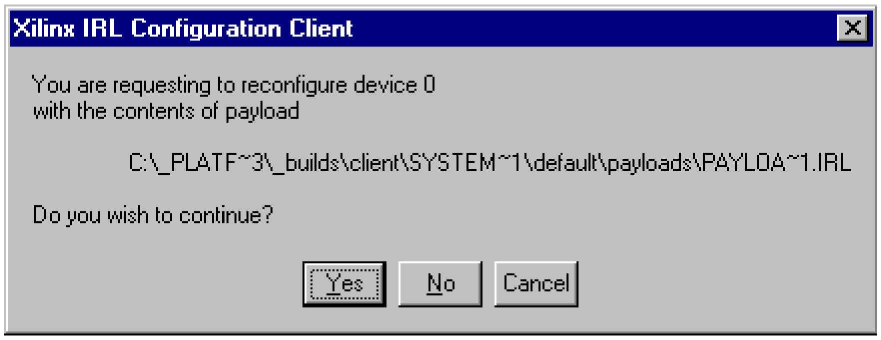
   ```
   -> pwd
   D:/_platform_mcp750admxrc/_builds/client/systemimage/default
   ```

   Start the push script (use the "<" symbol before the filename)

   ```
   -> <push
   ```

4. Start the client application by running `\bin\release\bin\startclient.bat` from the host.

5. To change the application with Push, drag and drop payloads (`.irl` files) icons onto the `\bin\release\bin\configurationclient.bat` file. Payloads files are under `\_builds\client\systemimage\default\payloads\`. When this works successfully, you will see a dialog as shown below:


Xilinx IRL Configuration Client

You are requesting to reconfigure device 0
with the contents of payload

C:_PLATF~3_builds\client\SYSTEM~1\default\payloads\PAYLOA~1.IRL

Do you wish to continue?

Figure D-7:

6. To stop this application, perform the following steps:
   a. In the Tornado shell, issue this command (including the "**<**"):
      ```
      -> <stop
      ```
   b. From the host run **`\bin\release\bin\shutdown.bat`**
   c. Type ctrl-c in the `startclient.bat` window. Both the Target and Host side tasks should now be terminated and the system is ready to run another application.

FFT

Design partitioning is an important concept for IRL Architectures. The design can be viewed in three paradigms:

- Modeling -The algorithm can be modeled on the host processor. This modeling can be done in Mathwork's Matlab.
- Embedding / Fielding - The Target processor can run the algorithm with code compiled from C.
- Acceleration - The FPGA increases the speed of the algorithm to multiple times the performance of either the host or target CPU.

How you divide the work between the processor (host or embedded) and the FPGA can make a tremendous difference in speed for the final product. Computational-intensive tasks such as large FFTs and Encryption / Decryption can find a large performance increase when run on an FPGA. With this in mind, we have included an FFT application that interacts with Matlab to show how a design can be partitioned in a system. For this application Matlab is providing data to the nodes doing the FFT computation; Matlab receives the processed data and plots it for all three implementations (Host CPU, Target CPU, and Target FPGA).

Running the FFT Application

This set of steps assumes you have the ADM-XRC hardware installed in a MCP750 and have Matlab installed on your host. For simplicity, the following paths will be under `D:\_platform_mcp750admxrc\` unless listed as part of a file.

1. Edit the `\bin\release\bin\startsimulation.m` file. Replace the IP Address "127.0.0.1" with the actual IP address being used on your Target (e.g. MCP750 board) system. If your system is not "Big Endian" set that to false.
   ```
   theFunctionControl.bServerBigEndian =   'TRUE';
   theFunctionControl.szIPAddress      = 127.0.0.1;
   ```
2. Edit `\_builds\client\systemimage\default\fft` and set the IP address to match the target's IP address; set the host name and path appropriately.
   ```
   wStatus = ioDefPathSet("irlhost01:
   ```

```
d:/_platform_mcp750admxrc/_builds/client/systemimage/default");
szIP = "127.0.0.1"
```

If you haven't done so from the Push and Pull applications, edit the following files so their IP addresses match the target's IP address and, in the `configurationclient.bat`, set the path:

- `\bin\release\bin\startclient.bat`
- `\bin\release\bin\configurationclient.bat`
- `\bin\release\bin\shutdown.bat`

3. VisualC++ Build - NT / Win2000 Build - The Host-side executables and batch files are present in the framework and *do not* require rebuilding. However, Visual Studio workspace (*.dsw) files are provided to rebuild these executables. If you need to rebuild the file follow these steps:
   - Open `\_builds\host\admxrc\clientapplications\clientapplication\x86-win32\admxrc_clientapplication_ntdsw` file in VisualC++.
   - The FileView tab should look as shown below.

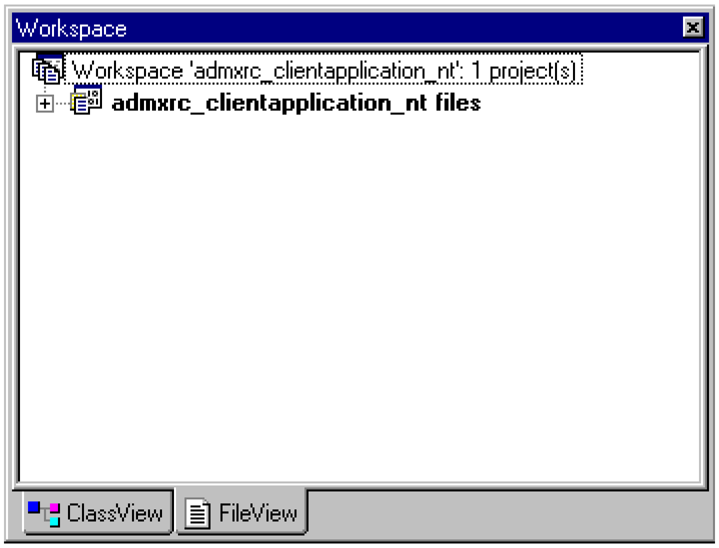

Workspace
Workspace 'admxrc_clientapplication_nt': 1 project(s)
admxrc_clientapplication_nt files
ClassView
FileView

Figure D-8:

- Select the following menu items:
  - **`Build -> Set Active Configuration...`** and set it to release configuration.
  - **`Build -> Clean`**
  - **`Build -> Rebuild All`**

Visual Studio workspaces are also present in:

`\_builds\host\admxrc\clientapplications\configurationclient\x86-win32\`

`\_builds\host\admxrc\clientapplications\shutdownx86-win32\`

`\_builds\host\admxrc\simulations\mexfunctiontemplate\x86-win32\`

4. Open `\_builds\_\admxrc.wsp` in Tornado. Build the following projects:
   - `ppc405gnu_admxrc_controlcpld`
   - `ppc405gnu_admxrc_plx9080`
   - `ppc405gnu_admxrc_serverapplication_vx`
   - `ppc405gnu_admxrc_virtexengine`
5. Open the tornado shell. You must be in `\_builds\client\systemimage\default` directory; you can verify this by the **pwd** command.

```
-> pwd
D:/_platform_mcp750admxrc/_builds/client/systemimage/default
```

Start the FFT script (use the "<" symbol before the filename)

```
-> <fft
```

6. Start the client application by running `\bin\release\bin\startclient.bat` from the host.
7. Start Matlab on Host NT or Win2000 system. Within Matlab command window

```
>> cd D:\_platform_mcp750admxrc\bin\release\bin
>> startsimulation.m
```

You should now see a single peaked frequency spectrum sweep back and forth on a Matlab plot window, as seen below. The first picture shows the FFT computation being performed on the MCP750 CPU. The second shows the FFT being performed on the Host CPU. The last shows the FFT being run on the FPGA on the ADM-XRC. The Host CPU and Target CPU use floating point math; the Target FPGA uses integer math.

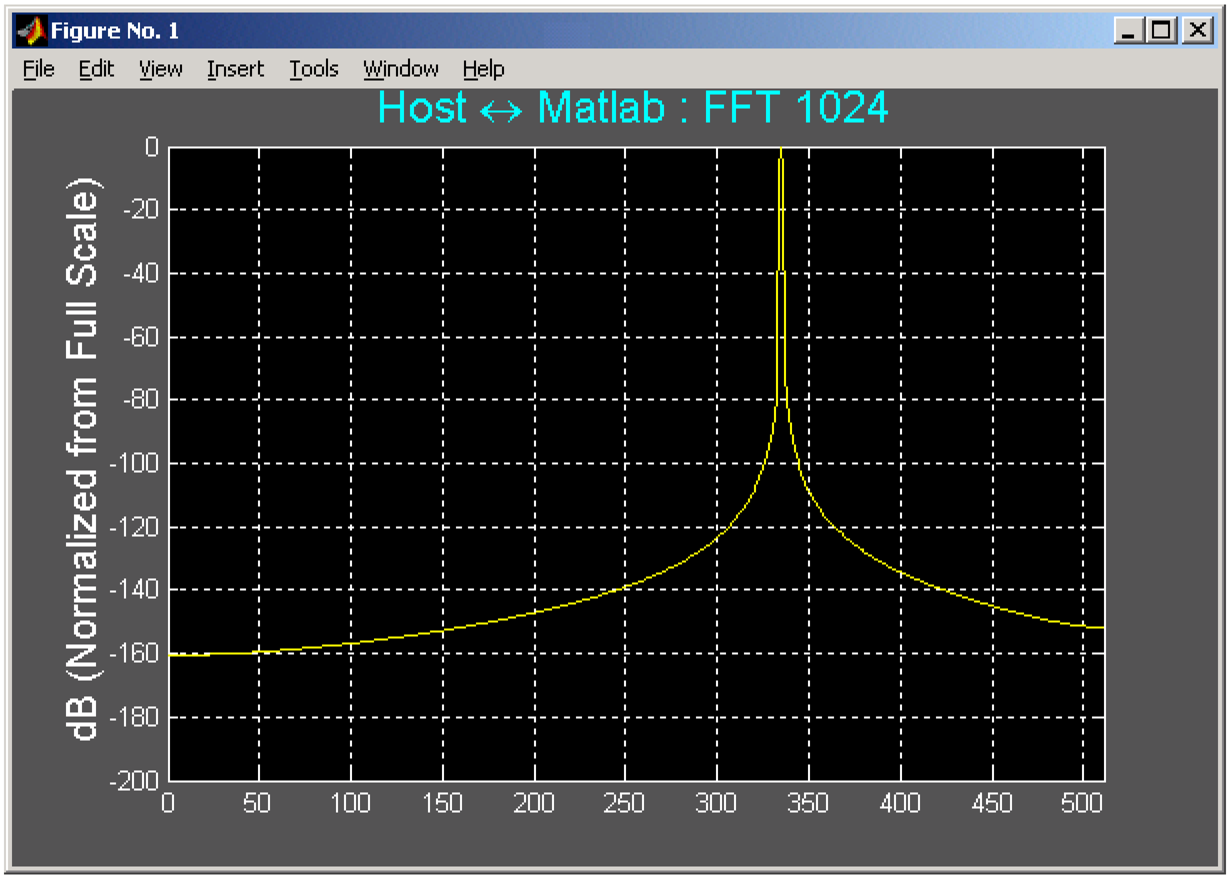
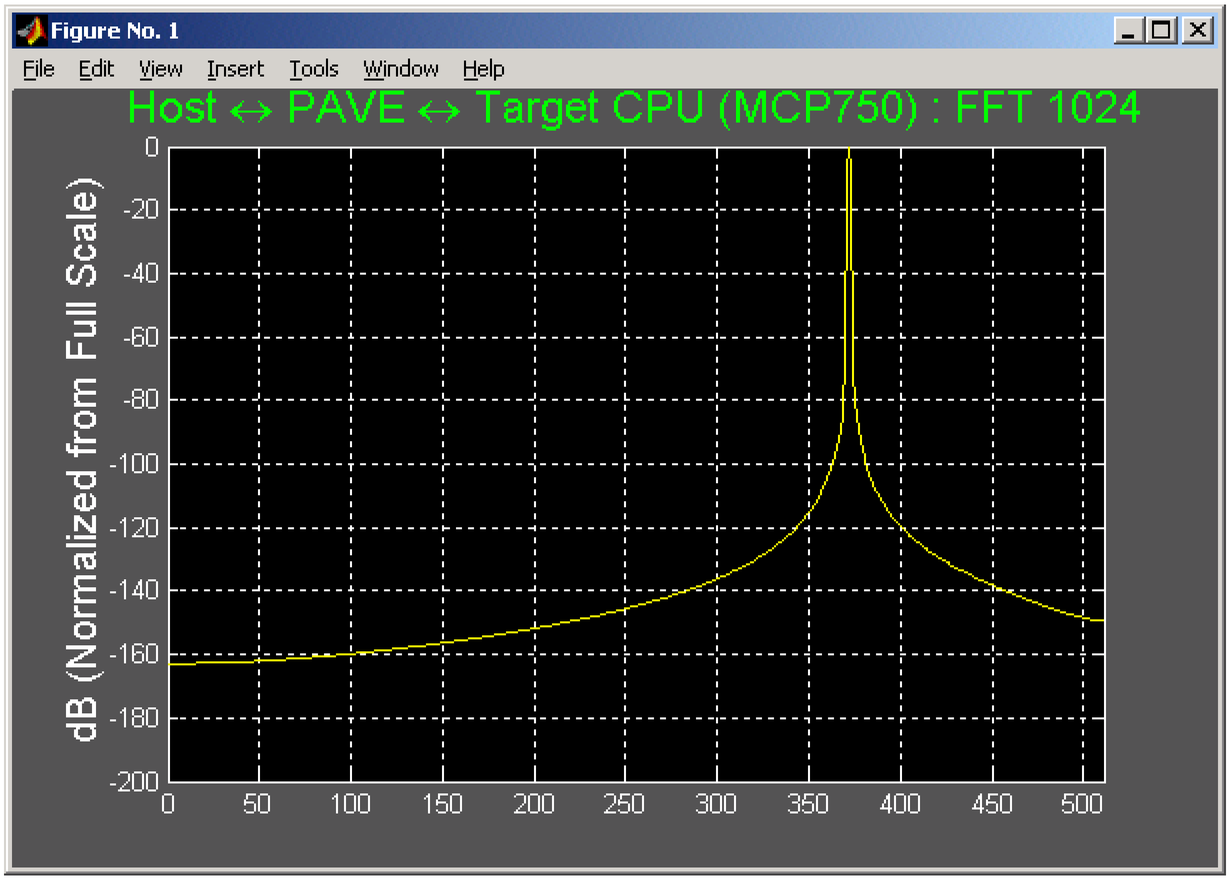


Figure D-9: **1024 Point FFT on the Host CPU**



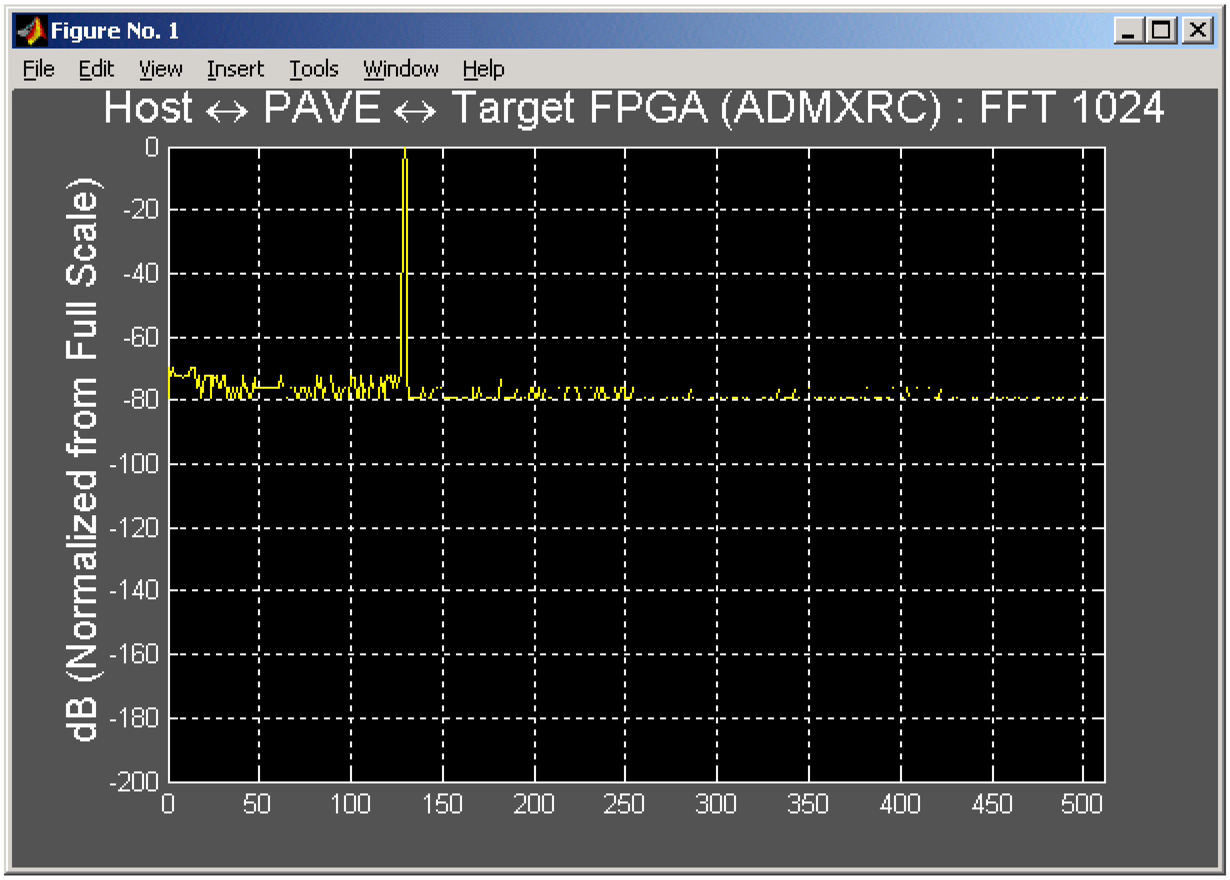


Figure D-10: **1024 Point FFT on the Target CPU (top) and ADM-XRC V1000E FPGA (bottom)**

8. To stop the FFT application, perform the following steps:

   a. In the Tornado shell, issue this command (including the "**<**"):

   ```
   -> <stop
   ```

   b. From the host run **`\bin\release\bin\shutdown.bat`**

   c. Type ctrl-c in the Matlab window.

ADM-XRC Payloads

The payloads included with the PAVE frameworks have been generated for a V1000E based ADM-XRC board. Creating payloads for an ADM-XRC with a different FPGA is done by

1. Place the bit file with the appropriate naming convention into the `\_builds\client\systemimage\default\fpga` directory.
2. Edit `\_builds\_\generatepayload.bat` to reflect the new bit file name.
3. Run **`generatepayload.bat`**. This will create a `.irl` payload file in `\_builds\client\systemimage\default\payloads\`

   Note: The `generatepayload.bat` script may take a few minutes to complete.

Resources

- Motorola - Additional details on the Motorola MCP750 can be found at:

  http://www.mcg.mot.com/cfm/templates/product.cfm?PageID=895&ProductID=22&PageTypeID=1
- Alpha Data Ltd. - Additional details on the ADM-XRC board can be found at:

  http://www.alphadata.co.uk/dsheet/adm-xrc.html
- Mathworks - Producers of the Matlab program.

  http://www.mathworks.com/

Appendix E

PAVE API Summary

API Reference Manual

This chapter provides detailed descriptions of the objects that comprise the PAVE Object Model and associated API.

The classes referenced in this chapter are:

- The `classRegister` object.
- The `classDevice` object.
- The `classIRLDevice` object
- The `classSignalBuffer` object.
- The `classStateMachine` object.

`classRegister::classRegister`

Synopsis

```
classRegister::classRegister (
    apiPHYSICALADDRESS     ptrOffset,
    apiWORD                wStartBit,
    apiWORD                wNumberOfBits,
    apiBOOL                bReadable,
    apiBOOL                bWriteable,
    apiBOOL                bInitialize,
    apiDWORD               dwInitializedValue,
    size_t                 wAddressableWidth,
    void                   *ptrParentDevice,
    enAPIReturnCodes       (*pfnDeviceModelFunction)(apiBYTE, void *),
    apiBOOL                bEnablePAL,
    pntr_classIRLPlatform  ptrPlatform
);
```

Description

This function is the default constructor for a `classRegister` object. The `classRegister` object is an abstraction of a register in the programming interface of an IRL-enabled device.

Parameters

- `ptrOffset: Offset address of the register within the IRL-enabled device.`
- `wStartBit: The starting bit of the register field within a 32 bit word.`
- `wNumberOfBits: The width of the register field in bits.`
- `bReadable: Indicates whether or not the register is readable.`
- `bWriteable: Indicates whether or not the register is writable.`
- `bInitialize: Indicates whether or not an initial value should be written to the register when it is instantiated.`
- `dwInitializedValue: Indicates the value to write to the register.`
- `wAddressableWidth: Indicates the addressing boundary on which the register field is addressed.`
- `ptrParentDevice: A pointer to the device object in which this register resides.`
- `pfnDeviceModelFunction: A pointer to a function which is called when this register is written to. See chapter 4, Simulation with the SIF.`
- `bEnablePAL: Enables the uses of platform specific routines for reading and writing to registers.`
- `ptrPlatform: The pointer to the platform abstraction layer.`

Return Values

`Pointer to the newly instantiate register object.`

See Also

`classRegister::~classRegister`

`classRegister::~classRegister`

Synopsis

```
classRegister::~classRegister (
);
```

Description

This function is the destructor for a `classRegister` object.

Parameters

N/A

Return Values

See Also

`classRegister::classRegister`

`classRegister::Upload`

Synopsis

```
enAPIReturnCodes    classRegister::Upload (
apiDWORD            dwValue
);
```

Description

This function results in the value in `dwValue` being written to the register that this instance of the register object abstracts. The register is properly address and the data aligned accordingly based on the register specification.

Parameters

- `dwValue: Indicates the value to write to the register.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`

`classRegister::Modify`

`classRegister::Download`

Synopsis

```
enAPIReturnCodes    classRegister::Download (
apiDWORD            *ptrValue
);
```

Description

This function results in the register contents being read into the `apiDWORD` pointed to by `ptrValue`. The register is properly address and the data aligned accordingly based on the register specification.

Parameters

- `ptrValue: a pointer to a apiDWORD into which the register contents are to be read.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Upload`

`classRegister::Modify`

classRegister::Modify

Synopsis

```
enAPIReturnCodes    classRegister::Modify (
apiDWORD            dwValue
);
```

Description

This function results in the register contents being modified by the contents of `apiDWORD dwValue`. The Modify method is particularly useful for addressing fields within a register. The register is properly address and the data aligned accordingly based on the register specification.

Parameters

- `dwValue: an apiDWORD from which the register contents are to be written.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`

`classRegister::Upload`

classRegister::UploadSignals

Synopsis

```
enAPIReturnCodes    classRegister::UploadSignals (
apiDWORD            *ptrBuffer,
apiDWORD            dwBufferLength,
apiDWORD            dwSetupTime,
apiDWORD            dwHoldTime
);
```

Description

This function results in the contents of a signal buffer being sequentially written to the register which the instance of this register object abstracts.

Parameters

- ptrBuffer: a pointer to a buffer of data values to write to the register.
- dwBufferLength: the length of the buffer.
- dwSetupTime: the delay in platform ticks to wait prior to writing each value to the register.
- dwHoldTime: the delay in platform ticks to wait after writing each value to the register.

Return Values

enAPIFunctionSuccessful – if the function completes successfully.

enAPIFunctionFailed – if the function fails.

See Also

classRegister::Download

classRegister::Upload

classRegister::Modify

`classRegister::ToggleSignal`

Synopsis

```
enAPIReturnCodes    classRegister::UploadSignals (
apiDWORD            dwAssertCondition,
apiDWORD            dwDeassertedCondition,
apiDWORD            dwNumToggles,
apiDWORD            dwSetupTime,
apiDWORD            dwHoldTime
);
```

Description

This function results in the contents of a register being sequentially written to with the `dwAssertedCondition` and `dwDeassertedCondition` values.

Parameters

- `dwAssertCondition: The value to write to the register as the asserted condition.`
- `dwDeassertedCondition: The value to write to the register as the deasserted condition.`
- `dwNumToggles: The number of iterations to assert the signal on the port.`
- `dwSetupTime: the delay in platform ticks to wait prior to writing each value to the register.`
- `dwHoldTime: the delay in platform ticks to wait after writing each value to the register.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`
`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`
`classRegister::Upload`
`classRegister::Modify`

`classRegister::Reset`

Synopsis

```
enAPIReturnCodes classRegister::Reset (
);
```

Description

This function results in the initial value being written to the register that this instance of the register object abstracts.

Parameters

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`

`classRegister::Modify`

`classRegister::Clear`

Synopsis

```
enAPIReturnCodes classRegister::Clear (
);
```

Description

This function results in a `0x00000000` being written to the register that this instance of the register object abstracts.

Parameters

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`

`classRegister::Modify`

classRegister::Initialize

Synopsis

```
enAPIReturnCodes classRegister::Initialize (
);
```

Description

This function results in the initial value being written to the register that this instance of the register object abstracts.

Parameters

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`

`classRegister::Modify`

`classRegister::Display`

Synopsis

```
enAPIReturnCodes classRegister::Display (
char                 *szAnnotation
);
```

Description

This function prints to stdout the contents of the register that this instance of the register object abstracts.

Parameters

- `szAnnotation: A annotating text string that you wish to have printed out in conjunction with the register contents.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classRegister::Download`

`classDevice::classDevice`

Synopsis

```
classDevice::classDevice (
);
```

Description

This function is the default constructor for device objects that have a register-level programming model. Note that this is different from the IRL-enabled Device object, which is re-configurable.

Parameters

Return Values

`Pointer to a new classDevice object.`

See Also

`classDevice::~classDevice`

`classDevice::classDevice`

Synopsis

```
classDevice::classDevice (
apiPHYSICALADDRESS       ptrBaseAddress,
pntr_classIRLPlatform    ptrPlatform
);
```

Description

This function is a constructor for device objects that have a register-level programming model. Note that this is different from the IRL-enabled Device object, which is re-configurable.

Parameters

- `ptrBaseAddress: The mapped physical base address of the device that this instance of the classDevice object abstracts.`
- `ptrPlatform: The pointer to the platform abstraction layer object.`

Return Values

`Pointer to a new classDevice object.`

See Also

`classDevice::~classDevice`

classDevice::classDevice

Synopsis

```
classDevice::classDevice (
apiWORD             wDeviceIndex,
apiWORD             wDeviceInstance,
apiPHYSICALADDRESS       ptrBaseAddress,
pntr_classIRLPlatform    ptrPlatform
);
```

Description

This function is a constructor for device objects that have a register-level programming model. Note that this is different from the IRL-enabled Device object, which is re-configurable.

Parameters

- `wDeviceIndex: The index of the device on the board.`
- `WDeviceInstance: The instance of this particular class of device on the board.`
- `ptrBaseAddress: The mapped physical base address of the device that this instance of the classDevice object abstracts.`
- `ptrPlatform: The pointer to the platform abstraction layer object.`

Return Values

`Pointer to a new classDevice object.`

See Also

`classDevice::~classDevice`

classDevice::~classDevice

Synopsis

```
classDevice::~classDevice (
);
```

Description

This function is a destructor for device objects that have a register-level programming model. Note that this is different from the IRL-enabled Device object, which is re-configurable.

Parameters

Return Values

See Also

classDevice::classDevice

classDevice::UploadRegister

Synopsis

```
enAPIReturnCodes         classDevice::UploadRegister(
apiPHYSICALADDRESS       ptrOffset,
apiWORD                  wStartBit,
apiWORD                  wNumberOfBits,
apiDWORD                 dwValue,
size_t                   wAddressableWidth
);
```

Description

This function uploads the contents of `dwValue` to the register pointed to by `ptrOffset`.

Parameters

- `ptrOffset: The offset to the devices register.`
- `wStartBit: The start bit of the register field to upload.`
- `wNumberOfBits: The number of bits in the register field.`
- `dwValue: The value to upload to the register.`
- `wAddressableWidth: How to address the register (apiBYTE, apiWORD, apiDWORD)`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classDevice::classDevice`

`classIRLDevice::classIRLDevice`

Synopsis

```
classIRLDevice::classIRLDevice (
apiPHYSICALADDRESS  ptrBaseAddress,
pntr_classIRLPlatformptrPlatform
);
```

Description

The default constructor of the `classIRLDevice` object. The `classIRLDevice` object is an abstraction of an IRL-enabled device. This is a virtual base class from which system component specific IRL-enabled device objects can be derived.

Parameters

- `ptrBaseAddress: The mapped physical base address of the IRL-enabled device that this instance of the classIRLDevice object abstracts.`
- `ptrPlatform: The pointer to the platform abstraction layer object.`

Return Values

`This function returns a pointer to a new instance of a classIRLDevice object.`

See Also

`classIRLDevice::~classIRLDevice`

classIRLDevice::classIRLDevice

Synopsis

```
classIRLDevice:: classIRLDevice (
apiWORD             wDeviceIndex,
apiWORD             wDeviceInstance,
apiPHYSICALADDRESS  ptrBaseAddress,
pntr_classIRLPlatformptrPlatform
);
```

Description

The constructor of the `classIRLDevice` object. The `classIRLDevice` object is an abstraction of an IRL-enabled with a register programming model.

Parameters

- `wDeviceIndex: The index of the device on the board.`
- `WDeviceInstance: The instance of this particular class of device on the board.`
- `ptrBaseAddress: The mapped physical base address of the IRL-enabled device that this instance of the classDevice object abstracts.`
- `ptrPlatform: The pointer to the platform abstraction layer object.`

Return Values

`This function returns a pointer to a new instance of a classIRLDevice object.`

See Also

`classIRLDevice::~classIRLDevice`

classIRLDevice::~classIRLDevice

Synopsis

```
classIRLDevice::~classIRLDevice (
);
```

Description

The default destructor of the `classDevice` object. This function deinstantiates the `classIRLDevice` object.

Parameters

Return Values

See Also

`classIRLDevice::classIRLDevice`

classIRLDevice::CachePayload

Synopsis

```
enAPIReturnCodes    classIRLDevice::CachePayload (
apiSTRING           szPayloadURL,
void                **pptrBitstreamBuffer,
void                **pptrModuleBuffer
);
```

Description

This method reads in the specified payload object from the local file system and places the bitstream and object module segments in memory.

Parameters

- `szPayloadURL`: the fully qualified path of the payload object which is to be uploaded into the device that this instance of the device object abstracts.
- `pptrBitstreamBuffer`: pointer a pointer to the buffer where the configuration bitstream is to be stored.
- `pptrModuleBuffer`: pointer to a pointer to the buffer where the binary object module is to be stored.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.

`enAPIFunctionFailed` – if the function fails.

See Also

classIRLDevice::classIRLDevice

classIRLDevice::GetPayloadSize

Synopsis

```
enAPIReturnCodes    classIRLDevice::GetPayloadSize (
apiSTRING           szPayloadURL,
apiDWORD            *ptrBitstreamSize,
apiDWORD            *ptrModuleSize
);
```

Description

This method returns the sizes of the configuration bitstream and binary object module segments in bytes. This information is extracted from the header of the payload object.

Parameters

- `szPayloadURL: the fully qualified path of the payload object which is to be uploaded into the device that this instance of the device object abstracts.`
- `ptrBitstreamSize: a pointer to an apiDWORD where the configuration bitstream size is to be stored.`
- `ptrModuleSize a pointer to an apiDWORD where the binary object module size is to be stored.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::classIRLDevice`

classIRLDevice::GetPayloadOffset

Synopsis

```
enAPIReturnCodes    classIRLDevice::GetPayloadOffset (
apiSTRING           szPayloadURL,
apiDWORD            *ptrBitstreamOffset,
apiDWORD            *ptrModuleOffset
);
```

Description

This method returns the offset of the configuration bitstream and binary object module segments in bytes. This information is extracted from the header of the payload object.

Parameters

- szPayloadURL: the fully qualified path of the payload object which is to be uploaded into the device that this instance of the device object abstracts.
- ptrBitstreamOffset: a pointer to an apiDWORD where the configuration bitstream size is to be stored.
- ptrModuleOffset a pointer to an apiDWORD where the binary object module size is to be stored.

Return Values

enAPIFunctionSuccessful – if the function completes successfully.

enAPIFunctionFailed – if the function fails.

See Also

classIRLDevice::classIRLDevice

classIRLDevice::GetPayloadChecksum

Synopsis

```
enAPIReturnCodes    classIRLDevice::GetPayloadOffset (
apiSTRING           szPayloadURL,
apiDWORD            *ptrBitstreamChecksum,
apiDWORD            *ptrModuleChecksum
);
```

Description

This method returns the checksum of the configuration bitstream and binary object module segments in bytes. This information is extracted from the header of the payload object.

Parameters

- `szPayloadURL`: the fully qualified path of the payload object which is to be uploaded into the device that this instance of the device object abstracts.
- `ptrBitstreamChecksum`: a pointer to an apiDWORD where the configuration bitstream size is to be stored.
- `PtrModuleChecksum`: a pointer to an apiDWORD where the binary object module size is to be stored.

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::classIRLDevice`

classIRLDevice::UploadPayloadFromBuffer

Synopsis

```
enAPIReturnCodes    classIRLDevice::UploadPayloadFromBuffer (
apiDWORD            dwBitstreamBufferSize,
void                *ptrBitstreamBuffer,
apiDWORD            dwModuleBufferSize,
void                *ptrModuleBuffer
);
```

Description

This method uploads the cached configuration bitstream to the IRL-enabled device that this instance of the `classIRLDevice` object abstracts. Additionally, the cached binary object module is reloaded.

Parameters

- `dwBitstreamBufferSize: the size of the configuration bitstream cache in bytes.`
- `ptrBitstreamBuffer: a pointer to the buffer where the configuration bitstream is contained.`
- `DwModuleBufferSize: the size of the object module buffer.`
- `ptrModuleBuffer: a pointer to the buffer where the binary object module is contained.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::DownloadPayloadToBuffer`

`classIRLDevice::UploadPayloadFromFile`

classIRLDevice::UploadPayloadFromFile

Synopsis

```
enAPIReturnCodes    classIRLDevice::UploadPayloadFromFile (
apiSTRING           szHeaderPath,
apiSTRING           szBitstreamPath,
apiSTRING           szModulePath
);
```

Description

This method uploads the file system resident configuration bitstream to the IRL-enabled device that this instance of the `classIRLDevice` object abstracts. Additionally, the cached binary object module is reloaded.

Parameters

- `szHeaderPath: the fully qualified path of the payload object header which is to be uploaded into the device that this instance of the device object abstracts.`
- `szBitstreamPath: the fully qualified path of the configuration bitstream which is to be uploaded into the device that this instance of the device object abstract.`
- `szModulePath: the fully qualified path of the binary object module which is to be uploaded into the device that this instance of the device object abstract.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::DownloadPayloadToFile`

`classIRLDevice::UploadPayloadFromBuffer`

classIRLDevice::SplitPayload

Synopsis

```
enAPIReturnCodes    classIRLDevice::SplitPayload (
apiSTRING           szPayloadPath,
apiSTRING           szHeaderPath,
apiSTRING           szBitstreamPath,
apiSTRING           szModulePath
);
```

Description

This method splits a payload object into its constituent components.

Parameters

- `szPayloadPath`: the fully qualified path of the payload object
- `szHeaderPath`: the fully qualified path of the payload object header.
- `szBitstreamPath`: the fully qualified path of the configuration bitstream.
- `szModulePath`: the fully qualified path of the binary object.

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

N/A.

`classIRLDevice::DownloadPayloadToBuffer`

Synopsis

```
enAPIReturnCodes    classIRLDevice::DownloadPayloadToBuffer (
void                *ptrBitstreamBuffer,
void                *ptrModuleBuffer
);
```

Description

This method downloads the current device configuration into the indicated configuration bitstream buffer. The binary object module associated with this component is also downloaded.

Parameters

- `ptrBitstreamBuffer: the buffer into which the configuration bitstream is to be downloaded.`
- `ptrModuleBuffer: the buffer into which the binary object module is to be loaded.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::UploadPayloadFromBuffer`

`classIRLDevice::DownloadPayloadToFile`

classIRLDevice::DownloadPayloadToFile

Synopsis

```
enAPIReturnCodes    classIRLDevice::DownloadPayloadToBuffer (
apiSTRING           szBitstreamPath,
apiSTRING           szModulePath
);
```

Description

This method downloads the current device configuration into the indicated configuration bitstream file. The binary object module associated with this component is also downloaded to a file.

Parameters

- `szBitstreamPath: the fully qualified path of the configuration bitstream which is to be uploaded into the device that this instance of the device object abstract.`
- `szModulePath: the fully qualified path of the binary object module which is to be uploaded into the device that this instance of the device object abstract.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::UploadPayloadFromFile`

`classIRLDevice::DownloadPayloadToBuffer`

classIRLDevice::UnCachePayload

Synopsis

```
enAPIReturnCodes    classIRLDevice::UnCachePayload (
void                *ptrBitstreamBuffer,
void                *ptrModuleBuffer
);
```

Description

This method frees configuration bitstream and module buffers that were allocated by `CachePayload`. This function must be called at the completion of the updating process.

Parameters

- `ptrBitstreamBuffer`: a pointer to the configuration bitstream buffer.
- `ptrModuleBuffer`: a pointer to the binary object module buffer.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.

`enAPIFunctionFailed` – if the function fails.

See Also

`classIRLDevice::CachePayload`

`classIRLDevice::UploadPayloadFromBuffer`

classIRLDevice::UploadRegister

Synopsis

```
enAPIReturnCodes    classDevice::UploadRegister (
apiPHYSICALADDRESS  ptrOffset,
apiWORD             wStartBit,
apiWORD             wNumberOfBits,
apiDWORD            dwValue,
size_t              wAddressableWidth
);
```

Description

This method uploads a register field specified by the parameters with the value in `dwValue`.

Parameters

- `ptrOffset`: an offset from the base address of the device to the register in which this register field is located.
- `wStartBit`: the starting bit of the register field within the register.
- `wNumberOfBits`: the number of bits in the register field.
- `dwValue`: the value to upload to the register.
- `WAddressableWidth`: the address boundary of the register that contains the register field of interest.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.

`enAPIFunctionFailed` – if the function fails.

See Also

`classIRLDevice::DownloadRegister`

`classIRLDevice::ModifyRegister`

classIRLDevice::DownloadRegister

Synopsis

```
enAPIReturnCodes    classIRLDevice::DownloadRegister (
apiPHYSICALADDRESS  ptrOffset,
apiWORD             wStartBit,
apiWORD             wNumberOfBits,
apiDWORD            *ptrValue,
size_t              wAddressableWidth
);
```

Description

This method downloads the contents of a register field specified by the parameters into the location pointed to by `ptrValue`.

Parameters

- `ptrOffset: an offset from the base address of the device to the register in which this register field is located.`
- `wStartBit: the starting bit of the register field within the register.`
- `wNumberOfBits: the number of bits in the register field.`
- `ptrValue: the location of where to store the register contents.`
- `WAddressableWidth: the address boundary of the register that contains the register field of interest.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::UploadRegister`

`classIRLDevice::ModifyRegister`

classIRLDevice::ModifyRegister

Synopsis

```
enAPIReturnCodes    classIRLDevice::ModifyRegister (
apiPHYSICALADDRESS  ptrOffset,
apiWORD             wStartBit,
apiWORD             wNumberOfBits,
apiDWORD            dwValue,
size_t              wAddressableWidth
);
```

Description

This method performs a read / modify / write operation of `dwValue` to the contents of a register field specified by the parameters.

Parameters

- `ptrOffset`: an offset from the base address of the device to the register in which this register field is located.
- `wStartBit`: the starting bit of the register field within the register.
- `wNumberOfBits`: the number of bits in the register field.
- `dwValue`: value to modify the register field with.
- `wAddressableWidth`: the address boundary of the register that contains the register field of interest.

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classIRLDevice::UploadRegister`

`classIRLDevice::ModifyRegister`

classSignalBuffer::classSignalBuffer

Synopsis

```
classSignalBuffer::classSignalBuffer (
apiDWORD             dwLength
);
```

Description

The default constructor for a signal buffer object.

Parameters

- `dwLength: Specifies the length of the buffer. If length is 0 then the signal buffer object is instantiated but it's ptrBuffer element is NULL. This allows the signal buffer to have other buffers attached to it without the need to allocate them at instantiation time.`

Return Values

`N/A`

See Also

`classSignalBuffer::~classSignalBuffer`

classSignalBuffer::~classSignalBuffer

Synopsis

```
classSignalBuffer::~classSignalBuffer (
);
```

Description

The method deinstantiates the signal buffer object.

Parameters

- `N/A`

Return Values

`N/A`

See Also

`classSignalBuffer::classSignalBuffer`

classSignalBuffer::AttachBuffer

Synopsis

```
enAPIReturnCodes    classSignalBuffer::AttachBuffer (
apiDWORD            *ptrBuffer,
apiDWORD            dwLength
);
```

Description

The method attaches a `apiDWORD` array to a Signal Buffer object.

Parameters

- `ptrBuffer: the apiDWORD array which is to be attached to the Signal Buffer.`
- `dwLength: the length of the apiDWORD array.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::DetachBuffer`

classSignalBuffer::DetachBuffer

Synopsis

```
enAPIReturnCodes    classSignalBuffer::DetachBuffer (
);
```

Description

The method detaches an `apiDWORD` array from the Signal Buffer object.

Parameters

- N/A

Return Values

```
enAPIFunctionSuccessful – if the function completes successfully.
enAPIFunctionFailed – if the function fails.
```

See Also

classSignalBuffer::AttachBuffer

classSignalBuffer::AssertSignal

Synopsis

```
enAPIReturnCodes    classSignalBuffer::AssertSignal (
apiDWORD            dwAssertedState,
apiDWORD            dwDeassertedState,
apiDWORD            dwDelayBeforeAssertion,
apiDWORD            dwDurationOfAssertion,
apiWORD             wSignalOffsetInRegister
);
```

Description

The method sets a bitfield within a word of a `classSignalBuffer` to `dwAssertedState` for `dwDurationOfAssertion` number of elements. The assertion is delayed from the start of the buffer by `dwDelayBeforeAssertion` elements.

Parameters

- `dwAssertedState: the value to be inserted into the bitfield in the asserted elements of the Signal Buffer.`
- `dwDeassertedState: the value to be inserted into the bitfield in the non-asserted elements of the Signal Buffer.`
- `dwDelayBeforeAssertion: the number of elements that are non-asserted in the Signal Buffer prior to assertion.`
- `wSignalOffsetInRegister: the position of the bitfield within the words of the Signal Buffer.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::SerializeRepetitiveBitSequence`

`classSignalBuffer::SerializeRepetitivePattern`

`classSignalBuffer::SerializePatternStream`

`classSignalBuffer::SerializePatternDWord`

`classSignalBuffer::ParallelizeSerialStream`

`classSignalBuffer::ParallelizeSerialDWord`

classSignalBuffer::SerializeRepetitiveBitSequence

Synopsis

```
enAPIReturnCodes    classSignalBuffer::SerializeRepetitiveBitSequence(
apiDWORD            dwBufferOffset,
apiDWORD            dwBitSequence,
apiWORD             dwNumBitsInSequence,
apiDWORD            dwNumberRepetitions,
apiWORD             wSignalOffsetInRegister
);
```

Description

The method will repetitively insert a bit sequence specified by `dwBitSequence` into successive elements of a Signal Buffer in bit position `wSignalOffsetInRegister`. Figure E-1 below illustrates how this method functions under the following invocation:

```
this->SerializeRepetitiveBitSequence(1, 0x16, 5, 3, 2);
```

In this example the 5 bit pattern word, 0x16, serially-shifted into bit position 0 of the Signal Buffer three times, offset by 1 word.

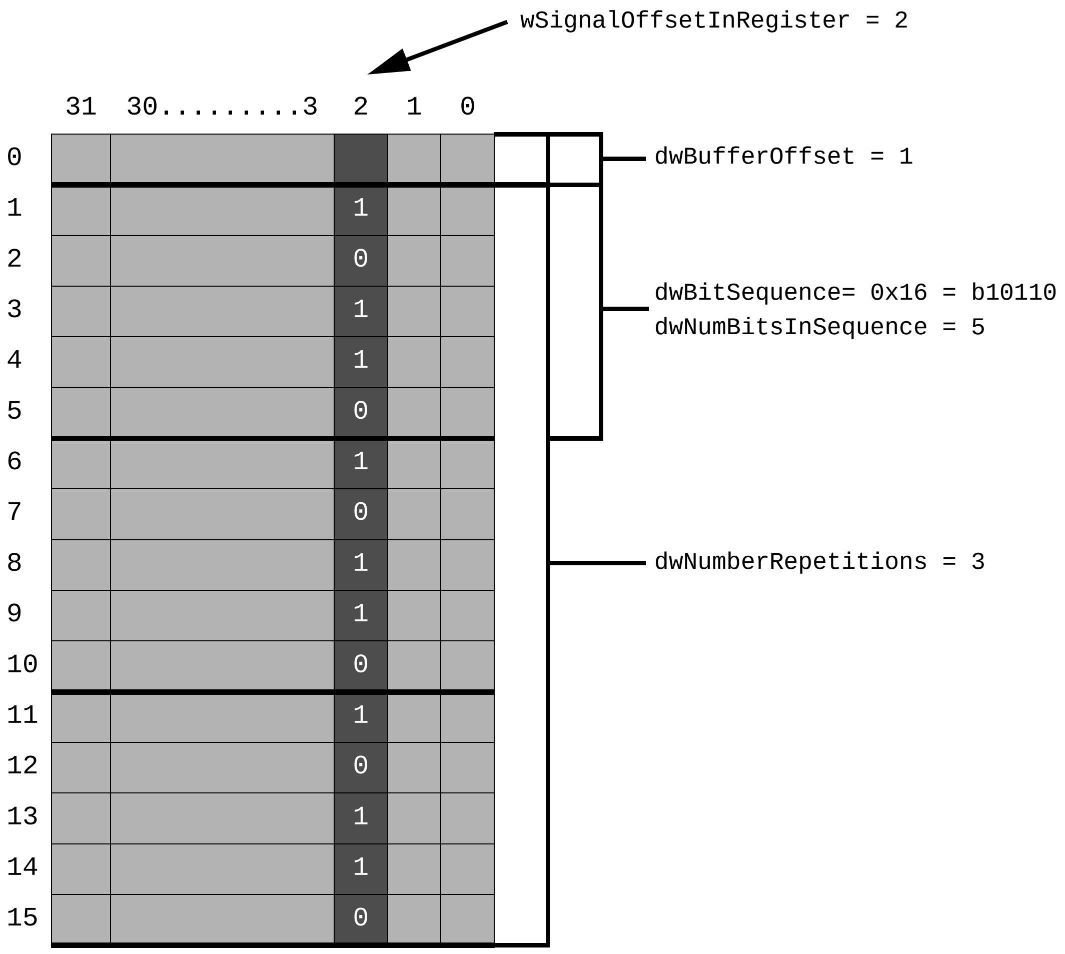


Figure E-1: **this->SerializeRepetitiveBitSequence(1, 0x16, 5, 3, 2);**

Parameters

- `dwBufferOffset`: the starting offset into the Signal Buffer into which the pattern is serially shifted.
- `dwBitSequence`: the pattern to be serially shifted into the Signal Buffer.
- `dwNumBitsInSequence`: the number of bits in the pattern to shift into the Signal Buffer.
- `dwNumberRepetitions`: the number of iterations to repeat the pattern.
- `wSignalOffsetInRegister`: the position of the bitfield within the words of the Signal Buffer.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.

`enAPIFunctionFailed` – if the function fails.

See Also

classSignalBuffer::SerializeRepetitivePattern

classSignalBuffer::SerializePatternStream

classSignalBuffer::SerializePatternDWord

classSignalBuffer::ParallelizeSerialStream

classSignalBuffer::ParallelizeSerialDWord

classSignalBuffer::SerializeRepetitivePattern

Synopsis

```
enAPIReturnCodes    classSignalBuffer::SerializeRepetitivePattern(
apiDWORD            dwBufferOffset,
apiDWORD            dwClockLength,
apiDWORD            dwPattern,
apiWORD             wSignalOffsetInRegister
);
```

Description

This method is a variant of `SerializeRepetitiveBitSequence`. The method will repetitively insert a 32 bit pattern specified by `dwPattern` into successive elements of a Signal Buffer in bit position `wSignalOffsetInRegister`. The pattern is repeated for an integral number of `dwClockLength`/32 iterations. Figure E-2 below illustrates how this method functions under the following invocation:

```
this->SerializeRepetitivePattern(2, 1024, 0x0000000B, 0);
```

In this example the pattern word, 0x0000000B, serially-shifted into bit position 0 of the Signal Buffer 32 times, offset by 0 words.

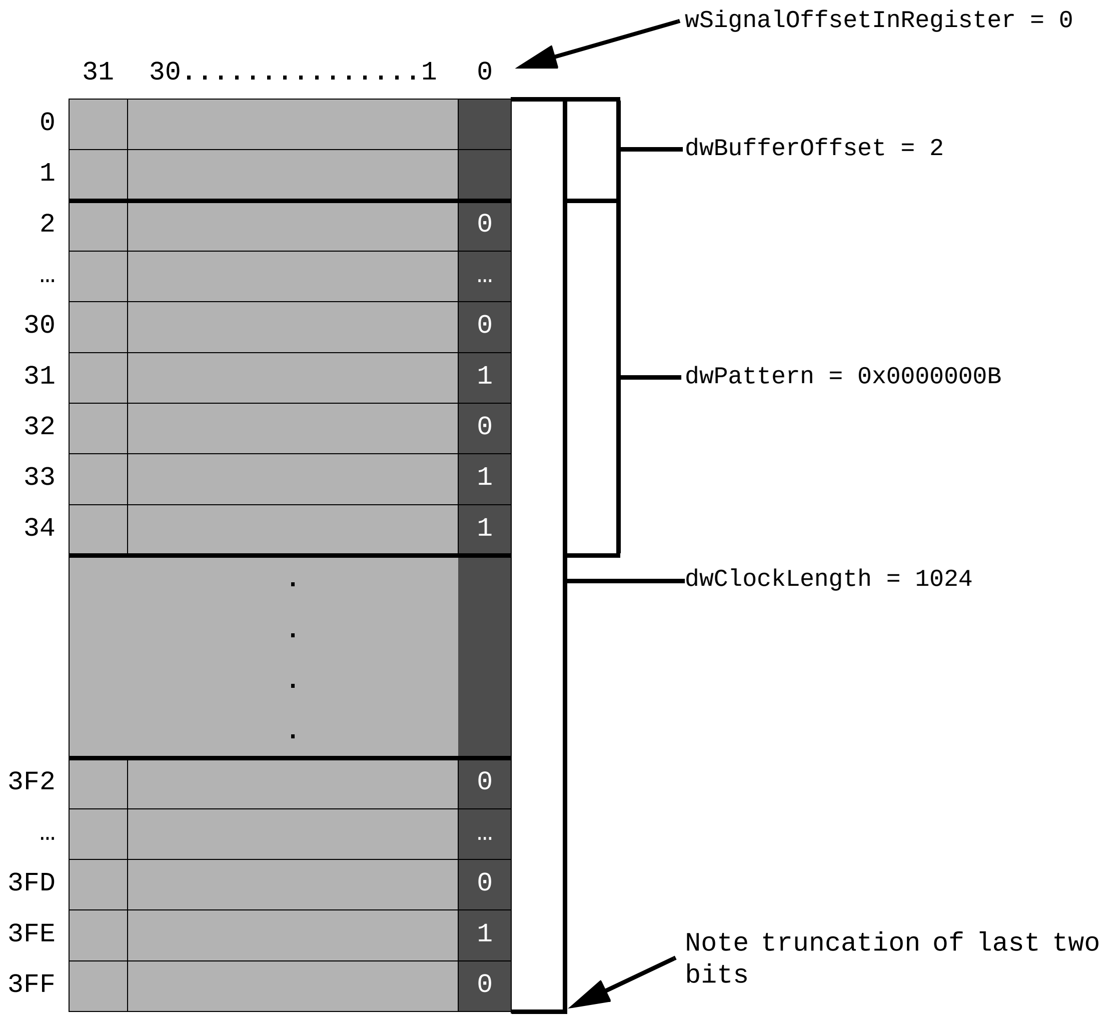


Figure E-2: **this->SerializeRepetitivePattern(2, 1024, 0x0000000B, 0);**

Parameters

- `dwBufferOffset`: the starting offset into the Signal Buffer into which the pattern is serially shifted.
- `dwClockLength`: the total length of the pattern to be written.
- `dwPattern`: the 32 bit pattern to be written into the Signal Buffer.
- `dwNumberRepetitions`: the number of iterations to repeat the pattern.
- `wSignalOffsetInRegister`: the position of the bitfield within the words of the Signal Buffer.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.

`enAPIFunctionFailed` – if the function fails.

See Also

classSignalBuffer::SerializeRepetitiveBitSequence

classSignalBuffer::SerializePatternStream

classSignalBuffer::SerializePatternDWord

classSignalBuffer::ParallelizeSerialStream

classSignalBuffer::ParallelizeSerialDWord

classSignalBuffer::SerializePatternStream

Synopsis

```
enAPIReturnCodes    classSignalBuffer::SerializePatternStream(
apiDWORD            dwBufferOffset,
apiDWORD            *ptrPatternArray,
apiDWORD            dwNumBitsInPatternArray,
apiBOOL             bLeftShift,
apiWORD             wSignalOffsetInRegister
);
```

Description

This method serially shifts an array of pattern words (containing `dwNumBitsInPatternArray` total) into a Signal Buffer in bit position `wSignalOffsetInRegister`, offset by `dwBufferOffset`.

```
this->SerializePatternStream(0, ptrArray, 32*1024, true, 0);
```

In this example, the array `ptrArray` is serially left-shifted into bit position 0 of the Signal Buffer.

Parameters

- dwBufferOffset: the starting offset into the Signal Buffer into which the pattern is serially shifted.
- ptrPatternArray: the Array in which the sequence of pattern bits reside.
- dwNumBitsInPatternArray: the total number of bits in the pattern. If ptrPatternArray consisted of 32 elements, then dwNumBitsInPatternArray would be 1024 bits.
- bLeftShift: if true the bits of each element are left-shift out of the element. If false, the bits are right-shifted out.
- wSignalOffsetInRegister: the position of the bitfield within the words of the Signal Buffer.

Return Values

enAPIFunctionSuccessful – if the function completes successfully.

enAPIFunctionFailed – if the function fails.

See Also

classSignalBuffer::SerializeRepetitiveBitSequence

classSignalBuffer::SerializeRepetitivePattern

classSignalBuffer::SerializePatternDWord

classSignalBuffer::ParallelizeSerialStream

classSignalBuffer::ParallelizeSerialDWord

classSignalBuffer::SerializePatternDWord

Synopsis

```
enAPIReturnCodes    classSignalBuffer::SerializePatternDWord(
apiDWORD            dwBufferOffset,
apiDWORD            dwPattern,
apiWORD             wNumBitsInPattern,
apiBOOL             bShiftLeft,
apiWORD             wSignalOffsetInRegister
);
```

Description

This method serially shifts a pattern word, `dwPattern`, (containing `wNumBitsInPattern`) into a Signal Buffer in bit position `wSignalOffsetInRegister`, offset by `dwBufferOffset`.

```
this->SerializePatternDWord(0, 0x1C, 5, true, 0);
```

In this example, the pattern 0x1C is serially left-shifted into bit position 0 of the Signal Buffer.

Parameters

- `dwBufferOffset: the starting offset into the Signal Buffer into which the pattern is serially shifted.`
- `dwPattern: the pattern word in which the sequence of pattern bits reside.`
- `dwNumBitsInPattern: the total number of bits in the pattern.`
- `bLeftShift: if true the bits of each element are left-shift out of the element. If false, the bits are right-shifted out.`
- `wSignalOffsetInRegister: the position of the bitfield within the words of the Signal Buffer.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::SerializeRepetitiveBitSequence`

`classSignalBuffer::SerializeRepetitivePattern`

`classSignalBuffer::SerializePatternStream`

`classSignalBuffer::ParallelizeSerialStream`

`classSignalBuffer::ParallelizeSerialDWord`

classSignalBuffer::ParallelizeSerialStream

Synopsis

```
enAPIReturnCodes    classSignalBuffer::ParallelizeSerialStream(
apiDWORD            dwSerialStreamOffset,
apiDWORD            *ptrPatternArray,
apiDWORD            dwPatternArrayLength,
apiDWORD            dwNumBitsInSerialStream,
apiWORD             wSignalOffsetInRegister
);
```

Description

This method parallelizes a serial stream of bits into an array of 32 bit words.

`this->ParallelizeSerialStream(1, ptrArray, 4, 128, 0);`

In this example, the bit-pattern in bit 0 of elements 0-3 of the Signal Buffer, offset by word 1, are serially left-shifted into an array of words pointed to by `ptrPatternArray`.

Parameters

- `dwSerialStreamOffset: the starting offset into the Signal Buffer into which the pattern is extracted.`
- `ptrPatternArray: the pattern array into which the sequence of pattern bits are shifted.`
- `dwPatternArrayLength: the total number of 32 words in the pattern array.`
- `dwNumBitsInSerialStream: the total number of bits in the serial stream to extract.`
- `wSignalOffsetInRegister: the position of the bitfield within the words of the Signal Buffer.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`
`enAPIFunctionFailed – if the function fails.`

See Also

classSignalBuffer::SerializeRepetitiveBitSequence
classSignalBuffer::SerializeRepetitivePattern
classSignalBuffer::SerializePatternStream
classSignalBuffer::SerializePatternDWord
classSignalBuffer::ParallelizeSerialDWord

classSignalBuffer::ParallelizeSerialDWord

Synopsis

```
enAPIReturnCodes    classSignalBuffer::ParallelizeSerialDWord(
apiDWORD            dwSerialStreamOffset,
apiDWORD            *ptrPattern,
apiWORD             wNumBitsInSerialDWord,
apiWORD             wSignalOffsetInRegister
);
```

Description

This method parallelizes a serial pattern of `wNumBitsInSerialDWord` bits, resident in bit `wSignalOffsetInRegister`, into a 32 bit word.

`this->ParallelizeSerialDWord (1, ptrPattern, 13, 0);`

In this example, the bit-pattern in bit 0 of elements 1-13 of the Signal Buffer, (*offset by 1 word), are serially left-shifted into a word pointed to by `ptrPattern`.

Parameters

- `dwSerialStreamOffset: the starting offset into the Signal Buffer into which the pattern is extracted.`
- `ptrPattern: the pattern word into which the sequence of pattern bits are shifted.`
- `dwNumBitsInSerialDWord: the total number of bits in the serial stream to extract.`
- `wSignalOffsetInRegister: the position of the bitfield within the words of the Signal Buffer.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::SerializeRepetitiveBitSequence`

`classSignalBuffer::SerializeRepetitivePattern`

`classSignalBuffer::SerializePatternStream`

`classSignalBuffer::SerializePatternDWord`

`classSignalBuffer::ParallelizeSerialStream`

classSignalBuffer::ParallelizeSerialDWord

Synopsis

```
enAPIReturnCodes    classSignalBuffer::ParallelizeSerialDWord(
apiDWORD            dwSerialStreamOffset,
apiDWORD            *ptrPattern,
apiWORD             wNumBitsInSerialDWord,
apiWORD             wSignalOffsetInRegister
r
);
```

Description

This method parallelizes a serial stream of bits into an array of 32 bit words.

`this->ParallelizeSerialStream(0, ptrArray, 4, 128, 0);`

In this example, the bit-pattern in bit 0 of elements 0-3 of the Signal Buffer are serially left-shifted into an array of words pointed to by `ptrPatternArray`.

Parameters

- `dwBufferOffset: the starting offset into the Signal Buffer into which the pattern is serially shifted.`
- `dwPattern: the pattern word in which the sequence of pattern bits reside.`
- `dwNumBitsInPattern: the total number of bits in the pattern.`
- `bLeftShift: if true the bits of each element are left-shift out of the element. If false, the bits are right-shifted out.`
- `wSignalOffsetInRegister: the position of the bitfield within the words of the Signal Buffer.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::SerializeRepetitiveBitSequence`

`classSignalBuffer::SerializeRepetitivePattern`

`classSignalBuffer::SerializePatternStream`

`classSignalBuffer::SerializePatternDWord`

`classSignalBuffer::ParallelizeSerialStream`

`classSignalBuffer::UploadFile`

Synopsis

```
enAPIReturnCodes    classSignalBuffer::UploadFile(
char                *szFileName
);
```

Description

This method uploads a binary file of data into the Signal Buffer object.

Parameters

- `szFileName: the fully qualified path to the binary file which is to be uploaded.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::DownloadFile`

classSignalBuffer::DownloadFile

Synopsis

```
enAPIReturnCodes    classSignalBuffer::DownloadFile(
char                *szFileName
);
```

Description

This method downloads a Signal Buffer into a binary file.

Parameters

- `szFileName: the fully qualified path to the binary file which is to be downloaded to.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::UploadFile`

classSignalBuffer::ClearBuffer

Synopsis

```
enAPIReturnCodes    classSignalBuffer::ClearBuffer(
);
```

Description

This method sets all elements of the Signal Buffer to 0.

Parameters

- `N/A.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`
`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::DisplayBuffer`

`classSignalBuffer::MoveIndex`

Synopsis

```
enAPIReturnCodes    classSignalBuffer::MoveIndex(
apiDWORD            dwNewIndexLocation
);
```

Description

This method moves the Signal Buffer cursor that points to the next fill position in the Signal Buffer. This cursor can be used as an offset in the fill functions. It is important to note that when the functions

`classSignalBuffer::SerializeRepetitiveBitSequence,`

`classSignalBuffer::SerializeRepetitivePattern,`

`classSignalBuffer::SerializePatternStream,`

`classSignalBuffer::SerializePatternDWord,`

`classSignalBuffer::ParallelizeSerialStream, and`

`classSignalBuffer::ParallelizeSerialDWord`

are used, they increment the cursor.

Parameters

- `N/A.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::GetIndex`

classSignalBuffer::GetIndex

Synopsis

```
enAPIReturnCodes    classSignalBuffer::GetIndex(
apiDWORD            *ptrIndexLocation
);
```

Description

This method gets the Signal Buffer index pointer that points to the next fill position in the Signal Buffer. This cursor can be used as an offset in the fill functions. It is important to note that when the functions

`classSignalBuffer::SerializeRepetitiveBitSequence`,
`classSignalBuffer::SerializeRepetitivePattern`,
`classSignalBuffer::SerializePatternStream`,
`classSignalBuffer::SerializePatternDWord`,
`classSignalBuffer::ParallelizeSerialStream`, and
`classSignalBuffer::ParallelizeSerialDWord`

are used, they increment the index pointer.

Parameters

- N/A.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.
`enAPIFunctionFailed` – if the function fails.

See Also

`classSignalBuffer::MoveIndex`

classSignalBuffer::DisplayBuffer

Synopsis

```
enAPIReturnCodes    classSignalBuffer::DisplayBuffer(
);
enAPIReturnCodes    classSignalBuffer::DisplayBuffer(
apiDWORD            dwNClocks
);
```

Description

This method displays all or N elements of the Signal Buffer on stdout.

Parameters

- N/A.

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`
`enAPIFunctionFailed – if the function fails.`

See Also

classSignalBuffer::ClearBuffer

`classSignalBuffer::FillTestPattern`

Synopsis

```
enAPIReturnCodes    classSignalBuffer::FillTestPattern(
apiWORD             wCommandLength
);
```

Description

This method fills the Signal Buffer with 8 test patterns in signal location 0,4,8,12,16,20,24,28,32.

Parameters

- `wCommandLength: number of elements to fill.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classSignalBuffer::DisplayBuffer`

classStateMachine::classStateMachine

Synopsis

```
classStateMachine::classStateMachine (
apiDWORD      dwNStates,
apiDWORD      dwNStimuli,
apiDWORD      dwInitialState
);
```

Description

This method is the constructor for a state machine object. It instantiates an abstraction of a finite state machine consisting of a set of State Elements of dimensionality `dwNStates` by `dwNStimuli`.

Parameters

`dwNStates: the set of all states through which the finite state machine can transition.`

`dwNStimuli: the set of all stimuli to which the finite state machine can respond.`

`dwInitialState: the initial state in which the state machine is set.`

Return Values

- `N/A.`

See Also

`classStateMachine::~classStateMachine`

`classStateMachine::~classStateMachine`

Synopsis

```
classStateMachine::~classStateMachine (
);
```

Description

This method is the destructor of the state machine object. It deinstantiates and frees the memory allocate for the State Element set.

Parameters

- `dwNStates: the set of all states through which the finite state machine can transition.`
- `dwNStimuli: the set of all stimuli to which the finite state machine can respond.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classStateMachine::classStateMachine`

classStateMachine::SetElement

Synopsis

```
EnAPIReturnCodes    classStateMachine::SetElement (
apiDWORD            dwState,
apiDWORD            dwStimuli,
apiDWORD            dwNextState,
apiDWORD            dwOutput,
enAPIReturnCodes    (*ptrTargetFunction)(void *)
);
```

Description

This method initializes an individual node in the `classStateMachine` state space.

Parameters

- `dwState: the state index of the node to be initialized.`
- `dwStimuli: the stimuli index of the node to be initialized.`
- `dwNextState: the state to which this node will transition.`
- `dwOutput: the output code to express at the completion of transitioning to the new state.`
- `ptrTargetFunction: a pointer to a void function which is executed at the transition to the new state.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classStateMachine::Jump`

`classStateMachine::Jump`

Synopsis

```
enAPIReturnCodes    classStateMachine::Jump (
apiDWORD            dwState,
apiDWORD            dwStimuli,
void                *ptrArgument
);
```

Description

This method executes the function that is associated with a node in the set of state elements.

Parameters

- `dwState: the state index of the node to be executed.`
- `dwStimuli: the stimuli index of the node to be executed.`
- `ptrArgument: a void pointer to an argument which is passed to the function.`

Return Values

`enAPIFunctionSuccessful – if the function completes successfully.`

`enAPIFunctionFailed – if the function fails.`

See Also

`classStateMachine::SetElement`

classStateMachine::Transition

Synopsis

```
enAPIReturnCodes    classStateMachine::Transition (
apiDWORD            dwStimuli,
void                *ptrArgument
);
```

Description

This method will step the state machine based on it’s current state and input stimuli.

Parameters

- `dwStimuli`: the stimuli that is to be applied to the state machine.
- `ptrArgument`: a void pointer to an argument which is passed to the function.

Return Values

`enAPIFunctionSuccessful` – if the function completes successfully.

`enAPIFunctionFailed` – if the function fails.

See Also

classStateMachine::Jump

Appendix F

Glossary

API

Application Program Interface, a set of routines, protocols, and tools for building software applications. An API makes it easier to develop a program by providing all the building blocks, which a programmer puts together.

APIs are implemented by writing function calls in the program, which provide the linkage to a specific sub-routine for execution. Thus, an API implies that some program module or routine is either already in place or must be linked in to perform the tasks requested by the function call.

Architecture

How the system or software is constructed.

Boot ROMs

The program used to start a computer that is stored in Read Only Memory. In VxWorks, boot ROMs are used to download the VxWorks kernel from a host computer over the network.

BSP

Board Support Package. The part of VxWorks that manages the CPU board on which VxWorks runs, that is, the part which handles the hardware like Ethernet, serial, etc.

Bus

The physical connection between components of a single computer system. Imagine it as a freeway with multiple lanes connecting various parts of the system. There are different types of busses (VME, EISA, PCI, etc.).

C

A high-level programming language developed at Bell Labs that is able to manipulate the computer at a low level like assembly language. C can be compiled into machine languages for almost all computers. For example, UNIX is written in C and runs in a wide variety of micros, minis and mainframes.

C++

An object-oriented version of the C programming language.

C++ Classes

In object-oriented C++ programming, a class is a category of objects. For example, there might be a class called shape that contains objects which are circles, rectangles, and triangles. The class defines all the common properties of the different objects that belong to it.

C++ Methods

The underlying function contained in a C++ class that preforms a specific operation.

C++ Objects

In object-oriented C++ programming, an object is a self-contained entity that consists of both data and procedures to manipulate the data.

Client

This term is a system-centric term. An object that requests another object to perform some function for it. For example, in a Compact PCI system the system slot board is a client to other special purpose, peripheral slot boards in the rack.

Client Domain

The client domain is a component of the PAVE Logical System Partitioning. It is comprised of those elements that are involved in the control of embedded system components.

Codebase

Fundamental underlying code of a program, without the associated data files.

Deinstantiate

Destroying an instance of an object.

Device element

A segment in a payload that contains the configuration information for a specific device.

Device object

An FPGA in an IRL-enabled System. PAVE views an FPGA as a standalone device object within a system component that software can modify at will, like any other software component.

Download

This term is object-centric. In the PAVE framework, objects download information. For example, the classRegister::Download method results in the contents of a register being read.

Driver

The software that communicates between a hardware peripheral and the rest of the system. Often called a “device driver.” In VxWorks, a device driver often refers only to those drivers which have a UNIX-like interface.

Embedded system

A specialized computer, often hidden from the end user, used to control devices such as automobiles, home and office appliances, hand-held units of all kinds as well as machines as sophisticated as space vehicles. Operating system and application functions are often combined in the same program. An embedded system implies a fixed set of functions programmed into a non-volatile memory (ROM, flash memory, etc.) in contrast to a general-purpose computing machine. Think of it like a self-contained system. An example would be a computer in a car that controls the ignition system. Because they often operate critically important applications, reliable real-time reactions are vital.

Endian Swap

Data can be ordered with the MSB at the beginning or end of multibyte data type. Depending on the system you are targeting you may or may not need to preform an endian swap to correctly align the data. More details on Byte ordering can be found at:

http://webopedia.internet.com/TERM/B/big_endian.html

Environment

A particular configuration of hardware or software. The environment refers to a hardware platform and the operating system that is used in it. A programming environment would include the compiler and associated development tools. Environment is used in other ways to express a type of configuration, such as a networking environment.

Flash memory

Memory that, unlike most RAM, retains its value when powered down, but can only be erased in bulk. Often used instead of PROMs.

FPGA

Field Programmable Gate Array, invented by Xilinx in 1984. FPGAs are CMOS SRAM-based devices, that reload their configuration each time they are powered up.

Host

A host (computer) communicates with a target (a CPU board running VxWorks) over a network.

Host Domain

The host domain is a component of the PAVE Logical System Partitioning. It is comprised of those elements that are used to develop and manage embedded applications. A workstation and its resident tool chain are generally host domain elements.

Instantiate

Creating a new instance of an object.

IP

Internet Protocol. The IP part of the TCP / IP communications protocol. IP implements the network layer (layer 3) of the protocol, which contains a network address and is used to route a message to a different network or subnetwork. IP accepts "packets" from the layer 4 transport protocol (TCP or UDP), adds its own header to it and delivers a "datagram" to the layer 2 data link protocol. It may also break the packet into fragments to support the maximum transmission unit (MTU) of the network.

IRL-enabled Device

A Xilinx FPGA with the configuration pins connected in an IRL-enabled Architecture.

IRL-enabled Architecture

The combination of a processor running PAVE, a memory mapped device configuration register, and a Xilinx FPGA. Figure F-1 shows the difference between an IRL-enabled Device and an IRL-enabled Architecture.

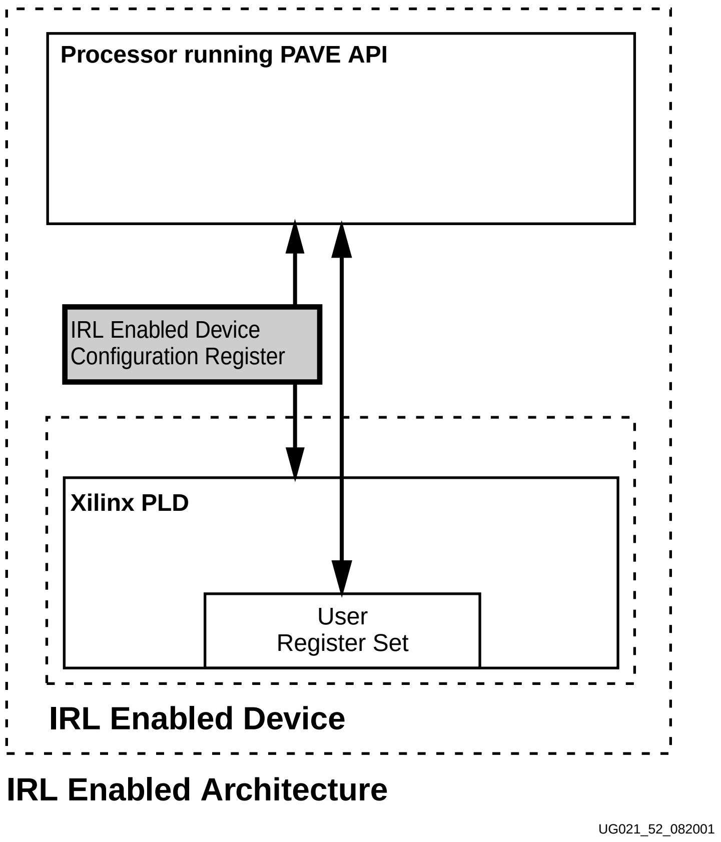


Figure F-1: **IRL Enabled Architecture**

Library

A collection of usually related subroutines in one file. Re-usable libraries are the basis of object-oriented programming.

Methods

See **C++ Methods**.

Object-centric

A view point that is self centered.

Object Model

Object oriented programming (OOP)

Programming that supports object technology. It is an evolutionary form of modular programming with more formal rules that allow pieces of software to be reused and interchanged between programs. OOP is thought to increase productivity by allowing programmers to focus on higher level software objects.

Payload

A combination of header information, configuration bitstreams, and device object modules that are concatenated into a single binary object and transported to a system.

PLD

Programmable Logic Device. Can be used generically for both SRAM-based FPGAs and macro-cell based devices such as CPLDs and PALs.

Real-time

Refers to the ability to respond to events quickly and predictably. In real-time programming a late response is a wrong response.

Real-time operating system (RTOS)

An operating system designed for use in a real-time computer system. A master control program that can provide immediate response to input signals and transactions. VxWorks is an RTOS.

Real-time server

That part of a computer network that handles the real-time needs of the system. VxWorks is often used as a real-time server.

SBC

SIngle Board Computer, typically a system board used in a CompactPCI or VME system.

SelectMAP

SelectMAP is an 8-bit bidirectional databus interface to the Virtex configuration logic that may be used for both configuration and readback.

Server

This term is a system-centric term. An object that provides some functionality to other objects. For example, in a Compact PCI system the peripheral slot boards are servers to the system slot board.

Server Domain

The client domain is a component of the PAVE Logical System Partitioning. It is comprised of those elements that provide specialized functionality within an embedded system.

SIF

System Integration Framework. Part of the PAVE Framework that customizes your software environment for your hardware setup.

Socket

A software protocol developed by UC Berkeley that allows a program on one system to talk to a program on another system.

Source code

The human readable version of a software program. The source code is then compiled and linked to generate code that a computer can run (binary executable). In other words, programmers write (or read) source code; computers run binary executables.

System-centric

A view point that is evaluated in the context of the overall system.

TCP/IP

Transport Control Protocol / Internet Protocol. Often used to refer to the Internet Protocol (IP) in general. Internet Protocol has different ways of handling and packaging data for transport over a network, such as TCP and UDP (User Datagram Protocol).

Target

The computer into which a program is loaded and run.

Target agent

In Tornado, the software on the target hardware that is responsible for communicating with the host computer.

Target server

In Tornado, the software on the host computer responsible for managing communications with the target hardware.

Upload

This term is object-centric. In the PAVE framework, objects upload information. For example, the `classRegister::Upload` method results in the register being written.

XSVF

Xilinx Serial Vector Format.

Index

A

About This Manual 13
ADMXRC 33, 99 - 100
ADMXRC Applications 101
ADMXRC Payloads 109
API Calls
- classDevice 56
  - ~classDevice 126
  - classDevice 123 - 125
  - UploadRegister 127
- classIRLBStreamDeliveryModule 55 - 56
- classIRLDevice
  - ~classIRLDevice 130
  - CachePayload 131
  - classIRLDevice 128 - 129
  - DownloadPayloadToBuffer 138
  - DownloadPayloadToFile 139
  - DownloadRegister 142
  - GetPayloadChecksum 134
  - GetPayloadOffset 133
  - GetPayloadSize 132
  - ModifyRegister 143
  - SplitPayload 137
  - UnCachePayload 140
  - UploadPayloadFromBuffer 135
  - UploadPayloadFromFile 136
  - UploadRegister 141
- classIRLPlatform 50, 56
- classPCIDevice 101
- classPlatform 49
- classRegister 49, 56
  - ~classRegister 113
  - classRegister 112
  - Clear 120
  - Display 122
  - Download 115
  - Initialize 121
  - Modify 116
  - Reset 119
  - ToggleSignal 118
  - Upload 114
  - UploadSignals 117
- classSignalBuffer 49, 56
  - ~classSignalBuffer 145
  - AssertSignal 148
  - AttachBuffer 146
  - classSignalBuffer 144
  - ClearBuffer 160
  - DetachBuffer 147
  - DisplayBuffer 163
  - DownloadFile 159
  - FillTestPattern 164
  - GetIndex 162
  - MoveIndex 161
  - ParallelizeSerialDWord 156 - 157
  - ParallelizeSerialStream 155
  - SerializePatternDWord 154
  - SerializePatternStream 153
  - SerializeRepetitiveBitSequence 149
  - SerializeRepetitivePattern 151
  - UploadFile 158
- classStateMachine 49, 56
  - ~classStateMachine 166
  - classStateMachine 165
  - Jump 168
  - SetElement 167
  - Transition 169

API Structure 53

B

Building a Custom Framework 41

C

C++
- Methods 172
- Resources 14
- Training 13

D

Developer Productivity 22
Durango 33, 85
Durango Code Examples 88
Durango Framework 87
Durango Hardware Features 94
Durango PCB Layout 96
Durango Reference Design 93

F

FFT 105

I

Installation 24
Installing to an alternate drive 33
IRL Register model 35
IRL-Enabled 20

J

JTAG
- Code example 62
- Format 44
- Programmer 45
- SVF2XSVF 47
- TAP Controller 61
- XSVF 44, 62

L

Low Cost 21

M

Motorola MCP750 85, 100

N

Network
- Configuration 69
- Domains 69
- TCP/IP 71

Network Configuration 69

P

PAVE
- API 19, 49
- API Classes 56
- Framework 19, 29
- Framework Partitioning 69
- generatedevice.exe 27, 38
- generatepayload.bat 42
- generatepayload.exe 42
- generatepayload.exe usage 43
- generateserver.exe 27, 40
- generatesystem.bat 38, 40
- INP Format 40 - 41
- SIF 19, 35
- SIF Flow 37
- SIF Tutorial 25
- Skills Required 13
- Software License iii
- Training 14
- VxSim Tutorial 30

Payload 51
Payload Configuration Segment Structure 58
Payload Diagram 42
Payload Generator 42
Payload Header Structure 58
Payloads 58
Pull 71, 103
Push 71, 103

R

Reliability 21

S

Scalability 20
SelectMAP 44, 65
  Bitstreams 66
  Code example 66
Summary 93
System Domains 70
System Generator 26, 38
System Requirements 23

T

Text Editors 33
Typographical Conventions 17

W

Wind River Systems
  Support 14
  Tornado 26
  Training 14

X

Xilinx
  Support 14 - 15
XIRL Interface Register 49, 61
  Map 36
  Rewiring 63, 67
  Signal Names 91
  Tables 90