USER'S GUIDE

NAVIGATOR Board Support Package

for Pentek's Jade Family of Products

Version 2.4

PENTEK

Pentek, Inc.
One Park Way
Upper Saddle River, NJ 07458
(201) 818–5900
http://www.pentek.com/

Manual Part Number: 800.48145 Rev: 2.4 – April 13, 2018

Manual Revision History

Date	Version	Comments
5/8/17	1.0	Initial Release
8/3/17	2.0	Version 2.0 release adds support for Models 71131 and 71841. Revised Sect 1.2, Sect 1.3.1, Sect 3.3.1, Sect 3.3.2, Sect 3.3.6, Sect 3.4.1, Sect 3.3, Sect 3.3.6.2, Sect 3.4.6.2, Sect 3.4.6.2, Sect 3.4.6.6, Sect 7.5, Sect 7.7, Sect 7.8, Sect 7.9, Sect 7.10, and Sect 9.2. Appendix A: added and updated command line arguments. Appendix C: revised code snippets in all scenarios; also revised Sect C.7.
10/4/17	2.1	Version 2.1 release adds support for Models 71821 and 71851. Added Sect 9.5 (for KB Case 1540). Revised -adcdatasrc, −dacdatamode, −dacdatasrc, −xfersize, and Sect A.3.
12/15/17	2.2	Version 2.2 release adds support for Model 71862. It also adds support for Models 7192/7192A/ 9192/9192A via the I2C interface on a Model 71841. Revised Sect 1.3.2, Sect 3.4, and Sect 8.3. In Appendix A, revised -adcdatasrc, −adcoption, −brdoption, −clkoption, −dacoption, −ddcoption, −dmaoption, −ducoption, −gateoption, −progoption, −syncoption, −vhost, −vport, and Sect A.3. Revised Sect C.1. Revised code snippet in Sect C.5. Revised text and snippet in Sect C.6.
3/9/18	2.3	Version 2.3 release adds support for Model 71132. Updated the driver version to 12.60. Revised Sect 3.3.1. In Appendix A, added −adcgatedly, −adcindly, −adcsyncdly, −caddr, −cport, −dac−gatedly, −dacsyncdly, −saddr, −sport, −vchanmask, and −vsubchan. Revised -adcfreq, -adcdat−amode, −dacfreq, −dacdatamode, −dacppssrc, −tstart, −tstop, −vhost, and Using Command−Line Arguments. In Appendix B, revised Sect B.3.
4/13/18	2.4	Version 2.4 release adds support for Model 71141. Revised −dacdatamode.

Software License Agreement

Software Limitations

Copyright

Trademarks

Printed in the United States of America.

Table of Contents

Page

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Page

This page is intentionally blank

Chapter 1: Introduction

1.1 Navigator BSP: Part of the Navigator Design Suite

The Navigator™ Design Suite supports Pentek's Jade® family of products, which are XMC modules based on the Xilinx® Kintex® Ultrascale® FPGAs. Jade XMCs are offered on a variety of carriers. The Navigator Design Suite consists of two parts:

❑ **Navigator™ Board Support Package (BSP)** – Provides C–callable software routines for accessing and controlling a Pentek Jade XMC module.

❑ **Navigator™ FPGA Design Kit (FDK)** – Pentek's IP design tool, which uses Vivado's block diagram design interface and AXI–4 (Advanced eXtensible Interface), which enables a modular approach when creating IP.

More information about the Navigator Design Suite and the Navigator BSP's role in the Suite is provided in Chapter 2.

This document describes the Navigator™ Board Support Package (BSP) and how to install and use it. Supplemental information is provided in the appendices.

1.2 Documentation for the Navigator Design Suite

- *Navigator BSP User's Guide* (this document)

- The *API Reference Guide for the Navigator BSP* is available in both HTML and PDF format (**`API_Reference.html`** and **`API_Reference.pdf`**). For more information, refer to Section 4.2. The *API Reference Guide* is available in the following location:

  Windows: **`C:\Pentek\BSP\BSP_X.Y\docs`** (where **`X.Y`** is the version number)
  or
  `%NAVBSP%\docs`

  Linux: **`/home/username/Pentek/BSP/BSP_X.Y/docs`** (where **`X.Y`** is the version number)
  or
  `$NAVBSP/docs`

- Provided for FDK users, the *Navigator FDK User's Guide* describes how to install and use the FDK software. The ***IP Core Conventions Guide and Example Labs*** tutorial (807.48111) describes the style, conventions, and standards used in all of Pentek's Navigator IP Cores and interfaces, and provides procedures for creating your own IP cores so they are compatible with the standards and interoperable with the Navigator IP Core Library. Other supplemental manuals also are available. You can get the Navigator FDK and associated user manuals by contacting sales@pentek.com.

- An *IP Core Manual* is provided for each Pentek Navigator IP core in the FDK. These can be accessed via the Vivado IP Integrator. Some also can be accessed from the operating manual for the Jade board.

1.3 System Requirements for the Navigator BSP

- ❑ National Instruments™ LabVIEW™ software
- ❑ Microsoft® Visual Studio® Professional 2015

NOTE: To run the executable from the Windows Navigator BSP, the following software components are required:

- Microsoft Visual C++ 2015 Redistributable package for running the Debug and Release versions. This may have already been installed on your computer when you installed other Windows applications. It may be included on your Microsoft Visual Studio installation disk.
- Windows SDK for running the Debug version of the executable. It is included with Microsoft Visual Studio.
- Microsoft Visual Studio 2015 IDE for utilizing the included solution/project files in developing your application.

1.3.1 Windows

- Processor: Intel® Pentium® 4 G1 (or equivalent) or later
- RAM: 1 GB
- Screen resolution: 1024 x 768 pixels
- Operating system (only 64–bit editions are supported):

  Microsoft Windows® 10/8.1/8/7 SP1

  Windows Server 2012 R2

  Windows Server 2008 R2 SP1
- Disk space: 5 GB

1.3.2 Linux

- Processor, RAM, and screen resolution: same as Windows
- Linux distributions (only 64–bit editions are supported): Debian GNU/Linux 7, 8, 9; Ubuntu 14.04, 15.04, 16.04, 17.04; Fedora 21, 22, 23, 24; RedHat Enterprise Linux® 6, 7; CentOS 6, 7; Scientific Linux 6, 7; Arch Linux 2017.09.01; openSUSE® 13.2, Leap 42.3; SUSE Linux Enterprise 12 SP3
- Eclipse IDE: The project files included in the Navigator BSP were created and tested with Eclipse IDE version 4.6.1, but they should also be compatible with older versions of Eclipse.
- Disk space: 2.2 GB

Chapter 2: Understanding the Navigator Design Suite

2.1 The Navigator Design Suite

Because the Navigator BSP is part of the Navigator Design Suite, it is important to understand the purpose and role of the Suite as a whole and how it is used. Pentek's Navigator Design Suite was designed to work with Pentek's Jade architecture products and it provides a new approach that helps users to more easily navigate through the task of IP and control software creation and compatibility.

The Navigator Design Suite contains two separate but closely related products:

- ❑ The **Navigator FPGA Design Kit (FDK)** for integrating custom IP into Pentek–sourced designs
- ❑ The **Navigator Board Support Package (BSP)** for creating host applications.

Users are able to work efficiently at the API level for software development and with an intuitive graphical interface for IP design.

2.2 Use Case Scenarios for the Navigator Design Suite

It will be easier to understand the Navigator Design Suite if we look at how it might be used. Let's look at two use case scenarios and how the Navigator Design Suite can be used in each.

2.2.1 Scenario 1: No FPGA Customization Needed

Each Pentek data acquisition and processing hardware product has an FPGA. Pentek's Jade family uses a Xilinx® Kintex® Ultrascale® FPGA. Like all Pentek products, Jade includes a full suite of built–in FPGA–based functions. For example, in the case of an A/D converter product, these functions include:

- an A/D data acquisition engine,
- a fully–programmable DDC,
- power meters and a threshold detect function,
- a timestamp and metadata creation engine, and
- a linked list DMA engine that allows users to customize data transfers to a host computer.

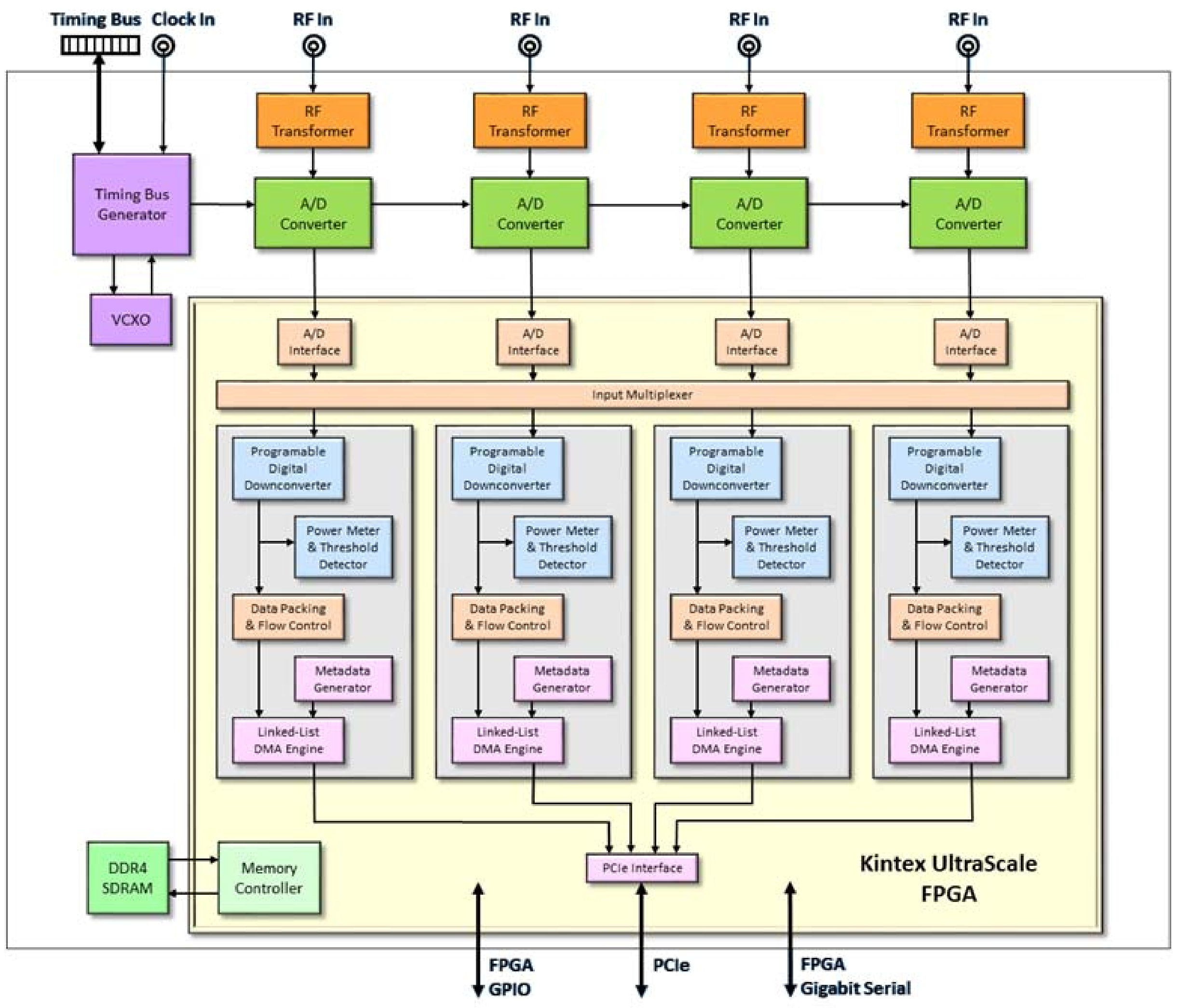
In many cases, users will find that the built–in FPGA–based functions satisfy all the requirements of their application and no custom FPGA IP is needed (see Figure 2–1). For these users, the FPGA looks like just another piece of hardware with fixed functions and a fixed interface for status and control.

2.2 Use Case Scenarios for the Navigator Design Suite (continued)

2.2.1 Scenario 1: No FPGA Customization Needed (continued)

In this situation, the Navigator BSP API is the best solution for creating appli–cations that control the Jade hardware. Provided as a C–callable high–level API, many of the most commonly used built–in functions can be controlled with simple commands. In addition, example programs (see Chapter 6 and Chapter 7) and the included Signal Viewer (see Appendix B) allow users to immediately start acquiring and displaying data in the time and frequency domains without the need for creating any code.

Figure 2–1: Jade Model 71861 Built–in FPGA Functions

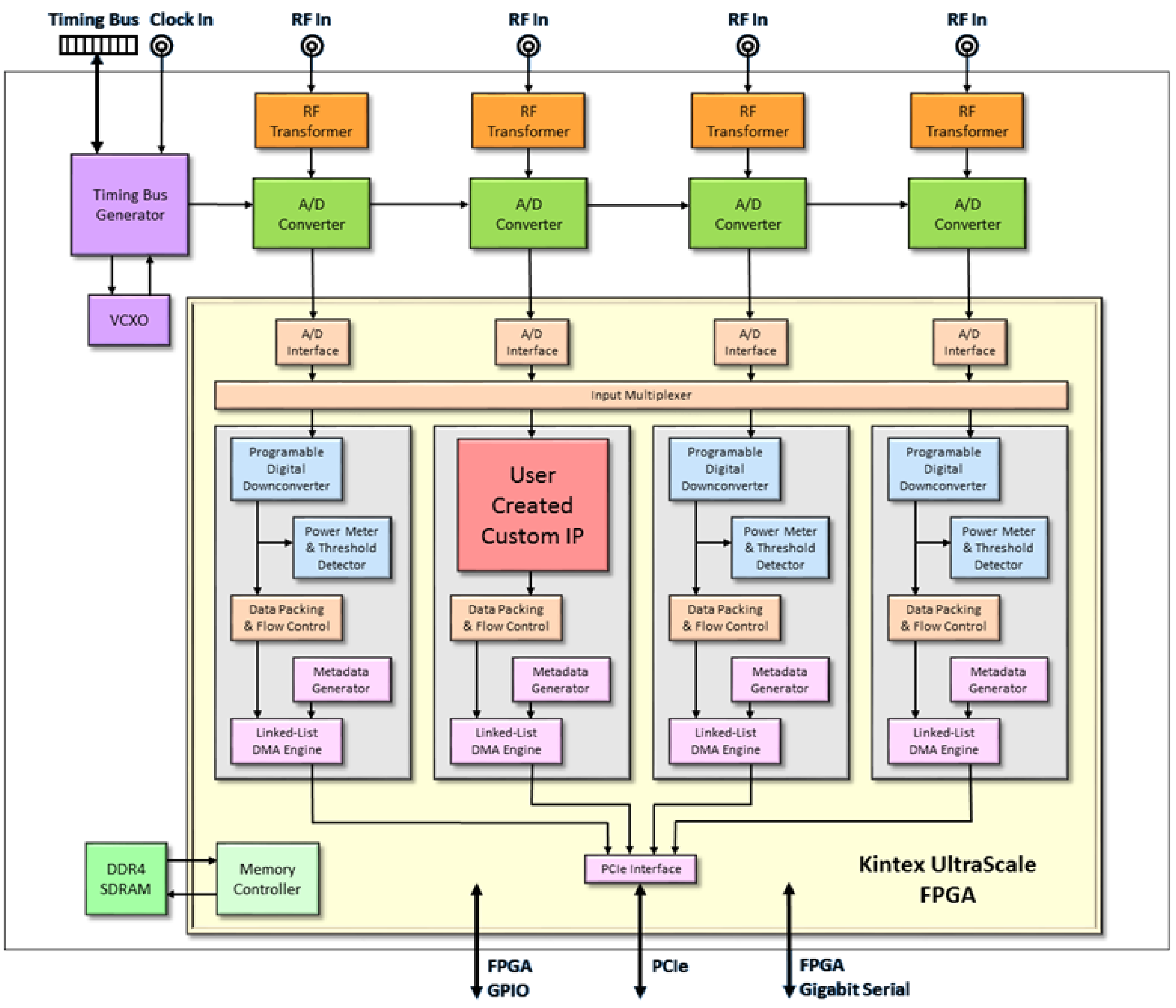


2.2 Use Case Scenarios for the Navigator Design Suite (continued)

2.2.2 Scenario 2: Customizing the FPGA IP

However, if an application requires special processing that only custom IP can provide (Figure 2–2), the solution is the Navigator FDK. It was created to work directly with the Xilinx Vivado® Design Suite and creates a seamless environment for developing IP on Pentek products. [1]

Figure 2–2: User–Created Custom IP



1.You can get the Navigator FDK and associated user manuals by contacting sales@pentek.com.

2.3 The Navigator FDK

The Navigator FDK leverages two new features in Vivado to greatly streamline IP devel–opment: the AXI4 standard and the IP Integrator.

2.3.1 The AXI4 Standard

AXI4 is the 4th generation of an interface specification from ARM® commonly used in the semiconductor industry. Xilinx has adopted this standard to create AXI4–compliant plug–and–play IP. The benefits can be seen immediately:

- AXI4 consolidates an array of different possible interfaces into a single well–defined interface.
- AXI4 manages differences in bus speed and width when connecting IP blocks.
- AXI4 allows easier integration of IP from different sources when all IP is using the same interface.
- AXI4 still allows enough flexibility to enable IP designers to tailor the interconnects to meet system performance requirements.

Navigator FDK follows the AXI4 standard. For Pentek's Jade data acquisition and processing products, the FDK includes the complete IP that is factory–installed in the board. This includes all interface, processing, data formatting, DMA functions, etc. IP designers can modify or replace functions as needed to match application requirements, and will find immediate compatibility with Xilinx IP and third–party IP that uses AXI4. Designers who create their own custom IP using the AXI4 standard will find integration with the Pentek–sup–plied IP straightforward.

2.3.2 The IP Integrator

So how is the FPGA design actually edited? This is where Navigator FDK exploits another new feature of Vivado: the IP Integrator. The concept of cre–ating FPGA designs by connecting blocks in a graphical interface, similar to drawing a schematic, is not new, but Xilinx's IP Integrator makes it a practical solution.

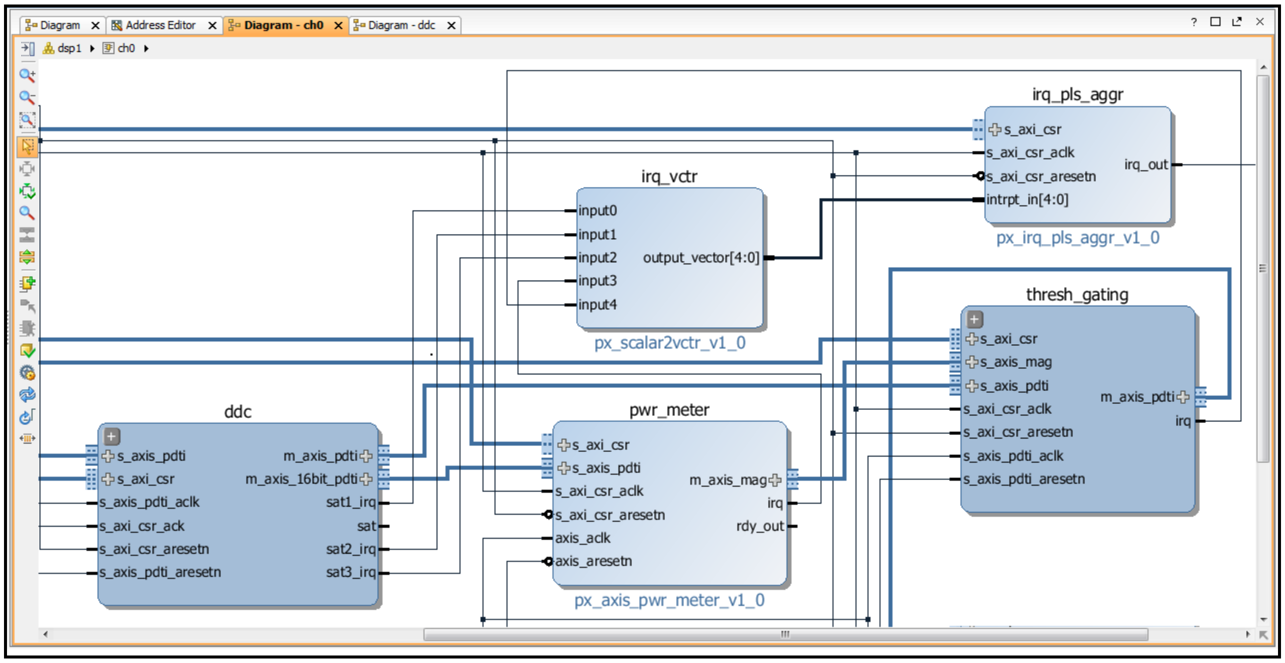
To edit a Pentek product design, an FPGA engineer opens the Navigator FDK design in Vivado. He then has immediate access to the entire board design as a block diagram. Individual IP cores can be removed, modified, or replaced with custom IP to meet the application's processing requirements. Because all blocks have AXI4 interfaces, connections between blocks can simply be "drawn" with wires or buses and the AXI4 interface handles the "housekeep–ing" of different bus speeds or widths.

2.3 The Navigator FDK (continued)

2.3.2 The IP Integrator (continued)

Viewing an FPGA design as a block diagram (see Figure 2–3) enables the designer to see the data flow and simplifies the design processes by working at the “interface” and not the “signal” level. If, at any time, a designer needs to work with the VHDL code directly, it is always accessible in a source window. Full on–line documentation for every Pentek IP core also is available via the IP Integrator. A complete procedure for creating your own IP core is provided in *IP Core Conventions Guide and Example Labs* (807.48111 – obtain by contact–ing sales@pentek.com).

Figure 2–3: IP Integrator



Once a board's function has been modified by changing FPGA IP, it is most likely that changes will need to be made in the software controlling the board to support the new function. While the Navigator's API is ideal for creating applications for the board, it assumes the board functions have not changed from the factory–installed set.

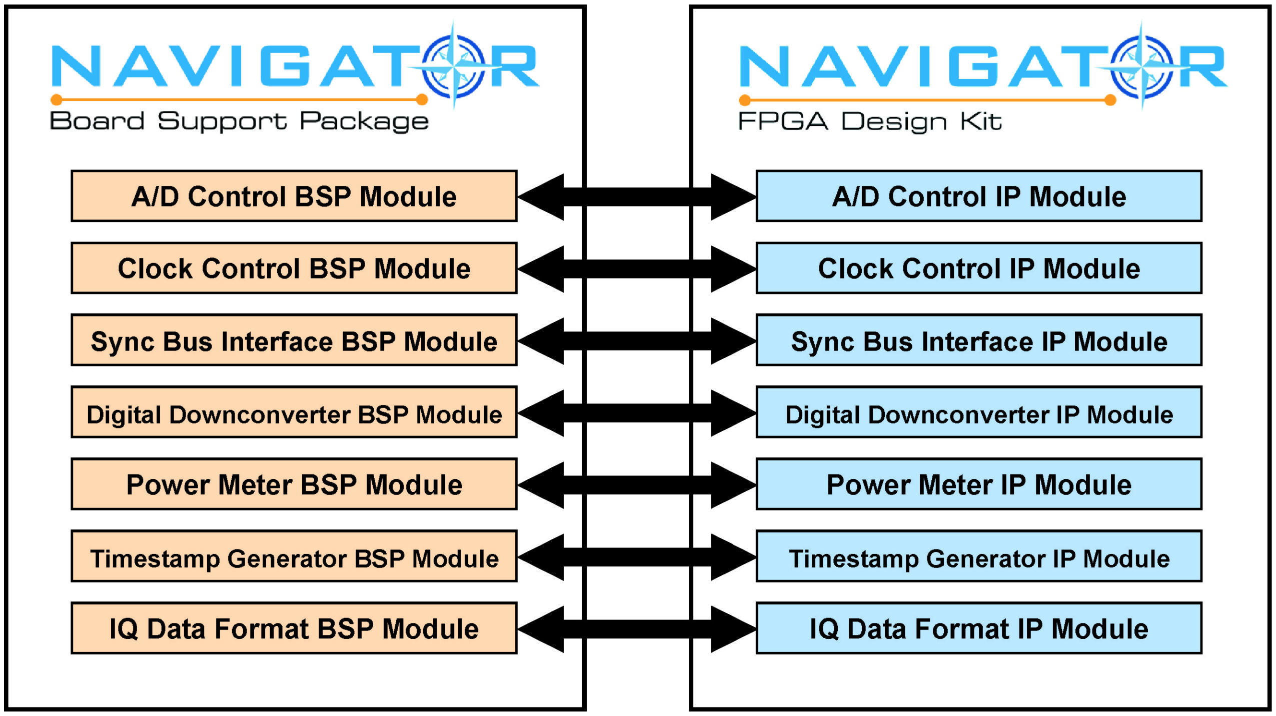
2.3 The Navigator FDK (continued)

2.3.3 The Role of Navigator BSP

Once custom IP is introduced in the FPGA, the Navigator BSP module library is the solution for modifying or creating new software. Designed to work with the Navigator FDK, the Navigator BSP is structured to simplify this process.

Each Navigator IP core module found in the FDK has an equivalent software module with a similar name in the Navigator BSP. Changes made to an IP module can be easily traced back to the BSP module to make the necessary changes to control the new IP. This one–to–one relationship between IP and software greatly simplifies the task of keeping IP and software in sync.

Figure 2–4: Navigator BSP and FDK



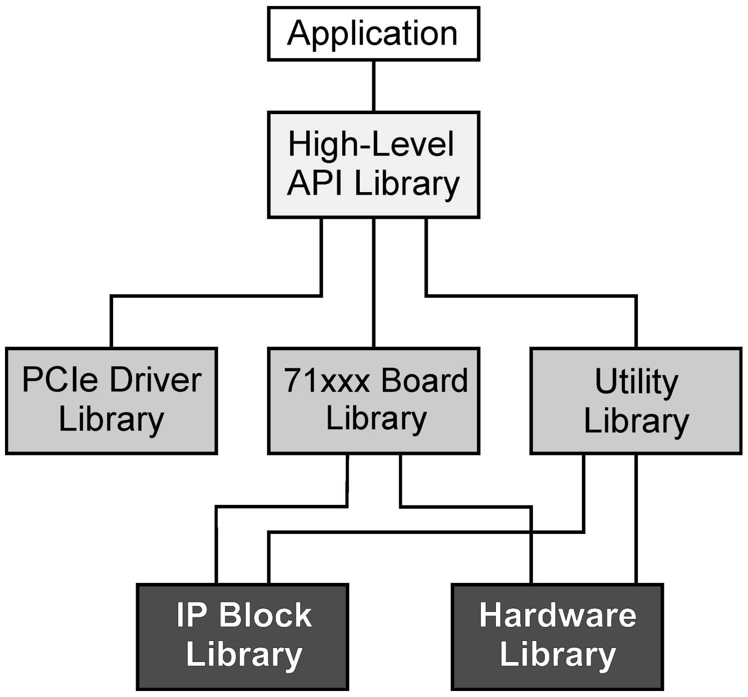
2.4 "Building Block" FPGA Design and Layered BSP Architecture

The Navigator FPGA Design Kit (FDK) for Pentek's Jade family of boards is character–ized by a "building–block" approach to FPGA design. FPGA code is developed by combining IP core modules to create the board functionality.

The Navigator BSP was designed to correspond to the Navigator FDK's "building–block" approach to FPGA design. The BSP is a "layered architecture" with four layers:

- **Application** – User programs are application layer code. These programs call routines in the high–level API library.
- **High–Level API** – High–level API library routines call routines in the board–specific, PCIe driver, and utility libraries.
- **Board–Specific, PCIe Driver, and Utility** – These libraries consist of routines that call the lowest level library routines that interface to the FPGA IP core module or hardware devices.
- **IP Block / Hardware –** These libraries communicate with the IP block and hardware.

Figure 2–5: BSP Layer Diagram



2.5 BSP Library Layers

This section describes the BSP library layers in more detail, starting with the bottom layer (nearest the hardware).

2.5.1 IP Block / Hardware Layer

This layer consists of two components: the IP block library and the hardware library.

- **IP block library:** This library provides the software interface to the IP core modules and communicates directly with them. It is tightly integrated with the IP core modules and provides a single source and single header file to support each module. All files are compiled into a single IP core dynamic–link library (DLL). Functions in this DLL are accessed only via board–specific layer functions.

- **Hardware library:** This library communicates with hardware devices that are external to the FPGA. Currently, there are no memory–mapped hardware devices. There are hardware devices outside the FPGA, such as the Si571 VCXO and the CDCM7005 clock divider, but these are not memory–mapped. They are accessed via IP core modules, such as an I2C interface. Any functions in a DLL for any future memory–mapped devices will be accessed only from board–specific layer functions.

2.5.2 Board–Specific / PCIe Support / Utility Layer

This layer consists of three components: board–specific libraries, PCIe support library, and utility library.

- **Board–specific libraries** are single DLLs tailored for each XMC board model family. For example, there is a Model 71861 board family DLL, a 71851–family DLL, etc. Routines in the DLLs are built around calls to routines in the IP block/hardware libraries. Hardware debugging routines, such as register dumps, are included in these libraries. Board–specific libraries also include routines to access hardware devices on the board that are external to the FPGA but accessed via IP core modules. For example, the ADS5485 ADC is programmed via the **`px_ads5485intrfc`** IP core module. For more information about IP core modules, see Section 2.3.

- **PCIe support library:** Single DLLs are provided for PCIe support. The PCIe driver interfaces to the PCIe bus and does not call external BSP DLLs.

- **Utility library:** This library provides operating system routines for file I/O, printing, command line parsing, etc. The utility library also contains files that support the Signal Viewer. For more information about the command line utility, see Appendix A. Appendix B describes the Signal Viewer.

2.5 BSP Library Layers (continued)

2.5.3 High–Level API Layer

The high–level API layer provides routines called by user or example pro–grams to program and control a Pentek Jade family board. These routines are generic in nature, calling board–specific routines via function pointers. There is no direct communication to the IP core or hardware library routines. This layer makes it unnecessary for users to become directly involved with board–specific details.

For more information, refer to the *Navigator Board Support Package API Refer–ence Guide*, which is available in HTML and PDF format (**`API_Reference.html`** and **`API_Reference.pdf`**) in the following location:

Windows: **`C:\Pentek\BSP\BSP_X.Y\docs`**
(where **`X.Y`** is the version number)
or
`%NAVBSP%\docs`

Linux: **`/home/username/Pentek/BSP/BSP_X.Y/docs`**
(where **`X.Y`** is the version number)
or
`$NAVBSP/docs`

2.5.4 Application Layer

The application layer is the code the user develops for his desired application. General–purpose examples are provided with the BSP.

This page is intentionally blank

Chapter 3: Installing the Navigator BSP

3.1 Introduction

This chapter provides the installation procedures for the Navigator BSP:

- ❑ Section 3.3 – Windows Installation Procedures
- ❑ Section 3.4 – Linux Installation Procedures

3.2 BSP Components

The Navigator BSP consists of three components, which are listed by their names in the directory structure:

1) **driver:** Universal driver
2) **bsp:** Universal board support package (main component)
3) **71nnn**: Board–specific **example programs** and MATLAB scripts for processing output, where **71nnn** is the model number of the Pentek Jade board.

The components are installed in the order given above.

3.3 Windows Installation Procedures

This section contains the following:

- Section 3.3.1 – Step 1: Install the Navigator Driver
- Section 3.3.2 – Step 2: Install the Navigator Board Support Package
- Section 3.3.3 – Step 3: Install the Navigator Example Programs
- Section 3.3.4 – Step 4 : Install the LabVIEW Runtime Engine
- Section 3.3.5 – Step 5: Windows Hardware Initialization
- Section 3.3.6 – How to Manually Install and Uninstall the Navigator BSP Device Driver

3.3 Windows Installation Procedure (continued)

3.3.1 Step 1: Install the Navigator Driver

1) Install the Navigator driver package from the distribution CD provided using the **`setup.exe`** program on the CD. DOS commands are used for illustration.

```
cd <CDROM_drive>:\NavDriver\Driver_12.60
run setup.exe
```

2) Define the Navigator directory environment variable. Navigator uses the Windows environment variable **`NAVBSP_DRVR`** to determine the location on the disk where the driver package has been installed. The variable should be set as follows:

Variable:	**`NAVBSP_DRVR`**
Value:	[Navigator driver base directory] (e.g., **`c:\Pentek\BSP\Driver_12.60`**)

NOTE: This environment variable is created automatically during installation and should not have to be set manually.

3) Add the Navigator driver directory search path to the Windows PATH environment variable: **`%NAVBSP_DRVR%`**.

The Path variable is found in the User Variables section of System Properties on a Windows System. PATH is modified during this installation. It should be the first element in PATH following this installation.

NOTE: The PATH environment variable is modified automatically during installation and should not have to be set manually.

3.3 Windows Installation Procedure (continued)

3.3.2 Step 2: Install the Navigator Board Support Package

1) Install the Navigator BSP package from the distribution CD provided using the **`setup.exe`** program on the CD. DOS commands are used for illustration. **`<VERSION>`** will be the version of the BSP (e.g., 1.0, 1.1, 2.0, etc.).

   ```
   cd <CDROM_drive>:\NavBSP\BSP_<VERSION>
   run setup.exe
   ```

2) Define the Navigator BSP directory environment variable. Navigator uses the Windows environment variable **`NAVBSP`** to determine the location on the disk where the BSP package has been installed. The variable should be set as follows:

Variable:	**`NAVBSP`**
Value:	[Navigator BSP base directory] (e.g., **`c:\Pentek\BSP\`**`BSP_2`**`.3`**)

   <u>NOTE:</u> This environment variable is created automatically during installation and should not have to be set manually.

3) Add the Navigator BSP directory search path to the Windows PATH environment variable: **`%NAVBSP%\lib`**.

   The Path variable is found in the User Variables section of System Properties on a Windows System. PATH is modified during this installation. It should be the first element in PATH following this installation, moving the driver path set in Section 3.3 to the second element.

   <u>NOTE:</u> The PATH environment variable is modified automatically during installation and should not have to be set manually.

3.3.3 Step 3: Install the Navigator Example Programs

Install the Navigator example programs package from the distribution CD provided using the **`setup.exe`** program on the CD. DOS commands are used for illustration.

```
cd <CDROM_drive>:\4815\71nnn_X.Y.dir
```

(where **`71nnn`** is the model number and **`X.Y`** is the version number)

```
run setup.exe
```

3.3 Windows Installation Procedure (continued)

3.3.4 Step 4 : Install the LabVIEW Runtime Engine

Install the LabVIEW Runtime Engine from the distribution CD. DOS com– mands are used for illustration.

```
cd <CDROM_drive>:\LabView

run LVRTE2015_f3Patchstd.exe
```

3.3.5 Step 5: Windows Hardware Initialization

After you have installed the Pentek Navigator package in accordance with Sections 3.3, 3.3, and you can boot your host Windows system. When Win– dows first starts up after installation of a Jade board, you may see a set of screens from the Windows New Hardware Wizard.

If the New Hardware Wizard starts, follow the steps below to respond to it.

1) First screen: **Welcome to the Found New Hardware Wizard**

   This is the first screen displayed. The New Hardware Wizard must search for Windows software components needed to initialize the new Jade board. This screen gives three options.

   Select **Yes, this time only** and click on **Next**.

2) Next screen: **The wizard helps you install software for:**
   PENTEK 71xxx Multi–Channel Transceiver

   This step installs Windows software elements found in the search from the prior screen. Since you have already installed the Pentek Navigator software from the distribution CD, do not insert the Pentek installation CD for this step. This screen gives two options.

   Select **Install the software automatically** and click on **Next**.

3) Next screen: **Please wait while the wizard searches...**

   This screen requires no response. Just wait until the operation completes, as indicated by the following screen.

4) Last screen: **Completing the Found New Hardware Wizard**

   This screen indicates completion of the New Hardware Wizard—just click on **Finish**.

3.3 Windows Installation Procedure (continued)

3.3.5 Step 5: Windows Hardware Initialization (continued)

You may verify installation of the required Windows software as follows:

a) Select Windows **Start** –> **Control Panel**.

b) In the **Control Panel** window click on **System** and **Security**.

c) In the next panel, click on Device Manager under **System**.

d) The **Device Manager** window lists the devices installed in your Windows system. Click on the pointer symbol to the left of the **System devices** line to expand the system devices, including any Pentek devices.

3.3.6 How to Manually Install and Uninstall the Navigator BSP Device Driver

The Windows install program automatically configures the Windows device drivers for the Pentek Jade board. The DOS procedures in Sections 3.3 and 3.3 are provided for you to use only if configuration is not possible using the `setup.exe` program provided on the Pentek distribution CD.

The following environment variable and search path is set during Navigator driver installation.

<u>**Device Driver Environment Variable**</u>

`NAVBSP_DRVR` – Navigator driver base directory
(e.g., `C:\Pentek\BSP\Driver_12.60`)

This environment variable is created during driver installation.

<u>**Device Driver Search Path**</u>

`%NAVBSP_DRVR%`

The path variable is found in the User Variables section of System Properties on a Windows system. Path is modified during driver installation. It should be the second element in PATH, following the BSP library path.

3.3 Windows Installation Procedure (continued)

3.3.6 How to Manually Install and Uninstall the Navigator BSP Device Driver (continued)

3.3.6.1 Manually Installing the Driver

These are the steps performed by the Windows install program when installing the Navigator driver installation package.

1) **<u>For Windows 10</u>**, in a DOS command window, change to the following directory:

```
cd %NAVBSP_DRVR%\win64_10
```

<u>For earlier versions of Windows</u>, in a DOS command window, change to the following directory:

```
cd %NAVBSP_DRVR%\win64
```

<u>NOTE:</u> The **NAVBSP_DRVR** environment variable is set during distribution installation. You can verify it from the DOS command prompt by typing **echo %NAVBSP_DRVR%** and pressing the **ENTER** key.

2) Execute the following commands:

```
wdreg -inf windrvr1260.inf install

wdreg -inf nav718x.inf install

copy KP_718x.sys c:\windows\system32\drivers

wdreg -name KP_718x install
```

3.3 Windows Installation Procedure (continued)

3.3.6 How to Manually Install and Uninstall the Navigator BSP Device Driver (continued)

3.3.6.2 Manually Uninstalling the Driver

These steps are already performed by the Windows uninstall pro– gram when uninstalling the Navigator driver installation package.

1) **For Windows 10**, in a DOS command window, change to the following directory:

```
cd %NAVBSP_DRVR%\win64_10
```

For earlier versions of Windows, in a DOS command window, change to the following directory:

```
cd %NAVBSP_DRVR%\win64
```

NOTE: The **NAVBSP_DRVR** environment variable is set during distribution installation. You can verify it from the DOS command prompt by typing **echo %NAVBSP_DRVR%** and pressing the **ENTER** key.

2) Execute the following commands:

```
wdreg -name KP_718X uninstall
wdreg -inf nav718x.inf uninstall
wdreg -inf windrvr1260.inf uninstall
del c:\windows\system32\drivers\KP_718X.sys
```

3.4 Linux Installation Procedures

Navigator BSP for Linux comes equipped with an interactive script that automates the process of installing the BSP on a Linux host. This script currently supports the Linux distributions listed in Section 1.3.2. The installer may also work on newer versions of the supported distributions or on other distributions derived from them but it has not been tested. In addition, the dependency of the driver package on Linux kernel ver– sions may cause incompatibility. Installing a compatible kernel should solve such problems.

The following sections describe how to use the interactive installer and also how to manually install the Navigator BSP:

- Section 3.4.1 – Step 1: Prepare for the Installation
- Section 3.4.2 – Step 2: Install the Navigator Driver
- Section 3.4.3 – Step 3: Install the Navigator Board Support Package
- Section 3.4.4 – Step 4: Install the Navigator Example Programs
- Section 3.4.5 – Removing or Upgrading the Installed Packages
- Section 3.4.6 – How to Manually Install the Navigator Packages
- Section 3.4.7 – Installing the Driver for ReadyFlow Alongside the Driver for Navigator

3.4 Linux Installation Procedures (continued)

3.4.1 Step 1: Prepare for the Installation

To access the Navigator installer present on the CD, you will need to mount it. Most Linux distributions will automatically mount an optical disc when it is inserted into the computer. A disc also can be mounted manually. Depending on the platform, superuser permissions may be required.

The commands shown here assume that the optical disc drive is **/dev/cdrom**. Open a terminal program and mount the Navigator distribution CD to a location of your choice (for example, **/mnt/cdrom**) and change to that directory:

```
$ mkdir /mnt/cdrom
$ mount -t iso9660 /dev/cdrom /mnt/cdrom
$ cd /mnt/cdrom
```

Execute the interactive script to begin installation:

```
$ sh ./INSTALL.sh
```

This interactive script will try to install the three components of Navigator: device driver, BSP library, and example programs. For each component, you are given four choices, one of which must be selected by entering an associated letter at the prompt:

- **Install** [**i**] : Perform the full installation, including setting up the environment variables and init scripts. A superuser password is required for this option.
- **Extract** [**e**] : Just extract the archive to the chosen location. Manual steps will be necessary afterwards.
- **Uninstall** [**u**] : Completely remove the package, including the environment variables and init scripts. A superuser password is required for this option.
- **Skip** [**s**] : Skip the component altogether and move on.

The installer will first print introductory messages to the console identifying the included Navigator package versions and the system information. The installer can identify several popular Linux distributions as noted above. However, it may happen that a particular distribution cannot be identified. In such a case, the installer will show a list of supported distributions and require you to select the one that is the closest match to your system.

3.4 Linux Installation Procedures (continued)

3.4.2 Step 2: Install the Navigator Driver

Once the distribution has been identified, the installer will ask which action is required for the Navigator driver package. Type in **`i`** and press the **Enter** key to select the **`Install`** option. The text below shows the messages printed by the interactive installer. The last line, in bold text, shows user input.

```
...
Do you wish to Install/Extract/Uninstall/Skip the Navigator Driver
package? [i/e/u/s]?
i
```

Next, type the path where you wish to install the Navigator Driver. By default, packages are installed within the directory in the user's home direc– tory that contains the Navigator Design Suite. If the default path shown on your screen is acceptable, just press the **Enter** key.

```
Please specify the full path where Navigator Device Driver will be
installed (e.g. /home/username/Pentek/BSP)
A new directory for the driver will be created within this location.
Press return to use default location (i.e. /home/johndoe/Pentek/BSP)
```

To successfully build and insert modules in the kernel, you must have access to the Linux kernel source or at least its header files. These source or header files must be the ones used to build the kernel in which the module is to be used. The interactive installer will attempt to locate the headers for the run– ning kernel. If the headers cannot be located, you will be asked to provide the full path to them.

Superuser privileges are necessary to complete the driver installation. When prompted, provide password for the **`root`** user account.

```
...
Please enter the 'root' account password when prompted.
Password: <yourpassword>
```

The installer will then proceed to extract and build the driver package at the specified location.

3.4 Linux Installation Procedures (continued)

3.4.2 Step 2: Install the Navigator Driver (continued)

In the final phase, the installer will set up a helper script (**`pentek-navdriver`**) to manage driver modules and then associate it with the **`init`** system to auto–matically load Pentek device driver modules at system startup. Two init sys–tems are supported: **`systemd`** on all distributions, and **`SysV`** init on Debian and RedHat families.

The helper script is installed at the location **`/usr/local/bin/pentek-navdriver`**. This script can be used to load or unload the driver modules on demand.

Usage:

```
sudo pentek-navdriver {start|stop|status}
```

Another script named **`pentek-nv-env.sh`** is created or updated to load the environment variables necessary to link programs with the device driver. It is located at:

`/usr/local/bin/pentek-nv-env.sh`

This script initializes the **`NAVBSP_DRVR`** variable with the path to Navigator driver installation. It also updates the **`PATH`** and **`LD_LIBRARY_PATH`** environment variables. A symbolic link to this helper script is created as

`/etc/profile.d/pentek-nv-env.sh`

Depending on the distribution and desktop environment, this will provide the environment variables to all programs.

The global **`bashrc`** file is modified to invoke the **`pentek-nv-env.sh`**, ensuring that all instances of the Bash interactive shell have access to the necessary variables. If you are using other shells, modify the respective files to obtain a working environment.

The installer also updates the **`/etc/security/limits.conf`** file to raise the resource limit on POSIX message queues. This is necessary for smooth opera–tion of the DMA callback mechanism in the BSP. Consult Appendix C for more information.

If there is an error at any stage, the installer will print a message at the point of failure.

3.4 Linux Installation Procedures (continued)

3.4.3 Step 3: Install the Navigator Board Support Package

After the installation for the driver package finishes successfully, or if it is skipped entirely, the installer will move on to the Navigator BSP and ask which action is required. Type in **i** and press the **Enter** key to select the **Install** option.

```
...
Do you wish to Install/Extract/Uninstall/Skip the Navigator BSP
package? [i/e/u/s]?
i
```

Next, type the path where you wish to install the Navigator BSP. If the default path is acceptable, just press the **Enter** key.

```
Please specify the full path where Navigator BSP will be installed
(e.g. /home/username/Pentek/BSP)
A new directory for the BSP will be created within this location.
Press return to use default location (i.e. /home/johndoe/Pentek/BSP)
```

Superuser privileges are necessary to complete the BSP installation. When prompted, provide the password for the **root** user account.

```
...
Please enter the 'root' account password when prompted.
Password: <yourpassword>
```

The installer will then proceed to extract and build the BSP at the specified location. The driver package must already be installed in order for the build to succeed.

Similar to the driver installation, the script named **pentek-nv-env.sh** is created or updated to load the environment variables necessary to compile and link programs with the BSP libraries. It is located at

/usr/local/bin/pentek-nv-env.sh

This script initializes the **NAVBSP** variable with the path to the Navigator BSP installation. It also updates the **LD_LIBRARY_PATH** environment variable.

3.4 Linux Installation Procedures (continued)

3.4.3 Step 3: Install the Navigator Board Support Package (continued)

A symbolic link to this helper script is created as

`/etc/profile.d/pentek-nv-env.sh`

Depending on the distribution and desktop environment, this will provide the environment variables to all programs.

The global **`bashrc`** file is also modified to invoke the **`pentek-nv-env.sh`** to ensure that all instances of the Bash interactive shell have access to the neces– sary variables. If you are using other shells, modify the respective files to obtain a working environment.

3.4.4 Step 4: Install the Navigator Example Programs

After the installation of the driver and BSP packages finishes successfully, or if they are skipped, the installer will move on to the Navigator examples pack– age for a particular Jade board and ask what action is required. Type in **`i`** and press the **Enter** key to select the **`Install`** option.

```
...
Do you wish to Install/Extract/Uninstall/Skip the 71861 Examples
package? [i/e/u/s]?
i
```

Next, type the path where you wish to install the Navigator examples pack– age. If the default path is acceptable, just press the **Enter** key.

```
Please specify the full path where Navigator BSP samples for 71861
will be installed (e.g. /home/username/Pentek/BSP)
A new directory for the Board-specific samples will be created
within this location.
Or press return to use default location (i.e. /home/johndoe/Pentek/
BSP)
```

The installer will then proceed to extract and build the board–specific exam– ple programs at the specified location. Both the driver and BSP packages must already be installed in order for the build to complete successfully.

3.4 Linux Installation Procedures (continued)

3.4.5 Removing or Upgrading the Installed Packages

If you wish to remove packages installed through the interactive script, you just need to re–run the script and select the **`Uninstall`** option by typing **`u`** when prompted. Uninstalling one package leaves the other packages intact.

Make sure you use the same version of the script for both installation and un–installation. We recommend that you remove the old package using the old installer before upgrading to a new version.

3.4.6 How to Manually Install the Navigator Packages

If the interactive installation fails for some reason, the Navigator packages can be installed manually.

3.4.6.1 Extract the Driver Package from the Distribution Disc

Create a directory to contain the Navigator Design Suite and change to that directory:

```
$ mkdir ~/Pentek ~/Pentek/BSP
$ cd ~/Pentek/BSP
```

Extract the Navigator driver package from the distribution CD:

`tar -xvf /mnt/cdrom/NavDriver/Driver_<VERSION>.tar`

where **`<VERSION>`** will be 12.50, 12.60, 13.00, etc. and **`/mnt/cdrom`** is the mount point for the optical disc drive.

We will be using the driver version 12.60 for the example com–mands below:

```
$ tar -xvf /mnt/cdrom/NavDriver/Driver_12.60.tar
```

3.4 Linux Installation Procedures (continued)

3.4.6 How to Manually Install the Navigator Packages (continued)

3.4.6.2 Set up the Environment Variables for the Driver Package

Define the Navigator driver directory environment variable. The Navigator driver uses the environment variable **NAVBSP_DRVR** to determine the location on the disk where the package has been installed.

- If using **tcsh** or **csh**, enter in the shell:

```
setenv NAVBSP_DRVR /home/user/Pentek/BSP/Driver_12.60
```

- If using **bash**, **sh**, or **ksh**, enter in the shell:

```
export NAVBSP_DRVR=/home/user/Pentek/BSP/Driver_12.60
```

Add the Navigator Driver library directory to the **LD_LIBRARY_PATH** environment variable:

- If using **tcsh** or **csh**, enter in the shell:

```
LD_LIBRARY_PATH $LD_LIBRARY_PATH:$NAVBSP_DRVR/kplugin
```

- If using **bash**, **sh**, or **ksh**, enter in the shell:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$NAVBSP_DRVR/kplugin
```

3.4.6.3 Build and Install the Driver Package

In order to build the driver modules, header files for the running kernel must be installed. Depending on the distribution and its installation, these files may or may not get installed automatically. To install the header files for the kernel supplied by your distribution, run one of the following commands as root (note that the ` symbols in the commands are backticks and not apostrophes):

- For Debian and Ubuntu:

```
apt-get install linux-headers-`uname -r`
```

- For Fedora, RedHat, and CentOS:

```
yum install kernel-headers-`uname -r` kernel-devel-`uname -r`
```

- For other distributions, consult their official documentation.

3.4 Linux Installation Procedures (continued)

3.4.6 How to Manually Install the Navigator Packages (continued)

3.4.6.3 Build and Install the Driver Package (continued)

The driver package uses the standard Unix install process based on Autotools. Log in as **`root`** and change to the **`redist`** directory in the extracted package. Then execute the configure script and Makefile as shown below:

```
$ su
# cd $NAVBSP_DRVR/redist
# ./configure --disable-usb-support --enable-kbuild --
with-kernel-source=/lib/modules/`uname -r`/build
# make
```

In order to run example programs as a regular user, execute the following command:

```
# chmod 666 /dev/windrvrXXXX
```

where **`XXXX`** is the first four digits of the driver version without the dot character (i.e., **`1260`**).

To verify that the drivers are installed, execute the following com–mand:

```
$ lsmod | grep KP
```

Output should look similar to the following:

```
KP_718X            49508      0
windrvr1260     28765  1      KP_718X
```

3.4 Linux Installation Procedures (continued)

3.4.6 How to Manually Install the Navigator Packages (continued)

3.4.6.3 Build and Install the Driver Package (continued)

To automatically load the driver on each boot, add the following two lines to **/etc/rc.local** or any other init script (change the driver version as appropriate):

```
/home/user/Pentek/BSP/Driver_12.60/redist/wdreg windrvr1260 auto
chmod 666 /dev/windrvr1260
modprobe KP_718X
```

These commands can also be executed as root in the shell after system startup.

Navigator BSP has built–in support for managing DMA opera–tions in parallel threads (see Appendix C for more information). For smooth execution in all scenarios, you must increase the resource limits imposed by Linux.

To raise the default limit on the maximum memory allowed for use by POSIX message queues, copy the following two lines to your **/etc/security/limits.conf** file:

```
*           hard    msgqueue        4915200
*           soft    msgqueue        4915200
```

To increase the maximum depth of message queues, execute the following command as root on system startup (or place this line in an init script like **/etc/rc.local**):

```
echo 2048 > /proc/sys/fs/mqueue/msg_max
```

3.4 Linux Installation Procedures (continued)

3.4.6 How to Manually Install the Navigator Packages (continued)

3.4.6.4 Extract the BSP Package from the Distribution Disc

If not already done, create a directory to contain the Navigator Design Suite and change to that directory:

```
$ mkdir ~/Pentek ~/Pentek/BSP
$ cd ~/Pentek/BSP
```

Extract the Navigator BSP package from the distribution CD:

```
tar -xvf /mnt/cdrom/NavBSP/BSP_<VERSION>.tar
```

where **`<VERSION>`** will be 1.0, 1.1, 2.0, etc., and **`/mnt/cdrom`** is the mount point for the optical disc drive. We will be using the BSP version 1.0 for the example commands below:

```
$ tar -xvf /mnt/cdrom/NavBSP/BSP_1.0.tar
```

3.4.6.5 Set up the Environment Variables for the BSP Package

Define the Navigator BSP directory environment variable. Navi–gator BSP uses the environment variable **`NAVBSP`** to determine the location on the disk where the package has been installed.

- If using **`tcsh`** or **`csh`**, enter in the shell:

```
setenv NAVBSP /home/user/Pentek/BSP/BSP_1.0
```

- If using **`bash`**, **`sh`**, or **`ksh`**, enter in the shell:

```
export NAVBSP=/home/user/Pentek/BSP/BSP_1.0
```

3.4 Linux Installation Procedures (continued)

3.4.6 How to Manually Install the Navigator Packages (continued)

3.4.6.5 Set up the Environment Variables for the BSP Package (continued)

Add the Navigator driver library directory to the **LD_LIBRARY_PATH** environment variable:

- If using **tcsh** or **csh**, enter in the shell:

  ```
  setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$NAVBSP/lib
  ```

- If using **bash**, **sh**, or **ksh**, enter in the shell:

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$NAVBSP/lib
  ```

3.4.6.6 Extract the Example Programs from the Distribution Disc

If not already done, create a directory to contain the Navigator Design Suite and change to that directory:

```
$ mkdir ~/Pentek ~/Pentek/BSP
$ cd ~/Pentek/BSP
```

Extract the Navigator examples package for your board from the distribution CD:

```
tar -xvf /mnt/cdrom/4814/<MODEL><VERSION>.tar
```

where **<MODEL>** will be 71861, 71841, etc., **<VERSION>** will be the BSP version without the dot character (like 10, 11, 20, etc.), and **/mnt/cdrom** is the mount point for the optical disc drive. As an example, we will use the model 71861 package version 1.0 (such that **<VERSION>** is 10) for the example command below:

```
$ tar -xvf /mnt/cdrom/4814/7186110.tar
```

No environment variables need to be created to point to the examples package. To build and run the Navigator examples or any user application based on the Navigator BSP, we need the environment variables set up in Sections 3.4.6.2 and 3.4.6.5.

3.4 Linux Installation Procedures (continued)

3.4.7 Installing the Driver for ReadyFlow Alongside the Driver for Navigator

Your situation may require you to use both Pentek's ReadyFlow and Navigator Board Support Packages on the same machine. To maintain compatibility with the kernel, it may be necessary to use the latest drivers for ReadyFlow and Navigator.

Driver packages supplied by Pentek are built upon Jungo's WinDriver framework. It is possible to install an older version and a newer version of WinDriver on the same machine. For example, a system with Linux kernel v3.10 can have WinDriver 11.80 for ReadyFlow as well as WinDriver 12.60 for Navigator (because Navigator does not work with lower versions of WinDriver).

Now if the aforementioned system needs to be upgraded with Linux kernel 4.6, the WinDriver version for ReadyFlow must also be upgraded because v11.80 is not compatible with Linux 4.6. Users facing this issue may request an upgraded driver package for ReadyFlow that is compatible with newer kernels.

Such a package must be installed separately in accordance with the steps outlined in the User's Guide for ReadyFlow. This package may use the same WinDriver version as its base as that of the Navigator driver. Since Pentek kernel modules depend on Jungo's WinDriver modules, all of them must be inserted or removed together.

The helper script placed by Navigator interactive installer at the location **`/usr/local/bin/pentek-navdriver`** has a variable within it that can be used to enable support for the legacy ReadyFlow kernel modules.

Edit that script as root and set:

```
LEGACY_SUPPORT=1
```

Once that is done, executing **`pentek-navdriver {start|stop}`** will insert/remove all Pentek and Jungo kernel modules together. This will be necessary only if both ReadyFlow and Navigator driver packages use the same WinDriver version as their base.

Chapter 4: Navigator BSP Files

4.1 Navigator BSP Directory Structure

The directory structure shown below is created under the **\Pentek\BSP\BSP_x.y** directory (where **x.y** is the Navigator BSP version number). **71nnn** is the Jade board model number.

71nnn	**examples**	Base directory for all sample programs. A **readme** file describes the examples and how to run them. Each sample program has its own directory.	
71nnn	**scripts**	MATLAB scripts for processing program output	
BSP	**bin**	BSP binaries, such as Signal Viewer	
BSP	**docs**	BSP documentation files	
BSP	**filters**	FIR filter files	
BSP	**include**	Include files for API layer libraries	
BSP	**include**	**[nnn]_include**	Include files for board layer board–specific libraries, where **nnn** in the directory is the last three digits of the board's family model number. For example, the board–specific directory for Model 71861 is **860_include**.
BSP	**include**	**dev_include**	Include files for hardware layer device libraries
BSP	**include**	**ip_include**	Include files for hardware layer IP core libraries
BSP	**include**	**util_include**	Include files for board layer utility libraries
BSP	**source**	Source code for the high–level interface	
BSP	**source**	**[nnn]_source**	Source code for board layer board–specific libraries, where **nnn** in the directory is the last three digits of the board's family model number. For example, the board–specific directory for Model 71861 is **860_source**.
BSP	**source**	**dev_source**	Source code for hardware layer device libraries
BSP	**source**	**ip_source**	Source code for hardware layer IP core libraries
BSP	**source**	**util_source**	Source code for board layer utility libraries
BSP	**lib**	Pre–compiled Navigator libraries and build tools	
BSP	**util**	Utility programs	
Driver	Support files for the device driver		

4.2 How the Navigator BSP API Reference is Organized

This section describes how the *Navigator BSP API Reference Guide*, is organized. The *API Reference Guide* describes the source and include files in the BSP. It is provided in HTML and PDF formats (**`API_Reference.html`** and **`API_Reference.pdf`**) in the following location:

Windows: **`C:\Pentek\BSP\BSP_X.Y\docs`** (where **`X.Y`** is the version number)
or
`%NAVBSP%\docs`

Linux: **`/home/username/Pentek/BSP/BSP_X.Y/docs`** (where **`X.Y`** is the version number)
or
`$NAVBSP/docs`

The *Navigator API Reference Guide* consists of three main parts:

- ❑ **Main Page** (HTML) or **Overview** (PDF) – Provides software release information.
- ❑ **Data Structures** (the PDF divides this into two chapters: data structure index and data structure documentation) – Provides information about the data structures. The data structures are listed alphabetically. The listing for each data structure names the **header (include) file** that contains it.
- ❑ **Files** (the PDF divides this into two chapters: file index and file documentation) – Provides information about what is in all the source and include files. The **Files** are grouped as follows:
    - **General** include and source files – All the BSP header files are in the **`include`** folder and all the BSP source files are in the **`source`** folder. There are three sub–folders in each of these folders:
        - **Board–specific** (e.g., **`860_include`** or **`860_source`**, where **`860`** identifies a specific board) – Specific to a Jade board family. A board family includes a specific XMC module plus boards with that module on various carriers**.**
        - **Device–specific** (**`dev_include`** or **`dev_source`**) **–** Specific to a device on the module that is outside the FPGA such as the Texas Instruments CDCM7005 clock synchronizer or the Silicon Laboratories Si571 VCXO.
        - **IP core module** files (**`ip_include`** or **`ip_source`**) – Specific to an IP core module.
        - **Utility** – Operating system routines, for file I/O, printing, command line parsing, etc. The utility library also contains files that support the Signal Viewer (described in Appendix B).

4.3 BSP Files and IP Core Files Listed by Topic

The **`navNNN_`** prefix indicates a board–specific file.

Topic	Navigator BSP Files
ADC	navNNN_adc.c navNNN_adc.h nav_adc.c nav_adc.h
API	nav_api.c nav_api.h
Board	navNNN_board_info.c navNNN_board_info.h nav_board_info.c nav_board_info.h
Clock	navNNN_cdcm.c navNNN_cdcm.h navNNN_clocksync.c navNNN_clocksync.h nav_clocksync.c nav_clocksync.h nav_dev_cdcm7005.c nav_dev_cdcm7005.h
Command Line Utility	nav_cmd.c nav_cmd.h
Debug	nav_debug.c nav_debug.h
Decimation	nav_ddc.c nav_ddc.h
Device	nav_dev.h nav_dev_cdcm7005.c nav_dev_cdcm7005.h nav_dev_common.h nav_dev_lm83.c nav_dev_lm83.h nav_dev_lm95234.c nav_dev_lm95234.h nav_dev_ltc2990.c nav_dev_ltc2990.h nav_dev_si571.c nav_dev_si571.h
Digital Downconverter (DDC)	navNNN_ddc.c naNNNv_ddc.h nav_ddc.c nav_ddc.h

Topic	Navigator BSP Files
Direct Memory Access (DMA)	navNNN_dma.c navNNN_dma.h nav_dma.c nav_dma.h nav_dma_common.c nav_dma_common.h nav_dmathread.c nav_dmathread.h
Gate / Trigger	navNNN_gatetrig.c navNNN_gatetrig.h nav_gatetrig.c nav_gatetrig.h
Interrupts	navNNN_intr.c navNNN_intr.h nav_intr.c nav_intr.h nav_ip_general_intr.h
IP	nav_ip_common.h nav_ip_core.h Each Navigator IP core has a corresponding source and include file.
IP Generic Interrupt	nav_ip_general_intr.c nav_ip_general_intr.h
Memory Map	navNNN_mem_map.h
Operating System	nav_os.h nav_sys.h nav_sys.c
Power Meter	navNNN_pwr_meter.c navNNN_pwr_meter.h nav_pwr_meter.c nav_pwr_meter.h
Signal Viewer	nav_view.c nav_view.h
Test Signal	navNNN_testsig_gen.c navNNN_testsig_gen.h nav_testsig_gen.c nav_testsig_gen.h

NOTE: Refer to the *Navigator BSP API Reference Guide* for a complete list of files. (see Section 1.2).

Chapter 5: Building Navigator BSP Libraries

5.1 Introduction

This chapter provides procedures for building the Navigator BSP libraries:

❑ Section 5.2 – Windows Procedures

❑ Section 5.3 – Linux Procedures

5.2 Windows Procedures

The Navigator source files are located in the **BSP\source** and **BSP\include** directories. The **BSP\lib** directory contains Microsoft Visual Studio project files to build the Navi–gator Libraries. All libraries are built using Microsoft Visual Studio 2015.

5.2.1 Navigator Libraries

The Navigator BSP employs multiple dynamic link libraries (DLLs). They support a layered architecture consisting of three software layers: API layer, board layer, and hardware layer.

- **NavBSP_API -** API layer DLL. Contains general routines to program and control a Jade board. These routines call routines in board–layer DLLs.
- **NavBSP_nnn** (where **nnn** = board model) – Board layer board–specific DLL. Calls routines in the board layer utility DLL and the hardware layer DLLs.
- **NavBSP_Util -** Board layer general–purpose DLL. Calls routines in the hardware layer DLLs.
- **NavBSP_Dev -** Hardware–layer DLL. Contains routines to interface to Pentek board hardware devices other that the FPGA. Calls routines in the hardware layer IP DLL.
- **NavBSP_IP -** Hardware–layer DLL. Contains routines to interface to IP modules in the FPGA.

5.2 Windows Procedures (continued)

5.2.2 Building Libraries Using Msbuild

A solution file, **`NavBSP.sln`**, is provided to build all libraries. Project files are provided to build the individual libraries. Project file names use the library names (see Section 6.2). For example, the IP core library project filename is **`NavBSP_ip.vcxproj`**. These files are found in the **`lib`** directory. Compiling with **`msbuild.exe`**, provided in the Visual Studio package, assumes the Navigator **`NAV_BSP_DRVR`** environment variable is properly set.

To build all libraries, call **`msbuild`** with the following syntax:

```
msbuild NavBSP.sln /p:Configuration=Debug
```

To build individual libraries, call **`msbuild`** with the following syntax (where **`library`** is the desired library):

```
msbuild NavBSP.sln /t:library /p:configuration=Debug
```

or

```
msbuild NavBSP.sln /t:library /p:configuration=Release
```

For example, to build the IP core library for release:

```
msbuild NavBSP.sln /t:NavBSP_IP /p:configuration=Release
```

5.2.3 Building Libraries Using Microsoft Visual Studio 2015

Microsoft Visual Studio files are provided for building the Navigator library. The **`BSP\lib`** directory contains one solution file (**`NavBSP.sln`**) and several proj–ect files (**`*.vcxproj`**) as listed in Section 5.2.1.

5.2.3.1 Loading the Project

1) Start Microsoft Visual Studio.

2) Click on **File** on the menu bar.

3) Click on **Project/Solution** in the pull–down menu.

4) Browse to the library directory
(e.g., **`c:\Pentek\BSP\BSP_X.Y\lib`**)
(where **`X.Y`** is the Navigator BSP version number).

5) Double–click on the **`NavBSP.sln`** solution file.

6) The selected project is automatically loaded and ready to be built.

5.2 Windows Procedures (continued)

5.2.3 Building Libraries Using Microsoft Visual Studio 2015 (continued)

5.2.3.2 Building the Project

Once the selected project is loaded (as described above) the project can be built as follows:

1) Click on **Build** and then select **Batch Build**.

2) Select one of the following:

   `NavBSP_API\Debug`
   `NavBSP_API\Release`
   `NavBSP_nnn\Debug` (where **`nnn`** = board model)
   `NavBSP_nnn\Release` (where **`nnn`** = board model)
   `NavBSP_Util\Debug`
   `NavBSP_Util\Release`
   `NavBSP_Dev\Debug`
   `NavBSP_Dev\Release`
   `NavBSP_IP\Debug`
   `NavBSP_IP\Release`

3) Click on **Build**.

4) A Build Output window will open, showing the build process. When build completes, the following message is displayed:

```
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
```

NOTE: If you had selected both **`Debug`** and **`Release`** in step 2, the number succeeded should be **`2`**.

5.3 Linux Procedures

The Navigator source files are located in the **BSP/source** and **BSP/include** directories. The **BSP/lib** directory contains a **Makefile** to build the Navigator Libraries.

5.3.1 Navigator Libraries

The Navigator BSP employs multiple dynamic link libraries (shared objects). They support a layered architecture consisting of three software layers: API layer, board layer, and hardware layer.

- **libnav.so -** Monolithic shared library containing all the routines available in Navigator BSP.
- **libnavapi.so -** API layer shared library. Contains general routines to program and control a Pentek Jade board. These routines call routines Board layer shared libraries.
- **libnavutl.so -** API layer general purpose shared library. Calls routines in the Board or Hardware layer libraries.
- **libnavNNN.so -** (where **NNN** = board model) Board layer library for a specific Jade model. Calls routines in the API layer utility library and the Hardware layer libraries.
- **libnavdev.so -** Hardware layer shared library. Contains routines to interface to Pentek board hardware devices other than the FPGA. Calls routines in the Hardware Layer IP library.
- **libnavip.so -** Hardware layer shared library. Contains routines to interface to IP–cores in the FPGA.

Each of these shared libraries has a corresponding static library with a **.lib** extension. The Navigator examples use the shared libraries by default but the user applications can be compiled against either shared or static libraries.

5.3.2 Building the Libraries

The **lib** directory (located below the BSP root directory) contains a **Makefile** that is used to build the libraries and also to generate documentation from the source code through Doxygen. The Navigator source and header files are located in the **source** and **include** directories located below the BSP root directory. These top–level directories contain the API–level files and are used to build the API libraries. They also contain sub–directories that have files for other libraries.

The **Makefile** has several invocation targets. Executing **make** or **make all** will compile the sources to create libraries. Refer the documentation within the **Makefile** for a detailed explanation of all available targets.

5.3 Linux Procedures (continued)

5.3.3 Building Libraries Using Eclipse IDE for C/C++

Navigator BSP for Linux systems includes project files for building and de–bugging the BSP library in Eclipse IDE. The Eclipse project relies on `Makefile` (mentioned in Section 5.3.2) to build the libraries. However, the IDE provides a superior experience while editing, building, and debugging the source code.

The relevant project description files (`.project`, `.cproject`) are present in the BSP root directory. To be able to successfully import this project into Eclipse, the CDT (C/C++ Development Tooling) plugin must be installed in the base IDE. Eclipse Foundation also provides customized versions of the base Eclipse IDE which focus on certain classes of users.

Go to http://www.eclipse.org/downloads/packages/ and download and install the latest **Eclipse IDE for C/C++ Developers** package to quickly get started with development.

5.3.3.1 Loading the Project

1) Start **Eclipse IDE for C/C++**. Make sure that the Eclipse graphical environment has access to the `NAVBSP`, `NAVBSP_DRVR`, and `LD_LIBRARY_PATH` environment variables. An easy way to accomplish this is to export those variables from a terminal window and then launch the Eclipse binary from there. These commands can also be bundled into a script for ease of use.

2) Create or open the Workspace for Navigator. We recommend using a dedicated workspace for working on the Navigator library and associated applications.

3) Click on **File** on the menu bar.

4) Click on **Import...** in the pull–down menu.

5) A popup window will be shown to choose the Import Wizard. Expand **General** and select **Existing Projects into Work–space**. Click on **Next**.

6) On the next screen, check the radio button that says **Select the root directory**. Click on **Browse...** and navigate to the folder containing Navigator BSP installation. Click **OK**.

7) A project named `NavigatorBSP` should be visible in the **Projects:** list box. Make sure the checkbox next to the project is selected. Click **Finish**.

The NavigatorBSP project should now be visible in the **Project Explorer** view.

5.3 Linux Procedures (continued)

5.3.3 Building Libraries Using Eclipse IDE for C/C++ (continued)

5.3.3.2 Building the Project

After loading the project as described above, the BSP libraries can be built via Eclipse as follows:

1) Make sure that **NavigatorBSP** is selected in the **Project Explorer**.

2) Click on **Project** on the menu bar.

3) Click on **Build** all in the pull–down menu.

This invokes the target `all` inside `lib/Makefile`.

A specific target from the `Makefile` can be invoked as follows:

1) Make sure that **NavigatorBSP** is selected in the **Project Explorer**.

2) Click on Project on the menu bar.

3) Click on **Build Targets** in the pull–down menu and select **Build…** from the sub–menu.

4) In the **Build Targets** popup window, choose a desired target and click **Build**.

Check the **Console** view to see the output from **CDT Builder**. The **Problems** view will also aggregate all the warnings and errors from the compiler.

Chapter 6: Building Navigator BSP Example Programs

6.1 Introduction

This chapter provides procedures for building the Navigator BSP libraries:

❑ Section 6.2 – Windows Procedures

❑ Section 6.3 – Linux Procedures

6.2 Windows Procedures

Example programs are provided for Pentek Jade boards. All libraries and programs are built using Microsoft Visual Studio Professional 2015. Microsoft Visual Studio files are provided for building each example program. The **`71nnn\examples`** directory contains one solution file (**`*.sln`**) and one project file (**`*.vcxproj`**) for each example program. A **`readme.txt`** file describes the example programs. It is recommended that the solution file (file extension **`.sln`**) be loaded first.

<u>NOTE:</u> Pentek provides pre–built libraries and executable example programs. The "release" versions of the examples can be run directly from the system command prompt. When initially running the "debug" versions under Microsoft Visual Studio, a warning will be issued:

This project is out of date:
Would you like to build it?

This is because Pentek does not provide object files. Pentek does provide debug database files, so the programs can be executed without building them. Attempting to debug any of the library routines will also cause warning mes–sages.

To avoid any warning messages, first rebuild the library, then rebuild the example program.

6.2.1 Building Example Programs Using Msbuild

Solution and project files are provided for each example program. For exam–ple, the **`show_info`** example has **`show_info.sln`** and **`show_info.vcxproj`**. These files are found in the example program directory. Compiling with **`msbuild.exe`**, provided in the Visual Studio package, assumes the Navigator **`NAVBSP`** and **`NAV_BSP_DRVR`** environment variables are properly set.

6.2 Windows Procedures (continued)

6.2.1 Building Example Programs Using Msbuild (continued)

To build the example program from the command line, call **msbuild** with the following syntax (where **example** is the name of the desired example pro– gram):

```
msbuild example.sln /t:example /p:configuration=Debug
```

or

```
msbuild example.sln /t:example /p:configuration=Release
```

For example, to build the **show_info** program for release:

```
msbuild show_info.sln /t:show_info /p:configuration=Release
```

6.2.2 Building Example Programs Using Microsoft Visual Studio

Microsoft Visual Studio files are provided for building each example pro– gram. The **71nnn\examples** directory contains one solution file and one project file for each example program. A **readme.txt** file describes the example pro– grams.

6.2.2.1 Loading a Project

1) Start Microsoft Visual Studio.
2) Click on **File** on the menu bar.
3) Click on **Open Project/Solution** in the pull–down menu.
4) Browse to one of the example directories (e.g., **c:\Pentek\BSP\BSP_X.Y\71nnn\examples\examplename**) (where **X.Y** is the Navigator BSP version number, **71nnn** is the board model number, and **examplename** is the name of the example).
5) Double click on the desired file, either ***.sln** or ***.vcxproj** (for example, **acquire.sln**).
6) The selected project (**acquire,** for example) is automatically loaded and ready to be built.

6.2 Windows Procedures (continued)

6.2.2 Building Example Programs Using Microsoft Visual Studio (continued)

6.2.2.2 Building a Project

Once the selected project (`acquire`, for example) is loaded (as described in Section 6.2.2.1), the project can be built as follows:

1) Click on **Build** and then select **Build acquire.exe.**

2) A Build Output window will open, showing the build process. When the build completes, the following message is displayed:

```
acquire - 0 error(s), 0 warning(s)
```

3) The `exe` file is created in a projects sub–directory (under the `71nnn\examples` directory), `Debug` or `Release`.

   To choose between Debug and Release configurations, click on **Build** and select **Active Configuration**. A popup window will be displayed with choices for either configuration. Select the desired one and click **OK**.

6.2.2.3 Executing an Example Program

To execute the example program, click on **Debug** and either select **Start Debug** to start the Window Debugger, or select **Start with–out Debugging** to just run the example.

__NOTE:__ Starting the debugger will require rebuilding the example.

6.3 Linux Procedures

Board–specific example programs utilizing the Navigator BSP are supplied as part of the Navigator Suite. The **`71nnn\examples`** directory contains a **`Makefile`** to build the example programs.

NOTE: Pentek does not provide pre–built libraries and executable example programs for the Linux platform.

6.3.1 Building the Example Programs from the Command Line

Individual Navigator example programs are placed within sub–directories of the **`71nnn\examples`** directory. Each sub–directory also contains an example–specific **`Makefile`**. This way, individual examples can be built from within their sub–directories and also from the parent directory. All examples can be built at once from the parent directory.

The makefiles have several invocation targets. Executing **`make`** or **`make all`** from the **`71nnn\examples`** directory will build all the example programs for that board. Refer to the documentation within a **`Makefile`** for a detailed explana– tion of all available targets.

6.3.2 Building Example Programs Using Eclipse IDE for C/C++

Navigator Suite for Linux systems includes project files for building and debugging the board–specific example programs in Eclipse IDE. The Eclipse project relies on Makefile (mentioned in Section 6.3.1) to build the executables. However, the IDE provides a superior experience while editing, building, and debugging the source code.

The Navigator examples projects reference the BSP library project to provide a seamless development experience. Step–by–step debugging as well as attaching the debugger to a running application are also possible.

The relevant project description files (**`.project`**, **`.cproject`**) are present in the BSP root directory. To be able to successfully import this project into Eclipse, the CDT (C/C++ Development Tooling) plugin must be installed in the base IDE. Eclipse Foundation also provides customized versions of the base Eclipse IDE which focus on certain classes of users.

Go to http://www.eclipse.org/downloads/packages/ and download and install the latest **Eclipse IDE for C/C++ Developers** package to quickly get started with development.

6.3 Linux Procedures (continued)

6.3.2 Building Example Programs Using Eclipse IDE for C/C++ (continued)

6.3.2.1 Loading the Project

1) Start **Eclipse IDE for C/C++**. Make sure that the Eclipse graphical environment has access to the **`NAVBSP`**, **`NAVBSP_DRVR`**, and **`LD_LIBRARY_PATH`** environment variables. An easy way to accomplish this is to export those variables from a terminal window and then launch the Eclipse executable from there. These commands can also be bundled into a script for ease of use.

2) Create or open the Workspace for Navigator. We recommend using a dedicated workspace for working on the Navigator library and associated applications.

3) Click on **File** on the menu bar.

4) Click on **Import...** in the pull–down menu.

5) A popup window will be shown to choose the Import Wizard. Expand **General** and select **Existing Projects into Workspace**. Click on **Next**.

6) On the next screen, check the radio button that says **Select the root directory**. Click on **Browse...** and navigate to the folder containing Navigator Examples installation for a particular board model. Click **OK**.

7) A project named **`NavigatorExamples_71nnn`** should be visible in the **Projects:** list box. Make sure the checkbox next to the project is selected. Click **Finish**.

The **`NavigatorExamples_71nnn`** project should now be visible in the **Project Explorer** view. To be able to successfully resolve the sym– bols and function names used in the example source files, the **Nav– igatorBSP** project must have been imported in the Workspace.

6.3 Linux Procedures (continued)

6.3.2 Building Example Programs Using Eclipse IDE for C/C++ (continued)

6.3.2.2 Building the Project

After loading the project as described above, the BSP libraries can be built via Eclipse as follows:

1) Make sure that `NavigatorExamples_71nnn` is selected in the **Project Explorer**.
2) Click on **Project** on the menu bar.
3) Click on **Build all** in the pull–down menu.

This invokes the target `all` inside `lib/Makefile`.

A specific target from the Makefile can be invoked as follows:

1) Make sure that `NavigatorExamples_71nnn` is selected in the **Project Explorer**.
2) Click on **Project** on the menu bar.
3) Click on **Build Targets** in the pull–down menu and select **Build...** from the sub–menu.
4) In the **Build Targets** popup window, choose a desired target and click **Build**. New targets can also be added this way.

Check the **Console** view to see the output from **CDT Builder**. The **Problems** view will also aggregate all the warnings and errors from the compiler.

6.3.2.3 Executing an Example Program from the Project

Each example program has an associated Eclipse launch configu– ration for its executable. After the executable has been built, run it as follows:

1) Click on **Run** on the menu bar.
2) Click on **Run Configurations...** in the pull–down menu.
3) In the **Run Configurations** popup window, choose a desired configuration under **C/C++ Application** and click **Run**. New configurations can also be added this way. The launch settings such as command line arguments and working directory can also be changed from here. Consult the Eclipse documentation for details.

6.3 Linux Procedures (continued)

6.3.2 Building Example Programs Using Eclipse IDE for C/C++ (continued)

6.3.2.3 Executing an Example Program from the Project (continued)

Alternatively, you can run the executables directly from the **Proj–ect Explorer**. Expand the **Binaries** tree item in the entry for a par–ticular project and right–click on an executable. Click on **Run As** and select **C/C++ Application**.

To debug the executable, run it as follows:

1) Click on **Run** on the menu bar.

2) Click on **Debug Configurations...** in the pull–down menu.

3) In the **Debug Configurations** popup window, choose a desired configuration under **C/C++ Application** and click **Debug**. New configurations can also be added this way. The launch settings like command line arguments and working directory can also be changed from here. Consult the Eclipse documentation for details.

Alternatively, you can debug the executables directly from the **Project Explorer**. Expand the **Binaries** tree item in the entry for a particular project and right–click on an executable. Click on **Debug** As and select **C/C++ Application**.

To attach the debugger to a running executable:

1) Click on **Run** on the menu bar.

2) Click on **Debug Configurations...** in the pull–down menu.

3) In the **Debug Configurations** popup window, choose a desired configuration from **C/C++ Attach to Application** and click **Debug**. You will then be asked to select a running process on your system. New configurations can also be added from this window. Consult the Eclipse documentation for details.

6.3 Linux Procedures (continued)

6.3.3 Creating an Eclipse Project for Custom Applications

Having an Eclipse project is very useful while working on applications based on the Navigator BSP Library. Users can simply create a new sub–directory within the `71nnn\examples` directory and modify the `Makefile` from another example to work for their program.

However, isolating the user applications into a separate project of their own is safer because the BSP and examples can be upgraded without affecting user applications. The following steps describe how to create a new Eclipse project for an application that relies on Navigator BSP.

1) Start **Eclipse IDE for C/C++**. Make sure that the Eclipse graphical environment has access to the `NAVBSP`, `NAVBSP_DRVR` and `LD_LIBRARY_PATH` environment variables. An easy way to accomplish this is to export those variables from a terminal window and then launch the eclipse binary from there. These commands can also be bundled into a script for ease of use.

2) Create or open the Workspace for Navigator. We recommend using a dedicated workspace for working on Navigator library and associated applications.

3) Click on **File** on the menu bar.

4) Click on **New** and select one of the following:

    - **Makefile Project with Existing Code**

      Choose this option if the source code and `Makefile` already exist or if you plan to add them yourself later on. Makefiles from `71nnn\examples` can be reused for this purpose. This is the recommended choice.

    - **C++ Project or C Project**

      Choose one of these to create a new C++ or C project, respectively, using a configuration wizard. By default, Eclipse will automatically generate Makefiles for these project types. In such a case, advanced configuration must be done to set compiler and linker flags. After creating such a project, change the compiler settings from **Project > Properties > C/C++ Build > Settings**.

6.3 Linux Procedures (continued)

6.3.3 Creating Eclipse Project for Custom Applications (continued)

5) Once a project has been created, it needs to be configured such that the Eclipse C/C++ indexer is able to resolve the various symbols and function names present in the Navigator Library. When these symbols and func– tions are used in the client application, Eclipse will be able to provide code completion and helpful tooltips. An XML file with the required settings is present in the `71nnn\examples` directory.

    a) Go to **Project > Properties > C/C++ General > Paths and Symbols**.

    b) Click on **Import Settings...**

    c) Click on Browse in the popup widow and navigate to the `71nnn\examples` directory.

    d) Select the `EclipseProjectSettings_Exported.xml` file and click **OK**.

    e) Make sure that the correct project is selected in the window and click **Finish**.

    A list of required include locations and symbols is given below in case manual entry becomes necessary.

    <u>Includes:</u>

    - `${NAVBSP}/include`
    - `${NAVBSP_DRVR}`
    - `${NAVBSP_DRVR}/include`
    - `${NAVBSP_DRVR}/kplugin`

    <u>Symbols:</u>

    - `LINUX`
    - `__KERNEL__`

6) Add a reference the **NavigatorBSP** project to make sure dependencies are built first. Go to **Project > Properties > Project References**. Select the **NavigatorBSP** project if it is present and click **OK**.

6.3 Linux Procedures (continued)

6.3.3 Creating Eclipse Project for Custom Applications (continued)

7) Rebuild the index to make sure all tokens are resolved.

   Select **Project > C/C++ Index > Rebuild**.

8) After updating the source files, build the executable as follows:

   Select **Project > Build all**.

Chapter 7: Anatomy of a Typical Application

In this chapter we will examine how a typical user application based on the Navigator BSP is constructed. The API calls available in the BSP library simplify the tasks of set–ting up and operating the hardware resources available in Pentek boards. Additional API calls help in performing utilitarian tasks such as obtaining program arguments, storing/analyzing the signal, or creating register dumps for debugging.

Readers can follow along with the **acquire** example in the Navigator BSP to see code pertaining to the logical sections of a typical application described below. For conve–nience, code snippets are also shown in this document. Bold text in the snippets is used to highlight important tokens.

7.1 Obtain Program Arguments

Most applications are built with a degree of flexibility to cater to different scenarios. The runtime behavior of the application can be changed through arguments supplied either at launch or during execution. The Navigator BSP Command Line Utility was designed to take care of these needs by supporting some common arguments out of the box. This utility also supports reading program arguments stored in a file (**.ini** file). For more information about the command line utility, see Appendix A.

All the high–level API routines in Navigator BSP return a specific status code that can be used to determine the cause of failure, if any. A special array of strings holds a tex–tual description for each status code. Its usage is also shown in the snippet below.

```
int32_t status;
NAV_CMD_PARAMS cmdParams = {0};
status = NAVcmd_ConstructArgs(&cmdParams,
                              argc, &argv[0]); /* arguments to main() function*/
printf("ConstructArgs Result: %s\n", NavApiStatus[status]);
```

7.2 Initialize the Device Driver

The bedrock of Navigator BSP is the device driver. The device driver works with the operating system and allows us to open a PCIe device for read/write access. It maps the internal memory of the hardware to the host RAM such that a user–space applica–tion can read or write to it. The driver also manages hardware interrupts from the device. Before an application can start interacting with the device driver, the driver resources must be initialized.

7.2 Initialize the Device Driver (continued)

The Navigator BSP provides a simple wrapper function to initialize the driver. The ini–tialization must be done once per application. More than one application can utilize the device driver simultaneously. However, the same board must not be accessed concur–rently from different applications.

```
status = NAV_BoardStartup();
```

7.3 Open a Board

Once the driver has been initialized, it can be asked to open a particular board model. A list of all Pentek boards found on the host system also can be obtained to narrow down the available choices. When a board is opened, the user application receives an abstract reference (a handle) to that board's resources. The handle is intended to be opaque from the perspective of user applications. This handle must be used in future API calls to set up the board. For details, refer to the documentation in the **nav_api.c** file.

```
uint32_t          numBoards;
NAV_DEVICE_INFO  *pciDeviceInfo[NAV_MAX_BOARDS];
void             *boardHandle;

/* Find all Pentek Boards */
status = NAV_BoardFind(0, pciDeviceInfo, &numBoards);

/* Or, find all 71861 boards */
status = NAV_BoardFind(0x71861, pciDeviceInfo, &numBoards);

/* Select a board to open via an interactive list */
boardHandle = NAV_BoardSelect(numBoards, 0, pciDeviceInfo[0], 0);
```

7.4 Initialize Application–Specific Resources

Every application needs to initialize or allocate some resources to perform its task. This is a good time to initialize some of those resources since we can be sure that a board has been found and opened. If there was an error while initializing the driver or opening a board, the application can avoid initializing resources that will never be used.

```
NAV_SYS_CONTEXT appSysContext;
FILE *applicationLog;

/* Initialize the context created for
 * system-specific resources (like semaphores) */
NAVsys_Init(&appSysContext);

/* Open files or allocate memory as needed... */
applicationLog = fopen("logfile.txt", "w");
```

Introduced in the snippet above is the Navigator BSP's built–in support for some platform–specific system calls. The BSP provides a unified interface to operating–system–specific mechanisms, which primarily deal with multi–threading and synchronization primitives but also provide other helper routines.

The context structure is used to hold the states or handles of system resources used within a piece of code. For details about this interface, refer to the documentation in the **nav_sys.c** file.

7.5 Set Up Board Resources

After a board has been opened, the user application should initialize it for a particular use case. Although it is technically possible to access individual registers in the board, the Navigator BSP provides a high–level API to set up logically separate resources of the board in one or two API calls. For example, the board clock, A/D converter, and DMA engine can all be configured with only a few function calls. Note that other features of the hardware like sync source, gate source, and triggering mechanism also must be set up before starting the acquisition.

These function calls are present in resource–specific source files (e.g., **nav_adc.c**, **nav_ddc.c**, **nav_dma.c**) and have the word **Setup** in their names (e.g., **NAV_AdcSetup()**, **NAV_DdcSetup()**, **NAV_DmaSetup()**). These high–level functions accept generic arguments for all boards, perform standard validation on them, and then call their board–specific counterparts, which carry out the actual implementation for a particular board. This approach helps in maintaining a uniform API for all user applications while still allowing flexibility of implementation within the library for different hardware products.

7.5 Set Up Board Resources (continued)

All of these setup functions also accept a 64–bit integer argument named 'options' (see Appendix A). This argument is only applicable to certain boards and provides a way for users to choose some feature that is available on that board itself. For example, on Model 71841 the ADC chip supports a dual–edge sampling mode (DES mode) which effectively doubles the sampling rate but sacrifices the second input channel. If you want to use this feature, you should supply the proper values via the 'options' argu– ment. For most boards, this value should be 0. The list of valid values can be seen in the description for the setup function.

Program arguments obtained in the first step of the application can be used while call– ing these “Setup” functions, as demonstrated in Navigator BSP example programs like **`acquire`**. The snippet on the next page shows hardcoded arguments for simplicity and clarity.

7.5 Set Up Board Resources (continued)

```
/* CLOCK
 * Use 200 MHz internally generated clock for ADC sampling rate,
 * with a 10 MHz reference supplied to front-panel CLK connector. */
status =
NAV_ClockSetup(boardHandle,
                   NAV_CLOCK_INT,   /* Board Clock Source */
                   200.0e6,         /* Board Clock Frequency */
                   10.0e6,          /* Reference Clock Frequency */
                   200.0e6,         /* ADC Clock Frequency */
                   0.0e6,           /* DAC Clock Frequency */
                   0.0e6,           /* ADC Test Signal Frequency */
                   0.0e6,           /* DAC Test Signal Frequency */
                   0);              /* Board-specific options */
}

/* ADC
 * Set ADC Channel 1 in Gate mode, generating 16-bit real samples
 * with data source as front-panel IN1 connector. */
status =
NAV_AdcSetup(boardHandle,
             NAV_CHAN_1,                        /* Channel being configured */
             NAV_CHAN_1,                        /* Data source */
             NAV_ADC_FORMAT_16BIT_REAL_PACKED,    /* Data packing format */
             0);                                  /* Board-specific options */


/* DMA
 * Set DMA Channel 1 in Continuous mode, acquiring 2 MiB of data per buffer
 * segment with 20 such buffers in use. */
status =
NAV_DmaSetup(boardHandle,
             NAV_CHANNEL_TYPE_ADC,      /* ADC data */
             NAV_CHAN_1,                        /* Channel being configured */
             20,                                /* Number of buffers */
             2U * (1<<20),                      /* Buffer size */
             NAV_DMA_METADATA_ENABLE,           /* FPGA will provide metadata */
             NAV_DMA_RUN_MODE_CONTINUOUS_LOOP,  /* Operating mode */
             NAV_SYS_WAIT_STATE_MILSEC(15000),  /* Timeout period */
             &dmaCallbackHandler,               /* DMA callback handler */
             boardHandle,                       /* Data pointer for the handler */
             0);                                /* Board-specific options */
```

7.6 Dump the Register State for Debugging Purposes

The user application can dump the register state of the device at any particular time to help with future debugging efforts. The register contents can be dumped to the console or to a file.

```
printf("Register state after resource configuration:\n");
status = NAV_IPRegDump(boardHandle, stdout);
```

7.7 Start the Data Flow

After configuration is complete, the user application must arm the trigger control state machine for the acquisition channel. Arming activates the acquisition channel such that it is ready to accept a gate or trigger signal which will start the acquisition process. The application can then start data flow by opening gates or generating triggers.

Gates and triggers are signals that tell the FPGA to start data transfer. Depending on the hardware, the gate/trigger signal can be supplied internally via register bit tog– gling or externally via a connector on the front panel. (For details about a particular product, please refer to its operating manual.) For simplicity, internal gating is shown in the snippet below. API calls are available to manipulate these gates for both incom– ing (A/D) and outgoing (D/A) directions.

```
/* Arm the trigger control state machine for ADC Channel 1 */
status = NAV_TrigArm(boardHandle,
                     NAV_CHANNEL_TYPE_ADC,
                     NAV_CHAN_1);

/* Open internal gate for ADC Channel 1 (local gate will affect only this channel)
*/
status = NAV_LocalGateOpen(boardHandle,
                           NAV_CHANNEL_TYPE_ADC,
                           NAV_CHAN_1);
```

7.8 Manage Data Transfer

Pentek boards use Direct Memory Access (DMA) to efficiently transfer data between the device and host memory. A special software framework for handling the minutiae of DMA operation is built within the Navigator BSP. If an application has requested DMA support while calling **`NAV_DmaSetup()`**, the library will provide a callback to a function within the user code, in response to certain events of interest. This callback will provide a status code along with pointers to DMA buffers in host memory and application–specific data. By looking at the status code, the application's DMA callback handler can decide which action to take. See Appendix C for details on this framework.

```
/* Typical implementation of a DMA callback handler for incoming A/D data.
 * This function will be executed by the library in a separate thread
 * for each channel, whenever user code needs to be notified of a DMA event.
 * For example, when the hardware has filled up a DMA buffer, the user code
 * can start using that data from this function. */
void dmaCallbackHandler (int32_t channel, int32_t dmaStatus,
                         void *dataBuffer, void *metaDataBuffer,
                         void *userData)
{
   NAV_DMA_ADC_META_DATA *metaData;
   void *boardHandle = userData;

    /* If a DMA buffer has been filled, act on it */
    if(dmaStatus & NAV_STAT_DMA_LINK_END)
    {
        /* Print useful metadata */
        metaData = (NAV_DMA_ADC_META_DATA *)metaDataBuffer;
        printf(stdout, "MetaData ValidBytes %u\n",
                metaData->validBytes);
        printf(stdout, "MetaData Counter    %d\n\n",
                metaData->packetCounter);

        /* Save data buffer to a file */
        NAV_WriteDataFile(dataFile[channel], dataBuffer,
                          metaData->validBytes,
                          NAV_DEBUG_FILE_FORMAT_BIN,
                          NAV_DEBUG_DATA_WIDTH_16BIT);

        /* Send portion of the data buffer to Signal Viewer */
        NAVview_SendData(boardHandle,
                         channel,
                         dataBuffer,
                         NAV_VIEW_BLK_SIZE_DEFAULT);
    }
}
```

7.8 Manage Data Transfer (continued)

As shown in the code snippet above, the Navigator BSP includes a function to save the data buffer to a file. Data can be stored in both binary and ASCII formats. For more information, refer to the documentation in the **`nav_debug.c`** file.

Navigator BSP also provides a a Signal Viewer that enables you to look at an incoming signal during acquisition. Both time−domain and frequency−domain representations are shown. For more information about the Signal Viewer, see Appendix B.

7.9 Handle Hardware Interrupts

Depending on the requirements, a user application may need to be notified of certain hardware events. For example, an application running on an externally supplied clock signal should stop normal operation when the clock signal is lost. This application can enable an interrupt for a clock−loss event such that the interrupt handler function is invoked by the device driver when the hardware discovers a loss of clock signal. The snippet on the next page shows how this can be achieved.

For more information, refer to the documentation in the **`nav_intr.c`** file. Available interrupt sources and events may vary from board to board, so please consult the oper− ating manual for the hardware as well.

The mask for the desired interrupt itself will also depend on the IP core that is imple− menting that interrupt source. For example, the Clock A interrupt source is imple− mented using **px_cdc_clk_intrfc** core on Model 71861 while the same source is implemented on Model 71841 using **px_sample_clk_rcvr** core.

7.9 Handle Hardware Interrupts (continued)

```
/* Enable the interrupt for clock-loss detection.
 * Do this while setting up the board resources. */
status = NAV_InterruptEnable(boardHandle,
                            NAV_INTR_DATA_IO_CLOCK_A, 0,
                            NAV_IP_CDC_CLK_INTRFC_INTR_CLK_NOT_OK,
                            &clockLostIntrHandler, NULL);



/* Interrupt handler for ClockNotOk event.
 * This function will be executed by the driver in a separate thread,
 * when the hardware raises the interrupt associated with this handler. */
void clockLostIntrHandler(void              *hDev,
                          int32_t            intSource,
                          int32_t            instance,
                          uint32_t           intFlag,
                          int32_t            numInterrupts,
                          int32_t            numLostInterrupts,
                          void              *pData)
{
    if (intFlag & NAV_IP_CDC_CLK_INTRFC_INTR_CLK_NOT_OK)
    {
        printf("Clock loss detected...\n");
        // Stop the application
    }
    else
        return;

}
```

7.10 Stop the Data Flow

If the goals of the application have been met or any other termination condition is encountered, the data flow should be stopped. The application must disarm the trigger control state machine for the acquisition channel. Disarming will deactivate the acqui–sition channel such that it will not respond to a new gate or trigger signal. If controlled by software, the application should also close the gates or stop the trigger generation. As noted earlier, the gate or trigger can also come from external signals. For simplicity, an internal gate is shown in the snippet below.

```
/* Disarm the trigger control state machine for ADC Channel 1 */
status = NAV_TrigDisarm(boardHandle,
                        NAV_CHANNEL_TYPE_ADC,
                        NAV_CHAN_1);

/* Clear existing trigger */
status = NAV_TrigClear(boardHandle,
                       NAV_CHANNEL_TYPE_ADC,
                       NAV_CHAN_1);



/* Close internal gate for ADC Channel 1 (local gate will affect only this
channel)*/
status = NAV_LocalGateClose(boardHandle,
                            NAV_CHANNEL_TYPE_ADC,
                            NAV_CHAN_1);
```

7.11 Free up the Resources

In preparation for exiting the application, any resources that were allocated during the execution will need to be properly freed.

```
/* Destroy the context created for
 * system-specific resources (like semaphores) */
NAVsys_UnInit(&appSysContext);

/* Close files or free memory as needed... */
fclose(applicationLog);
```

7.12 Close the Board

This step frees the resources allocated within the BSP library for working with a partic–ular board. Once a board is closed, it cannot be accessed by the handle that was received on opening the board.

```
status = NAV_BoardClose(boardHandle);
```

7.13 Uninitialize the Driver

Similar to the previous step, this frees up the resources created within the device driver for supporting the current application.

```
NAV_BoardFinish();
```

7.14 Exit the Application

This is the last step!

This page is intentionally blank

Chapter 8: Adding an IP Core to the Navigator BSP

8.1 Introduction

The process of creating your own IP core includes generating a document similar to the documents Pentek provides for each Pentek IP core. Your document will help guide you in creating support code to be added to the Pentek's Navigator Board Support Package (BSP).

For each Pentek IP core, there is a companion Navigator BSP header and source file (see Chapter 2 and Section 2.3). The filenames are the same, except for the filename extension. We recommend that you follow the same method.

The main problem with adding support for your IP module to the Navigator BSP is that, if the BSP is updated, you'll have to add code for your module to the updated BSP. To make this easier, we suggest that you create your own library. If you create your own library, only references to your library will have to be added to an updated BSP.

8.2 Create Your Library Files

The document you create for your IP core should include a section for the register space, which will define all register address offsets from the IP module base address. Since an IP module can be used multiple times in a design, the module base address is project–specific. For creating your support files, you will need the module base address. It is added to the Pentek library later.

The first file you'll need to create is a header file. This file will contain defines for register offsets, register bit fields, function prototypes, etc., to support your IP module.

If you take a look at a few Pentek IP core header files, you'll see that most of these files start with register offset defines. For example, in the header file **px_ads5485intrfc.h**, the first two register offset defines are as follows:

```
/* Control Register 1 */
#define NAV_IP_ADS5485_INTRFC_CTRL1_REG    0x00000000

/* Control Register 2 */
#define NAV_IP_ADS5485_INTRFC_CTRL2_REG    0x00000004
```

<u>NOTE:</u> The Pentek documentation shows offsets as 32–bit offsets but the defines are byte offsets in the header file.

These are followed by a section of bit field and mask defines. In our example header file (**px_ads5485intrfc.h**), the Control Register 1 has one 9–bit bit field, from bits D08 through D00, and one single–bit field at bit D09.

8.2 Create Your Library Files (continued)

The register defines are as follows:

```
#define NAV_IP_ADS5485_INTRFC_CTRL1_MASK            0x000003FF

#define NAV_IP_ADS5485_INTRFC_CTRL1_DLY_LD_CTL_IDLE  0x00000000
#define NAV_IP_ADS5485_INTRFC_CTRL1_DLY_LD_CTL_LOAD  0x00000200

#define NAV_IP_ADS5485_INTRFC_CTRL1_TAP_DELAY_MASK   0x000001FF
```

Note the naming convention of these defines:

- The **NAV_** prefix indicates that the define is part of the Navigator package.
- The **IP_** segment indicates that the define is in the IP Core library.
- **ADS5485_INTRFC_** indicates the IP core it supports.
- **CTRL1_** indicates the register name.
- This is followed by a bit field identifier and a function identifier.

Any special definitions, structures, etc. will follow. Function prototypes are at the end of the header file.

You will want to add an include directive to your header file for **nav_ip_common.h**. This will give you access to defines, macros, and other resources.

Once you have written the header file, you will create the C source file. The filename will be the same as the header file, but the extension will be **.c**. For example, the **px_ads5485intrfc.h** header file is matched with a **px_ads5485intrfc.c** source file.

Besides comments, the first line of code in your source file should be an include state–ment, which includes the path. For example, in **px_ads5485intrfc.c**, it is:

```
#include "ip_include/px_ads5485intrfc.h"
```

indicating that the header file is in **ip_include**, below the standard Navigator include directory.

A typical source file will contain separate **set** and **get** functions for each register bit field, as needed. There are typically higher–level routines that call these set and get function at the start of a file. For example, in **px_ads5485intrfc.c**, the first function found is **NAVip_Ads5485Intrfc_InitRegs()**, used to set all registers to power–on default values.

8.2 Create Your Library Files (continued)

A **get** function typically returns the masked contents of a register. An example from **adc5485intrfc.c** follows:

```
int32_t NAVip_Ads5485Intrfc_Ctrl1_GetTapDelay(volatile uint32_t *coreBase,
                                               uint32_t          *tapDelay)
{
    if (coreBase == NULL)
        return (NAV_IP_STAT_INVAL_ADDR);
    *(tapDelay) = NavRegRead((coreBase + NAV_IP_REG_OFFSET(NAV_IP_ADS5485_INTRFC_CTRL1_REG)),
                             NAV_IP_ADS5485_INTRFC_CTRL1_TAP_DELAY_MASK);
    return (NAV_IP_STAT_OK);
}
```

By convention, Pentek routines always return a status, which is a 32–bit value. They do not return the requested value. Instead, the address of a variable is supplied to the function. In the above example, the arguments to the function are pointers to the IP module base address and the variable to hold the return.

The function calls **NavRegRead()**, which is one of three inline routines used frequently in IP support functions. The other two commonly used inline routines are **NavRegWrite()** and **NavRegReadModWrite()**. For details, refer to the documentation in the **nav_inline.h** file.

A typical **set** function is shown below.

```
int32_t NAVip_Ads5485Intrfc_Ctrl1_SetTapDelay(volatile uint32_t *coreBase,
                                               uint32_t           tapDelay)
{
    if (coreBase == NULL)
        return (NAV_IP_STAT_INVAL_ADDR);
    NavRegReadModWrite((coreBase + NAV_IP_REG_OFFSET(NAV_IP_ADS5485_INTRFC_CTRL1_REG)),
                       NAV_IP_ADS5485_INTRFC_CTRL1_TAP_DELAY_MASK, tapDelay);
    return (NAV_IP_STAT_OK);
}
```

<u>**NOTE:**</u> All functions receive only the base address of the module and must supply the register offset internally.

8.3 Add Your Files to the Library

Once you've finished creating your header and source files, they must be added to the library in order to use them. We suggest that you bundle your code as an independent library or shared object and patch it into the Pentek library. This way, if the Navigator BSP is updated in the future, it will be easier to link your custom library to the application without any effect from updates to the Navigator BSP. Examine how the Pentek IP core library is built and create your own Dynamic Link Library / Shared Object.

Next, you need to decide how your routines will be called. In the Navigator BSP environment, the IP core routines are called by board–level routines. For example, for Pentek's Model 71861 module, code for board–level support is found in the **`860_include`** and **`860_source`** directories. If you are adding a new IP core to a 71861 module, you may want to create board–level source and header files in these directories.

Continuing with our Model 71861 example, assume you added a data modification module that is used at the output of the DDC for all four channels of the 71861. Its design modifies every data sample coming out of the DDC. It has a control register with three active bits:

- **`D00`** – Module Enable/Disable – switches the IP module in or out
- **`D01`** – Core Reset – resets the core to power–on state
- **`D02`** – Module Load – loads a new module value

It has one 32–bit wide data modification value register.

Your IP code has XXXX routines:

- **MyIPCore_ModEnableDisable()**
- **MyIPCore_ModReset()**
- **MyIPCore_ModLoad()**
- **MyIPCore_GetModValue()**
- **MyIPCore_SetModValue()**

Since there are four DDC channels, there are four instances of this new core. Assume the cores are accessed in User Block 1 and have base addresses as follows:

- DDC Channel 1: **`0x03000000`**
- DDC Channel 2: **`0x03001000`**
- DDC Channel 3: **`0x03002000`**
- DDC Channel 4: **`0x03003000`**

8.3 Add Your Files to the Library (continued)

Here are the rest of the steps for our Model 71861 example scenario:

1) Add defines for the four addresses in **`nav860_mem_map.h`** or create a special header for your IP core containing these addresses.

2) Create entries in the **`NAV_IP_ADDR_TABLE`** address table for the new base addresses. For example:

   ```
   volatile uint32_t *modDdc[NAV_MAX_DDC_CHANNELS];
   ```

3) Create a 71861 source file containing board–level routines. In this example, the low–level routines may be sufficient for calling from the API level but assume it's desirable to have one routine that sets mod value and loads it. So, you will write a board–level routine called **`MyDDCMod_LoadModValue()`** that calls **`MyIPCore_SetModValue()`** and **`MyIPCOre_ModLoad()`**.

4) Create a 71861 header file for your source file that contains function prototypes, etc., as needed.

5) Add an include statement for your header to **`nav860.h`**.

6) Add your source file to the 860 compile chain and rebuild.

7) Make a list of what you created and where you put it, so you can add your IP module to future BSP updates.

This page is intentionally blank

Chapter 9: Troubleshooting

If you cannot find the help you need in this chapter, refer to our website for more help: http://www.pentek.com/support

9.1 Cannot run the examples' executable (Windows)

Problem: After installing the Navigator BSP for Windows, you are not able to run the examples' executable (either the Debug or Release version).

Possible messages shown on executing the application:

- **`The program can't start because api-ms-win-crt-runtime-l1-1-0.dll is missing from your computer. Try reinstalling the program to fix this problem.`**
- **`The program can't start because ucrtbased.dll is missing from your computer. Try reinstalling the program to fix this problem.`**

Solution: Your system is missing runtime components for running Windows applica–tions. Refer to the NOTE in Section •.

9.2 DMA Thread cannot be created (Linux)

Problem: While running an application based on the Navigator BSP, you encounter a status code for **`NAV_STAT_DMA_THREAD_CREATION_FAIL`**.

Solution: This error usually indicates that a system–imposed limit on resource usage has been reached. The most common cause would be the exhaustion of available resources for message queues. Navigator BSP's DMA threads rely on message queues for inter–thread communication and will not run if such queues could not be created.

First, ensure that no orphaned message queues are present on the system. The BSP nor–mally destroys all message queues when they are no longer required. However, if the application crashes for some reason, the queues are left untouched.

To delete the message queues manually, navigate to **`/dev/mqueue`** and delete files belonging to orphaned message queues. Queues created by the BSP are named **`penteknav_<PID>_<ContextID>_<DMAChannel>`**. The PID (process identifier) field can be used to ensure queues belonging to a running application are not deleted. Rebooting the system also guarantees that all orphaned queues are destroyed.

9.2 DMA Thread cannot be created (Linux) (continued)

If there are no such orphaned message queues inside **/dev/mqueue**, the system may have a lower resource limit for POSIX message queues than required by the DMA Threads. Linux imposes a limit on the maximum memory allowed for use by POSIX message queues. A low value of this limit (819200 bytes by default) will prevent multiple DMA threads from running simultaneously, especially if other applications are also using POSIX message queues.

Copy the following two lines to your **/etc/security/limits.conf** file to raise the default limit.

```
*       hard    msgqueue        4915200
*       soft    msgqueue        4915200
```

The number in the last column can be changed, if needed. Consult the **ulimit** man page or the **limits.conf** file for further information. The installer for Navigator BSP on Linux will modify the **limits.conf** file during installation. If the installer is not used, the limit must be raised manually (as a superuser). It is necessary to log out and log back in for the changes in **limits.conf** to take effect.

A much rarer cause could be that the system–imposed limit on the number of threads was exceeded. Consult the manual entry for **pthread_create** to find out details on this limit.

9.3 Installing LabVIEW RTE on non–RPM–based systems (Linux)

Problem: The LabVIEW Runtime Environment fails to install with the following error message:

This install script supports only RPM-based package installation system. Installation aborted.

Solution: National Instruments only provides RPM packages for LabVIEW and officially supports Red Hat Enterprise Linux Desktop + Workstation 5 or later, openSUSE 11.4 or later, or Scientific Linux 6 or later. These distributions use the RedHat Package Manager (RPM).

For distributions that do not use RPM, a Linux utility called **alien** can be used to convert the **rpm** packages to a different format like Debian's **dpkg**. Be very careful while using the **alien** utility to install unsupported packages on a system. Usage instructions for **alien** on Ubuntu can be found here.

To successfully run a LabVIEW–based application like Navigator Signal Viewer, some prerequisites must also be installed as mentioned in this National Instruments Knowledge Base.

9.4 Illegible fonts in Navigator Signal Viewer (Linux)

Problem: When you launch the Navigator Signal Viewer, the text labels are illegible or appear to be in a language other than English.

Solution: This issue is caused by missing fonts for the X Window System. To fix the problem, install 100–dpi and 75–dpi versions of the X.Org Fonts package for your distribution. Corresponding package names for popular distributions are provided below:

Fedora / RedHat / CentOS	xorg–x11–fonts–100dpi xorg–x11–fonts–75dpi
Debian / Ubuntu	xfonts–100dpi xfonts–75dpi xfonts–scalable
Arch	xorg–fonts–100dpi xorg–fonts–75dpi

You may need to reboot the system before the changes will take effect.

9.5 Windows 10 update may cause driver and reserve memory issues

Problem: When Windows 10 updates overnight, the update may cause issues with the Pentek driver. Windows Device Manager will show that the driver for Pentek boards is installed and working properly, but the device will fail to open via BSP routines. Also, the updates may reset the reserve memory functionality used to dedicate a portion of the RAM for DMA purposes.

Solution: To fix this problem, reinstall the Navigator Driver and reapply the remove memory command.

1. Manually install the driver by following the instructions in Section 3.3.6.1.

2. If the reserved memory functionality is being used, reapply the command:

```
bcdedit /set removememory <size in megabytes>
```

This page is intentionally blank

Appendix A: Navigator BSP Command Line Utility

A.1 Introduction

The command line utility provides flexibility when running the Navigator BSP exam– ple programs. It allows you to change some common program options without needing to rebuild the examples. The command line utility is part of the BSP's utility library (**`NavBSP_util`**). It is available for the ADC, DDC, DAC, and DUC examples. It also pro– vides basic validations on the option values that are passed to the interface.

The command line utility provides two ways to change an example program's default execution options for different operating modes: via command prompt and **`ini`** files.

- ❑ **Command prompt** – This is the traditional command line argument feeding method. Program options can be changed by entering the different command line arguments and options at the command prompt. The syntax is **`-<argument> <option>`** (for example, **`-model 71861`**). Section A.2 lists all the command–line arguments.

- ❑ **Initialization files** (**`.ini`** files) – Rather than typing the command line arguments each time you run an application, the arguments are retrieved from an **`ini`** file. In the **`ini`** file, the argument and its option must be separated by an equals (**`=`**) sign (for example, **`model=71861`**).

  Each example in the BSP may have one or more associated **`ini`** files in the program's source directory. The default **`ini`** file is named **`<program>_default.ini`**.

  When you specify an **`ini`** file with a **`-ini <filename>`** command at the command prompt, the command line utility will read and process the specified **`ini`** file. If you don't specify an **`ini`** file, the command line utility will search in the current directory for the default **`ini`** file for the same program name. If the utility does not find the default **`ini`** file in the current directory, the utility will search inside the parent directory (**`..\`** for Windows, and **`../`** for Linux).

  For example, if you run the **`acquire`** example program, the command line utility will search in the directory from which the program is running for a file named **`acquire_default.ini`**. If the utility cannot find the file, it will search in the directory above the directory from which the program is running. If the utility still cannot find the **`acquire_default.ini`** file in the parent directory, some example programs may use the generic argument options stored in the library.

  Section A.2 describes all the initialization files. The files specific to your Jade board are located in the **`examples`** directory.

Note that you can use command line argument options together with saved **`ini`** file argument options. However, arguments issued on the command line have a higher priority than arguments saved in an **`ini`** file. For example, you can save the normal operating argument options in an **`ini`** file, and also modify an option at program run– time via the command prompt. Instead of opening and modifying the **`ini`** file, you can just enter the desired change at the command prompt with **`-<argument> <option>`**.

A.2 List of Command–Line Arguments

All arguments and their corresponding options for all the example programs are listed in this section. Typing the name of an example program followed by `-h` or `-?` will list all the command–line arguments.

NOTE: Argument names are not case–sensitive.

NOTE: Not all arguments or argument options are supported by every example program. Refer to the default argument list of an individual example program for the specified supported arguments.

The table below shows the arguments grouped by category. **Each argument has a brief description, but also is linked to a detailed description** in the list starting on page A–5.

System–specific arguments	
–brd	Board index in decimal format
–ini	Read from specified .ini file
–model	Choose model for program (Module ID)
Common arguments for any ADC, DAC, DDC, or DUC programs	
–brdclksrc	Board clock source
–brdfreq	Board clock frequency
–brdmode	Board operation mode
–chanmask	Channel mask for single or multichannel operation
–loop	Number of loops
–numbuf	Number of buffers
–reffreq	Reference clock frequency
–xfersize	Transfer size in number of bytes/samples
Generic optional arguments	
–brdoption	Optional option for board setup (unsigned 64–bits)
–clkoption	Optional option for clock setup (unsigned 64–bits)
–dmaoption	Optional option for DMA setup (unsigned 64–bits)
–gateoption	Optional option for gate setup (unsigned 64–bits)
–progoption	Optional option for program running (unsigned 64–bits)
–syncoption	Optional option for sync setup (unsigned 64–bits)
Arguments only for ADC and DDC programs	
-adcdatamode	ADC data mode
-adcdatasrc	ADC data source
-adcfreq	ADC clock frequency rate
–adcgatedly	ADC gate tap delay
–adcgatepol	ADC gate/trigger polarity

Arguments only for ADC and DDC programs (continued)	
–adcgatesrc	ADC gate/trigger signal source
–adcgtrigmode	ADC gate/trigger mode selection
–adcindly	ADC input tap delay
–adcoption	Optional option for ADC setup (unsigned 64–bits)
–adcppspol	ADC PPS polarity
–adcppssrc	ADC PPS signal source
–adcsyncdly	ADC sync tap delay
–adcsyncpol	ADC sync polarity
–adcsyncsrc	ADC sync signal source
Arguments only for DDC programs	
–ddc	DDC enabled
–ddcgain	DDC gain
–ddcinvert	DDC spectrum inversion enabled
–ddcoption	Optional option for DDC setup (unsigned 64–bits)
–ddcphase	DDC phase in degrees
–decim	DDC decimation
–tunefreq	Tuning frequency
Arguments only for DAC and DUC programs	
–dacdatamode	DAC data mode
–dacdatasrc	DAC data source
–dacfreq	DAC clock frequency rate
–dacgatedly	DAC gate tap delay
–dacgatepol	DAC gate/trigger polarity
–dacgatesrc	DAC gate/trigger signal source
–dacgtrigmode	DAC gate/trigger mode
–dacoption	Optional option for DAC setup (unsigned 64–bit)
`-dacoutdly`	DAC output tap delay
–dacppspol	DAC PPS polarity
–dacppssrc	DAC PPS signal source
–dacsyncdly	DAC sync tap delay
–dacsyncpol	DAC sync polarity
–dacsyncsrc	DAC sync signal source
Arguments only for DUC programs	
–duc	DUC enabled
–ducgain	DUC gain
–ducinvert	DUC spectrum inversion enabled
–ducoption	Optional option for DUC setup (unsigned 64–bits)

Arguments only for DUC programs (continued)	
–ducphase	DUC phase in degrees
–interp	Interpolation
–ncofreq	NCO frequency
Signal Viewer arguments	
–vchanmask	Signal Viewer viewing channel mask
–vhost	Signal Viewer server host mode
–vport	Signal Viewer port number
–vsubchan	Signal Viewer viewing sub–channel
ADC data file writing arguments	
–wdatafmt	Data file format to be written to file
–wfile	Data file name to be written with captured data.
–wsize	Data size to be written to the file
DAC data file reading arguments	
–rdatafmt	Data file format to be read from file
–rfile	File name of data to be read from
–rsize	Data size to be read from the file
Timer – in the form of YYYY:MM:DD, HH:MM:SS (I.e., 2017:12:25, 10:00:00)	
–tstart	Time start
–tstop	Time stop
Multi–board event synchronization server/client arguments	
–caddr	Client host IP address
–cport	Client port number
–saddr	Server host IP address
–sport	Server port number

A.2 List of Command–Line Arguments (continued)

`-adcdatamode` ADC data mode. You can use one of the following options:

- **rpk8** – 8–bit real with time packed (4 consecutive samples stored in one 32–bit word)
- **rpk16** – 16–bit real with time packed (2 consecutive samples stored in one 32–bit word)
- **iqpk16** – 16–bit I/Q packed (I and Q samples stored together in a single 32–bit word)
- **iqupk24** – 24–bit I/Q unpacked (I and Q samples stored separately in 32–bit words, with 24–bits of valid data padded with 0's in the lower 8 bits)
- **iqupk32** – 32 bit I/Q samples unpacked (I and Q samples are interleavedly stored in 32–bit words)

`-adcdatasrc` ADC data source. You can use one of the following options:

- **adc1** – Analog signal data comes from channel input 1.
- **adc2** – Analog signal data comes from channel input 2.
- **adc3** – Analog signal data comes from channel input 3.
- **adc4** – Analog signal data comes from channel input 4.
- **adc5** – Analog signal data comes from channel input 5.
- **adc6** – Analog signal data comes from channel input 6.
- **adc7** – Analog signal data comes from channel input 7.
- **adc8** – Analog signal data comes from channel input 8.
- **own** – Analog signal data comes from its own corresponding channel input.
- **sine** – Analog signal data comes from internal test SINE signal.
- **ramp** – Analog signal data comes from internal test RAMP signal.

`-adcfreq` ADC clock frequency rate, in floating point format. This value should be a factor of **`-brdfreq`**. Valid dividers are 1, 2, 3, 4, 6, 8, and 16.

`-adcgatedly` ADC gate tap delay

`-adcgatepol` ADC **GATE/TRIG** polarity. Valid options are:

- **normal** – As received, or Rising edge
- **invert** – Inverted from received, or Falling edge

A.2 List of Command–Line Arguments (continued)

`-adcgatesrc`	The source for ADC **GATE/TRIG** signals. Valid options are: • **off** – No signal is provided. A gate/trigger signal is not generated. Therefore, no ADC output is expected. • **reg** – Internal registers are used to generate signals. • **ttl** – Front panel **TRIG** input (SSMC connector, TTL voltage level) • **sbusttl** – Front panel **SYNC/GATE** input (26–pin sync bus connector, TTL voltage level) • **sbusdif** – Front panel **SYNC/GATE** input (26–pin sync bus connector, LVPECL voltage level)
`-adcgtrigmode`	ADC gate, trigger, or trigger hold selection. Valid options are: • **gate** – Global gate for all channels • **lgate** – Local gate for an individual channel • **trig** – Trigger mode • **trighold** – Trigger hold mode • **timestamp** – Timestamp mode
`-adcindly`	ADC input tap delay
`-adcoption`	A 64–bit optional value used for selecting a special feature of some boards during ADC setup.
`-adcppspol`	ADC **PPS** polarity. You can use one of the following options: • **normal** – As received, or Rising edge • **invert** – Inverted from received, or Falling edge
`-adcppssrc`	The source for ADC **PPS** signals. Valid options are: • **off** – No signal is provided. • **reg** – Internal registers are used to generate signals. • **syncsbusttl** – TTL **SYNC** signal from SBUS connector • **gatesrc** – Received **GATE** signal • **syncsbusdif** – Diff **SYNC** signal from front panel 26–pin SBUS connector • **gatesbusttl** – TTL **GATE** signal from front panel 26–pin SBUS connector • **gatessmcttl** – TTL **GATE** signal from front panel SSMC connector

A.2 List of Command–Line Arguments (continued)

`-adcsyncdly` ADC sync tap delay

`-adcsyncpol` ADC **SYNC** polarity. You can use one of the following options:

- **normal** – As received, or Rising edge
- **invert** – Inverted from received, or Falling edge

`-adcsyncsrc` The source for ADC **SYNC** signals. You can use one of the following options:

- **off** – No signal is provided. A sync signal is not generated. Therefore, no ADC output is expected.
- **reg** – Internal registers are used to generate signals.
- **ttl** – Front panel **TRIG** input (SSMC connector, TTL voltage level)
- **sbusttl** – Front panel **SYNC/GATE** input (26–pin sync bus connector, TTL voltage level)
- **sbusdif** – Front panel **SYNC/GATE** input (26–pin sync bus connector, LVPECL voltage level)
- **gatesrc** – Received **GATE** signal

`-brd` Board index in decimal format. Used when multiple Pentek boards are detected in the same system. Ranges from 1 to 19. In Navigator BSP example programs, a special value of 0 will provide the indexes of found boards and prompt you for selection.

`-brdclksrc` Board clock source. Choose one of the following clock selection modes for the primary board clock. This board clock is used to gen–erate other necessary clocks like those for ADC and DAC sampling frequency. You can use one of the following options:

- **int** – A clock is generated through the on–board VCXO and a reference clock is used from the **CLK** connector if available.
- **int_sbusref** – A clock is generated through the on–board VCXO and a reference clock is used from the 26–pin **SYNC/GATE** connector if available.
- **ext** – An externally generated clock is accepted at the **CLK** connector.
- **ext_sbus** – An externally generated clock is accepted at the 26–pin **SYNC/GATE** connector.

A.2 List of Command–Line Arguments (continued)

`-brdfreq`	Board clock frequency rate in floating point representation. In Navigator BSP example programs, it is used as the frequency of the externally supplied clock signal or the clock generated by the on–board oscillator.
`-brdmode`	Board operation mode. You can use one of the following options: • **alone** – Stand–alone mode. • **slv** – Slave mode. The target board accepts **CLK**, **GATE/TRIG**, or **SYNC/PPS** from another board. • **mstr** – Master mode. The target board provides **CLK**, **GATE/TRIG**, or **SYNC/PPS** to another board.
`-brdoption`	A 64–bit optional value used for selecting a special feature of some boards during board setup.
`-caddr`	Client host IP address
`-chanmask`	Channel mask for single or multichannel operation in hex representation with or without the **`0x`** prefix. It supports up to 32 channels. **`1`** – For channel 1; **`2`** – For channel 2; **`4`** – For channel 3; **`8`** – For channel 4; **`10`** – For channel 5; **`20`** – For channel 6; **`40`** – For channel 7; **`80`** – For channel 8; **`FF`** – For all 8 channels (1 to 8); **`FFFFFFFF`** – For all 32 channels (1 to 32)
`-clkoption`	A 64–bit optional value used for selecting a special feature of some boards during clock setup.
`-cport`	Client port number

A.2 List of Command–Line Arguments (continued)

`-dacdatamode` DAC data mode. You can use one of the following options:

- **timepk8** – 8–bit real, time packed
  (4 consecutive samples stored in a single 32–bit word)
- **timepk** – 16–bit real, time packed
  (2 consecutive samples stored in a single 32–bit word)
- **chanpk** – 16–bit real, channel packed
  (2 channel–interleaved samples stored in a single 32–bit word)
- **iqpk** – 16–bit I/Q packed
  (I and Q samples stored together in a single 32–bit word)

`-dacdatasrc` DAC data source. You can use one of the following options:

- **dma** – Data comes from its own specified DMA channel
- **ram** – Data comes from on–board DDR RAM
- **sine** – Digital data comes from test SINE signal
- **ramp** – Digital data comes from test RAMP signal

`-dacfreq` DAC clock frequency rate, in floating point format. This value should be a factor of **`brdfreq`**. Valid dividers are 1, 2, 3, 4, 6, 8 and 16.

`-dacgatedly` DAC gate tap delay

`-dacgatepol` The polarity for DAC **GATE/TRIG** signals. You can use one of the following options:

- **normal** – As received, or Rising edge
- **invert** – Inverted from received, or Falling edge

`-dacgatesrc` The source for DAC **GATE/TRIG** signals. You can use one of the following options:

- **off** – No signal is provided. A gate/trigger signal is not generated. Therefore, no DAC output is expected.
- **reg** – Internal registers are used to generate signals.
- **ttl** – Front panel **TRIG** input (SSMC connector, TTL voltage level)
- **sbusttl** – Front panel **SYNC/GATE** input (26–pin sync bus connector, TTL voltage level)
- **sbusdif** – Front panel **SYNC/GATE** input (26–pin sync bus connector, LVPECL voltage level)

A.2 List of Command–Line Arguments (continued)

`-dacgtrigmode` DAC gate, trigger, or trigger hold selection. You can use one of the following options:

- **gate** – Global gate for all channels
- **lgate** – Local gate for each channel
- **trig** – Trigger
- **trighold** – Trigger hold mode
- **timestamp** – Timestamp mode

`-dacoption` A 64–bit optional value used for selecting a special feature of some boards during DAC setup.

`-dacoutdly` DAC output tap delay

`-dacppspol` DAC **PPS** polarity. You can use one of the following options:

- **normal** – As received, or Rising edge
- **invert** – Inverted from received, or Falling edge

`-dacppssrc` The source for DAC **PPS** signals. You can use one of the following options:

- **off** – No signal is provided.
- **reg** – Internal registers are used to generate signals.
- **syncsbusttl** – TTL **SYNC** signal from SBUS connector
- **gatesrc** – Received **GATE** signal
- **syncsbusdif** – Diff **SYNC** signal from front panel 26–pin SBUS connector
- **gatesbusttl** – TTL **GATE** signal from front panel 26–pin SBUS connector
- **gatessmcttl** – TTL **GATE** signal from front panel SSMC connector

`-dacsyncdly` DAC sync tap delay

`-dacsyncpol` The polarity for DAC **SYNC** signals. You can use one of the following options:

- **normal** – As received, or Rising edge
- **invert** – Inverted from received, or Falling edge

A.2 List of Command–Line Arguments (continued)

`-dacsyncsrc`	The source for DAC **SYNC** signals. You can use one of the following options: • **off** – No signal is provided. A sync signal is not generated. Therefore, no DAC output is expected. • **reg** – Internal registers are used to generate signals. • **ttl** – Front panel **TRIG** input (SSMC connector, TTL voltage level) • **sbusttl** – Front panel **SYNC/GATE** input (26–pin sync bus connector, TTL voltage level) • **sbusdif** – Front panel **SYNC/GATE** input (26–pin sync bus connector, LVPECL voltage level) • **gatesrc** – Received **GATE** signal
`-ddc`	DDC enabled. You can use one of the following options: • **yes –** DDC enabled • **no –** DDC disabled
`-ddcgain`	DDC gain in integer value.
`-ddcinvert`	DDC spectrum inversion enabled. You can use one of the following options: • **yes –** Enable inversion. • **no –** Disable inversion.
`-ddcoption`	A 64–bit optional value used for selecting a special feature of some boards during DDC setup.
`-ddcphase`	DDC phase in degrees.
`-decim`	DDC decimation. This argument accepts the option in the form of **`<chan>:decimation`** where **`chan`** is the channel number to which the decimation applies and the decimation is the value that follows the colon (**`:`**). Note that **`<chan>`** ranges from **`1`** to **`20`**.
`-dmaoption`	A 64–bit optional value used for selecting a special feature of some boards during Direct Memory Access (DMA) setup.
`-duc`	DUC enabled. You can use one of the following options: • **yes –** DUC enabled • **no –** DUC disabled

A.2 List of Command–Line Arguments (continued)

`-ducgain`	DUC (digital upconverter) gain (integer).
`-ducinvert`	DUC spectrum inversion enabled. You can use one of the following options: • **yes –** Enable inversion. • **no –** Disable inversion.
`-ducoption`	A 64–bit optional value used for selecting a special feature of some boards during DUC setup.
`-ducphase`	DUC phase in degrees.
`-gateoption`	A 64–bit optional value used for selecting a special feature of some boards during gate setup.
`-ini`	User–specified **`.ini`** file with absolute path. This argument is only supported as a command prompt argument. It cannot be used in an **`ini`** file. For more information about **`ini`** files, see Section A.1.
`-interp`	Interpolation (integer value). This argument accepts the option in the form of **`<chan>:interpolation`** where **`chan`** is the channel number to which the interpolation applies and the interpolation is the value that follows the colon (**`:`**). Note that **`<chan>`** ranges from **`1`** to **`20`**.
`-loop`	Number of loops in decimal format. In Navigator BSP example programs, a special value of 0 implies that the program should run forever until it is manually stopped with a key hit.
`-model`	Module ID, in hex representation (for example: **`0x71131`**, **`0x71821`**, **`0x71841`**, **`0x71861`**, etc.). In the Navigator BSP example programs, it is used to find specific models of Pentek boards present in the system. Supplying a **`0`** for **`-model`** will make the BSP search for all Pentek Jade boards. In the case when multiple boards are detected, use **`-brd <n>`** for the desired board.
`-ncofreq`	NCO (numerically controlled oscillator) frequency, in floating point representation. This argument accepts the option in the form of **`<chan>:NCO freq`** where **`chan`** is the channel number to which the NCO frequency applies and the NCO frequency is the value that follows the colon (**`:`**). Note that **`<chan>`** ranges from **`1`** to **`32`**.

A.2 List of Command–Line Arguments (continued)

-numbuf	Number of buffers. In the Navigator BSP example programs, it is used as the number of DMA buffers of **xfersize** each. It employs the **numbuf** buffers to create a circular buffer for more efficient operation.
-progoption	A 64–bit optional value used for selecting a special feature for running programs.
-rdatafmt	Data file format to be read from file. Choose binary (**bin**) or ascii (**asci**). In the Navigator BSP example programs, it is used in the following way: • **bin** (binary) – The file contains samples in raw binary format and has a **.dat** extension. • **asci** (ascii) – The file contains the samples as signed integer values on separate lines and has a **.txt** extension.
-reffreq	This is an optional reference clock frequency in floating point representation. It is supplied at the **CLK** connector to help improve the accuracy of the internally generated board clock. This argument is ignored when an external clock is provided. Consult the Pentek board's hardware manual for a valid range of values. The value must be non–zero.
-rfile	File name of data to be read from. In the Navigator BSP example programs, if this is not provided, a default name will be generated by the program.
-rsize	Data size to be read from the file.
-saddr	Server host IP address
-sport	Server port number
-syncoption	A 64–bit optional value used for selecting a special feature of some boards during sync setup.
-tstart	Time Start for the timestamp. This argument should provide a calendar date in the following format: **YYYY:MM:DD,HH:MM:SS** (24–hour time). For example, **2018:12:31,11:00:01**. <u>**NOTE:**</u> This value must be in the future. In the Navigator BSP example programs, it is normally used for timestamp–based triggering.

A.2 List of Command–Line Arguments (continued)

-tstop

Time Stop for the timestamp. This argument should provide a calendar date in the following format: **YYYY:MM:DD,HH:MM:SS** (24–hour time). For example, **2018:12:31,11:59:59**.

<u>NOTE:</u> This value must be in the future and later than the value specified by **tstart**. In the Navigator BSP example programs, it is normally used for timestamp–based triggering.

-tunefreq

Tuning Frequency in floating point representation. This argument accepts the option in the form of **<chan>:Tuning freq** where **chan** is the channel number to which the tuning frequency applies and **Tuning freq** is the tuning frequency value that follows the colon (**:**). Note that **<chan>** ranges from **1** to **32**.

-vchanmask

Hex representation of the channel mask for channels to be displayed with the Signal Viewer. It supports up to 32 channels.

- 1 for channel 1
- 2 for channel 2
- 4 for channel 3
- 8 for channel 4
- 10 for channel 5
- 20 for channel 6
- 40 for channel 7
- 80 for channel 8
- FF for all 8 channels

-vhost

Host mode for the computing device where data stream is serving to the Signal Viewer. Choose one of the following:

- **lcl** (local mode) – The BSP library is set to automatically launch the Signal Viewer on the local machine.
- **rmt** (remote mode) – Manual launch of the Signal Viewer is required on the remote machine.

<u>NOTE:</u> In the Navigator BSP example program, this argument is ignored if **vport** is set to 0.

A.2 List of Command–Line Arguments (continued)

`-vport`	Signal Viewer port number in integer representation. Valid port numbers range from **`1`** to **`65534`**. However, to avoid posting conflicts with other systems in the IP network, users should use port num– bers from **`50000`** to **`65534`**. You can use one of the following options: **`0`** – Signal Viewer is not activated. **`<port number>`** – Use port numbers ranging from **`50000`** to **`65534`**.
`-vsubchan`	Sub–channel to display using the Signal Viewer. This parameter is used for the narrowband DDC and it accepts the option in the form of **`<chan>:vsubchan`** where **`chan`** is the channel number on which the sub–channel following the '**`:`**' is applied. NOTE: **`<chan>`** and **`<vsubchan>`** range from 1 to 32
`-wdatafmt`	Data file format to be written to a file. Choose binary (**`bin`**) or ascii (**`asci`**). In the Navigator BSP example programs, it is used in the following way: • **bin** (binary) – The file shall contain samples in raw binary format and will have a **`.dat`** extension. • **asci** (ascii) – The file shall contain the samples as ascii formatted signed integer values on separate lines, and the file name will have a **`.txt`** extension.
`-wfile`	File name of data to be written with captured data. In the Navigator BSP example programs, if this is not provided, default name(s) will be generated by the program.
`-wsize`	Data size to be written to a file. This value must not exceed the size specified in **`-xfersize`**. A zero value for this argument indicates no data is to be saved to a file.
`-xfersize`	Transfer size in number of bytes. In the Navigator BSP example programs, it is used as transfer size in bytes (individual buffer size). One or more buffers of **`xfersize`** each are employed by the example programs to perform DMA operations.

A.3 Using Command–Line Arguments

Following are examples of how to use command–line arguments:

`acquire`	Run the acquire program using its defaults.
`acquire -?`	List the command line options for the acquire example.
`acquire -h`	List the command line options for the acquire example.

`acquire -loop 0 -vport 50000 -vhost lcl`

Run the **`acquire`** example forever, with the Signal Viewer enabled. Local host mode is selected so the Viewer will be launched automatically on the local host machine. IP port number set to **`50000`**. The rest of arguments supported by the example remain at their default option settings.

`acquire   -chanmask 1 -xfersize 32768 -loop 100000`

Run the **`acquire`** example with the following options:

channel number = 1
transfer size = 32768 bytes
loop count = 100000

Appendix B: Navigator Signal Viewer

B.1 Introduction

The Navigator Signal Viewer allows you to monitor live input signals in both time and frequency domains. It provides signal analysis tools, including amplitude and frequency calculators, distortion calculators, averaging modes, zooming and panning controls, and cursors for exploring spectral components.

Signals displayed are "snapshot" blocks of data, processed by the Signal Viewer at its maximum rate. If the data rate into the Signal Viewer is faster than the Signal Viewer can process input samples, then input data is discarded until the Signal Viewer can accept another new block of samples. If the data rate into the Signal Viewer is slower than the Signal Viewer processing speed, then the Signal Viewer will wait for a complete block of samples before updating the Oscilloscope and Spectrum Analyzer displays, and no input data is discarded.

The Signal Viewer utility is built as a client–server application, which works with the Navigator example programs to display the data being sent via TCP/IP sockets. The Signal Viewer includes signal viewing and analysis tools with a virtual oscilloscope and a virtual spectrum analyzer. These tools allow you to monitor the live data coming out of the ADC or the DDC of the Jade series modules.

B.2 Description of the Signal Viewer Software

There are two parts to the Signal Viewer utility: the Viewer Client and Viewer Server.

B.2.1 The Viewer Client

This application is installed when the Navigator BSP is installed. It is provided as an executable under the directory of **`<NAVBSP>\bin`**, where **`<NAVBSP>`** is the base directory of the Navigator BSP. The Viewer Client is served to receive data from the Viewer Server via a TCP/IP data streaming socket connection. The data is then calculated, scaled, and displayed on the Signal Viewer window (Figure B–1), which opens when the Signal Viewer is launched.

The Client application depends on the Server to send it a View–Control word (a data structure which includes information about the data that the Server is about to send).

After the client receives the control word data information, it displays some of this information, such as the board model, channel, clock, center frequency (DDC only), etc. in the Signal Viewer window (Figure B–1). It also uses some of the data it receives from the Server to calculate the input frequency, amplitude, harmonics, etc. of the signal, and scale the signal on the time–domain and frequency–domain graphs.

B.2 Description of the Signal Viewer Software (continued)

B.2.1 The Viewer Client (continued)

For Windows and Linux platforms, this application is started automatically after the Server has initiated the TCP/IP data Streaming socket connections. This is all done inside the utility API.

B.2.2 The Viewer Server

The Viewer Server is served to initiate the TCP/IP data streaming socket con–nections to the Client application (the Signal Viewer program). Once the socket connections are established, it can send ADC or DDC data to the client for display via the socket connection. Each data buffer it sends to the Client must be accompanied by a View–Controlling data structure word, which should be sent before the data buffer.

The Viewer Server is included as part of the API utility. It consists of various functions, but mainly three high–level routines that can be called inside the user's data acquisition application. The routines can be invoked repeatedly for multiple channels of data acquisition. These functions are written in portable C source code and can be adapted under various platforms. The appropriate platform–specific socket library is linked with the server application under the individual platform in the Navigator example programs.

The three high–level routines, in the order of practical usage, are as follows:

`NAVview_OpenViewer()`

- Opens and configures all necessary TCP/IP sockets.
- Initializes and sets up basic Navigator parameters in the Viewer Control word, as required by the Viewer Client for proper display.
- Validates the Viewer Control word.
- Launches Signal Viewer Client application.
- Establishes socket connections between the Server and the Client.

`NAVview_SendData()`

- Sets up the remaining parameters of the Viewer Control word for the most up–to–date graphing criteria, such as sampling frequency, packing mode, block size, decimation (DDC only), center frequency (DDC only), etc.
- Sends display control word to the Viewer Client.
- Sends specified data block size to Viewer Client.

B.2 Description of the Signal Viewer Software (continued)

B.2.2 The Viewer Server (continued)

`NAVview_CloseViewer( )`

- Terminates the Viewer client.
- Closes all socket connections.

B.3 Sample Programming Scenario

The following is a sample programming scenario:

```
int                 *dataBuf;
void                *boardHandle;

/* Start Viewer using port number 6000 */
NAVview_OpenViewer(boardHandle,
                   NAV_CHAN_1,
                   "localhost",
                   6000);

/* Do 5000 loops */
count = 5000;
while( count-- )
{
    /* Acquire data here ...*/

    /* Send data to Viewer */
    NAVview_SendData(boardHandle,
                     NAV_CHAN_1,
                     NAV_CHAN_1,
                      dataBuf,
                     NAV_VIEW_BLK_SIZE_DEFAULT);

}

/* Close all socket connections */
NAVview_CloseViewer(boardHandle, NAV_CHAN_1);
```

NOTE: **`boardHandle`** is a handle of the board when the application opens a board using the **`NAV_BoardOpen()`** routine. See the demo example program for detail.

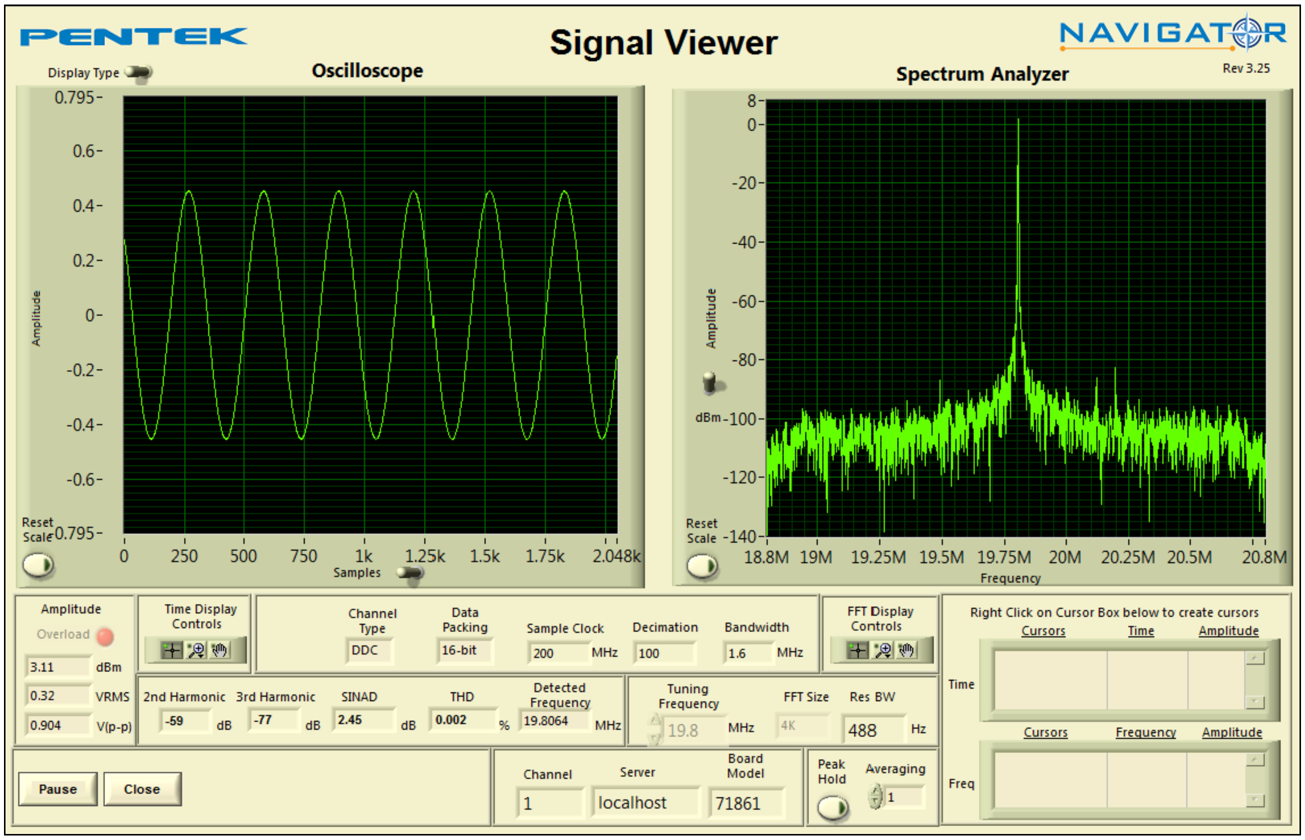
B.4 Signal Viewer Operation

The **Signal Viewer** application is installed during the Navigator installation. However, by default Signal Viewer operation is disabled because, in order to run it, you must install the LabVIEW runtime engine as described in Section B.4.

The Signal Viewer must not be started until the View Server initiates the TCP/IP data streaming socket connection in the Navigator program, as described in Section B.3. The View Client is launched as described in Section B.4.

After it is started, the Signal Viewer displays two windows: the **Oscilloscope** window on the left and the **Spectrum Analyzer** window on the right as shown in Figure B–1.

Figure B–1: Signal Viewer (Default Display)



Amplitude: Overload; 3.11 dBm; 0.32 VRMS; 0.904 V(p-p)

Channel Type	Data Packing	Sample Clock	Decimation	Bandwidth
DDC	16-bit	200 MHz	100	1.6 MHz

2nd Harmonic	3rd Harmonic	SINAD	THD	Detected Frequency
-59 dB	-77 dB	2.45 dB	0.002 %	19.8064 MHz

Tuning Frequency	FFT Size	Res BW
19.8 MHz	4K	488 Hz

Channel	Server	Board Model	Peak Hold	Averaging
1	localhost	71861		1

Right Click on Cursor Box below to create cursors

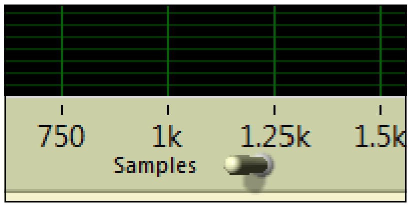
Cursors	Time	Amplitude

Cursors	Frequency	Amplitude

At the bottom of the **Oscilloscope** window is a **Samples** switch that allows you to select between samples and time (Figure B–2).

B.4 Signal Viewer Operation (continued)

Figure B–2: Samples/Time Switch



Click on the switch to change its position/setting. When the switch is set to **Samples**, the X–axis is scaled to show the signal in terms of the number of samples. When switched to **Time**, the X–axis is scaled to show the signal in terms of time in seconds.

At the top of the **Oscilloscope** window is a **Display Type** switch (Figure B–3). The Display Type switch allows you to toggle the left–side Signal Viewer window between the **Oscilloscope** display and the **Spectrogram** display. Click on the switch to change its position/setting.

Figure B–3: Display Type Switch

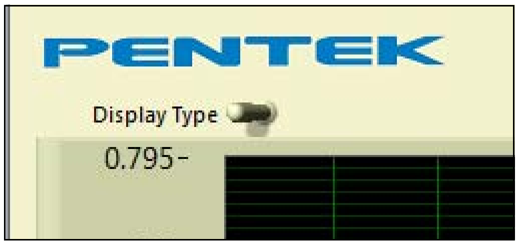
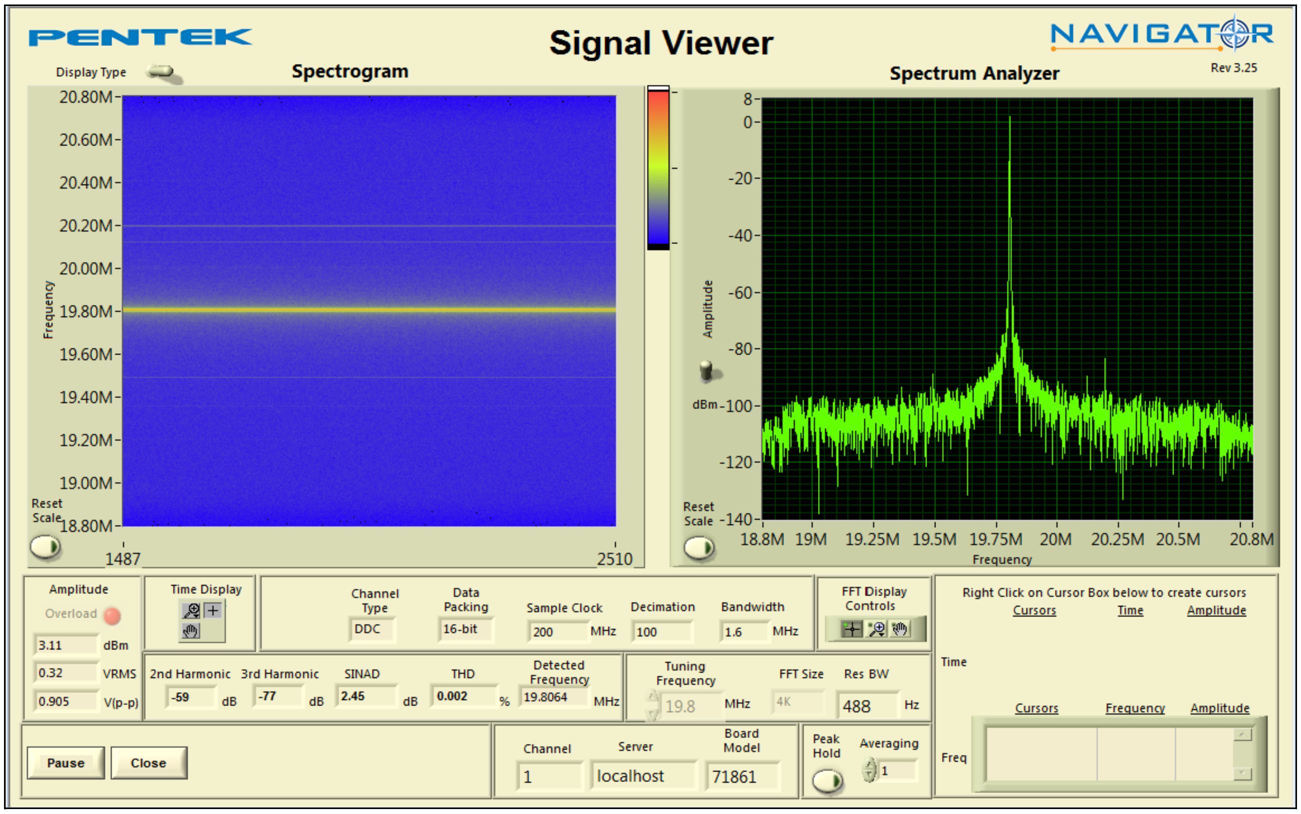


Figure B–1 shows the Signal Viewer with the **Oscilloscope** display in the left window and Figure B–4 shows the Signal Viewer with the **Spectrogram** display in the left window.

B.4 Signal Viewer Operation (continued)

When the **Display Type** switch is set to show **Spectrogram**, the left window displays a horizontal waterfall of the signal that is shown in the right window. See Figure B–4. The color corresponds to the intensity of the signal.

Figure B–4: Signal Viewer with Spectrogram Display in Left Window



Amplitude
Overload
3.11 dBm
0.32 VRMS
0.905 V(p-p)

Channel Type DDC
Data Packing 16-bit
Sample Clock 200 MHz
Decimation 100
Bandwidth 1.6 MHz

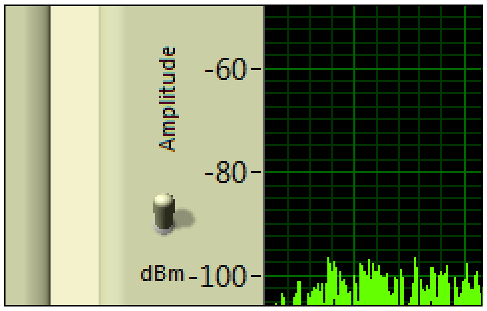
2nd Harmonic -59 dB
3rd Harmonic -77 dB
SINAD 2.45 dB
THD 0.002 %
Detected Frequency 19.8064 MHz

Tuning Frequency 19.8 MHz
FFT Size 4K
Res BW 488 Hz

Channel 1
Server localhost
Board Model 71861
Averaging 1

Regardless of what is displayed in the Signal Viewer's left window, the Signal Viewer's right window always shows the **Spectrum Analyzer** display. On the left side of the **Spectrum Analyzer** window is an **Amplitude** switch (Figure B–5) that allows you to toggle the amplitude scaling between **dBm** and **dBfs**. Click on the switch to change its position/setting.

Figure B–5: Amplitude Switch (Spectrum Analyzer Display)

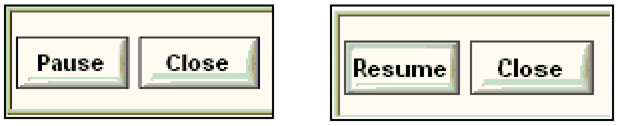


B.4 Signal Viewer Operation (continued)

B.4.1 Resume, Pause and Close

When started, the Signal Viewer begins displaying live data. Clicking on the **Pause** button in the lower left stops the live display and clicking **Resume** restarts it. The Signal Viewer is closed by clicking the **Close** button.

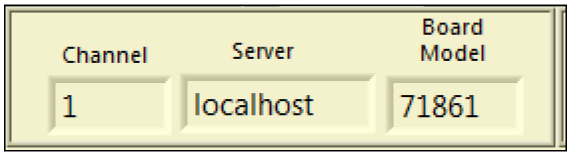
Figure B–6: Resume, Pause and Close Buttons



B.4.2 Channel, Server, and Board Model

The **Channel**, **Server**, and **Board Model** are displayed at the bottom part of the Signal Viewer window (Figure B–7). These provide information about the source of the signal you are viewing. Board Model is the model number of the Pentek Jade board being used.

Figure B–7: Channel, Server, and Board Model



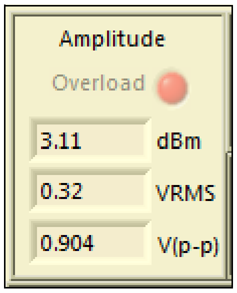
Channel	Server	Board Model
1	localhost	71861

B.4 Signal Viewer Operation (continued)

B.4.3 Amplitude Calculator

The **Amplitude** calculator window in the lower left of the Signal Viewer displays the calculated amplitude of the input signal in **dBm**, **VRMS**, and **V(p–p)**.

Figure B–8: Amplitude Calculator



3.11	dBm
0.32	VRMS
0.904	V(p-p)

dBm dBm = 20 log (V_{RMS}) – 20 log (0.2236)

where 0.2236 = the RMS Voltage of 1 mW_{RMS} into 50 Ω

V_{RMS} V_{RMS} calculates the root mean square of the input signal.

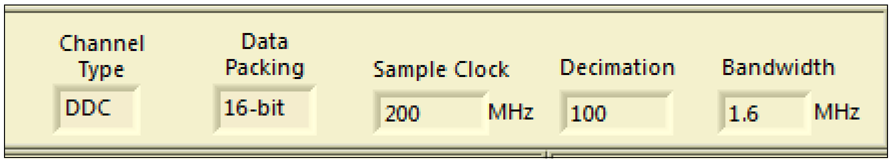
V(p–p) V(p–p) measures the difference from the most positive peak to the most negative peak.

NOTE: The **Overload** LED corresponds to the Overload LED on the Pentek board.

B.4.4 Signal Characteristics

The characteristics of the signal being viewed are summarized in the area to the right of the **Time Display Controls** and shown in Figure B–9 below. The displayed **Channel Type, Data Packing**, **Sample Clock**, **Decimation**, and **Bandwidth** values are derived from the Pentek Jade module's configuration settings and the selected signal source.

Figure B–9: Signal Characteristics



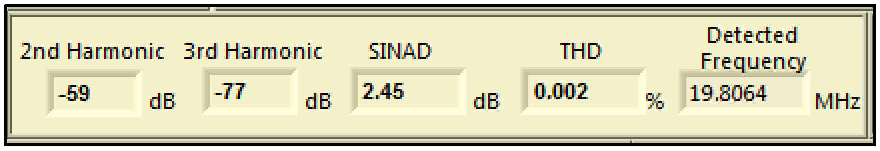
Channel Type	Data Packing	Sample Clock	Decimation	Bandwidth
DDC	16-bit	200 MHz	100	1.6 MHz

B.4 Signal Viewer Operation (continued)

B.4.5 Distortion Calculator

The distortion calculator window located to the right of the Amplitude Cal–culator displays the results of signal analysis algorithms.

Figure B–10: Distortion Calculator



2nd Harmonic	3rd Harmonic	SINAD	THD	Detected Frequency
-59 dB	-77 dB	2.45 dB	0.002 %	19.8064 MHz

- **2nd Harmonic** – Displays the level of the second harmonic component in dB relative to the fundamental signal level.

- **3rd Harmonic –** Displays the level of the third harmonic component in dB relative to the fundamental signal level.

- **SINAD –** Displays the measured signal noise and distortion (SINAD). SINAD is defined as the dB ratio of the RMS energy of all signals to the RMS energy of all signals minus the energy of the fundamental.

- **THD** – Displays the measured total harmonic distortion up to and including the highest harmonic component. THD is defined as the ratio of the RMS sum of the harmonic components to the amplitude of the fundamental signal. To compute THD as a percentage, multiply the displayed value by 100.

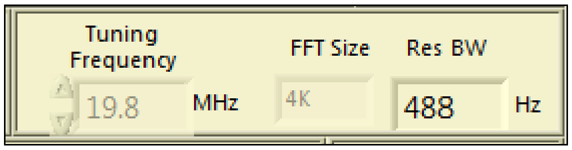
- **Detected Frequency** –Displays the calculated value based on a software frequency detector algorithm. This calculated value will not be accurate for certain signals with low levels or excessive noise.

B.4 Signal Viewer Operation (continued)

B.4.6 Tuning Frequency, FFT Size, and Resolution Bandwidth

This section of the Signal Viewer allows you to adjust the **Tuning Frequency** and **FFT Size**. The Tuning Frequency is adjusted using up and down buttons. By clicking on the FFT Size input field you can choose from several values. The resolution bandwidth calculator (**Res BW**) displays the resolution bandwidth based on the frequency bandwidth and FFT size.

Figure B–11: Tuning Frequency, FFT Size, and Res BW



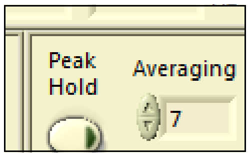
B.4.7 Spectrum Averaging

The **Averaging** control on the bottom right side of the screen provides exponential averaging of the frequency spectrum display. The number of averages is set with up and down buttons, or with direct numeric entry.

For an averaging value of N, exponential averaging sums the latest N values for each frequency display point (FFT point) and divides each value by N. The larger the value of N, the more averaging occurs.

If the **Peak Hold** button is pressed, the right–hand window will display the peak amplitudes across all the bandwidths.

Figure B–12: Averaging Control

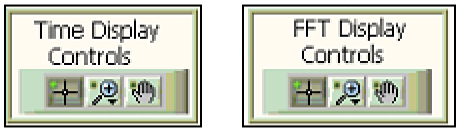


B.4 Signal Viewer Operation (continued)

B.4.8 Display Zooming

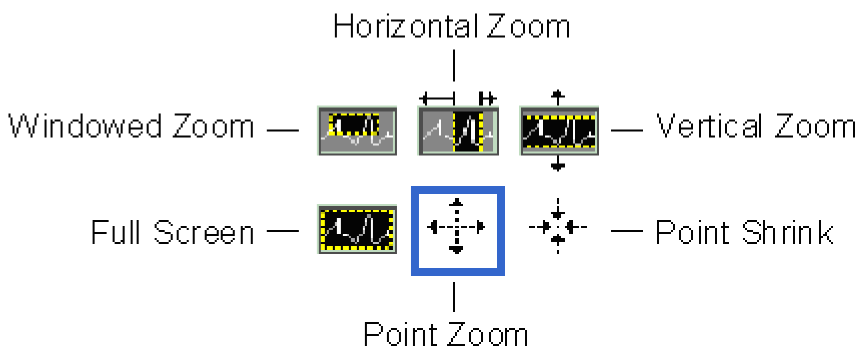
The **Time Display Controls** and **FFT Display Controls** both include **Zoom** buttons (center button) to provide tools for zooming in both the **Oscillo–scope** and **Spectrum Analyzer** display windows.

Figure B–13: Time and FFT (Frequency) Display Controls



Click on the magnifying glass **Zoom** button to invoke the zoom func–tion for either the Oscilloscope or Spectrum Analyzer display window. The Zoom menu screen appears as shown in Figure B–14 below.

Figure B–14: Time and FFT (Frequency) Zoom Controls



Click on any one of the six buttons in the menu to invoke the desired zoom functions described in the following paragraphs.

Zooming will change the horizontal or vertical scale, or both.

B.4.8.1 Horizontal Zoom

To zoom in the horizontal axis only, click the **Horizontal Zoom** button. Then place the magnifying glass cursor in the display window at the starting position for horizontal zoom, hold the left mouse button down, drag the cursor to the ending position for the horizontal zoom, and then release the mouse button.

B.4 Signal Viewer Operation (continued)

B.4.8 Display Zooming (continued)

B.4.8.2 Vertical Zoom

To zoom in the vertical axis only, click the **Vertical Zoom** button. Then place the magnifying glass cursor in the display window at the starting position for vertical zoom, hold the left mouse button down, drag the cursor to the ending position for the vertical zoom, and then release the mouse button.

B.4.8.3 Windowed Zoom

To zoom within a rectangular window, click the **Windowed Zoom** button. Then place the magnifying glass cursor in the display win– dow in one corner of the desired zoom rectangle, hold the left mouse button down, drag the cursor to the opposite corner of the desired zoom rectangle, and then release the mouse button.

B.4.8.4 Full Screen

To restore the full screen from a zoomed region, click the **Full Screen** button **Reset Scale** at the lower left of the Spectrum Ana– lyzer display window.

<u>**NOTE:**</u> The **Full Screen** function may not work properly for DDC displays, so use the **Reset Scale** button.

B.4.8.5 Point Zoom

To zoom from a single point in the display, click the **Point Zoom** button. Then place the quad outward–arrow cursor in the display window at the desired zoom position and click the left mouse but– ton. Click again for additional zooming.

B.4.8.6 Point Shrink

To shrink (unzoom) from a single point in the display, click the **Point Shrink** button. Then place the quad inward–arrow cursor in the display window at the desired shrink position and click the left mouse button. Click again for additional shrinking.

B.4 Signal Viewer Operation (continued)

B.4.9 Display Panning

The **Time Display Controls** and **FFT Display Controls** shown in Figure B–13 both include **Pan** buttons (right button) to provide panning of both dis–play windows.

Click on the **Pan** (hand) button to invoke the panning function for either the Oscilloscope or Spectrum Analyzer display window.

Then place the hand cursor in the display, hold down the left mouse button and move the display up, down, left or right. Release the mouse button when you are finished.

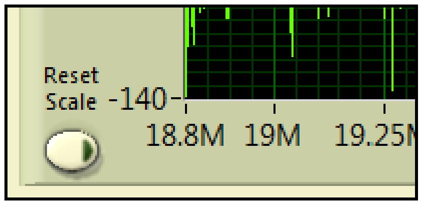
The panning operation will not change the vertical or horizontal scale.

The display can be restored by clicking the **Reset Scale** button.

B.4.10 Reset Scale – Oscilloscope and Spectrum Analyzer Displays

After panning and zooming in the Oscilloscope and Spectrum Analyzer dis–plays, you can quickly restore the initial default scaling, zoom and panning by clicking the **Reset Scale** buttons at the lower left corner of each display.

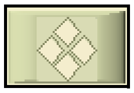
Figure B–15: Reset Scale Button



B.4.11 Cursor Operation

The buttons beneath the cursor legend (Figure B–16) can be used to control the position of the cursors. You can move the cursors vertically and horizon–tally by using the appropriate buttons. Moving the cursors by dragging them on the graph is also possible but the buttons provide fine–grained control and are helpful in snapping the cursors to points of interest (e.g., peaks).

Figure B–16: Cursor Position Adjustment Buttons



NOTE: Vertical movement is not available for multi–plot cursors because they do not track the X axis (i.e., amplitude).

B.4 Signal Viewer Operation (continued)

B.4.11 Cursor Operation (continued)

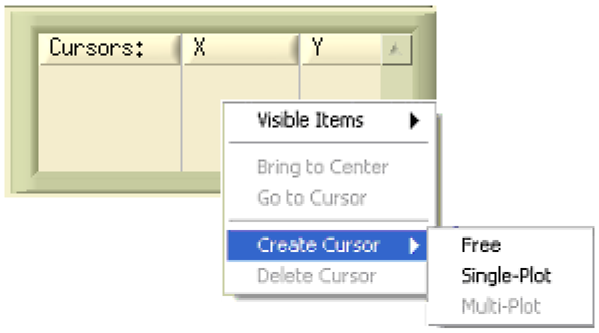
The **FFT Display Controls** shown in Figure B–13 includes the **Cursor** button (left button) which provides multiple cursors for exploring and identifying features in the **Spectrum Analyzer** display window.

NOTE: There is no cursor support for the Oscilloscope display window.

Click on the **Cursor** (crosshairs) button to invoke the Cursor function, to ensure that both **Zoom** and **Pan** functions are disabled.

Then right–click the mouse in the Cursors window at the lower left of the screen to bring up the Create Cursor menu as shown in Figure B–17.

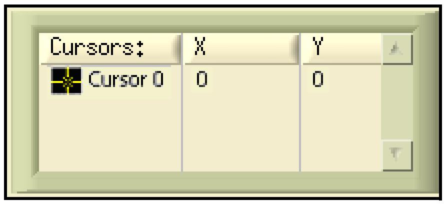
Figure B–17: Create Cursor Menu



Two types of cursors are available: **Free** and **Single–Plot**. **Free** cursors can be moved freely anywhere on the screen in horizontal and vertical direc–tions. **Single–Plot** cursors can be moved anywhere on the screen in the hor–izontal dimension, but the vertical position tracks the vertical value of the display plot.

By clicking on the **Free** cursor menu selection, a new entry is added to the cursor window as shown below in Figure B–18.

Figure B–18: New Free Cursor 0



Cursors:	X	Y
Cursor 0	0	0

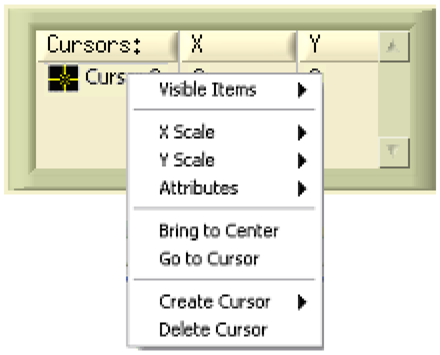
B.4 Signal Viewer Operation (continued)

B.4.11 Cursor Operation (continued)

Cursor 0 is a **Free** cursor that starts out with a vertical line at 0 Hz (at the left side of the Spectrum Analyzer display) and a horizontal line at 0 dB (at the top of the Spectrum Analyzer display). Notice that the frequency and amplitude values of the cursor are displayed in the cursor window.

By right clicking on **Cursor 0**, its properties can be displayed along with specific settings for that cursor as shown in Figure B–19.

Figure B–19: Cursor 0 Properties



For example, by clicking on **Bring to Center**, Cursor 0 is centered horizon–tally and vertically on the display, and the values in the cursor window are updated accordingly.

Because Cursor 0 is a free cursor, the frequency value can be changed by moving the vertical line with the mouse or entering a new frequency value in the cursor window, and the amplitude value can be changed by moving the horizontal line with the mouse or entering a new amplitude value in the cursor window.

Both horizontal and vertical values of a free cursor can be moved by moving the intersection crosshairs of the horizontal and vertical cursor lines with the mouse.

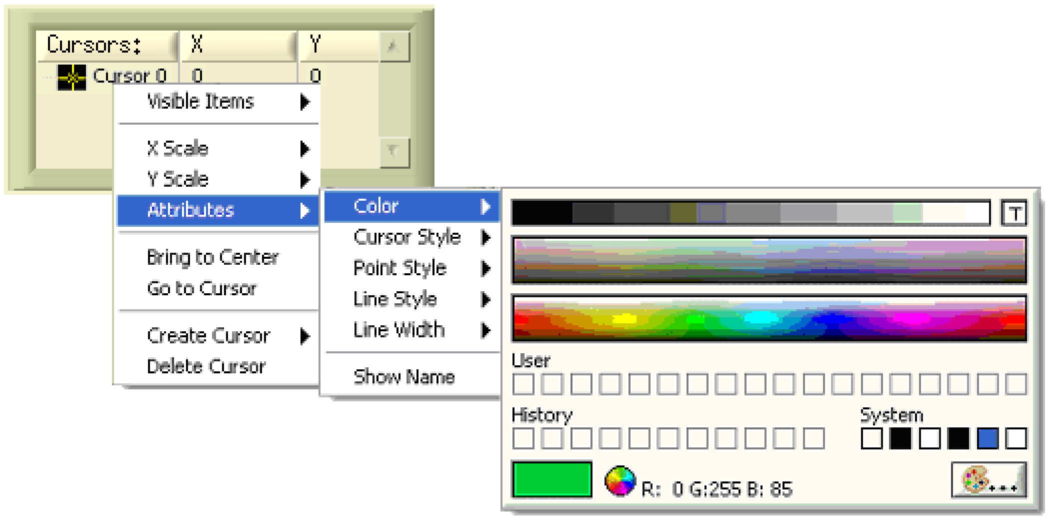
NOTE: Be sure the Cursor button in the FFT Display Controls window is selected (highlighted in dark gray) in order to move the cursor.

B.4 Signal Viewer Operation (continued)

B.4.11 Cursor Operation (continued)

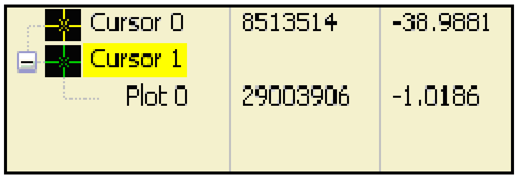
By clicking on the **Attributes** property, many different features of the cursor can be customized, such as the color of the cursor lines, as shown below.

Figure B–20: Cursor 0 Attributes



By right clicking on **Create Cursor** and then **Single–Plot**, a new Single–Plot cursor is created as **Cursor 1**, as shown below.

Figure B–21: Cursor Window with Single Free Cursor and Single–Plot Cursor



Cursor 0	8513514	-38.9881
Cursor 1		
Plot 0	29003906	-1.0186

Because Cursor 1 is a **Single–Plot** cursor, its frequency value can be changed by moving the vertical line with the mouse or by entering a new frequency value in the cursor window, but the amplitude value will track the amplitude of Plot 0 and is not user adjustable.

This feature can be useful for reading the amplitude of frequency compo–nents. In some cases, the peak value of a plot may be discovered by zooming the plot in the frequency scale to improve the horizontal resolution.

Appendix C: DMA Callbacks

C.1 Background and Usage

Direct Memory Access (DMA) is a capability provided by PCI bus architecture which allows direct data transfer between a connected device, such as a Pentek board, and the memory on the host computer, freeing the CPU from involvement with the data transfer and thus improving the host's performance. This is one of the most commonly employed features in Navigator−based user applications.

To ease the task of managing DMA operations across several channels in either direction, a framework is built within the BSP that provides asynchronous callbacks to a user−supplied handler function. When DMA engine is being set up for a channel and a non−null callback function pointer is provided, the library creates a dedicated DMA thread for that channel. The library also attaches its own interrupt handler for interrupts generated by the DMA hardware in the device. The DMA thread and interrupt handler work in conjunction to provide easy access to DMA buffers in user application and monitor breakdown in real−time performance.

The end result is that the users don't need to worry about maintaining concurrent threads or handling hardware interrupts. User code can concentrate on managing the data coming out of or going into the hardware. The Navigator BSP will take care of the tedious tasks in the background. This mechanism is similar to the event−driven programming paradigm of GUI applications. It is also analogous to signal handlers in C and even to interrupt handlers. To look at a typical usage scenario of this mechanism and corresponding API functions, please refer to Sections 7.5 (Set Up Board Resources) and 7.8 (Manage Data Transfer).

The callback mechanism is built around the PCIe DMA core's linked list descriptors in the FPGA. When DMA operation is being set up on a channel, the user provides the information required to create a ring buffer of sufficient size in the host RAM. Individual links of the DMA descriptor list are programmed to form equal−sized segments of this ring buffer. The last segment points to the first segment, forming a ring. The size of each segment and the number of such segments is specified by the user during setup.

The library enables an interrupt for each segment such that a hardware interrupt is generated whenever the DMA core has finished writing to a buffer segment (A/D application) or reading from a buffer segment (D/A application). The DMA core may also fill buffer segments partially in case the acquisition gate goes down before the segment is totally full. For this reason, we recommend that you enable metadata generation while setting up the DMA and use that metadata to obtain the number of valid bytes in each buffer segment.

C.1 Background and Usage (continued)

Under normal circumstances, a DMA interrupt is received when ADC data is available in a buffer. The library's DMA thread calls the user–supplied callback function with appropriate arguments so that the user application can make use of the available data. This frees up the application from the burden of creating concurrent threads and syn–chronizing them. The user's callback function is executed in a separate thread for each channel in use.

User code should still take precautions to synchronize accesses to shared resources from within the DMA callback handler. All hardware interrupts (including those from DMA core) can be used alongside the callback function in the user application. If the application has explicitly enabled DMA core interrupts along with the callback mecha–nism, the user's DMA interrupt handler will be invoked first and then a DMA callback will be provided as needed.

This callback mechanism also keeps track of undesirable conditions which degrade or break down real–time performance, such as FPGA FIFO being capped or buffer over–run/underrun. When such a condition is encountered, the data becomes unreliable and the DMA thread communicates these conditions to the user application for further action.

DMA callbacks can be set up in two operating modes:

- The first, and more commonly used mode is **`CONTINUOUS_LOOP`**. In this mode, the DMA runs continuously and callbacks are provided until stopped explicitly. This mode utilizes the ring buffer to provide continuous performance.

- The second mode is **`SINGLE_CHAIN`**. In this mode, the hardware pauses DMA automat–ically after executing the last DMA descriptor link. This is useful when precise control is required on the amount of data transfer. Only one set of data buffers is transferred on each run. The application can still restart the DMA after the end of the chain, if needed.

We will look at a few common use cases to understand how to set up the DMA callback framework. These scenarios are primarily meant to demonstrate the capabilities and usage of this framework. The solutions described in these scenarios are somewhat the–oretical and a real–world application will have greater complexity. Nevertheless, the same approaches can be used in more sophisticated applications.

C.2 Scenario 1: Signal Recorder

In this scenario, the requirement is to save all the signals received by an antenna. Assume that the signal from the antenna is available at an input connector on a Jade board's front panel. The application should continuously acquire and store data to disk. The storage configuration on the system provides an effective throughput of 450 MB/s. The ADC on a Jade 71861 board is configured to sample at 200 MSPS in 16–bit real data format, resulting in a data rate of 400 MB/s. After running storage benchmarks, it was found that, in the worst case, disk write throughput can briefly plummet to 100 MB/s because of the varying system load.

As is evident from the numbers above, some kind of buffering is needed for reliable signal acquisition. The DMA core can be configured to store ADC data in a ring buffer composed of '**n**' segments in host RAM, and as soon as a segment is filled, the application will copy it to disk. The value of '**n**' is determined by the worst–case disk performance.

Let's fix the size of each buffer segment as 4 MB (4 million bytes), assuming that the degraded write speeds of 100 MB/s last for 0.04 seconds or less. There should be enough buffers such that the DMA does not wrap around and overwrite a buffer segment while it is being copied to disk. In the ideal case, with a write speed of 450 MB/s, the buffer segments will be copied to disk in less time than it takes for the DMA core to fill them with ADC data. Each buffer segment will get copied to disk slightly before the next one is filled.

However, in the worst case, the disk write speed is a quarter of the incoming data rate. By the time the current buffer segment is saved on the disk, the DMA core would have completely filled the next four segments and moved onto the fifth one. If there are any less than five buffer segments, the data stored on the disk would not be reliable. Also, we need extra buffers to provide the quicker disk writes some time to catch up with the DMA core. The number of such extra buffers can be found through basic arithmetic:

```
(5 buffers * 4 MB) + 400 MB/s * T seconds = 450 MB/s * T seconds
T = 20/50 = 0.4 seconds; equivalent to 160 MB of data at 400 MB/s.
That 160 MB of data can be stored in 40 buffer segments of 4 MB each.
```

Thus, at least 45 buffer segments of 4 MB each are necessary to completely recover from occasional degradation of performance on this system. If the disk write speed degrades too often or if the maximum possible disk write speed is less than the incoming data rate, continuous data acquisition will not be feasible.

If a segment in the ring buffer is about to be (or has already been) overwritten by the DMA core before being processed by the application's DMA callback handler, the status code will have its **NAV_STAT_DMA_OVERRUN** bit set.

C.2 Scenario 1: Signal Recorder (continued)

A performance bottleneck could also manifest in the hardware. If the host machine is not able to maintain the required data rate when interacting with the PCIe device, the device FIFOs will eventually be used up to capacity. For example, if the PCIe link is not fast enough, the rate of getting data out of the device to host RAM will be slower than the rate at which it is being generated/sampled. The device will then end up with a totally full output FIFO. When this happens, the DMA status code will have its **`NAV_STAT_DMA_FIFO_CAPPED`** bit set.

The API call to set up DMA in Navigator for this scenario will be as shown below:

```
/* DMA
 * Set DMA Channel 1 in Continuous mode, acquiring 4 MB of data per buffer
 * segment with 45 such buffers in use. */
status =
NAV_DmaSetup(boardHandle,
             NAV_CHANNEL_TYPE_ADC,         /* ADC DMA*/
             NAV_CHAN_1,                        /* Channel being configured */
             45,                                /* Number of buffers */
             4.0e6,                             /* Buffer size */
             NAV_DMA_METADATA_ENABLE,           /* FPGA will provide metadata */
             NAV_DMA_RUN_MODE_CONTINUOUS_LOOP,  /* Operating mode */
             NAV_SYS_WAIT_STATE_MILSEC(15000),  /* Timeout period */
             &dmaCallbackHandler,               /* DMA callback handler */
             boardHandle,                       /* Data pointer for the handler */
             0);                                /* Options (none needed) */


/* Implementation of a DMA callback handler for this scenario.
 * When the hardware has filled up a DMA buffer, the user code
 * can start using that data from this function. */
void dmaCallbackHandler (int32_t channel, int32_t dmaStatus,
                         void *dataBuffer, void *metaDataBuffer,
                         void *userData)
{
    /* If DMA buffer segment has been filled without missing any samples,
     * and has not been overwritten, save it to disk. */
    if( (dmaStatus & NAV_STAT_DMA_LINK_END)  &&
       !(dmaStatus & NAV_STAT_DMA_OVERRUN)   &&
       !(dmaStatus & NAV_STAT_DMA_FIFO_CAPPED) )
    {
        /* Save buffer segment to disk */

        //...

    }
 }
```

C.3 Scenario 2: Single Snapshot

In this scenario, the requirement is to save one second of a continuous signal from four ADC channels for offline analysis. Assume that all signals are available at input connectors on a Jade board's front panel. The application should store one second worth of data from each channel onto the disk. The storage configuration is composed of commodity–grade solid state drives (SSDs) in RAID0 configuration providing an effective throughput of 1200 MB/s with worst–case performance of 800 MB/s. All four ADCs on a Jade 71861 board are configured to sample at 200 MSPS in 16–bit real data format, resulting in a combined data rate of 1600 MB/s.

Buffering of DMA data is necessary to ensure that all of it is reliably saved to the disk. With individual buffer segments of 4 MB (4 million bytes), 100 such buffers must be acquired on each channel to have one second of signal. The application must try to minimize its memory footprint as much as possible. Assuming a consistently worst–case disk write speed, the minimum number of buffers on each channel can be calculated as follows:

```
N = BURST_LENGTH * (1 - (Fout/Fin))
N = 100 * (1- 800/1600) = 50
```

Thus, at least 50 buffer segments of 4 MB each are needed. Since the disk write speed is half the incoming data rate, by the time the 50th buffer segment is being saved on the disk, the DMA core would have filled each of the 50 buffer segments twice (of which only the first pass is saved on disk). To prevent the unsaved buffer segments from being overwritten, it is necessary to stop the DMA when it has transferred exactly 100 buffer segments.

An interrupt handler must be used to stop the DMA in a timely manner. The DMA link–end interrupt is enabled and associated with a handler routine that counts the number of invocations for each channel separately and stops DMA on the 100th invocation for each channel. To account for the delay between the interrupt coming in and the DMA actually stopping, a few more buffers should be added.

The API calls to set up DMA in Navigator for this scenario will be as shown below:

C.3 Scenario 2: Single Snapshot (continued)

```
for(chan = NAV_CHAN_1; chan < NAV_CHAN_4; chan++)
{

/* DMA
 * Set DMA Channels in Continuous mode, acquiring 4 MB of data per buffer
 * segment with 55 such buffers in use. */
status =
NAV_DmaSetup(boardHandle,
             NAV_CHANNEL_TYPE_ADC,         /* ADC DMA*/
             chan,                              /* Channel being configured */
             55,                                /* Number of buffers */
             4.0e6,                             /* Buffer size */
             NAV_DMA_METADATA_ENABLE,           /* FPGA will provide metadata */
             NAV_DMA_RUN_MODE_CONTINUOUS_LOOP,  /* Operating mode */
             NAV_SYS_WAIT_STATE_MILSEC(15000),  /* Timeout period */
             &dmaCallbackHandler,               /* DMA callback handler */
             boardHandle,                       /* Data pointer for the handler */
             0);                                /* Options (none needed) */



/* Enable the interrupt for counting buffer transfers.
 * Do this while setting up the board resources. */
status = NAV_InterruptEnable(boardHandle,
                            NAV_INTR_DATA_ACQ_ADC_DMA, chan,
                            NAV_IP_DMA_PPKT2PCIE_INTR_LINK_END_INT,
                            &dmaLinkIntrHandler, NULL);
}


/* Interrupt handler for LinkEnd event.
 * This function will be executed by the driver in a separate thread,
 * when the hardware raises the interrupt associated with this handler.
 */
void dmaLinkIntrHandler(void              *hDev,
                        int32_t            intSource,
                        int32_t            instance,
                        uint32_t           intFlag,
                        int32_t            numInterrupts,
                        int32_t            numLostInterrupts,
                        void              *pData)
{
    void *boardHandle = pData;

    if (intFlag & NAV_IP_DMA_PPKT2PCIE_INTR_LINK_END_INT)
```

The above code example continues on the next page.

C.3 Scenario 2: Single Snapshot (continued)

The code example below is continued from the previous page.

```
    {
        if (++loopCount[instance] == 100)
        {
           NAV_DmaStop(boardHandle,
                       NAV_CHANNEL_TYPE_ADC, instance);
        }
    }
}

/* Implementation of a DMA callback handler for this scenario.
 * When the hardware has filled up a DMA buffer, the user code
 * can start using that data from this function.
 * To avoid code duplication, only one callback handler is used for all
channels
 * with separate state variables. If required, a different handler function
 * can be used for each channel.*/
void dmaCallbackHandler (int32_t channel, int32_t dmaStatus,
                         void *dataBuffer, void *metaDataBuffer,
                         void *userData)
{
    /* If DMA buffer segment has been filled, save it */
    if((dmaStatus & NAV_STAT_DMA_LINK_END) && !captureDone[channel])
    {
        /* Save buffer segment to disk */

        //...

        /* Increment the counter */
        ++counter[channel]

        /* Capture is complete if 100 segments have been saved to disk */
        if(counter[channel] == 100)
            captureDone[channel] = 1;
    }
}
```

C.4 Scenario 3: Repeated Snapshots

Like the previous scenario, the requirement for this scenario is to save one second of a continuous signal from four ADC channels for offline analysis. A different algorithm is used to analyze data from each channel independently. After the analysis is done, it must be re–run 10 times to average out the results. Assume that calibrated test signals are available at input connectors on a Jade board's front panel.

The memory footprint of the application is not a constraint. Deviating from the approach taken in the previous scenario, 100 buffer segments will be used per channel such that all the required data is available in the memory. This will greatly simplify the implementation as well. Bear in mind that there are limits on the maximum number of PCIe DMA core linked list descriptors in the FPGA and the maximum buffer size that can be successfully allocated by the operating system on the host machine.

Instead of relying on interrupt handlers, the DMA should be configured in **`SINGLE_CHAIN`** mode. This way, the hardware itself will pause the DMA after transferring 100 buffers. The application can then act on the data without any concern for an over–run condition. After an algorithm has analyzed all the data for a channel, the DMA for that channel can be restarted. Each channel can proceed at its own pace since the DMA restarts are channel–specific.

The API call to set up DMA in Navigator for this scenario will be as shown below:

```
for(chan = NAV_CHAN_1; chan <= NAV_CHAN_4; chan++)
{
/* DMA
 * Set DMA Channels in Single Chain mode, acquiring 4 MB of data per buffer
 * segment with 100 such buffers in use. */
status =
NAV_CHANNEL_TYPE_ADC,                     /* ADC DMA*/
             chan,                   /* Channel being configured */
             100,                              /* Number of buffers */
             4.0e6,                            /* Buffer size */
             NAV_DMA_METADATA_ENABLE,          /* FPGA will provide metadata */
             NAV_DMA_RUN_MODE_SINGLE_CHAIN,    /* Operating mode */
             NAV_SYS_WAIT_STATE_MILSEC(15000), /* Timeout period */
             &dmaCallbackHandler,              /* DMA callback handler */
             boardHandle,                      /* Data pointer for the handler */
             0);                               /* Options (none needed) */


  }
```

The above code example continues on the next page.

C.4 Scenario 3: Repeated Snapshots

The code example below is continued from the previous page.

```
/* Implementation of a DMA callback handler for this scenario.
 * When the hardware has filled up all DMA buffers, the user code
 * can start using that data from this function.
 * To avoid code duplication, only one callback handler is used for all channels
 * with separate state variables. If required, a different handler function
 * can be used for each channel. */
void dmaCallbackHandler (int32_t channel, int32_t dmaStatus,
                         void *dataBuffer, void *metaDataBuffer,
                         void *userData)
{
    /* If all DMA buffer segments have been filled, run the analysis */
    if(dmaStatus & NAV_STAT_DMA_CHAIN_END)
    {
        /* Run the analysis algorithm directly on the data in memory */

        //...

        /* Restart the chain */
        NAV_DmaAdvance(boardHandle, NAV_CHANNEL_TYPE_ADC, channel);
    }
}
```

C.5 Scenario 4: RADAR Receiver

Applications like radar systems require complex chains of acquisition periods and delays. Depending on the hardware capabilities, the acquisition gate can be generated internally in response to a trigger or it can be supplied as an external signal to the board. For more information on API calls for gate/trigger, refer to the documentation in the **`nav_gatetrig.c`** file.

The requirement for this scenario is to acquire data according to a set scheme and verify that no deviation has taken place. The gate signal will be generated externally via separate hardware.

The acquisition scheme is as follows:

1) Acquire 8192 16–bit real data samples on ADC channel 1.

2) Wait 10,000 clock cycles.

3) Acquire 16384 16–bit real data samples on ADC channel 1.

4) Wait 50,000 clock cycles.

5) Acquire 32768 16–bit real data samples on ADC channel 1.

6) Verify acquisition and wait for the next cycle.

Two features of the DMA core will be very useful here. First, the DMA core can be configured to generate an interrupt for partially filled buffers if the acquisition gate goes down. This mode is always enabled by the Navigator BSP while setting up DMA. If this mode is not enabled, the DMA core will wait for the acquisition gate to go up again such that it can fill the buffer and then generate an interrupt.

Second, the DMA core can optionally collect useful information about the ADC data transferred during the last DMA and automatically DMA this information to the host processor in a separate buffer. The user application can enable this feature by providing appropriate arguments while setting up DMA. When a callback is provided to the user code, pointers for both data buffers and metadata buffers are available as arguments in the callback handler. Each data buffer segment has an associated metadata buffer.

Information included in the meta data packet such as the channel number and number of bytes transferred as well as the timestamps can aid the user application in processing received data. Of particular concern to us in this scenario are the two fields that hold the number of valid bytes transferred and the sample clock count portion of the time–stamp.

Based on the acquisition scheme, we see that the largest chunk of samples to acquire is 32768 16–bit samples, so we will use three DMA buffers of 65536 bytes each. The DMA core will fill the buffers only when the acquisition is active and it will also stop the transfer midway if the gate goes down. Since the waiting period between successive cycles is arbitrary or unknown, we will not set the DMA to time out.

C.5 Scenario 4: RADAR Receiver (continued)

Apart from the snippet for DMA setup shown below, there would be some API calls required for setting up the ADC and gate/trigger.

The API call to set up DMA in Navigator for this scenario will be as shown below:

```
/* DMA
 * Set DMA Channel 1 in Continuous mode, acquiring maximum 64 KiB of data
 * per buffer segment with 3 such buffers in use. */
status =
NAV_DmaSetup(boardHandle,
             NAV_CHANNEL_TYPE_ADC,         /* ADC DMA*/
             NAV_CHAN_1,                        /* Channel being configured */
             3,                                 /* Number of buffers */
             65536,                             /* Buffer size */
             NAV_DMA_METADATA_ENABLE,           /* FPGA will provide metadata */
             NAV_DMA_RUN_MODE_CONTINUOUS_LOOP,  /* Operating mode */
             NAV_SYS_WAIT_STATE_FOREVER,        /* Timeout period */
             &dmaCallbackHandler,               /* DMA callback handler */
             boardHandle,                       /* Data pointer for the handler */
             0);                                /* Options (none needed) */


/* Implementation of a DMA callback handler for this scenario.
 * When the hardware has filled up a DMA buffer, the user code
 * can start using that data from this function.
 * The metadata can be used to verify acquisition length and delay. */
void dmaCallbackHandler (int32_t channel, int32_t dmaStatus,
                         void *dataBuffer, void *metaDataBuffer,
                         void *userData)
{
    void *boardHandle = userData;
    NAV_DMA_ADC_META_DATA *metaData = (NAV_DMA_ADC_META_DATA *) metaDatabuffer;


    /* If DMA buffer segment has been filled, save it */
    if((dmaStatus & NAV_STAT_DMA_LINK_END) && !captureDone[channel])
    {
        /* Check the metadata
         * validBytes should cycle between 16384, 32768 and 65536
         * ClockCount between successive callbacks should equal the delays. */
        printf("MetaData ValidBytes %u\n", metaData->validBytes);
        printf("MetaData ClockCount %u\n", metaData->timestampClockCount);

        //...
    }
}
```

C.6 Scenario 5: Custom Implementation

Using the DMA callback framework present in the Navigator BSP greatly simplifies general application development but it introduces latency that may be undesirable in certain use cases.

If the capabilities provided by the DMA callback framework are not sufficient, or if an application requires a predictable interrupt response time, then it is best to bypass the callback framework entirely by directly handling DMA interrupts and data buffers. This requires that DMA threads as well as the universal interrupt handler used inside the library are bypassed. This ends up being the more traditional approach, which may be more familiar to some users but also requires greater effort in getting the task done.

The easiest way to achieve this is to call the **NAV_DmaSetup()** routine with the **dmaCallback** argument as NULL and then override the interrupt handler by directly calling the driver–level routine. When dmaCallback is set to NULL, DMA threads are not started and thus no callbacks will be provided. However, inside this setup routine the library automatically attaches its own interrupt handler, which must be overridden by directly calling the driver–level routine **NAV718X_intEnable()**. Calling the library routine **NAV_InterruptEnable()** is not enough because that routine just makes sure that the DMA interrupts are forwarded from the library's universal interrupt handler to the user–specified interrupt handler.

Note that with a NULL callback, even though the library will not create any threads, the DMA descriptors will still be set up to form a ring buffer, as described in Section C.1.

The API call to set up DMA in Navigator for this scenario will be as shown below. The example continues on the next page.

```
/* DMA
 * Set DMA Channel 1 in Continuous mode, acquiring 4 MB of data per buffer
 * segment with 100 such buffers in use. */
status =
NAV_DmaSetup(boardHandle,
             NAV_CHANNEL_TYPE_ADC,              /* ADC DMA*/
             NAV_CHAN_1,                        /* Channel being configured */
             100,                               /* Number of buffers */
             4.0e6,                             /* Buffer size */
             NAV_DMA_METADATA_ENABLE,           /* FPGA will provide metadata */
             NAV_DMA_RUN_MODE_CONTINUOUS_LOOP,  /* Operating mode */
             NAV_SYS_WAIT_STATE_MILSEC(15000),  /* Timeout period */
             NULL,                              /* No DMA callback needed */
             NULL,                              /* No Data pointer */
             0);                                /* Options (none needed) */


/* If needed, use IP-core level routines to reconfigure the DMA Linked List
 * Descriptors. The implementation of NAV_DmaSetupDescriptorList() in the
 * nav_dma_common.c file can be used as a reference. */
```

C.6 Scenario 5: Custom Implementation (continued)

The code example below is continued from the previous page.

```
/* Override the interrupt handler for the DMA link-end event by calling a
 * driver-level routine. */
status = NAV718X_intEnable(((NAV_BOARD_RESRC *)boardHandle)->pciInfo.hDev,
                            NAV_INTR_DATA_ACQ_ADC_DMA, NAV_CHAN_1,
                            NAV_IP_DMA_PPKT2PCIE_INTR_LINK_END_INT,
                            NULL, &dmaLinkIntrHandler);


/* Interrupt handler for LinkEnd event.
 * This function will be executed by the driver in a separate thread,
 * when the hardware raises the interrupt associated with this handler. */
void dmaLinkIntrHandler(void                *hDev,
                        int32_t             intSource,
                        int32_t             instance,
                        uint32_t            intFlag,
                        int32_t             numInterrupts,
                        int32_t             numLostInterrupts,
                        void               *pData)
{
    void *boardHandle = pData;

    if (intFlag & NAV_IP_DMA_PPKT2PCIE_INTR_LINK_END_INT)
    {
        /* Take action when a buffer segment has been filled
         * (e.g. release a semaphore through NAVsys_SemaphorePost() function) */

        // ...
    }
}
```

C.7 Advanced Implementation Details

This section sheds some light on how the DMA callback mechanism has been imple– mented within the library. This information may prove useful while modifying the library or while debugging a problem.

When the DMA callback framework is active, code is being executed in three threads:

- First, there is the main application thread.
- Second, there is an interrupt handler thread, created for each application by the device driver, which services incoming interrupts in a queue. All the interrupt han– dler routines for the application are synchronously invoked by this interrupt thread in the driver. An interrupt handler specifically for DMA–related interrupts is present in the BSP library. This handler is called **`NAV_DmaInterruptHandler()`**.
- Finally, there is a channel–specific DMA thread, **`NAV_DmaThread()`**, created within the Navigator BSP library. When the DMA thread is launched, it starts to wait for a request for action from any other thread.

User applications and interrupt handlers can communicate with DMA threads using message queues. Each DMA thread has its own message queue. A special routine, named **`NAV_DmaDispatchThreadCommand()`**, is available in the library to send messages to a particular DMA thread. The BSP library uses this routine to send messages to DMA threads on behalf of the user application. Normally, there would be no need to invoke this function from code outside the BSP.

At present, the communication with DMA threads is only one–way and utilizes the POSIX.1 and Win32 API on Linux and Windows, respectively. However, DMA threads do acknowledge the receipt of certain messages by releasing a semaphore. The mini– mum recommended depth for these message queues is 2048 on Linux and 20000 on Windows. Default queue depths set by the operating system are 10 on Linux and 10000 on Windows.

Queue depth can be changed under Linux by writing to:

```
/proc/sys/fs/mqueue/msg_max
```

This must be done every time the system reboots. The installer for Navigator BSP on Linux provides an **`init`** script that attempts to do this during boot up. If the installer is not used or if it fails for some reason, the limit must be raised manually (as a super– user).

Linux imposes another limit on the maximum memory allowed for use by POSIX mes– sage queues. A low value of this limit will prevent multiple DMA threads from running simultaneously.

C.7 Advanced Implementation Details (continued)

Copy the following two lines to your **`/etc/security/limits.conf`** file to raise the default limit.

```
*       hard    msgqueue        4915200
*       soft    msgqueue        4915200
```

The number in the last column can be changed if needed (the value mentioned above should be enough to support about 20 DMA threads). Consult the 'ulimit' man page or the limits.conf file for further information. The installer for Navigator BSP on Linux will modify the limits.conf during installation. If the installer is not used, the limit must be raised manually (as a superuser). It is necessary to log out and log back in for the changes in limits.conf to take effect.

Under Windows, the queue depth can be changed through the registry key:

```
HKEY_LOCAL_MACHINE
 SOFTWARE
  Microsoft
   Windows NT
    CurrentVersion
     Windows
      USERPostMessageLimit
```

Like other registry changes, this change is permanent and need not be repeated after reboot. Be careful while changing this limit because a value less than 4000 can result in system errors and an inability to log in. Always take a backup of the registry prior to changing any keys in it.

If the queue depth is insufficient (too many messages are being sent in rapid succes– sion), some of the messages will get dropped and cause malfunctions in DMA threads. This scenario is detected in the DMA interrupt handler and a notification is displayed on the console.

Taking the ADC as an example, when DMA is in progress and one of the buffers is filled, an interrupt is generated by the hardware. This interrupt is serviced by the driver and the library's DMA interrupt handler is invoked. If the user has also enabled DMA interrupts, the library's DMA interrupt handler calls the user's interrupt handler before proceeding. After returning from user code, the library's DMA interrupt handler sends an asynchronous message to the DMA thread regarding the buffer being filled and then quickly exits.

The DMA interrupt handler does not wait for an acknowledgment of message receipt by the DMA thread. Apart from the messages sent by the DMA interrupt handler, all other messages are acknowledged by the DMA thread. Code sending non–interrupt messages is blocked until the DMA thread acknowledges the receipt.

C.7 Advanced Implementation Details (continued)

To avoid a deadlock, do not call **`NAV_DmaStart()`** or **`NAV_DmaStop()`** from within the DMA callback handler, because then the thread would start waiting on a semaphore released by the thread itself. The interrupt handler and message dispatcher implementations are present in the **`nav_dma_common.c`** file.

A DMA thread is normally waiting for a message to arrive. There are two states in which the thread listens for messages: **`DMA started`** and **`DMA not started`**. When created, the thread is in **`DMA not started`** state and will wait indefinitely for a message to arrive.

When the application calls **`NAV_DmaStart()`**, a message arrives with the command **`START`** and the thread switches its state to **`DMA started`**. In this state, it only waits as long as the user–supplied DMA timeout for a message to arrive. If no message arrives within this period, it implies that no interrupt was received and that DMA has timed out.

The user–supplied DMA callback handler function is then invoked with a status code denoting the timeout condition. When the application calls **`NAV_DmaStop()`**, a message arrives with the command **`STOP`** and the thread switches its state back to **`DMA not started`**.

Similarly, when the DMA thread receives the message about a recently filled buffer, it invokes the user–supplied DMA callback handler function with a status code denoting this condition. The status code supplied with any callback denotes the situation that prompted the callback. Status codes can come OR'ed together to signify one or more conditions occurring simultaneously.

Based on this status, the user code can decide which action to take. If the main application thread ends DMA operation by calling **`NAV_DmaCleanup()`**, the library notifies the DMA thread via another message carrying the command **`QUIT`**, and the thread then exits on receipt of such a message. The implementation for the DMA thread is present in the **`nav_dmathread.c`** file.

There is no provision in the library to easily override the way the DMA descriptors are set up. This is because the existing DMA callback mechanism relies on the descriptors being set up in a specific way. If DMA callbacks are not being used, the implementation in **`NAV_DmaSetupDescriptorList()`** can be changed to fit unique requirements.