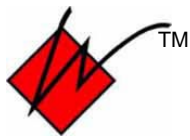


WILDSTAR™-II Host Software Reference Manual (including PRO-Series Boards)

12928-0000 Revision 7.0

© Copyright 2000-2006 by Annapolis Micro Systems, Inc. All Rights Reserved. Printed and published in the United States of America. WILDFIRE™, WILDFIRE™-XL, WILDCHILD™, WILDFORCE™, WILDFORCE™-XL, WILD-ONE™, WILD-ONE™-XL, WILDTIME™, WILDCARD™, STARFIRE™, WILDSTAR™, WILDSTAR™-II, WILDSTAR™-E, WSDP™, WILDWARE™, C2WILD™, CoreFire™, WILD™, and FIREBIRD™ are trademarks of Annapolis Micro Systems, Inc. All other trademarks and registered trademarks are owned by their respective owners.



ANNAPOLIS MICRO SYSTEMS, INC. - LICENSE AGREEMENT

WILDSTAR™-II Host Software, Device Drivers, Models, VHDL, Examples, Tools and PCI Controller are supplied with a License Agreement. This License Agreement also covers the PLD designs and Flash content supplied with the board.

Do not install or use this product and/or break the seal on the CD-ROM until you have read and agreed to the following terms and conditions. If you agree to these terms and conditions, you should sign and return the License and Registration Certificate. You will not be entitled to support or updates until the License and Registration Certificate is received by Annapolis Micro Systems, Inc. Should you choose not to be bound by the terms and conditions of this agreement, you should promptly return this product.

YOU ARE BOUND TO THE TERMS OF THIS AGREEMENT BY BREAKING THE SEAL ON THE CD-ROM

Under the terms of this License, you:

- may make copies of the Licensed Product
- may not transfer the Licensed Product to an unlicensed party
- may modify the VHDL and the Examples
- may not modify any other parts of the Licensed Product
- may not decompile, reverse assemble or otherwise reverse engineer the Licensed Product
- may run this product ONLY on an Annapolis Micro Systems, Inc. board

The Licensed Product is owned and copyrighted by Annapolis Micro Systems, Inc. You may not remove the copyright notice from the Licensed Product. You must use your best efforts to prevent any unauthorized copying of the Licensed Product.

The Licensed Product is provided “as is” without warranty of any kind including warranties for merchantability or fitness for a particular purpose. Annapolis Micro Systems, Inc. shall not be liable for any loss of profits, loss of use, interruption of business, nor for indirect, special, incidental or consequential damages of any kind whether under this agreement or otherwise.

Although Annapolis Micro Systems, Inc. does not warrant the functions contained in the Licensed Product, the medium on which the Licensed Product is furnished is warranted to be free from defects in materials and workmanship under normal use for a period of 90 days from date of delivery to you as evidenced by a copy of your receipt. Annapolis Micro Systems’ entire liability to you and your exclusive remedy shall be replacement of the Licensed Product if the medium on which the Licensed Product is furnished proves to be defective.

You understand that the Licensed Product may require a license from the US Department of Commerce or other government agency before it may be taken or sent outside of the United States. You agree to obtain any required licenses before taking or sending the Licensed Product out of the United States. You will not permit the re-export of the Licensed Product without obtaining required licenses or letter of further assurance.

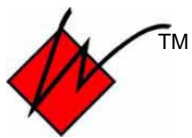


TABLE OF CONTENTS

1.	ABOUT THIS MANUAL	1-1
1.1	INTRODUCTION	1-1
1.2	CONTENTS	1-1
1.2.1	Chapter One	1-1
1.2.2	Chapter Two	1-1
1.3	Conventions.....	1-2
1.4	Icons	1-2
1.5	Key Words and Definitions	1-3
2.	SOFTWARE REFERENCE	2-1
2.1	Enumerations and Defines	2-1
2.2	Structures	2-3
2.3	Introduction to the WILDSTAR™-II API Functions.....	2-4
2.3.1	General Functions.....	2-4
2.3.1.1	<i>WSII_GetApiVersion</i>	2-4
2.3.1.2	<i>WSII_GetSlotNum</i>	2-5
2.3.1.3	<i>WSII_IsBoardHandleValid</i>	2-6
2.3.1.4	<i>WSII_Open</i>	2-7
2.3.1.5	<i>WSII_Close</i>	2-10
2.3.1.6	<i>WSII_BoardReset</i>	2-11
2.3.1.7	<i>WSII_GetInformation</i>	2-12
2.3.1.8	<i>WSII_GetBoardDescriptionField</i>	2-19
2.3.1.9	<i>WSII_SetVmeTransferMode</i>	2-20
2.3.1.10	<i>WSII_GetVmeTransferMode</i>	2-21
2.3.1.11	<i>WSII_GetMainBoardBaseAddr</i>	2-23
2.3.2	Register Read/Write Functions.....	2-24
2.3.2.1	<i>WSII_ReadReg_32</i>	2-24
2.3.2.2	<i>WSII_WriteReg_32</i>	2-25
2.3.2.3	<i>WSII_ReadRegs_32</i>	2-26
2.3.2.4	<i>WSII_WriteRegs_32</i>	2-27
2.3.3	Processing Element API Functions	2-28
2.3.3.1	<i>WSII_PeProgram</i>	2-28
2.3.3.2	<i>WSII_PeProgramDaisyChain</i>	2-31
2.3.3.3	<i>WSII_PeProgramOnStartup</i>	2-33
2.3.3.4	<i>WSII_PeDeprogram</i>	2-34
2.3.3.5	<i>WSII_PeReset</i>	2-35
2.3.3.6	<i>WSII_PeEnableSramDll</i>	2-36
2.3.3.7	<i>WSII_PeExtIoInterfaceEnable</i>	2-37
2.3.3.8	<i>WSII_ResetPciDcm</i>	2-38
2.3.3.9	<i>WSII_ProgrammedPes</i>	2-39
2.3.4	Clock API Functions.....	2-40
2.3.4.1	<i>WSII_GetClockFrequency</i>	2-40
2.3.4.2	<i>WSII_SetClockFrequency</i>	2-41
2.3.4.3	<i>WSII_GetStartupFrequency</i>	2-43
2.3.4.4	<i>WSII_SetStartupFrequency</i>	2-44
2.3.4.5	<i>WSII_SetUClockSource</i>	2-46
2.3.4.6	<i>WSII_SetPClockSource</i>	2-47
2.3.4.7	<i>WSII_GetUClockSource</i>	2-48
2.3.4.8	<i>WSII_GetPClockSource</i>	2-49
2.3.4.9	<i>WSII_SetCrystalOsc</i>	2-50

2.3.4.10	<i>WSII_GetCrystalOscFreq</i>	2-51
2.3.5	Thermal Management API Functions	2-53
2.3.5.1	<i>WSII_GetTemperature</i>	2-54
2.3.5.2	<i>WSII_GetTemperatureThresh</i>	2-56
2.3.5.3	<i>WSII_SetTemperatureThresh</i>	2-57
2.3.5.4	<i>WSII_LimitTemperatureThresh</i>	2-58
2.3.5.5	<i>WSII_QueryTemperatureEvents</i>	2-60
2.3.5.6	<i>WSII_ResetTemperatureEvents</i>	2-61
2.3.5.7	<i>WSII_PeSetFanMode</i>	2-62
2.3.6	Power API Functions	2-63
2.3.6.1	<i>WSII_GetPower</i>	2-63
2.3.6.2	<i>WSII_GetPowerStatus</i>	2-65
2.3.6.3	<i>WSII_ClearPowerStatus</i>	2-67
2.3.7	Interrupt API Functions	2-69
2.3.7.1	<i>WSII_InterruptQueryStatus</i>	2-69
2.3.7.2	<i>WSII_InterruptReset</i>	2-71
2.3.7.3	<i>WSII_InterruptWait</i>	2-72
2.3.7.4	<i>WSII_InterruptRegisterCallback</i>	2-73
2.3.7.5	<i>WSII_InterruptGetVmeVector</i>	2-74
2.3.7.6	<i>WSII_InterruptSetVmeVector</i>	2-75
2.3.8	Error Handling and Callback API Functions	2-76
2.3.8.1	<i>WSII_SetErrorCallback</i>	2-76
2.3.8.2	<i>WSII_GetLastError</i>	2-79
2.3.8.3	<i>WSII_GetErrorString</i>	2-80
2.3.9	LED Display API Functions	2-81
2.3.9.1	<i>WSII_UpdateDisplay</i>	2-81
2.3.9.2	<i>WSII_SetDisplayLevel</i>	2-82
2.3.10	DMA API Functions	2-83
2.3.10.1	<i>Working with DMA</i>	2-83
2.3.10.2	<i>DMA Under Windows NT® and Windows® 2000®</i>	2-86
2.3.10.3	<i>DMA Under Linux® on an x86</i>	2-88
2.3.10.4	<i>WSII_DmaMemAlloc</i>	2-90
2.3.10.5	<i>WSII_DmaMemFree</i>	2-91
2.3.10.6	<i>WSII_DmaBind</i>	2-92
2.3.10.7	<i>WSII_DmaUnbind</i>	2-94
2.3.10.8	<i>WSII_DmaErrorOp</i>	2-95
2.3.10.9	<i>DMA Under Solaris™</i>	2-96
2.3.10.10	<i>WSII_DmaMemAllocWithId</i>	2-97
2.3.10.11	<i>WSII_DmaGetMap</i>	2-98
2.3.10.12	<i>WSII_DmaSync</i>	2-99

1. ABOUT THIS MANUAL

1.1 Introduction

This *WILDSTAR™-II Software Reference Manual* provides software reference information and host programming guidelines for all WILDSTAR™-II and WILDSTAR™-II PRO-series boards produced by Annapolis Micro Systems.



INFORMATION NOTE

In this document, “WILDSTAR™-II” stands for WILDSTAR™-II /VME and WILDSTAR™-II /PCI motherboards, including PRO-series boards, unless specifically noted otherwise.

1.2 Contents

The WILDSTAR™-II Software Reference Manual describes how to program WILDSTAR™-II through common software functions.

1.2.1 Chapter One

Chapter One outlines the conventions, icons, and key words used throughout the manual.

1.2.2 Chapter Two

Chapter Two contains software reference information, including types, API functions, and memory interfaces.




1.3 Conventions

A variety of text styles are used throughout the manual to call attention to specific items.

Convention	Description
Text represented as screen display	This typeface is used to represent displays appearing on the screen, such as at the “A:\” prompt.
Text represented as commands	This typeface is used to represent commands that are to be entered, such as: type “setup.”
Text represented as code	This typeface is used to represent ‘C’ code.
Keys	When specific keys are referenced, they are designated by their labels, such as “the Enter key” or “the Escape key,” or they may be shown as [Enter] or [Esc]. When two or more keys are to be pressed simultaneously, the keys are linked with a plus sign (+). For example: [Ctrl] + [Alt] + [Del].
Text represented as <i>bold italics</i>	This typeface is used to designate equations being used in the text.

1.4 Icons

Throughout the manual, important information is highlighted with icons.

Icon	Type	Description
	Information Note	Information Notes call attention to important features or instructions.
	Caution	Cautions are directions that must be followed to avoid loss of system data or damage to hardware.
	Warning	Warnings are directions that must be followed to ensure personal safety.

1.5 Key Words and Definitions

Some of the terms used throughout the manual are defined below.

API

Application Programming Interface.

Application Programming Interface

A set of communication functions coded in the C language

Block RAM

An architectural feature of the Virtex™ FPGA in which blocks are reserved specifically for Random Access Memory.

CLB

Configurable Logic Block.

Configurable Logic Block

CLBs provide the functional elements for constructing logic within the PE. Each Virtex™ CLB contains four Logic Cells, organized in two similar Slices.

DMA

Direct Memory Access.

Driver

Software to handle communication to WILDSTAR™-II boards.

Euro I/O Card

Eurocard form factor I/O daughter card for WILDSTAR™-II /VME.

External I/O

External Input/Output.

External I/O Cards

External Input/Output Cards for WILDSTAR™-II /VME and PCI. Includes both Euro I/O cards and PCI I/O cards.

FPGA

Field Programmable Gate Array.

KCLK

System bus clock used for Local Address Data Bus transactions.

LAD Bus

Local Address Data Bus. Bus between the PCI Controller, PE0, PE1, PE2, and the External I/O card.

MCLK

Reprogrammable Memory clock.

N/C

No Connection.

PCI

Peripheral Component Interconnect.

PCI I/O Card

PCI form factor I/O daughter card for WILDSTAR™-II /PCI.

PCLK

Reprogrammable Processing Element Clock synchronous to MCLK.

PE

Processing Element.

PE0

Processing Element 0

PE1

Processing Element 1

PE2

Processing Element 2

PMC

PCI Mezzanine Card

Processing Element

A Xilinx® Virtex™ Field Programmable Gate Array (FPGA), which comprises the basic processing unit on the WILDSTAR™ board.

Slice

A Slice contains two look-up tables and two flip-flops for implementing logic within a CLB. There are two slices per CLB.

SBC

Single Board Computer. A host system that resides on a single printed circuit board (e.g., VMIC x86 VME, Motorola 68000 VME)

UCLK

Configurable User clock for use in the PEs and External I/O cards.

Virtex™ FPGA

Virtex™ Field Programmable Gate Array.

Virtex™-E FPGA

High density FPGA offering higher performance, larger capacity, and higher speed than the Virtex™ FPGA.

VME

Versa Module Eurocard.

WILDSTAR™-II, WILDSTAR™-II PRO Host Software

Application Programming Interface, Device Driver, and utilities.

WILDSTAR™-II VHDL Models

Hardware models used for board-level VHDL simulation of application.

ZBT

Zero Bus Turnaround.



2. SOFTWARE REFERENCE

This chapter describes Constants, Types, and API Functions of WILDSTAR™-II software.

i

INFORMATION NOTE

WILDSTAR™-II motherboards operate with **WILDSTAR™-II Host Software and Device Drivers**. Refer to the *WILDSTAR™-II Hardware Reference Manual* for board-specific hardware and VHDL Models information.

2.1 Enumerations and Defines

Many API calls require variables or constants to be of a certain type. Using enumerated types rather than #defined constants helps prevent sending improper values into API calls (although the API does check many parameters for incorrect values). #defined constants are used for creating bit masks, which in turn are used for calls that can specify more than one item (e.g., waiting for multiple PE interrupts in WSII_InterruptWait, resetting multiple PEs in WSII_PeReset).

The WILDSTAR™-II enumerated types include:

WSII_BOARD_SITE – Specifies which board in a slot to target, the WILDSTAR™-II main board or an external I/O card.

WSII_EXTIO_SITE – Specifies an external I/O card.

WSII_FLAGS – Flags that may be logically “OR’ed” together for the WSII_Open call. See the WSII_Open call for more information.

WSII_PE_NUM – Usually specifies the target Processing Element for a function call, but can also specify a FLASH memory bank. Some enumerations are specific to WILDSTAR™-II-PRO boards with larger flash parts. Read the comment next to the enumerator to make sure the enumerator is valid for the target board type.

WSII_CLOCK_SRC - Specifies the target clock for a function call. Some enumerations are specific to WILDSTAR™-II boards, and others are specific to WILDSTAR™-II-PRO boards. Read the

comment next to the enumerator to make sure the enumerator is valid for the target board type.

WSII_U_CLOCK_INPUT_SRC – Enumerates the possible sources for U clock. This enumeration is only valid for WILDSTAR™-II-PRO boards.

WSII_P_CLOCK_INPUT_SRC – Enumerates the possible sources for P clock. This enumeration is only valid for WILDSTAR™-II-PRO boards.

WSII_CRYSTAL_OSC – Enumerates the possible crystal clock sources for the Rocket I/O clocks. This enumeration is only valid for certain WILDSTAR™-II-PRO boards.

WSII_POWER_SRC - Specifies voltage or current source to be measured. All of the enumerators for this enumeration are valid for WILDSTAR™-II boards. Only four are valid for WILDSTAR™-II-PRO boards. Read the comment next to the enumerator to make sure the enumerator is valid for the target board type.

WSII_TEMPERATURE_SRC – Specifies temperature to be measured. For the physical location of the WSII_TEMP_SRC_LOCX members, see the WILDSTAR™-II hardware reference manual. Read the comment next to the enumerator to make sure the enumerator is valid for the target board type.

WSII_TEMP_THRESHOLD – Specifies temperature threshold to be assigned or examined.

WSII_RESET_OP – Specifies to assert, de-assert, or pulse the PE's RESET line. Used in the WSII_PeReset call

WSII_FAN_MODE – Specifies the operating mode of a cooling fan. Used in the WSII_PeSetFanMode call.

WSII_DISPLAY_LEVEL – Specifies intensity of the LED display

WSII_INFO_REQ – Specifies a type of information to be retrieved by the WSII_GetInformation function call.

WSII_DMA_SYNC – Enumeration for cache coherency between memory and the CPU or the target device.

WSII_VME_XFER_MODE – Used to specify the VME bus transfer mode to be used for outbound data transfers when the WILDSTAR™-II PRO/VME or WILDSTAR™-II PRO ACE for VME is the master (used with VxWorks® only).

WSII_VME_2ESST_RATE – Used to specify the 2eSST transfer rate when the VME bus transfer mode is either 2eSST or 2eSST broadcast for the WILDSTAR™-II PRO/VME or WILDSTAR™-II PRO ACE for VME (used with VxWorks® only).

2.2 Structures

Several WILDSTAR™-II API function calls return data in the form of a data structure, allowing multiple pieces of related data to be returned to a user. Besides removing a large number of function parameters, these calls also help organize the data.

WSII_MEM_INFO – Contains information about a particular memory element, such as the size of the memory, the memory speed, and the width (in bits) of each port in the memory element.

WSII_DMA_ERRORS – Contains an array of DWORDs, one DWORD per DMA channel. Each DWORD contains a bit-mask of any errors encountered on that DMA channel. This structure is used in the WSII_DmaErrorOp function call.

WSII_DAISSY_CHAIN_BUFFER – Contains a pointer to a PE Programming buffer and the size (in DWORDs) of that programming buffer. An array of these structures is used in the WSII_PeProgramDaisyChain function call.

WSII_ADC_CHANNEL – Specifies a particular channel on an ADC External I/O card.

WSII_ADC_CHANNEL_INFO – Contains information about a particular channel on an ADC External I/O card, such as the chip type, the sample frequency, and the sample frequency accuracy.

2.3 Introduction to the WILDSTAR™-II API Functions

This section describes the API functions available to the application programmer. Each API function description includes the valid call syntax for the function, the list of parameters to the function, and valid return codes from the function.

2.3.1 General Functions

General functions have been provided to allow the application programmer to determine the version of the WILDSTAR™-II API, generate and destroy access handles to WILDSTAR™- II boards, display return status from API functions, and get physical board attributes.

2.3.1.1 WSII_GetApiVersion

```
WSII_RetCode WSII_GetApiVersion (DWORD *pApiVer)
```

Parameters:

pApiVer pointer to API version number

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

On return, the DWORD pointed to by pApiVer will contain the API's version in the following format:

BYTE 3: RESERVED, always 0x0

BYTE 2: Major Version Number

BYTE 1: Minor Version Number

BYTE 0: Second Minor Version Number

For example, version 3.5.2 would be returned as 0x00030502.

Example:

```
DWORD apiVer;  
rc = WSII_GetApiVersion (&apiVer);  
printf("API Version = 0x%x\n", apiVer);
```

2.3.1.2 WSII_GetSlotNum

DWORD WSII_GetSlotNum (WSII_BOARD hBoard)

Parameters:

hBoard Board handle returned by WSII_Open()

Returns:

Zero-based slot number. On PCI systems, slot numbers are assigned sequentially, starting at zero. On VME systems, the board number corresponds to the VME slot ID when the VME geographical address is used; otherwise, it corresponds to the slot number that is set with the WILDSTAR™-II board's configuration switches. (See Chapter 4 of the *WILDSTAR™-II Hardware Reference Manual* for more information about setting configuration switches).

Example:

```
DWORD SlotNum;  
SlotNum = WSII_GetSlotNum (BoardHandle);  
printf("Slot Number = %d\n", SlotNum);
```

2.3.1.3 WSII_IsBoardHandleValid

```
BOOLEAN WSII_IsBoardHandleValid (WSII_BOARD hBoard)
```

Parameters:

hBoard Board handle returned by WSII_Open()

Returns:

This API simply validates a given board handle.

Example:

```
BOOLEAN bIsHndValid;  
bIsHndValid = WSII_IsBoardHandleValid (BoardHandle);  
if (bIsHndValid)  
    printf("Board handle is valid.\n");  
else  
    printf("Board handle is not valid.\n");
```

2.3.1.4 WSII_Open

```
WSII_BOARD WSII_Open      (DWORD      dSlotNumber ,
                          DWORD      ApiCode
                          WSII_RetCode *pErrorCode,
                          DWORD      dFlags)
```

Parameters:

dSlotNumber	Zero-based board number to open
ApiCode	Must pass in the #defined constant "WSII_API_CODE"
pErrorCode	Indicates WSII_SUCCESS or reason for failure
Flags	One or more #defined WSII_FLAGS, logically OR'ed together

Returns:

Board handle that is used in subsequent calls to the API.

Remarks:

If the call was not successful, the call returns `NULL` and `pErrorCode` will contain a numerical code from *wsii.h* indicating the reason for failure. The caller must pass in the constant, `WSII_API_CODE`, defined in *wsii.h*. Using this constant at compile time tells all future versions of the API library which version of *wsii.h* was used to compile the application.

The `dSlotNumber` parameter is the location on the host system bus of the WILDSTAR™-II board to be accessed. On PCI-based host buses, slot numbering starts at zero for the first board located in the system, incrementing by one for each additional board. For VME-based host buses, `dSlotNumber` corresponds to the number displayed on the WILDSTAR™-II front panel display (see Chapter 3 of the *WILDSTAR™-II Hardware Reference Manual* for more information on the location of the front panel display).

On host systems having both a PCI and a VME bus (e.g., a PCI system with an SBS BIT-3 PCI-VME bridge) it is important NOT to assign a VME slot number to a WILDSTAR™II /VME that conflicts with a WILDSTAR™-II/PCI. Make sure that the first VME board's slot number is greater than the number of WILDSTAR™-II/PCI boards in the system. (For more information on assigning VME slot numbers to WILDSTAR™-II /VME boards, see Chapter 4 of the *WILDSTAR™-II Hardware Reference Manual*.)

The dFlags parameter is a bit mask, which is created by logically OR'ing together one (or more) of the following #defined flags with WSII_FLAGS_OPEN:

WSII_FLAGS_FLASH_SHOW_DOTS
WSII_FLAGS_BUS_TARGET_PCI
WSII_FLAGS_BUS_TARGET_VME_VMIC
WSII_FLAGS_BUS_TARGET_VME_BIT3
WSII_FLAGS_BUS_TARGET_VME_SBS_SBC
WSII_FLAGS_DISABLE_SRAM_DLL

WSII_FLAGS_FLASH_SHOW_DOTS – When this flag is sent passed into WSII_Open(), the API sends a series of ellipses (“dots”) to stdout during FLASH memory operations. This can help when programming FLASH memory with Processing Element images (“PE Images”). These PE images can be several megabytes in length, and can take up to several minutes to complete the programming operation. Printing “dots” to the display screen helps users know that the system is still responding and performing a lengthy task.

**WSII_FLAGS_BUS_TARGET_PCI ,
WSII_FLAGS_BUS_TARGET_VME_VMIC,
WSII_FLAGS_BUS_TARGET_VME_BIT3**

WSII_FLAGS_BUS_TARGET_VME_SBS_SBC– These flags are mutually exclusive, and specifying more than one of these flags to a WSII_Open call is an error. Since a host system can have more than one data I/O bus, this flag will tell the WSII_Open call on which bus to look for a WILDSTAR™-II at the specified dSlotNumber. Generally, systems with multiple host data I/O buses are one of the following configurations:

- VMIC/VME Host SBC: System has both a PCI bus and a VME bus
- BIT3 PCI-VME Bridge: System has both a PCI bus and a VME bus

WSII_FLAGS_DISABLE_SRAM_DLL – Starting with API version 3.8.2, all WILDSTAR™-II API libraries will, after programming a Processing Element, turn on or off that Processing Element’s SRAM DLL (prior versions of the WILDSTAR™-II API only turned the DLL on) based on this flag. By default, the DLL is turned on; passing this flag in will turn it off.

Example:

```
WSII_BOARD    BoardHandle;
WSII_RetCode  RetCode;

BoardHandle = WSII_Open (0x0,
                        WSII_API_CODE,
                        &RetCode,
                        WSII_FLAGS_OPEN);

if (BoardHandle == NULL)
{
    printf( "Unable to get Handle to specified board\n");
    printf( "Error = %d\n", RetCode);
    exit(-1);
}

/* ...rest of program... */
```

2.3.1.5 WSII_Close

WSII_RetCode WSII_Close (WSII_BOARD hBoard)

Parameters:

hBoard Board handle returned by WSII_Open()

Returns:

WSII_SUCCESS if the board was successfully closed.

Remarks:

After a successful call to WSII_Open(), it is important to call WSII_Close() before a program exits. Calling this function cleans up resources and frees memory. Once WSII_Close is called, hBoard is invalidated and an application *must not* use the handle further, unless hBoard is re-initialized with another call to WSII_Open.



CAUTION

Failure to call this function on some host systems may cause errors if the user attempts to re-open the board at a later time.

2.3.1.6 WSII_BoardReset

WSII_RetCode WSII_BoardReset (WSII_BOARD hBoard)

Parameters:

hBoard Board handle returned by WSII_Open()

Returns:

WSII_SUCCESS if the board was successfully reset.

Remarks:

This call does a board-level reset on a WILDSTAR™-II board. It is not typically called from applications, but provided as a way for users who cannot physically access their boards to simulate a power-on reset.

For more information, see Chapter 4 of the *WILDSTAR™-II Hardware Reference Manual*.

CAUTION

Making this call on a VME board with the “board reset select” switch set to OFF (the default setting) will cause a reset of the PCI controller. If this call is made, all communication will be lost to the WILDSTAR™-II board. It will be necessary to close the application that made this call, and then restart it in order for the software to re-initialize the WILDSTAR™-II board to a point where it can be used again.

2.3.1.7 WSII_GetInformation

```

WSII_RetCode WSII_GetInformation (WSII_BOARD    hBoard,
                                WSII_INFO_REQ dInfoType,
                                void          *pExtra,
                                void          *pData)

```

Parameters:

hBoard Board handle returned by WSII_Open()
dInfoType Type of information requested
pExtra Additional information the API may require to complete the information request, such as PE number or I/O site
pData Pointer to user-allocated storage that the API will fill in with the requested data.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This general-purpose call acquires information about various aspects of a WILDSTAR™-II main board or an attached I/O board. The caller must supply storage for **pExtra** and **pData**. For requests that return a text string, **pData** must point to at least WSII_MAX_STRING_LENGTH bytes of storage.

Table 2-1: WSII_GetInformation Quick Reference

INFORMATION TYPE:	pExtra INFO:	pData RETURNS:
WSII_PE_TYPE	WSII_PE_NUM	NULL-Terminated String
WSII_PE_PKG	WSII_PE_NUM	NULL-Terminated String
WSII_PE_SPEED	WSII_PE_NUM	NULL-Terminated String
WSII_PE_REV_CODE	WSII_PE_NUM	NULL-Terminated String
WSII_PE_IO_OPTION	WSII_PE_NUM	NULL-Terminated String
WSII_BOARD_P0_EXT_5V_OPT	N/A	DWORD
WSII_BOARD_TYPE	WSII_BOARD_SITE	NULL-Terminated String
WSII_SERIAL_NUMBER	WSII_BOARD_SITE	NULL-Terminated String
WSII_BOARD_REVISION	WSII_BOARD_SITE	ASCII Character
WSII_PRODUCT_CFG_CODE	WSII_BOARD_SITE	NULL-Terminated String
WSII_REVISION_LEVEL_CODE	WSII_BOARD_SITE	NULL-Terminated String
WSII_HOST_BUS_TYPE	N/A	NULL-Terminated String
WSII_HOST_BUS_SPEED	N/A	DWORD
WSII_HOST_BUS_WIDTH	N/A	DWORD
WSII_PCI_TYPE	N/A	NULL-Terminated String
WSII_SRAM_INFO	WSII_PE_NUM	WSII_MEM_INFO structure
WSII_DRAM_INFO	WSII_PE_NUM	WSII_MEM_INFO structure

INFORMATION TYPE:	pExtra INFO:	pData RETURNS:
WSII_API_VERSION	N/A	DWORD
WSII_DRIVER_VERSION	N/A	DWORD
WSII_PCI_VERSION	N/A	DWORD
WSII_PLD_VERSION	WSII_PE_NUM	DWORD
WSII_API_DATE	N/A	NULL-Terminated String
WSII_API_BUILD_VERSION	N/A	DWORD
WSII_DMA_STATUS	NULL	DWORD
WSII_DMA_BUFFER_PCI_ADDR	WSII_DMA_HANDLE	DWORD*
WSII_ADC_CHANNEL_CONFIG	WSII_ADC_CHANNEL	WSII_ADC_CHANNEL_INFO
WSII_ADC_MAX_CHANNELS	WSII_BOARD_SITE	DWORD
WSII_ADC_CURRENT_CAL_VAL	WSII_BOARD_SITE	WSII_ADC_CURRENT_CAL
WSII_SAMP_PCLK_FREQ_KHZ	NULL	DWORD
WSII_RAW_INFO	WSII_BOARD_SITE	NULL-Terminated String

Specifying **WSII_PE_TYPE**, **WSII_PE_PKG**, **WSII_PE_SPEED**, **WSII_PE_REV_CODE**, or **WSII_PE_IO_OPTION** for `dInfoType` requires that `pExtra` point to a variable of type **WSII_PE_NUM** that contains one of the following values:

WSII_PE0, **WSII_PE1**, **WSII_PE2**, **WSII_EXTIO_0_PE0**,
WSII_EXTIO_0_PE1, **WSII_EXTIO_0_PE2**, **WSII_EXTIO_0_PE3**,
WSII_EXTIO_1_PE0, **WSII_EXTIO_1_PE1**, **WSII_EXTIO_1_PE2**, or
WSII_EXTIO_1_PE3

These calls also require that `pData` point to a (user-allocated) character string that is at least **WSII_MAX_STRING_LENGTH** bytes long.

WSII_PE_TYPE – On return, the character string pointed to by `pData` will contain either a string representing the Xilinx® FPGA part type for the specified processing element, or it will have a NULL (i.e., 0x0) in the 0th member of the character string pointed to by `pData`. For example, requesting the part type of PE0 on a WILDSTAR™-II /PCI board populated with Xilinx Virtex-II 6000 FPGAs will return the string “XC2V6000”.

Requesting the part type of PE2 on the same board would return a NULL in the 0th position of `pData` because PCI-based WILDSTAR™-II boards do not have a PE2 element.

WSII_PE_PKG – On return, the character string pointed to by `pData` will contain either a string representing the package type for the specified processing element, or it will have a NULL (i.e., 0x0) in the 0th member of the character string pointed to by `pData`. For example, requesting the package type of PE0 on a WILDSTAR™-II/PCI board will return the string “FF1517”.

Requesting the package type of PE2 on the same board would return a NULL in the 0th position of `pData` because PCI-based WILDSTAR™-II boards do not have a PE2 element.

WSII_PE_SPEED – On return, the character string pointed to by pData will contain either a string representing the PE’s Xilinx-specified speed grade, or it will have a NULL (i.e., 0x0) in the 0th member of the character string pointed to by pData. For example, requesting the package type of PE0 on a WILDSTAR™-II /PCI board could return the string “4C-0765”.

Requesting the package type of PE2 on the same board would return a NULL in the 0th position of pData because PCI-based WILDSTAR™-II boards do not have a PE2 element.

WSII_PE_REV_CODE – On return, the character string pointed to by pData will contain either a string representing the PE’s Xilinx®-specified revision code, or it will have a NULL (i.e., 0x0) in the 0th member of the character string pointed to by pData. This data may be required by Annapolis Micro Systems support personnel, and is not intended for customer use.

WSII_PE_IO_OPTION – On return, the character string pointed to by pData will contain either a string representing the PE’s I/O configuration. It is useful for determining the configuration of a PE’s I/O pins. Some possible values are: “**SYSTOLIC=SINGLE-ENDED**”, “**P0:DIFFERENTIAL;P2:RACEWAY**”, etc.

Specifying **WSII_BOARD_TYPE**, **WSII_SERIAL_NUMBER**, or **WSII_BOARD_REVISION** for dInfoType requires that pExtra point to a variable of type **WSII_BOARD_SITE** that contains one of the following values:

WSII_MAIN_BOARD, **WSII_EXTIO_0**, or **WSII_EXTIO_1**

WSII_BOARD_P0_EXT_5V_OPT – When specifying this for the dInfoType, pData must point to a (user-allocated) DWORD variable. On return, the DWORD pointed to by pData will contain TRUE or FALSE, indicating whether the VME target board has the P0 external 5V power supply option.

WSII_BOARD_TYPE – Specifying this for the dInfoType also requires that pData point to a (user-allocated) character string that is at least **WSII_MAX_STRING_LENGTH** bytes long.

On return, the character string pointed to by pData will contain either a string representing the specified board’s name or it will have a NULL (i.e., 0x0) in the 0th member of the character string pointed to by pData if the board does not exist (e.g., if an external I/O board is not present).

Possible board types are:

- WILDSTAR™-II-PRO
- WILDSTAR™-II
- FPDF External I/O Card
- Myrinet-E External I/O Card
- LVDS/TTL External I/O Card
- Virtex™ External I/O Card
- E3 External I/O Card
- A/D 65MHz External I/O Card

- Quad G-Link External I/O Card
- Myrinet External I/O Card
- PCI PCI External I/O Card
- A/D 1.5 GHz External I/O Card
- WSDP™ External I/O Card
- Gigabit Ethernet I/O Card
- Quad 105 MHz I/O Card
- A/D 105MHz External I/O Card
- WSDP™ PCI External I/O Card
- FPDP-E External I/O Card
- ECL External I/O Card
- WSDP™-E External I/O Card
- Fibre Channel 2 I/O Card
- Dual GHz I/O Card

WSII_SERIAL_NUMBER – Specifying this for the `dInfoType` also requires that `pData` point to a (user-allocated) `DWORD`. On return, the `DWORD` pointed to by `pData` will contain either the board’s serial number or it will contain 0x0 if the board does not exist (e.g., if an external I/O board is not present).

WSII_BOARD_REVISION – Specifying this for the `dInfoType` also requires that `pData` point to a (user-allocated) character variable. On return, the character variable pointed to by `pData` will contain either the board’s revision code or it will contain 0x0 if the board does not exist (e.g., if an external I/O board is not present).

WSII_PRODUCT_CFG_CODE – Specifying this for the `dInfoType` also requires that `pData` point to a (user-allocated) character string that is at least **WSII_MAX_STRING_LENGTH** bytes long.

WSII_REVISION_LEVEL_CODE – Specifying this for the `dInfoType` also requires that `pData` point to a (user-allocated) character string that is at least **WSII_MAX_STRING_LENGTH** bytes long.

WSII_HOST_BUS_TYPE – When specifying this for the `dInfoType`, `pExtra` should be passed in as `NULL`. `pData` must point to a (user-allocated) character string of at least **WSII_MAX_STRING_LENGTH** bytes. On return, the character string pointed to by `pData` will contain one of the following: “PCI”, “CompactPCI”, “VME”, or “Cardbus”.

WSII_HOST_BUS_SPEED – When specifying this for the `dInfoType`, `NULL` must be passed in for `pExtra`. `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` variable pointed to by `pData` will contain the WILDSTAR™-II host board’s data I/O bus speed (usually 33 or 66 MHz for PCI and 66, 100, or 133 MHz for PCIX).

WSII_HOST_BUS_WIDTH – When specifying this for the `dInfoType`, `NULL` must be passed in for `pExtra`. `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain the WILDSTAR™-II host board’s data I/O bus width in bits (typically either 32 or 64 bits).

WSII_PCI_TYPE – When specifying this for the `dInfoType`, `pExtra` should be passed in as `NULL`. `pData` must point to a (user-allocated) character string of at

least `WSII_MAX_STRING_LENGTH` bytes. On return, the character string pointed to by `pData` will contain one of the following: “PCI” or “PCI-X”.

`WSII_SRAM_INFO` and `WSII_DRAM_INFO` – These require that `pExtra` point to a variable of type `WSII_PE_NUM` that contains one of the following values:

`WSII_PE0`, `WSII_PE1`, `WSII_PE2`, `WSII_EXTIO_0_PE0`,
`WSII_EXTIO_0_PE1`, `WSII_EXTIO_0_PE2`, `WSII_EXTIO_0_PE3`,
`WSII_EXTIO_1_PE0`, `WSII_EXTIO_1_PE1`, `WSII_EXTIO_1_PE2`, or
`WSII_EXTIO_1_PE3`

These calls also require that `pData` point to a (user-allocated) `WSII_MEM_INFO` structure that, upon return, will be filled in with information about the memory elements associated with the specified PE.

`WSII_API_VERSION`, `WSII_DRIVER_VERSION`, `WSII_PCI_VERSION`, and `WSII_PLD_VERISON` – When specifying this for the `dInfoType`, `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain the requested element’s version in the following format:

BYTE 3: RESERVED, always 0x0
BYTE 2: Major Version Number
BYTE 1: Minor Version Number
BYTE 0: Second Minor Version Number

For example, version 3.5.2 would be returned as 0x00030502. For `pExtra`, `WSII_API_VERSION`, `WSII_DRIVER_VERSION`, and `WSII_PCI_VERSION`, require that `NULL` must be passed in; for `WSII_PLD_VERISON`, `pExtra` must point to a variable of type `WSII_PE_NUM` that contains one of the following values:

`WSII_PE0`, `WSII_PE1`, `WSII_PE2`, `WSII_EXTIO_0_PE0`, or
`WSII_EXTIO_1_PE0`

`WSII_API_BUILD_VERSION` – When specifying this for the `dInfoType`, `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain the API build number. The API build number is additional API versioning information that should be provided to customer support in the event that support is required.

`WSII_DMA_STATUS` - When specified for the `dInfoType`, `pExtra` must be `NULL`, and `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain the DMA control and status register. This call is provided for legacy applications and should not be used. Instead, use the `WSII_DmaErrorOp` call.

`WSII_DMA_BUFFER_PCI_ADDR` – When specifying this for the `dInfoType`, `pExtra` must be a `WSII_DMA_HANDLE`, and `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain a 32-bit PCI address corresponding to the PCI address of the contiguous buffer held in the `WSII_DMA_HANDLE`.

WSII_ADC_CHANNEL_CONFIG – When specifying this for the `dInfoType`, `pExtra` must point to a variable of type `WSII_ADC_CHANNEL` that contains the board site and the ADC channel of interest, and `pData` must point to a (user-allocated) `WSII_ADC_CHANNEL_INFO` structure. On return, the `WSII_ADC_CHANNEL_INFO` type pointed to by `pData` will contain the ADC cards channel info.

WSII_ADC_MAX_CHANNELS – When specifying this for the `dInfoType`, `pExtra` must point to a variable of type `WSII_BOARD_SITE` that contains one of the following values: `WSII_MAIN_BOARD`, `WSII_EXTIO_0`, or `WSII_EXTIO_1`, and `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain the maximum number of ADC channels for the target External I/O ADC card.

WSII_ADC_CURRENT_CAL_VAL – When specifying this for the `dInfoType`, `pExtra` must point to a variable of type `WSII_BOARD_SITE` that contains one of the following values: `WSII_MAIN_BOARD`, `WSII_EXTIO_0`, or `WSII_EXTIO_1`, and `pData` must point to a (user-allocated) `WSII_ADC_CURRENT_CAL` structure. On return, the `WSII_ADC_CURRENT_CAL` type pointed to by `pData` will contain the ADC cards current calibration information for the target External I/O ADC card.

WSII_SAMP_PCLK_FREQ_KHZ – When specified for the `dInfoType`, `pExtra` must be `NULL`, and `pData` must point to a (user-allocated) `DWORD` variable. On return, the `DWORD` pointed to by `pData` will contain the sampled rate of P clock in KHz.

WSII_RAW_INFO – When specifying this for the `dInfoType`, `pExtra` must point to a variable of type `WSII_BOARD_SITE` that contains one of the following values: `WSII_MAIN_BOARD`, `WSII_EXTIO_0`, or `WSII_EXTIO_1`, `pData` must point to (user-allocated) character data of size `WSII_MAX_BOARD_DESCRIPTION_STRING_LENGTH`. On return, the character string pointed to by `pData` will contain the null terminated raw board description data.

Examples:

To get the board PE type:

```
WSII_RetCode rc;
WSII_PE_NUM PeNumber;
char pText[WSII_MAX_STRING_LENGTH];

/* open board, get board handle */
PeNumber = WSII_PE0;
rc = WSII_GetInformation( BoardHandle,
                        WSII_PE_TYPE,
                        &PeNumber,
                        (void*)pText);

if (rc == WSII_SUCCESS)
    printf("PE0 is an %s\n", pText);
```

OUTPUT:

```
PE0 is an XC2V6000
```

To get the board type:

```
WSII_RetCode      rc;
WSII_BOARD_SITE   BoardLocation;
char              BoardRevision;
char              pBoardType[WSII_MAX_STRING_LENGTH];
char              pBusType[WSII_MAX_STRING_LENGTH];

    /* open board, get board handle */
BoardLocation = WSII_MAIN_BOARD;
rc = WSII_GetInformation( BoardHandle, WSII_HOST_BUS_TYPE,
                          NULL, (void*)pBusType);
if (rc != WSII_SUCCESS)
    return (rc);

rc = WSII_GetInformation( BoardHandle, WSII_BOARD_TYPE,
                          &BoardLocation, (void*)pBoardType);
if (rc != WSII_SUCCESS)
    return (rc);
rc = WSII_GetInformation( BoardHandle, WSII_BOARD_REVISION,
                          &BoardLocation, (void*)&BoardRevision);
if (rc != WSII_SUCCESS)
    return (rc);

printf("Board #0 is a %s/%s Rev %c\n",
       WSII_GetSlotNum(BoardHandle),
       pBoardType, pBusType,
       BoardRevision );
```

OUTPUT:

```
Board #0 is a WILDSTAR_II/PCI Rev B
```

2.3.1.8 WSII_GetBoardDescriptionField

```
WSII_RetCode  
WSII_GetBoardDescriptionField (WSII_BOARD      hBoard,  
                               WSII_BOARD_SITE BoardSite,  
                               char            *pField,  
                               DWORD          userFieldSize,  
                               char          *pSearchStrings[],  
                               DWORD          numSearchStrings);
```

Parameters:

hBoard	Board handle returned by WSII_Open()
BoardSite	WSII_MAIN_BOARD, WSII_EXTIO_0 or WSII_EXTIO_1
pField	Pointer to user allocated storage for returning the requested string data
userFieldSize	Size in bytes allocated to pField
pSearchStrings[]	Array of character pointers that define a path to the desired data field
numSearchStrings	Number of search strings

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This call parses the specified board description information data and returns the target field string data. In most cases the API call `WSII_GetInformation` can be used to return any board description information required by the user. This call is provided for those cases that `WSII_GetInformation` does not handle.

2.3.1.9 WSII_SetVmeTransferMode

```
WSII_RetCode WSII_SetVmeTransferMode(WSII_BOARD hBoard,  
  
                                     WSII_VME_XFER_MODE dTransferMode,  
  
                                     WSII_VME_2ESST_RATE dTransferRate)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dTransferMode	VME bus transfer mode (power on default is MBLT)
dTransferRate	2eSST transfer rate when the transfer mode is 2eSST or 2eSST broadcast.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See wsii.h for a list of possible errors.

Remarks:

This function should be used before running DMA transfers with the WILDSTAR™-II PRO/VME or WILDSTAR™-II PRO ACE for VME under VxWorks® to ensure optimum throughput. The transfer mode and transfer rate should be selected based on the target system configuration.

Note: Selecting a mode or rate that the host board does not support will result in the failure of VME data transfers.

If the specified transfer mode is not 2eSST or 2eSST broadcast, the value selected for transfer rate will be ignored. Regardless of the specified mode, the value for transfer rate must be a valid value.

2.3.1.10 WSII_GetVmeTransferMode

```
WSII_RetCode WSII_GetVmeTransferMode (WSII_BOARD hBoard,  
  
                                       WSII_VME_XFER_MODE *pTransferMode,  
  
                                       WSII_VME_2ESST_RATE *pTransferRate)
```

Parameters:

- | | |
|---------------|--|
| hBoard | Board handle returned by WSII_Open() |
| pTransferMode | Pointer to user-allocated storage that the API will fill in with the current VME bus transfer mode (power on default is 160 MB/s) |
| pTransferRate | Pointer to user-allocated storage that the API will fill in with the current 2eSST transfer rate when the transfer mode is 2eSST or 2eSST broadcast. |

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See wsii.h for a list of possible errors.

2.3.1.11 WSII_GetMainBoardBaseAddr

```
WSII_RetCode WSII_GetMainBoardBaseAddr(WSII_BOARD    hBoard,
                                       DWORD          *pBaseAddr )
```

Parameters:

- hBoard Board handle returned by WSII_Open()

- pBaseAddr Pointer to user-allocated storage that the API will fill in with the base address of the target VME WILSTAR-II board.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See wsii.h for a list of possible errors.

Remarks:

This function is currently only supported when the target board is a VME form factor WILDSTAR-II board. The value of the 32 bit data at this target board base address is the vendor ID and device ID of the Universe II or Tsi148 bus bridge parts, depending on whether the target board is a WILDSTAR-II or a WILDSTAR-II PRO respectively.

Using this base address and the following tables for specific host configurations, the addresses of the various PE and I/O register space regions can be calculated:

VMIC Host		
Region	Byte Offset from Base Address	Size of Region in Dwords (32 bit words)
Processing Element 0	0x410000	0x80000
Processing Element 1	0x610000	0x80000
Processing Element 2	0x810000	0x80000
External I/O 0	0xA10000	0x80000
External I/O 1	0xC10000	0x80000

All Other Hosts		
Region	Byte Offset from Base Address	Size of Region in Dwords (32 bit words)
Processing Element 0	0x2400000	0x80000
Processing Element 1	0x2600000	0x80000
Processing Element 2	0x2800000	0x80000
External I/O 0	0x2A00000	0x80000
External I/O 1	0x2C00000	0x80000

2.3.2 Register Read/Write Functions

Below is a listing of Read/Write API functions for WILDSTAR™-II.

2.3.2.1 WSII_ReadReg_32

```
DWORD WSII_ReadReg_32 ( WSII_BOARD  hBoard,  
                        WSII_PE_NUM  dPeNum,  
                        DWORD         dOffset )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
dOffset	DWORD offset in to the PE's memory space

Returns:

A `DWORD` that was contained in the register specified by the `dOffset` parameter.

Remarks:

All API calls use the host computer as a frame of reference; “read” for this function therefore indicates that data will be transferred from the WILDSTAR™-II board to the host computer.

This call performs a Programmed I/O (PIO) read operation from the WILDSTAR™-II board, returning a single 32-bit piece of data (a `DWORD`) from `dPeNum`.

Example:

```
DWORD  dData;  
  
dData = WSII_ReadReg_32(BoardHandle, WSII_PE0, 0x100 );  
printf("DWORD Offset 0x100 = 0x%08X\n",dData);
```

2.3.2.2 WSII_WriteReg_32

```
WSII_RetCode WSII_WriteReg_32 (WSII_BOARD      hBoard,  
                               WSII_PE_NUM     dPeNum,  
                               DWORD           dOffset,  
                               DWORD           dDataToWrite )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
dOffset	DWORD offset in to the PE's memory space
dDataToWrite	DWORD containing the data. to write

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

All API calls use the host computer as a frame of reference; “write” for this function means that data will be transferred to the WILDSTAR™-II board from the host computer.

This call performs a Programmed I/O (PIO) write operation to the WILDSTAR™-II board, moving a single 32-bit value (a DWORD) to dPeNum.

Example:

```
WSII_RetCode rc;  
  
/* Write the value 0xDEADBEEF to DWORD offset 0x100 */  
rc=WSII_WriteReg_32(BoardHandle, WSII_PE0, 0x100, 0xDEADBEEF);
```

2.3.2.3 WSII_ReadRegs_32

```
WSII_RetCode WSII_ReadRegs_32 (WSII_BOARD      hBoard,  
                               WSII_PE_NUM     dPeNum,  
                               DWORD           dOffset,  
                               DWORD           dNumDwords,  
                               DWORD           *pData )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
dOffset	DWORD offset in to the PE's memory space
dNumDwords	Number of DWORDs to read
pData	Pointer to a DWORD buffer that will receive the data.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

All API calls use the host computer as a frame of reference; “read” for this function therefore indicates that data will be transferred from the WILDSTAR™-II board to the host computer.

This call performs a Programmed I/O (PIO) read operation from the WILDSTAR™-II board, moving dNumDwords pieces of 32-bit (DWORD) data from dPeNum into the user's pData buffer. The transfer will start from the PE's at dOffset and end at dOffset+dNumDwords. Note that the caller of this function must allocate sufficient storage, pointed to by pData.

Example:

```
int    i;  
DWORD *pBuffer;  
  
pBuffer = (DWORD*)malloc(100*sizeof(DWORD));  
if (pBuffer != NULL)  
{  
    WSII_ReadRegs_32 (BoardHandle, WSII_PE0, 0x0, 100, pBuffer);  
  
    /* Display the buffer */  
    for (i=0; i<100; i++)  
    {  
        printf("pBuffer[%2d]=0x%08x\n", i, pBuffer[i]);  
    }  
}  
free(pBuffer);
```

2.3.2.4 WSII_WriteRegs_32

```
void* WSII_WriteRegs_32 ( WSII_BOARD      hBoard,  
                          WSII_PE_NUM    dPeNum,  
                          DWORD          dOffset,  
                          DWORD          dNumDwords,  
                          DWORD          *pData )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
dOffset	DWORD offset in to the PE's memory space
dNumDwords	Number of DWORDs to write
pData	Pointer to a DWORD buffer that contains the data.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

All API calls use the host computer as a frame of reference; “write” for this function means that data will be transferred to the WILDSTAR™-II board from the host computer.

This call performs a Programmed I/O (PIO) write operation to the WILDSTAR™-II board, moving dNumDwords pieces of 32-bit (DWORD) data from the user's pData buffer to dPeNum

Example:

```
WSII_RetCode rc;  
DWORD       pBuffer[10];  
int         i;  
  
for (i=0; i<10; i++)  
{  
    pBuffer[i] = i;  
}  
  
/* This will write 10 DWORDs to PE0 at  
 * starting at DWORD offset 0x100 */  
rc = WSII_WriteRegs_32(BoardHandle, WSII_PE0, 0x100, 10,  
pBuffer);
```

2.3.3 Processing Element API Functions

The following API functions are used to manipulate the WILDSTAR™-II board PEs.

2.3.3.1 WSII_PeProgram

```
WSII_RetCode WSII_PeProgram ( WSII_BOARD      hBoard,  
                              WSII_PE_NUM    dPeNum,  
                              DWORD          *pBuffer,  
                              DWORD          NumDwords )
```

Parameters:

<code>hBoard</code>	Board handle returned by <code>WSII_Open()</code>
<code>dPeNum</code>	target PE number (see <i>wsii.h</i> for <code>WSII_PE_NUM</code> definition)
<code>pBuffer</code>	DWORD buffer containing a PE image
<code>dNumDwords</code>	Number of DWORDs to contained in <code>pBuffer</code>

Returns:

`WSII_SUCCESS`, if successful; otherwise a valid `WSII_RetCode`. See *wsii.h* for a list of possible errors.

Remarks:

This call either configures a Processing Element from a user-supplied FPGA image or from the WILDSTAR™-II PE's two local FLASH memory banks, or it programs a user-supplied FPGA image into one of the WILDSTAR™-II PE's two local FLASH memory banks.

Typically, the user will load the FPGA image file from a hard drive into host memory, and then call this function to program the WILDSTAR™-II FPGA Processing Element. Alternatively, this call can target the PE local FLASH memory.

To write a PE image into FLASH memory, the destination PE and PE Image Bank are selected by passing in an appropriate `dPeNum` (e.g., `WSII_WRITE_TO_FLASH_PE0_BANK0` to target PE0, Bank1), PE Image Buffer (`pBuffer`), and DWORD count (`NumDwords`). The API will then write the PE image into FLASH memory.

INFORMATION NOTE

Please be aware that FLASH memory operations can take several minutes to complete.



By default, the API will not display ellipses while the PE image is being programmed into FLASH. Since this operation may take while to complete, it may be desirable to have the API write a series of ellipses (...) to the screen (via `stdout`). If ellipses are desired, the board should be opened with the `WSII_FLAGS_FLASH_SHOW_DOTS` flag “OR’ed” with any other `WSII_Open()` flags.

After an FPGA image has been written to the WILDSTAR™-II’s FLASH memory, it will remain programmed until it is overwritten with another image.

To program a PE from FLASH memory, is it necessary only to specify its bank number (e.g., `WSII_PROG_FROM_FLASH_PE0_BANK0`), `pBuffer` and `NumDwords` can be `NULL` and `0x0`, respectively.

Example:

```

WSII_RetCode rc;
DWORD      *pBuffer;
char       pFileName[] = "PeImage.x86";
int        NumBytes;
FILE       *pFile;

if ( (pFile=fopen(pFileName,"rb")) != NULL )
{
    /* get the file's size */
    fseek( pFile, 0, SEEK_END );
    NumBytes = ftell(pFile);
    fseek( pFile, 0, SEEK_SET );

    /* make sure have DWORD alignment */
    if ( NumBytes % sizeof(DWORD) )
        NumBytes += (sizeof(DWORD) - (NumBytes % sizeof(DWORD)));

    pBuffer = (DWORD*)malloc( NumBytes );
    if( pBuffer != NULL )
    {
        memset(pBuffer,0x0,NumBytes);
        fread( pBuffer, 1, NumBytes, pFile );
        if( !ferror(pFile) )
        {
            rc = WSII_PeProgram( BoardHandle,
                                WSII_PE0,
                                pBuffer,
                                NumBytes>>2 );

            /* Write the PE image into FLASH */
            if (rc == WSII_SUCCESS)
            {
                rc = WSII_PeProgram( BoardHandle,
                                    WSII_WRITE_TO_FLASH_PE0_BANK0,
                                    (DWORD*)pBuffer,
                                    NumBytes>>2 );
            }
        }
    }
}

```

```

        /* Now program the PE from FLASH */
if (rc == WSII_SUCCESS)
{
    /* Deprogram the PE so we know that
    * the following PeProgram worked */
    WSII_PeDeprogram( BoardHandle, WSII_PE0 );
    rc = WSII_PeProgram( BoardHandle,
                        WSII_PROG_FROM_FLASH_PE0_BANK0,
                        NULL,
                        0x0 );
}
}
free(pBuffer);
}
fclose( pFile );
}

```

2.3.3.2 WSII_PeProgramDaisyChain

```
WSII_RetCode  
WSII_PeProgramDaisyChain (WSII_BOARD           hBoard,  
                          WSII_BOARD_SITE      BoardSite,  
                          WSII_DAISSY_CHAIN_BUFFER *pBuffer,  
                          DWORD                NumBuffers);
```

Parameters:

hBoard Board handle returned by WSII_Open()
BoardSite WSII_EXTIO_0 or WSII_EXTIO_1
pBuffer DWORD buffer containing a PE image
dNumBuffers Number of WSII_DAISSY_CHAIN_BUFFERs that the pBuffer
array points to.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h*
for a list of possible errors.

Remarks:

This call configures a daisy-chain of Processing Elements on a WSDP
External I/O board. The caller allocates an array of
WSII_DAISSY_CHAIN_BUFFER structures, then fills in each structure with the
address and length of a valid PE images.

Each of these structures describes a particular daisy-chain Processing
Element. For example, on an External I/O board attached to the #0 I/O card
site, the first element of the pBuffer (pBuffer[0]) corresponds to
WSII_EXTIO_0_PE1

This call should ONLY be used for programming WSDP External I/O cards.

Example:

```
WSII_DAISSY_CHAIN_BUFFER pDCBuff[3];  
char *pFileNames[] = {"F1.x86", "F2.x86", "F3.x86"};  
int NumBytes;  
DWORD index;  
FILE *pFile;  
  
for (index=0; index<3; index++)  
{  
    pFile = fopen(pFileNames[index], "rb");  
    if (pFile)  
    {  
        fseek( pFile, 0, SEEK_END );  
        NumBytes = ftell(pFile);  
        fseek( pFile, 0, SEEK_SET );  
  
        pDCBuff[index].pProgBuffer = malloc(NumBytes);
```

```
    if (pDCBuff[index].pProgBuffer)
    {
        fread( pDCBuff[index].pProgBuffer, 1, NumBytes, pFile);
        pDCBuff[index].dNumBufferDwords = NumBytes>>2;
        fclose(pFile);
    }
}
else
    return(-1);
}

WSII_PeProgramDaisyChain ( BoardHandle,
                           WSII_EXTIO_0,
                           pDCBuff,
                           3);
```

2.3.3.3 WSII_PeProgramOnStartup

```
WSII_RetCode  
WSII_PeProgramOnStartup( WSII_BOARD      hBoard,  
                          WSII_PE_NUM    PeNum,  
                          BOOLEAN        bProgramOnStartup )
```

Parameters:

hBoard Board handle returned by WSII_Open()
dPeNum PE number and Bank Number to target
bProgramOnStartup TRUE=Program on startup; FALSE=Don't program

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

The WILDSTAR™-II board has the ability to program FPGA Processing Elements when the board is powered up. When this call is made, the PeNum argument contains both the PE number and the FLASH Bank Number (either 0 or 1).

For example, to have the WILDSTAR-II board's firmware program PE0 from PE0 Flash Bank 1, set PeNum to WSII_PROG_FROM_FLASH_PE0_BANK1, and to bProgramOnStartup to TRUE. See the WSII_PE_NUM enumerated type in *wsii.h* for valid PE and FLASH Memory bank names.

INFORMATION NOTE

i

It is important write a valid Processing Element image into a PE's FLASH memory *before* calling this function to set PE programming on startup to TRUE.

Example:

```
WSII_PeProgramOnStartup( BoardHandle,  
                          WSII_PROG_FROM_FLASH_PE0_BANK0, TRUE );
```

2.3.3.4 WSII_PeDeprogram

```
WSII_RetCode WSII_PeDeprogram ( WSII_BOARD hBoard,  
                                WSII_PE_NUM dPeNum )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This function deprograms the Processing Element, placing it in a safe state. This API is typically called when a PE's temperature exceeds a desired level, or when a user has completed a task involving a PE and would like to "erase" an application before ending the session.

Note that on some I/O boards containing PEs in a daisy-chain (for example, the WSDP™ I/O boards), it is not possible to individually deprogram a PE with this call. All PEs on the target daisy-chain will be de-programmed simultaneously by this call if any PE in that daisy chain is targeted.

Example:

```
WSII_PeDeprogram (BoardHandle, WSII_PE0);
```

2.3.3.5 WSII_PeReset

```
WSII_RetCode WSII_PeReset (WSII_BOARD hBoard,  
                           DWORD dPeMask,  
                           WSII_RESET_OP Operation)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeMask	Valid PE Mask. See Remarks for more information.
Operation	ASSERT, DEASSERT, or PULSE

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This call asserts, de-asserts, or pulses one or more Processing Element reset lines. To create the PE bit-mask, the caller should logically conduct an OR operation on the mask constants found in *wsii.h*.

Example:

The following code toggles the PE reset lines going to PE0 and PE1 simultaneously:

```
WSII_RetCode rc;  
DWORD Mask;  
  
Mask = WSII_MASK_PE0 | WSII_MASK_PE1;  
rc = WSII_PeReset (BoardHandle, Mask, WSII_RESET_PULSE);
```

2.3.3.6 WSII_PeEnableSramDll

```
WSII_RetCode WSII_PeEnableSramDll (WSII_BOARD      hBoard,  
                                   DWORD           dPeMask,  
                                   BOOLEAN          bEnPeSramDll)
```

Parameters:

hBoard Board handle returned by WSII_Open()
dPeMask Valid PE Mask. See Remarks for more information.
bEnPeSramDll TRUE=Enable PE SRAM DLL; FALSE= Disable PE SRAM
 DLL

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This call enables or disables the SRAM DLL associated with the specified Processing Element(s). To create the PE bit-mask, the caller should logically conduct an OR operation on the mask constants found in *wsii.h*.



INFORMATION NOTE

This API call is only valid for WILDSTAR™-II boards, not WILDSTAR™-II PRO boards.

Example:

The following code enables the SRAM DLLs for PE0 and PE1:

```
WSII_RetCode rc;  
DWORD       Mask;  
  
Mask = WSII_MASK_PE0 | WSII_MASK_PE1;  
rc = WSII_PeEnableSramDll (BoardHandle, Mask, TRUE);
```

2.3.3.7 WSII_PeExtioInterfaceEnable

```
WSII_RetCode WSII_PeExtioInterfaceEnable (WSII_BOARD      hBoard,  
                                           WSII_EXTIO_SITE  ExtioSite,  
                                           BOOLEAN          bEnable)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ExtioSite	External I/O board number
bEnable	TRUE=Enable I/O board interface; FALSE= Disable I/O board interface

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This call enables or disables the External I/O board interfaces and is for VHDL users only. The VHDL that supports the Virtex™-II Pro PE parts does not enable the I/O interfaces until the application calls this API. The call checks to make sure that the interfaces that were instantiated are compatible. If they are, then it enables or disables them. If the interfaces instantiated are not compatible, then an error code is returned.

Example:

The following code enables the External I/O board 0:

```
WSII_RetCode    rc;  
WSII_EXTIO_SITE extioBoardNum;  
  
extioBoardNum = WSII_EXTIO_SITE_0;  
rc = WSII_PeExtioInterfaceEnable (BoardHandle, extioBoardNum,  
    TRUE);
```

2.3.3.8 WSII_ResetPciDcm

```
WSII_RetCode WSII_ResetPciDcm (WSII_BOARD  hBoard,  
                               WSII_PE_NUM  dPeNum)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This call resets the DCM in the PCI controller that is associated with the target PE number.

Example:

This code resets the DCM associated with PE0:

```
WSII_RetCode rc;  
  
rc = WSII_ResetPciDcm (BoardHandle, WSII_PE0);
```

2.3.3.9 WSII_ProgrammedPes

```
WSII_RetCode WSII_ProgrammedPes (WSII_BOARD          hBoard,  
                                 WSII_PE_PROGRAMMED *pProgPeMask)
```

Parameters:

hBoard Board handle returned by WSII_Open()
pProgPeMask Pointer to mask indicating currently programmed PEs.

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Example:

```
WSII_RetCode rc;  
WSII_PE_PROGRAMMED pes;  
  
rc = WSII_ProgrammedPes (BoardHandle, &pes);  
  
printf("Mask of programmed pes: 0x%x\n", pes);
```

2.3.4 Clock API Functions

Below is a listing of the clock API functions for the WILDSTAR™-II board.

2.3.4.1 WSII_GetClockFrequency

```
WSII_RetCode WSII_GetClockFrequency (WSII_BOARD    hBoard,  
                                     WSII_CLOCK_SRC ClockSrc,  
                                     DOUBLE         *pClockFreqMHz)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ClockSrc	Programmable clock generator chip to target
pClockFreqMHz	Frequency, in MHz, of the programmable clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors. Note that if the clock frequency has not yet been set, this call may return WSII_ERR_INVALID_FREQUENCY_ARG.

Remarks:

The clock generator chips achieve their variable output frequencies by using a ratio of clock multipliers and dividers. As a result, the exact frequency requested may not be attainable. If precise frequency control is critical, please check the frequency either with this function or with the value returned by WSII_SetClockFrequency() to make sure that the frequency requested is the frequency programmed into the desired clock generator.

The global ICLK cannot be modified, and it cannot be read back with this call. (See the *WILDSTAR™-II Hardware Reference Manual*, Chapter Six, for more information.)

Example:

```
WSII_RetCode rc;  
DOUBLE      SetFreqMhz;  
  
rc = WSII_GetClockFrequency (BoardHandle,  
                             WSII_CLK_PE0_A,  
                             &SetFreqMhz);  
  
printf("Clock is set to %3.2f MHz\n",SetFreqMhz);
```

2.3.4.2 WSII_SetClockFrequency

```
WSII_RetCode  
WSII_SetClockFrequency (WSII_BOARD      hBoard,  
                        WSII_CLOCK_SRC  ClockSrc,  
                        DOUBLE          fClockFreqMHz,  
                        DOUBLE          *pClockFreqMHz )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ClockSrc	Programmable clock generator chip to target
fClockFreqMHz	Desired clock frequency (in Megahertz)
pClockFreqMHz	Actual frequency, in MHz, of the programmable clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

For this function, it is critical to remember that there are many programmable clocks on the WILDSTAR™-II board; some are local to a particular Processing Element (PE's A,B,C), while some are global to the entire board (MCLK, PCLK). *Changing one of the global clocks affects the entire board.*

The WILDSTAR™-II PRO board has U clock for the PE's and the External I/O's, while its global clocks are P clock and REF clock.



CAUTION

Changing P clock on WILDSTAR™-II-PRO boards is not recommended if the PEs are already programmed. It is recommended that P clock be set during application initialization before the PEs are programmed.



CAUTION

Changing one of the global clocks affects the entire board.



INFORMATION NOTE

For WILDSTAR™-II boards, PE0's ACLK drives UCLK for external I/O card 0. Likewise, PE1's ACLK drives UCLK for external I/O card 1.

Note also that the clock generator chips achieve their variable output frequencies by using a ratio of clock multiplier and dividers. Consequently, the exact frequency requested may not be attainable. If precise frequency

control is critical, please check the frequency returned by this function in `pClockFreqMHz` to make sure that the frequency requested is, in fact, the frequency programmed.

The global ICLK cannot be set by users. See the *WILDSTAR™-II Hardware Reference Manual* for more information.

Example:

```
WSII_RetCode rc;
DOUBLE      SetFreqMhz;

rc = WSII_SetClockFrequency( BoardHandle,
                             WSII_CLK_PE0_A,
                             50.0,
                             &SetFreqMhz);

if (SetFreqMhz != 50.0)
    printf("Clock was programmed to %3.2f MHz\n",SetFreqMhz);
```

2.3.4.3 WSII_GetStartupFrequency

```
WSII_RetCode  
WSII_GetStartupFrequency( WSII_BOARD      hBoard,  
                          WSII_CLOCK_SRC  ClockSrc,  
                          DOUBLE          *pClockFreqMHz)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ClockSrc	Programmable clock generator chip to target
pClockFreqMHz	Frequency, in MHz, of the programmable clock

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This API function call will get the frequency that a particular clock will be set to at board power-up or at board RESET. Only the global “M” and “P” clocks can be programmed at power-up, and they will be programmed to the **same** frequency; therefore, the only valid ClockSrc values are WSII_CLK_GLOBAL_P and WSII_CLK_GLOBAL_M.

Example:

```
WSII_RetCode rc;  
DOUBLE FreqMhz;  
  
rc = WSII_GetStartupFrequency( BoardHandle,  
                              WSII_CLK_GLOBAL_M,  
                              &FreqMhz );  
  
printf("At power on, the 'M' and 'P' clocks will be  
      programmed to %3.2f MHz\n", FreqMhz);
```

2.3.4.4 WSII_SetStartupFrequency

```
WSII_RetCode  
WSII_SetStartupFrequency( WSII_BOARD      hBoard,  
                          WSII_CLOCK_SRC  ClockSrc,  
                          DOUBLE         fClockFreqMHz,  
                          DOUBLE         *pClockFreqMHz )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ClockSrc	Programmable clock generator chip to target
fClockFreqMHz	Desired clock frequency (in Megahertz)
pClockFreqMHz	Actual frequency, in MHz, of the programmable clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

The WILDSTAR™-II board is able to program the “M” and “P” clocks to the same user-specified frequency at startup. To do this, the user executes this call, specifying WSII_CLK_GLOBAL_M or WSII_CLK_GLOBAL_P, and a desired frequency. The frequency is written to FLASH, and a bit is set in FLASH indicating that the specified clock is to be programmed at startup. To have both “M” and “P” clocks program at startup, this call must be called twice: once for WSII_CLK_GLOBAL_M, and once for WSII_CLK_GLOBAL_P. For more information on clock programming, see WSII_SetClockFrequency(), above, or see the *WILDSTAR™-II Hardware Reference Manual*.

The WILDSTAR™-II-PRO board is able to program the “REF”, “P”, and “U” clocks to a user-specified frequency at startup.

INFORMATION NOTE



This call does not program the specified clock when called. This call writes the user-specified frequency to the non-volatile FLASH memory so that the WILDSTAR™-II board's firmware can program both the MCLK and PCLK with the clock frequency at power-on.

Example:

```
WSII_RetCode rc;  
DOUBLE      SetFreqMhz;  
  
rc = WSII_SetStartupFrequency( BoardHandle, WSII_CLK_GLOBAL_M,  
                              66.0, &SetFreqMhz);  
printf("MCLK will be programmed to %3.2f MHz at startup.\n",
```

```
SetFreqMhz );
```

2.3.4.5 WSII_SetUClockSource

```
WSII_RetCode  
WSII_SetUClockSource (WSII_BOARD           hBoard,  
                      WSII_PE_NUM         dPeNum,  
                      WSII_U_CLOCK_INPUT_SRC ClockInputSrc)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
ClockInputSrc	Selects the source for U clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II-PRO boards.

This call allows the user to select the input source for U clock.

Example:

```
WSII_RetCode rc;  
WSII_U_CLOCK_INPUT_SRC  ClockInputSrc;  
  
ClockInputSrc = WSII_PCLK;  
  
rc = WSII_SetUClockSource( BoardHandle,  
                           WSII_PE0,  
                           ClockInputSrc);  
  
If( rc == WSII_SUCCESS )  
{  
    printf("U Clock is now using P clock as its source \n");  
}
```

2.3.4.6 WSII_SetPClockSource

```
WSII_RetCode  
WSII_SetPClockSource (WSII_BOARD          hBoard,  
                     WSII_P_CLOCK_INPUT_SRC ClockInputSrc)
```

Parameters:

hBoard Board handle returned by WSII_Open()
ClockInputSrc Selects the source for P clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II PRO boards.

This call allows the user to select the input source for P clock.

Example:

```
WSII_RetCode rc;  
WSII_P_CLOCK_INPUT_SRC  ClockInputSrc;  
  
ClockInputSrc = WSII_PCLK_SYNTHESIZER;  
  
rc = WSII_SetPClockSource( BoardHandle,  
                           ClockInputSrc);  
If( rc == WSII_SUCCESS )  
{  
  printf("P Clock is now using the P clock synthesizer as its  
  source \n");  
}
```

2.3.4.7 WSII_GetUClockSource

```
WSII_RetCode  
WSII_GetUClockSource (WSII_BOARD           hBoard,  
                      WSII_PE_NUM         dPeNum,  
                      WSII_U_CLOCK_INPUT_SRC *pClockInputSrc)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
pClockInputSrc	On return contains the current source for U clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II-PRO boards.

This call allows the user to retrieve the current input source for U clock.

Example:

```
WSII_RetCode rc;  
WSII_U_CLOCK_INPUT_SRC  ClockInputSrc;  
  
rc = WSII_GetUClockSource( BoardHandle,  
                          WSII_PE0,  
                          &ClockInputSrc);  
  
If( rc == WSII_SUCCESS )  
{  
    printf("U Clock is sourcing from %d\n", ClockInputSrc);  
}
```

2.3.4.8 WSII_GetPClockSource

```
WSII_RetCode  
WSII_GetPClockSource (WSII_BOARD          hBoard,  
                      WSII_P_CLOCK_INPUT_SRC *pClockInputSrc)
```

Parameters:

hBoard Board handle returned by WSII_Open()
pClockInputSrc On return contains the source for P clock

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II PRO boards.

This call allows the user to retrieve the current input source for P clock.

Example:

```
WSII_RetCode rc;  
WSII_P_CLOCK_INPUT_SRC ClockInputSrc;  
  
rc = WSII_GetPClockSource( BoardHandle,  
                           &ClockInputSrc);  
If( rc == WSII_SUCCESS )  
{  
    printf("P Clock is currently sourcing from %d\n",  
          ClockInputSrc);  
}
```

2.3.4.9 WSII_SetCrystalOsc

```
WSII_RetCode  
WSII_SetCrystalOsc (WSII_BOARD          hBoard,  
                   WSII_CLOCK_SRC      ClockSrc,  
                   WSII_CRYSTAL_OSC    CrystalOsc)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ClockSrc	Specifies the clock source
CrystalOsc	Specifies which crystal will be used

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II PRO boards. Further this API is only supported on a WILDSTAR™-II PRO ACE VME Rev B or greater, or a WILDSTAR™-II PRO PCI Rev C or greater, or on any WILDSTAR™-II PRO VME

This call sets the crystal oscillator that is to be used by the Rocket I/O clocks. Valid clock sources for this API are WSII_CLK_GLOBAL_REF (clock 0 for rocket I/O) or WSII_CLK_GLOBAL_REF_1 (clock 1 for rocket I/O). Note that WSII_CLK_GLOBAL_REF_1 is only present on VME boards.

Example:

```
WSII_RetCode rc;  
WSII_CLOCK_SRC  ClockInputSrc;  
WSII_CRYSTAL_OSC CrystalOsc;  
  
ClockInputSrc = WSII_CLK_GLOBAL_REF;  
CrystalOsc    = WSII_CRYSTAL_OSC_0  
  
rc = WSII_SetCrystalOsc ( BoardHandle, ClockInputSrc,  
                          CrystalOsc );  
  
If( rc == WSII_SUCCESS )  
{  
    printf("Ref clock 0 is now sourcing crystal 0\n");  
}
```

2.3.4.10 WSII_GetCrystalOscFreq

```
WSII_RetCode  
WSII_GetCrystalOscFreq (WSII_BOARD      hBoard,  
                        WSII_CLOCK_SRC  ClockSrc,  
                        WSII_CRYSTAL_OSC CrystalOsc,  
                        DOUBLE          *pFreqMHz)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ClockSrc	Specifies the clock source
CrystalOsc	Specifies which crystal will be used
pFreqMHz	On return contains the frequency of the crystal

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II PRO boards. Further this API is only supported on a WILDSTAR™-II PRO ACE VME Rev B or greater, or a WILDSTAR™-II PRO PCI Rev C or greater, or on any WILDSTAR™-II PRO VME

This call gets the frequency of the specified crystal oscillator. Valid clock sources for this API are, WSII_CLK_GLOBAL_REF (clock 0 for rocket I/O) or WSII_CLK_GLOBAL_REF_1 (clock 1 for rocket I/O). Note that WSII_CLK_GLOBAL_REF_1 is only present on VME boards.

Example:

```
WSII_RetCode rc;  
WSII_CLOCK_SRC  ClockInputSrc;  
WSII_CRYSTAL_OSC CrystalOsc;  
DOUBLE          freqMHz;  
  
ClockInputSrc = WSII_CLK_GLOBAL_REF;  
CrystalOsc    = WSII_CRYSTAL_OSC_0  
  
rc = WSII_GetCrystalOscFreq ( BoardHandle, ClockInputSrc,  
                             CrystalOsc, &freqMHz);  
  
If( rc == WSII_SUCCESS )
```

```
{  
    printf("Crystal 0 has a frequency of %f MHz\n", freqMHz);  
}
```

2.3.5 Thermal Management API Functions

Various Xilinx® components on the WILDSTAR™-II have special features designed to monitor temperature and manage thresholds to prevent excessive heat buildup on the board.

The Xilinx® parts monitored include:

- Processing Elements, excluding those on external I/O cards
- PCI Controller
- Board-ambient temperature at several locations (Not available on WILDSTAR™-II-PRO boards)

If the temperature monitor detects that a Processing Element (PE) or the PCI Controller have exceeded a configurable temperature threshold, an interrupt is generated. If the temperature of a PE exceeds the shutdown threshold, the PE is deprogrammed by the hardware and a separate interrupt is generated. The PCI Controller can generate a shutdown interrupt event itself, but it will not be deprogrammed.

2.3.5.1 WSII_GetTemperature

```
WSII_RetCode  
WSII_GetTemperature ( WSII_BOARD           hBoard,  
                     WSII_TEMPERATURE_SRC TemperatureSrc,  
                     DOUBLE              *pTemperature )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
TemperatureSrc	Temperature sensor to target, must be of type WSII_TEMPERATURE_SRC.
pTemperature	The “junction” or “die” temperature of a particular PE or the temperature of a specific area of the WILDSTAR™-II board in Celsius.

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

The temperature of a WILDSTAR™-II board depends on many factors, including the total number of flip-flops in a design, the frequency at which it is run, the ambient air temperature, airflow over the board, and devices such as heat sinks and fans attached directly to the PEs. It is important to monitor PE temperature for a target application so avoid overheating.

To help monitor the board’s temperature, this function returns the temperature, in Celsius, of a geographic location on the WILDSTAR™-II board or temperature of a specific Processing Element.

The `TemperatureSrc` argument determines which temperature sensor is targeted by this call; the PCI controller, a PE, or an on-board sensor (WILDSTAR™-II only). The sensors in the PEs are located on the FPGA die and reflect the actual die temperature of the PE. The remaining temperature sensors on the board are labeled `WSII_TEMP_SRC_LOC0` - `WSII_TEMP_SRC_LOC3`. For the locations of these sensors, see “Thermal Management,” located in Chapter Six of the *WILDSTAR™- II Hardware Reference Manual*.

Example:

```
WSII_RetCode rc;  
DOUBLE      MeasuredTempC;  
  
rc = WSII_GetTemperature ( BoardHandle,  
                          WSII_TEMP_SRC_PE1,  
                          &MeasuredTempC );
```

```
printf("PE1 die temperature is %3.2f C\n", MeasuredTempC);
```

2.3.5.2 WSII_GetTemperatureThresh

```
WSII_RetCode  
WSII_GetTemperatureThresh ( WSII_BOARD           hBoard,  
                             WSII_TEMP_THRESHOLD Thresh,  
                             DOUBLE              *pTemperature)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
Thresh	Temperature threshold to target, must be of type WSII_TEMP_THRESHOLD.
pTemperature	The warning or shutdown threshold temperature in Celsius.

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This function returns the temperature, in Celsius, at which the temperature monitor will generate an interrupt and/or deprogram the part.

The Thresh argument determines whether the warning or shutdown threshold is targeted by this call. The sensors in the PEs are located on the FPGA die. The same warning or shutdown level is used to monitor all of the PEs as well as the PCI controller.

Example:

```
WSII_RetCode rc;  
DOUBLE ThreshTempC;  
  
rc = WSII_GetTemperatureThresh (BoardHandle,  
                                WSII_WARNING_THRESHOLD,  
                                &ThreshTempC);  
  
printf("Warning Threshold temperature is %3.2f C\n",  
       ThreshTempC);
```

2.3.5.3 WSII_SetTemperatureThresh

```
WSII_RetCode  
WSII_SetTemperatureThresh ( WSII_BOARD           hBoard,  
                           WSII_TEMP_THRESHOLD  Thresh,  
                           DOUBLE              fTemperature )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
Thresh	Temperature threshold to target, must be of type WSII_TEMP_THRESHOLD.
fTemperature	The warning or shutdown threshold temperature in Celsius.

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This function is used to set the threshold that is used by the temperature monitor to determine when a part will be deprogrammed or warning event generated. The maximum allowed threshold value is 125.0 degrees Celsius unless set to be unlimited via call to WSII_LimitTemperatureThresh.



CAUTION

Allowing the FPGA to reach a temperature above 125 degrees Celsius may permanently damage the part.

The Thresh argument determines whether the warning or shutdown threshold is targeted by this call. The sensors in the PEs are located on the FPGA die. The same warning or shutdown level is used to monitor all of the PEs as well as the PCI controller.

Example:

```
WSII_RetCode rc;  
DOUBLE ThreshTempC = 100.0;  
  
printf("Setting Shutdown Threshold temperature to %3.2f C\n",  
ThreshTempC);  
  
rc = WSII_SetTemperatureThresh ( BoardHandle,  
                               WSII_SHUTDOWN_THRESHOLD,  
                               ThreshTempC );
```

2.3.5.4 WSII_LimitTemperatureThresh

```
WSII_RetCode  
WSII_LimitTemperatureThresh ( WSII_BOARD          hBoard,  
                               BOOLEAN            bEnable )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
bEnable	Enable or disable limit checking for warning or shutdown thresholds.

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This function allows the application to disable the limit checking performed on the `fTemperature` parameter used by the `WSII_SetTemperatureThresh` routine. When limit checking is enabled, the maximum allowed threshold value is 125 degrees Celsius. If limit checking is disabled, the threshold values are not limited.



CAUTION

If an FPGA or other board part reaches a temperature higher than 125 degrees Celsius it may be seriously damaged.

For WILDSTAR™-II boards the shutdown mechanism is completely disabled when the threshold value is set above 390 degrees Celsius.

For WILDSTAR™-II-PRO boards the shutdown mechanism is completely disabled when the threshold value is set above 127 degrees Celsius.

Operating above 85 degrees Celsius exceeds the recommended operating conditions for a commercial part. The default operation of `WSII_SetTemperatureThresh` performs limit checking on the `fTemperature` temperature threshold parameter.

Example:

```
WSII_RetCode rc;  
DOUBLE ThreshTempC = 130.0;  
  
printf("Setting Shutdown Threshold temperature to %3.2f C\n",  
ThreshTempC);  
  
rc = WSII_LimitTemperatureThresh ( BoardHandle,  
                                   FALSE ); /* Allow threshold > 125 */
```

```
rc = WSII_SetTemperatureThresh ( BoardHandle,  
                                WSII_SHUTDOWN_THRESHOLD,  
                                ThreshTempC );
```

2.3.5.5 WSII_QueryTemperatureEvents

```
WSII_RetCode  
WSII_QueryTemperatureEvents (WSII_BOARD          hBoard,  
                             DWORD              *pEventMask )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pEventMask	Bit mask of warning or shutdown events reported by temperature monitor.

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

On return, the *pEventMask* parameter contains a map describing any temperature events that have occurred.

The bits are assigned as follows:

- 0 - WSII_MASK_PCI_TEMPER_WRN
- 1 - WSII_MASK_PCI_TEMPER_SHTDN
- 2 - WSII_MASK_PE0_TEMPER_WRN
- 3 - WSII_MASK_PE0_TEMPER_SHTDN
- 4 - WSII_MASK_PE1_TEMPER_WRN
- 5 - WSII_MASK_PE1_TEMPER_SHTDN
- 6 - WSII_MASK_PE2_TEMPER_WRN
- 7 - WSII_MASK_PE2_TEMPER_SHTDN

If a bit is set, that temperature threshold has been exceeded.

A set of #defines are included in the *wsii.h* file to help the caller test the various temperature event bits. See *wsii.h* for more information.

Example:

```
WSII_RetCode rc;  
DWORD TempEvents;  
  
rc = WSII_QueryTemperatureEvents ( BoardHandle,  
                                  &TempEvents );  
  
printf("Temperature event mask is 0x%x\n", TempEvents);
```

2.3.5.6 WSII_ResetTemperatureEvents

```
WSII_RetCode  
WSII_ResetTemperatureEvents (WSII_BOARD    hBoard,  
                             DWORD        dResetMask)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dResetMask	Bit mask of warning or shutdown events to be reset.

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

The user sets the bits in dResetMask, then calls this function to reset Temperature events after they have occurred. The bits are assigned as follows:

- 0 - WSII_MASK_PCI_TEMPER_WRN
- 1 - WSII_MASK_PCI_TEMPER_SHTDN
- 2 - WSII_MASK_PE0_TEMPER_WRN
- 3 - WSII_MASK_PE0_TEMPER_SHTDN
- 4 - WSII_MASK_PE1_TEMPER_WRN
- 5 - WSII_MASK_PE1_TEMPER_SHTDN
- 6 - WSII_MASK_PE2_TEMPER_WRN
- 7 - WSII_MASK_PE2_TEMPER_SHTDN

If a bit is set, that temperature event will be cleared; otherwise, it will be left alone.

A set of #defines are included in the *wsii.h* file to help the caller test the various temperature event bits. See *wsii.h* for more information.

2.3.5.7 WSII_PeSetFanMode

```
WSII_RetCode WSII_PeSetFanMode(WSII_BOARD    hBoard,  
                               WSII_PE_NUM   dPeNum,  
                               WSII_FAN_MODE FanMode )
```



INFORMATION NOTE

Only the WILDSTAR™-II /PCI motherboard has a cooling fan option.

Parameters:

hBoard	Board handle returned by WSII_Open()
dPeNum	Valid PE number to target
FanMode	WSII_FAN_OFF or WSII_FAN_ON

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This function sets the fan in one of two operating modes: WSII_FAN_ON or WSII_FAN_OFF. If WSII_FAN_ON is selected, the cooling fan will run constantly. If WSII_FAN_OFF is selected, the fan will only run when the on-board temperature exceeds the temperature warning threshold.

Example:

```
WSII_RetCode rc;  
  
rc = WSII_PeSetFanMode(BoardHandle, WSII_PE0, WSII_FAN_ON);
```

2.3.6 Power API Functions

The voltage and current consumption of various sections of the WILDSTAR™-II are measured and monitored continuously by the WILDSTAR™-II's onboard firmware. The API can read these measurements from the hardware through the WSII_GetPower API function call.

2.3.6.1 WSII_GetPower

```
WSII_RetCode  
WSII_GetPower ( WSII_BOARD      hBoard,  
                WSII_POWER_SRC  PowerSrc,  
                DOUBLE          *pPowerReading )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
PowerSrc	Voltage or current source to be measured
pPowerReading	Voltage or current measurement of a particular PE, power plane, or the PCI controller.

Returns:

WSII_SUCCESS if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

On return, the pPowerReading parameter contains voltage in volts, and current in amps. For WILDSTAR™-II-PRO boards the only valid enumerators from the WSII_POWER_SRC enumeration are WSII_HOST_3_30I, WSII_HOST_5_00I, WSII_BOARD_3_30V, and WSII_BOARD_5_00V.

Example:

```
WSII_RetCode rc;  
DOUBLE      Voltage;  
DOUBLE      Current;  
  
rc = WSII_GetPower(BoardHandle, WSII_BOARD_3_30V, &Voltage);  
if (rc == WSII_SUCCESS)  
{  
    rc = WSII_GetPower(BoardHandle, WSII_HOST_3_30I, &Current);  
    if (rc == WSII_SUCCESS)  
    {  
        printf("Host 3.3v supply is %3.2f volts, ",Voltage);  
        printf ("and is supplying %3.2famps\n",Current);  
    }  
}
```

```
if (rc!=WSII_SUCCESS)
    printf("The WILDSTAR-II API returned the following
          error: %d (%s)\n",rc,WSII_GetErrorString(rc));
```

2.3.6.2 WSII_GetPowerStatus

```
WSII_RetCode  
WSII_GetPowerStatus ( WSII_BOARD          hBoard,  
                      WSII_PRO_POWER_STATUS *pStatusMask )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pStatusMask	Pointer to storage area where the mask of voltage status values are stored.

Returns:

WSII_SUCCESS if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II-PRO boards.

On return, the `pStatusMask` parameter contains a mask of any voltage sources that have been out of specification. These status bits hold their state once a power status is out of specification and must be cleared with a call to the API, `WSII_ClearPowerStatus`.

The mask bit definitions for the parameter, `pStatusMask` are:

WSII_PRO_POWER_STATUS_VBATTBAD	(0x00100000)	/* VBATT voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PCIV15BAD	(0x00080000)	/* PCI Controller 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_V25BAD	(0x00040000)	/* 2.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_V33BAD	(0x00020000)	/* 3.3V out of spec mask bit */
WSII_PRO_POWER_STATUS_5VBAD	(0x00010000)	/* 5V out of spec mask bit */
WSII_PRO_POWER_STATUS_IO1V25BAD	(0x00008000)	/* External I/O 1 2.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_IO1V33BAD	(0x00004000)	/* External I/O 1 3.3V out of spec mask bit */
WSII_PRO_POWER_STATUS_IO0V25BAD	(0x00002000)	/* External I/O 0 2.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_IO0V33BAD	(0x00001000)	/* External I/O 0 3.3V out of spec mask bit */

```

WSII_PRO_POWER_STATUS_PE2RAMBAD (0x00000400) /*
PE2 SRAM voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE2AUXBAD (0x00000200) /*
PE2 VCC AUX voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE2V15BAD (0x00000100) /*
PE2 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_PE1RAMBAD (0x00000040) /*
PE1 SRAM voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE1AUXBAD (0x00000020) /*
PE1 VCC AUX voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE1V15BAD (0x00000010) /*
PE1 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_PE0RAMBAD (0x00000004) /*
PE0 SRAM voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE0AUXBAD (0x00000002) /*
PE0 VCC AUX voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE0V15BAD (0x00000001) /*
PE0 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_MASK (0x001FF777) /* Mask
of all power status bits */

```

Example:

```

WSII_RetCode rc;
WSII_PRO_POWER_STATUS vStatus;

rc = WSII_GetPowerStatus(BoardHandle, &vStatus);
if (rc == WSII_SUCCESS)
{
    printf("Voltage status mask = 0x%x\n", vStatus);
}

if (rc!=WSII_SUCCESS)
    printf("The WILDSTAR-II API returned the following
error: %d (%s)\n",rc,WSII_GetErrorString(rc));

```

2.3.6.3 WSII_ClearPowerStatus

```
WSII_RetCode  
WSII_ClearPowerStatus ( WSII_BOARD          hBoard,  
                        WSII_PRO_POWER_STATUS ClearMask )
```

Parameters:

hBoard Board handle returned by WSII_Open()
ClearMask Mask of power status bits to clear.

Returns:

WSII_SUCCESS if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:



INFORMATION NOTE

This API is only valid on WILDSTAR™-II-PRO boards.

This API clears any power status bits that have been set as a result of voltage source(s) being out of specification.

The mask bit definitions for the parameter, `ClearMask` are:

```
WSII_PRO_POWER_STATUS_VBATTBAD    (0x00100000) /*  
VBATT voltage out of spec mask bit */  
WSII_PRO_POWER_STATUS_PCIV15BAD   (0x00080000) /* PCI  
Controller 1.5V out of spec mask bit */  
WSII_PRO_POWER_STATUS_V25BAD     (0x00040000) /* 2.5V  
out of spec mask bit */  
WSII_PRO_POWER_STATUS_V33BAD     (0x00020000) /* 3.3V  
out of spec mask bit */  
WSII_PRO_POWER_STATUS_5VBAD      (0x00010000) /* 5V  
out of spec mask bit */  
WSII_PRO_POWER_STATUS_IO1V25BAD   (0x00008000) /*  
External I/O 1 2.5V out of spec mask bit */  
WSII_PRO_POWER_STATUS_IO1V33BAD   (0x00004000) /*  
External I/O 1 3.3V out of spec mask bit */  
WSII_PRO_POWER_STATUS_IO0V25BAD   (0x00002000) /*  
External I/O 0 2.5V out of spec mask bit */  
WSII_PRO_POWER_STATUS_IO0V33BAD   (0x00001000) /*  
External I/O 0 3.3V out of spec mask bit */  
WSII_PRO_POWER_STATUS_PE2RAMBAD   (0x00000400) /*  
PE2 SRAM voltage out of spec mask bit */
```

```

WSII_PRO_POWER_STATUS_PE2AUXBAD (0x00000200) /*
PE2 VCC AUX voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE2V15BAD (0x00000100) /*
PE2 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_PE1RAMBAD (0x00000040) /*
PE1 SRAM voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE1AUXBAD (0x00000020) /*
PE1 VCC AUX voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE1V15BAD (0x00000010) /*
PE1 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_PE0RAMBAD (0x00000004) /*
PE0 SRAM voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE0AUXBAD (0x00000002) /*
PE0 VCC AUX voltage out of spec mask bit */
WSII_PRO_POWER_STATUS_PE0V15BAD (0x00000001) /*
PE0 1.5V out of spec mask bit */
WSII_PRO_POWER_STATUS_MASK (0x001FF777) /* Mask
of all power status bits */

```

Example:

```

WSII_RetCode rc;
WSII_PRO_POWER_STATUS ClearMask;

rc = WSII_GetPowerStatus(BoardHandle, &ClearMask);
if (rc == WSII_SUCCESS)
{
    printf("Voltage status mask = 0x%x\n", ClearMask);
}

rc = WSII_ClearPowerStatus(BoardHandle, ClearMask);
if (rc == WSII_SUCCESS)
{
    printf("Voltage status mask was cleared.\n");
}

if (rc!=WSII_SUCCESS)
    printf("The WILDSTAR-II API returned the following
           error: %d (%s)\n",rc,WSII_GetErrorString(rc));

```

2.3.7 Interrupt API Functions

The WILDSTAR™-II board can generate interrupt from a variety of sources. WILDSTAR™-II interrupt calls operate like “level-sensitive” interrupts; in other words, once an interrupt has occurred it can later be detected by querying its status. Similarly, waiting for a PE interrupt after that interrupt has already occurred (but has not yet been reset by this call), will cause the “wait” to immediately return, indicating that the interrupt has been satisfied.

2.3.7.1 WSII_InterruptQueryStatus

```
WSII_RetCode  
WSII_InterruptQueryStatus (WSII_BOARD hBoard, DWORD *pIntMask)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pIntMask	Bit mask of interrupt sources that have interrupted

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

On return, the pIntMask parameter contains a map describing any interrupts that have occurred:

The bits are assigned as follows:

- 0 - WSII_MASK_PE0
- 1 - WSII_MASK_PE1
- 2 - WSII_MASK_PE2
- 3 - WSII_MASK_EXT0
- 4 - WSII_MASK_EXT1
- 5 - WSII_MASK_PIO_TO
- 6 - WSII_MASK_PIO_ERR
- 7 - WSII_MASK_PIO_OVRFLO
- 8 - WSII_MASK_TEMPER_WRN
- 9 - WSII_MASK_TEMPER_SHTDN
- 10 - WSII_MASK_DMA_ERROR
- 11 - WSII_MASK_SPLIT_COMP_ERR (PCI-X systems only)
- 12 - WSII_MASK_READ_RETRY_ERR (PCI-X systems only)
- 13 - WSII_MASK_WRITE_RETRY_ERR (PCI-X systems only)

If a bit is set, that interrupt source has interrupted. Note that on power-up, the interrupt lines are in an undefined state and should be explicitly enabled and cleared before use.

A set of #defines are included in the *wsii.h* file to help the caller test the various interrupt source bits. See *wsii.h* for more information.

2.3.7.2 WSII_InterruptReset

WSII_RetCode

WSII_InterruptReset(WSII_BOARD hBoard, DWORD dResetMask)

Parameters:

hBoard	Board handle returned by WSII_Open()
dResetMask	Bit mask of interrupt sources to be reset

Returns:

WSII_SUCCESS, if successful; otherwise, a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

The user sets the bits in dResetMask, then calls this function to reset interrupts after they have occurred. The bits are assigned as follows:

- 0 - WSII_MASK_PE0
- 1 - WSII_MASK_PE1
- 2 - WSII_MASK_PE2
- 3 - WSII_MASK_EXT0
- 4 - WSII_MASK_EXT1
- 5 - WSII_MASK_PIO_TO
- 6 - WSII_MASK_PIO_ERR
- 7 - WSII_MASK_PIO_OVRFLO
- 8 - WSII_MASK_TEMPER_WRN
- 9 - WSII_MASK_TEMPER_SHTDN
- 10 - WSII_MASK_DMA_ERROR
- 11 - WSII_MASK_SPLIT_COMP_ERR (PCI-X systems only)
- 12 - WSII_MASK_READ_RETRY_ERR (PCI-X systems only)
- 13 - WSII_MASK_WRITE_RETRY_ERR (PCI-X systems only)

If a bit is set, that interrupt source will be cleared; otherwise, it will be left alone. Note that on power-up, the interrupt lines are in an undefined state and should be explicitly enabled and cleared before use.

A set of #defines are included in the *wsii.h* file to help the caller test the various interrupt source bits. See *wsii.h* for more information.

2.3.7.3 WSII_InterruptWait

```
WSII_RetCode
WSII_InterruptWait      ( WSII_BOARD      hBoard,
                          DWORD          dWaitMask,
                          DWORD          *pIntMask,
                          DWORD          dTimeoutMs)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dWaitMask	Bit mask of interrupt(s) to wait for
pIntMask	Bit mask of the interrupt(s) that satisfied the wait
dTimeoutMs	Number of milliseconds to wait for an interrupt

Returns:

WSII_SUCCESS, if the wait was satisfied by an interrupt, or
WSII_ERR_INTERRUPT_TIMEOUT if dTimeoutMs milliseconds elapsed
without an interrupt.

Remarks:

The user sets bits in dWaitMask, corresponding to the interrupts he wants to wait for, and passes in a value for the timeout. On return, this call will have either timed out (and return WSII_ERR_INTERRUPT_TIMEOUT) or it contains a bit mask of the interrupt(s) that satisfied the wait (pIntMask) . This call behaves as a “wait for any interrupt” call; if an application needs to wait for multiple interrupts and have all of those interrupts satisfied before continuing program execution, the application can test the returned pIntMask to determine which interrupts occurred. Then, if necessary, the application can make another call into this API function call to wait for any unsatisfied interrupts.

Note that on power-up, the interrupt lines are in an undefined state and should be explicitly enabled and cleared before use. A set of #defines are included in the *wsii.h* file to help the caller test the various interrupt source bits. See *wsii.h* for more information.



INFORMATION NOTE

See important note about performing interrupts on the first page of section 2.3.7.

2.3.7.4 WSII_InterruptRegisterCallback

```
WSII_RetCode  
WSII_InterruptRegisterCallback (WSII_BOARD      hBoard,  
                                WSII_INTR_CBACK pIntrCback,  
                                void            *pUserContext)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pIntrCback	Pointer to interrupt callback function to be registered
pUserContext	Pointer to user context, which is passed to the interrupt callback function

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode.
See wsii.h for a list of possible errors.

Remarks:

If an interrupt callback function is registered then calls to the APIs, WSII_InterruptReset and WSII_InterruptWait will not succeed if the target interrupt masks are PE interrupt types (WSII_MASK_PE0, WSII_MASK_PE1, WSII_MASK_PE2, WSII_MASK_EXT0, WSII_MASK_EXT1). However, use of these calls is permitted if the target interrupt types are not PE types (WSII_MASK_PIO_TO, WSII_MASK_PIO_ERR, WSII_MASK_PIO_OVRFLO, WSII_MASK_TEMPER_WRN, WSII_MASK_TEMPER_SHTDN, WSII_MASK_DMA_ERROR).

To disable the use of an interrupt callback function the application can close and reopen the board or pass NULL as the pIntrCback parameter.

The interrupt callback function is called from an interrupt context, therefore only operations that are legal from interrupt context may be called in the callback function. Please see the VxWorks® documentation to learn what can be done in a function running at interrupt level context.



INFORMATION NOTE

This API is only supported under VxWorks®.

2.3.7.5 WSII_InterruptGetVmeVector

```
WSII_RetCode
WSII_InterruptGetVmeVector (WSII_BOARD      hBoard,
                           USHORT          *pIntLevel,
                           USHORT          *pIntVector)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pIntLevel	Pointer to VME interrupt level
pIntVector	Pointer to VME interrupt vector

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode.
See wsii.h for a list of possible errors.

Remarks:

This API allows you to get the VME interrupt level and vector that the target VME WILDSTAR-II™ board has been set to.



INFORMATION NOTE

This API is only supported on VME systems.

2.3.7.6 WSII_InterruptSetVmeVector

```
WSII_RetCode  
WSII_InterruptSetVmeVector (WSII_BOARD      hBoard,  
                             USHORT          wIntLevel,  
                             USHORT          wIntVector)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
wIntLevel	VME interrupt level to set for target board
wIntVector	VME interrupt vector to set for target board

Returns:

WSII_SUCCESS, if successful, otherwise a valid WSII_RetCode.
See wsii.h for a list of possible errors.

Remarks:

This API allows you to set the VME interrupt level and vector for the target VME WILDSTAR-II™ board.



INFORMATION NOTE

This API is only supported on VME systems.

2.3.8 Error Handling and Callback API Functions

2.3.8.1 WSII_SetErrorCallback

```
WSII_RetCode  
WSII_SetErrorCallback ( WSII_BOARD      hBoard,  
                       WSII_FLAGS     dFlags,  
                       void            *pUserFunction,  
                       void            *pUserContext )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dFlags	Must be set to WSII_FLAGS_RESERVED
pUserFunction	Function that the user wishes to be called on error
pUserContext	Pointer to user-allocated storage (or NULL)

Returns:

WSII_SUCCESS if successful; otherwise, it will be a numeric code indicating the error number.

Remarks:

The WILDSTAR™-II API allows a user to register an error-handling function with the API. If a user registers a call-back routine with the API, the function will be called whenever the API encounters an error. As a result, the function will be given the BoardHandle used in the call that received the error, the numeric value of the error, and a pointer that can point to whatever the user wishes. The prototype for the user's callback function is as follows:

```
void ErrorCallbackFunction(WSII_BOARD      hBoard,  
                          WSII_RetCode    ErrorCode,  
                          void            *pUserContext)
```



CAUTION

There are only two WILDSTAR™-II API calls that may be made from a callback routine: WSII_Close(), and WSII_GetErrorString(). *Do not attempt any other calls.*

Upon exiting the callback routine, the program will return back into the API, then immediately return to the user code at the point where the original call was made.

Example:

```
WSII_RetCode main()
{
    WSII_BOARD    BoardHandle=0x0;
    WSII_RetCode  rc;
    DWORD        pData[10];

    BoardHandle = WSII_Open ( 0x0, WSII_API_CODE,
                              &rc, WSII_FLAGS_OPEN );
    if (rc!=WSII_SUCCESS)
    {
        printf("Error Opening the board!\n");
        printf("(%d: %s)\n",rc,WSII_GetErrorString(rc));
    }

    rc = WSII_SetErrorCallback( BoardHandle, WSII_FLAGS_RESERVED,
                                ErrorCallbackFunction, NULL );
    if (rc!=WSII_SUCCESS)
    {
        printf("Error Registering the callback routine!\n");
        printf("(%d: %s)\n",rc,WSII_GetErrorString(rc));
        WSII_Close(BoardHandle);
    }

    /* From here on, we won't check error codes;
     * if an error is encountered in the code,
     * the callback function will display the error,
     * close the board, and exit the program */

    /* Assuming that we have a WILDSTAR-II PCI with
     * 2 PEs, the following code is valid, and should
     * not generate an error */
    WSII_ReadRegs_32 (BoardHandle, WSII_PE0, 0x0, 10, pData );

    /* The same board, however, would cause the
     * API to generate an error with the following line because
     * there can never be a WSII_PE2 on a WILDSTAR-II/PCI;
     * there are only WSII_PE0 and WSII_PE1 */
    WSII_ReadRegs_32 (BoardHandle, WSII_PE2, 0x0, 10, pData );
}

void
ErrorCallbackFunction( WSII_BOARD    BoardHandle,
                      WSII_RetCode  ErrorCode,
                      void          *pUserContext)
{
    printf("Callback received an API error!\n");
    printf("(%d: %s)\n",ErrorCode,WSII_GetErrorString(ErrorCode));
    WSII_Close(BoardHandle);
    exit(ErrorCode);
}
```

}

2.3.8.2 WSII_GetLastError

```
WSII_RetCode WSII_GetLastError ( WSII_BOARD hBoard )
```

Parameters:

hBoard Board handle returned by WSII_Open()

Returns:

The last error that the API encountered.

Remarks:

If a callback function is not registered (see WSII_SetErrorCallback), each API function's return code should be checked after every board operation to make sure that the call has completed successfully. For single-threaded programs, this is one way of checking for completion of the call.

Example:

```
WSII_RetCode rc;
DWORD pData[10];

WSII_ReadRegs_32 (BoardHandle, WSII_PE2, 0x0, 10, pData );
rc = WSII_GetLastError(BoardHandle);
if (rc !=WSII_SUCCESS)
{
    printf("Error reading from the board!\n");
    printf("API returned error %#d\n",rc);
    printf ("%s\n",WSII_GetErrorString(rc));
}
```

2.3.8.3 WSII_GetErrorString

```
char* WSII_GetErrorString (WSII_RetCode Error)
```

Parameters:

Error Error Code return by the API (see wsii.h for possible values)

Returns:

A pointer to a NULL-Terminated text string describing the error.

Example:

```
WSII_RetCode rc;
DWORD pData[10];

WSII_ReadRegs_32 (BoardHandle, WSII_PE2, 0x0, 10, pData );
rc = WSII_GetLastError(BoardHandle);
if (rc !=WSII_SUCCESS)
{
    printf("Error reading from the board!\n");
    printf("API returned error #d\n",rc);
    printf("%s\n",WSII_GetErrorString(rc));
}
```

2.3.9 LED Display API Functions

2.3.9.1 WSII_UpdateDisplay

```
WSII_RetCode WSII_UpdateDisplay (WSII_BOARD    hBoard,  
                                char          *pDisplayString )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pDisplayString	Pointer to user-allocated NULL-terminated text

Returns:

WSII_SUCCESS if successful; otherwise, it will be a numeric code indicating the error number.

Remarks:

The WILDSTAR™-II /VME has a four-character LED display. The Update Display call allows a user to update the display's text. The display can show numbers and letters (uppercase only) as well as spaces. pDisplayString must be NULL-terminated and must be four characters or less. To display a space, either a space character or an underscore may be used.

From power-up, the display is completely dimmed, so a call to WSII_SetDisplayLevel() must be made, and WSII_DISPLAY_LEVEL must be >= WSII_LEVEL_1.

Note that it is not valid to make this API function call on a PCI-based WILDSTAR™-II.

2.3.9.2 WSII_SetDisplayLevel

```
WSII_RetCode  
WSII_SetDisplayLevel ( WSII_BOARD          hBoard,  
                       WSII_DISPLAY_LEVEL  Level )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
Level	Specifies the intensity level of the LED display

Returns:

WSII_SUCCESS if successful; otherwise, it will be a numeric code indicating the error number.

Remarks:

The WILDSTAR™-II-VME has a four-character LED display. From power-up, the display is completely dimmed, so a call to WSII_SetDisplayLevel() must be made, and WSII_DISPLAY_LEVEL must be \geq WSII_LEVEL_1.

The following are valid Level enumerations:

WSII_LEVEL_0	Display is turned off
WSII_LEVEL_1	Dimmest setting
WSII_LEVEL_2	
WSII_LEVEL_3	
WSII_LEVEL_4	Brightest Setting

2.3.10 DMA API Functions

2.3.10.1 Working with DMA

Direct Memory Access (DMA) between the WILDSTAR™-II board and host memory is accomplished through five WILDSTAR™-II Application Programming Interface (API) function calls:

```
WSII_DmaMemAlloc  
WSII_DmaMemFree  
WSII_DmaUnbind  
WSII_DmaBind  
WSII_DmaErrorOp
```

Most of these function calls use a `WSII_DMA_HANDLE`, an opaque data structure that the API uses to keep track of its various operations. Each host memory allocation made via `WSII_DmaMemAlloc` will return a unique `WSII_DMA_HANDLE`. The detailed contents of this structure are not important to the user, but it is critical to use this handle in all subsequent DMA API calls that wish to use that buffer. Note that it is possible to have multiple `WSII_DMA_HANDLES`, each describing a different host memory buffer.

The `WSII_DmaMemAlloc` call and the `WSII_DmaMemFree` call are used for allocating (and releasing) a host memory buffer and its associated `WSII_DMA_HANDLE`. The buffer returned by the `WSII_DmaMemAlloc` call can be used as any memory would be used; files can be read from disk into and out of it, data copied into and out of it, manipulations can be made on the data stored in it, and so on.

Before the user performs a DMA operation, a `WSII_DmaBind` must first be called. This call performs three operations:

1. It locks the specified segment of the buffer associated with the `WSII_DMA_HANDLE` into physical host memory so it can't be paged out to disk during DMA operations.
2. It initializes the WILDSTAR™-II hardware to map host memory between the PE's LAD bus and the PCI bus.
3. It returns a range of addresses to be used by the PEs to read and write host memory.



CAUTION

The WILDSTAR™-II hardware that maps host memory between the PE's LAD bus and the PCI bus is a scatter/gather table that contains 8192 entries, each referencing 4096 bytes of contiguous host memory. `WSII_DmaBind` is called to bind host memory, allocated with `WSII_DmaMemAlloc`, to this scatter/gather table. The number of entries in the scatter/gather table used to bind the host memory is equal to $\text{ceil}((\text{mem_bytes})/4096) + 1$. The additional entry that is added is used to terminate a binding to prevent the PE from trying to access memory beyond the bounded target memory segment. The total amount of memory that can be bound at one time is dependent on the number of bindings, but the total is always less than 32 MB.

The `WSII_DmaBind` API call takes several parameters:

`hBoard`: the board handle returned from the call to `WSII_Open`.

`DmaHandle`: the DMA handle returned from a call to `WSII_DmaMemAlloc`

`pBuffer`: a `DWORD` pointer. This parameter allows a user to specify the beginning of the buffer segment that will be locked into physical memory and mapped into the WILDSTAR™-II board. This must point to a `DWORD` within the buffer allocated by `WSII_DmaMemAlloc`.

`dRequestedLengthDwords`: This is the number of `DWORD`s within the host buffer that the user would like to have locked into physical host memory. Note that this parameter, along with `pBuffer` will tell the WILDSTAR™-II device driver what segment of the buffer returned by `WSII_DmaMemAlloc` to use. It is perfectly acceptable to simply bind the entire buffer; just use the `dNumDwords` parameter passed into `WSII_DmaMemAlloc`, and the pointer returned by `WSII_DmaMemAlloc`.

The `HardwareAddress` parameter returned by `WSII_DmaBind` is a 32-bit value that represents the lowest possible address that the PE may use for a DMA operation. The highest address that a PE may use is the `HardwareAddress + GrantedLengthDwords`. It is important to note that, due to hardware constraints, it may not always be possible bind the caller's entire request length, so it is important to check this value to make sure it is equal to `dRequestedLengthDwords`. If it is

not, then the user may only DMA from the starting address up to `GrantedLengthDwords`.

Once the host buffer has been locked and the WILDSTAR™-II hardware has been initialized with the “bind” call, the user may write the `HardwareAddress` into the desired PE, and the PE may begin doing DMA operations (see the *WILDSTAR™-II Hardware Reference Manual* for more information). The host will have no knowledge or interaction with the DMA operations; it is completely up to the PE and its design to determine when to initiate DMA, when to stop, and what length to transfer (up to the maximum length granted by the “bind” call).

If the host needs to know when the PE has completed a DMA operation, the PE may generate an interrupt, or the host may poll the PE, or the PE may use DMA write a particular value into a pre-determined host memory location that the host application periodically polls; it is entirely up to the application designer. Since DMA is typically used so that the application is not consuming host CPU cycles, it is desirable to use interrupts; a thread simply waits for a PE interrupt. That thread consumes no CPU cycles until the PE generates an interrupt.

When the application has completed its DMA operation, it should signal any PEs that might be doing DMA to stop using DMA, then call `WSII_DmaUnbind`. After calling `WSII_DmaUnbind`, the PE will no longer be able to do DMA using memory bound with the handle specified to `WSII_DmaUnbind` (though it may still do DMA to/from memory bound by other DMA handles); the PCI controller will prevent it from accessing the PCI bus. If there is another portion of the host buffer that it would like to bind, it may do so (only a single segment of a DMA buffer may be bound at any one time), otherwise the application should call `WSII_DmaMemFree`.

`WSII_DmaErrorOp` is the only call that does not use a `WSII_DMA_HANDLE` parameter. It does not require that DMA memory to be bound, nor that memory even be allocated at all. This call is used for checking (and optionally) clearing the error state of a one (or more) of the ten DMA channels. Until now, the WILDSTAR™-II software manual has not discussed the concept of a “DMA Channel” because the API primarily serves to allocate and bind host memory for DMA use.

That memory, once properly allocated and bound, can be used by any PE connected to the LAD bus on a WILDSTAR™-II. If used correctly, there are no errors, but during development of an application errors may occur, and the `WSII_DmaErrorOp` call is used to recover from them.

Each processing element has two channels associated with it; a read channel and a write channel. There can be up to five PEs in a WILDSTAR™-II system (three main PEs and two External I/O PEs), for a total of ten DMA channels (#defined by the

constant `WSII_NUM_DMA_CHANNELS`). The channel numbers correspond to PEs as follows:

Bit 0: DMA to main board PE0	Bit 5: DMA from main board PE2
Bit 1: DMA from main board PE0	Bit 6: DMA to External I/O board 0
Bit 2: DMA to main board PE1	Bit 7: DMA from External I/O board 0
Bit 3: DMA from main board PE1	Bit 8: DMA to External I/O board 1
Bit 4: DMA to main board PE2	Bit 9: DMA from External I/O board 1

Improper use of any of these channels can generate errors, which is where the `WSII_DmaErrorOp` call will be used.

There are four types of errors that a particular channel can generate. A “general” error is typically generated due to the PE generating an incorrect LAD Bus cycle. A “page” error can be generated in one of two ways: the PE attempted to DMA to or from an invalid `HardwareAddress`, or it attempted a DMA transfer that went beyond the end of the bound memory (`HardwareAddress + pGrantedLengthDwords`). “Target Abort” and “Master Abort” errors should never occur in user applications and are provided only to assist Annapolis Micro Systems technical support personnel.

The `WSII_DmaErrorOp` call is used both to check for DMA errors and to clear them. To check for errors without clearing them, simply pass in 0x0 for the `ChannelMaskToClear` parameter. The call will fill in the (user allocated) `WSII_DMA_ERRORS` structure, and the user may then check the `ChannelErrors[]` array contained in the `WSII_DMA_ERRORS` structure to determine if any errors occurred on a particular channel. The index into the `ChannelErrors[]` array corresponds to the channel number, and the following bits are defined for each array element:

- Bit0: General Error
- Bit1: Page Error
- Bit2: Target Abort
- Bit3: Master Abort

To clear down all errors on a particular channel, set the bit in the `ChannelMaskToClear` parameter corresponding to the channel to be cleared. For example, to clear all errors on channels 3 and 8, `ChannelMaskToClear` should be set to 0x00000108, corresponding to the third and eighth bits being set (zero-based).

2.3.10.2 DMA Under Windows NT® and Windows® 2000®

The Windows® device driver reads a file called “WILDSTARII_DMA.cfg” located in the “<system root>\system32\driver\etc\” directory (where “<system root>” is usually “WinNT” or “Windows”). From that text file, the device driver determines how many buffers, and of what size, the user will require.

The “WILDSTARII_DMA.cfg” file is a file that a user may edit to have the driver reserve contiguous memory. It contains a section heading and a series of entries (see example below). Each of the entries contains the keyword “buffer” and a size (in BYTES) for that contiguous buffer. The following is an example of a “WILDSTARII_DMA.cfg” file that would have the driver attempt to allocate one 2M byte and three 1M byte buffers:

```
[BUFFERS]
BUFFER = 2097152
BUFFER = 1048576
BUFFER = 1048576
BUFFER = 1048576
[END]
```

As mentioned, the driver reads these entries and attempts to allocate a separate, contiguous buffer for each entry. The buffers will consist of contiguous memory, but the individual buffers themselves will not (necessarily) be contiguous to each other.

2.3.10.2.1 DMA Allocation Constraints Under Windows®

Every call to WSII_DmaMemAlloc starts its search for contiguous memory at the first block of contiguous buffer the driver allocated. Users should plan allocation calls carefully. As an example of the importance of careful planning, consider the following case:

In this example, two 200K byte buffers and one 1M byte buffer are desired. The user creates the following “WILDSTARII_DMA.cfg” file:

```
[BUFFERS]
BUFFER = 1048576
BUFFER = 524288
[END]
```

This file will allocate two blocks of contiguous memory; one for the 1M byte buffer and one for the two 200K byte buffers. But then the user allocates memory in the following order:

1. 200K byte buffer
2. 200K byte buffer
3. 1M byte buffer

The last allocation will fail because there is not enough contiguous memory remaining (see Allocation Method #1) even though 1.6M bytes remain unallocated. Had the user allocated the 1M byte buffer first, the allocations would have succeeded (see Allocation Method #2).

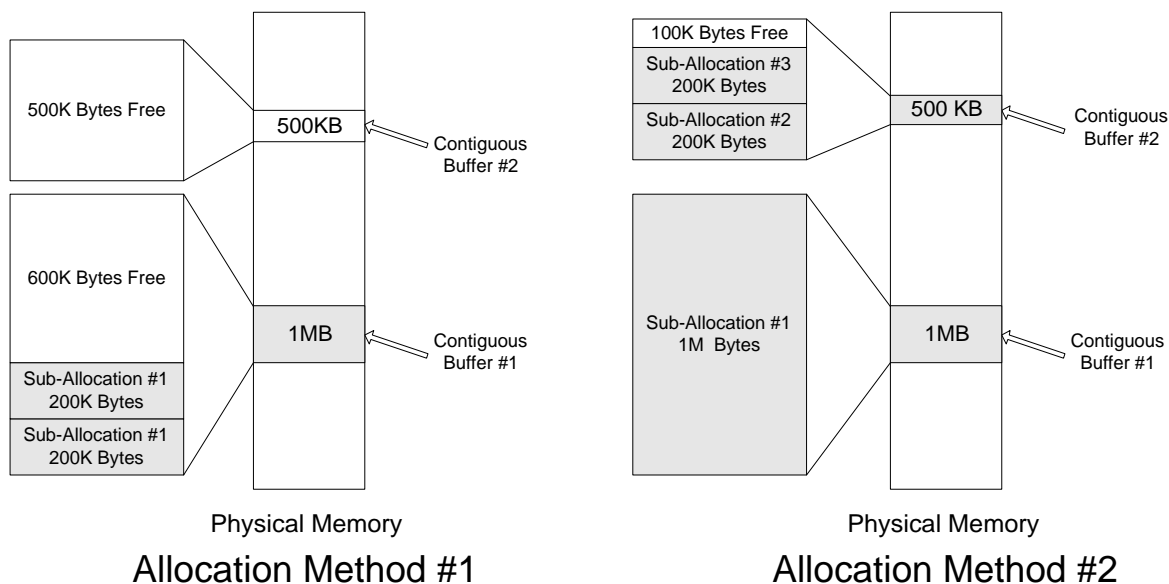


Figure 2–1: Memory Allocation (Windows® only)

2.3.10.3 DMA Under Linux® on an x86

The WILDSTAR™-II Device Driver under Linux® supports DMA transfers to and from contiguous buffers only. The buffer must be allocated via a call to `WSII_DmaMemAlloc`.

The WILDSTAR™-II device driver acquires physical memory prior to the operating system load. The user may specify the amount of physical memory that it reserves via a kernel boot parameter. The parameter indicates the number of physical pages (of size 4K) to be reserved by the device when the system starts. The parameter is set as follows:

```
wsiidmapages=<# of pages>
```

This parameter may be entered directly by the user at the lilo boot prompt or automated by adding the following line to `"/etc/lilo.conf"`:

```
append="wsiidmapages=<# of pages>"
```

After modifying `"/etc/lilo.conf,"` update the lilo table by running the command `"/sbin/lilo"` as a privileged user.

The following diagram illustrates a typical DMA flow:

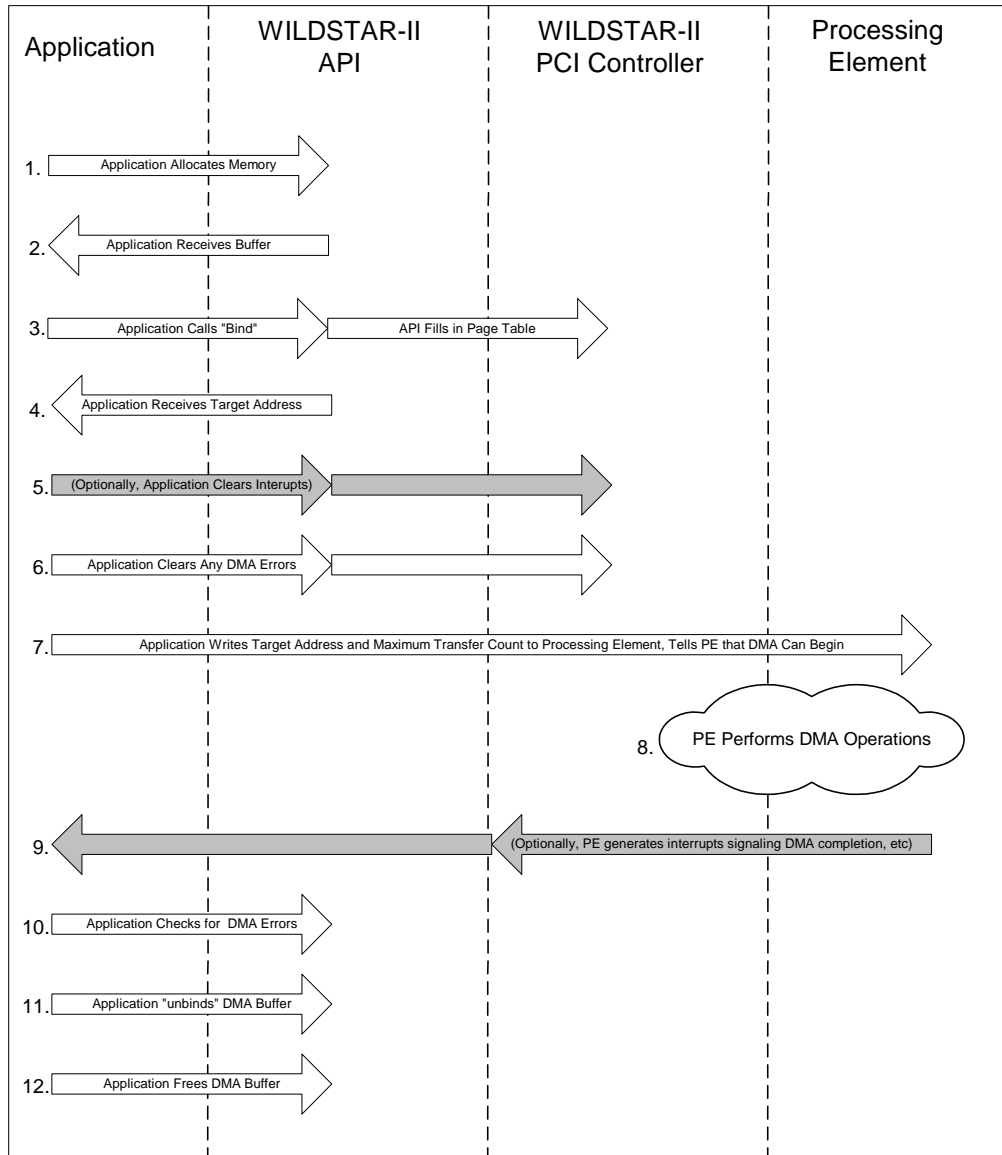


Figure 2-2: DMA Flow

The five DMA calls are described in detail in the following sections.

2.3.10.4 WSII_DmaMemAlloc

```
WSII_RetCode
WSII_DmaMemAlloc ( WSII_BOARD          hBoard,
                   WSII_DMA_HANDLE    *pDmaHandle,
                   DWORD               dNumDwords,
                   DWORD               **pBuffer)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pDmaHandle	Address of user's WSII_DMA_HANDLE
dNumDwords	Number of DWORDs (32-bit values) to allocate
pBuffer	Address of the pointer that will point to the user's buffer.

Returns:

WSII_SUCCESS, if successful, otherwise pBuffer will be NULL and rc will be a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

For a more thorough overall discussion of DMA, see the beginning of this section. On return, pBuffer will point to memory that can be used as any memory would be used; files can be read from disk into and out of it, data copied into and out of it, manipulations can be made on the data stored in it, etc.

Example:

```
WSII_RetCode    rc;
DWORD           *pBuffer;
WSII_DMA_HANDLE DmaHandle;

rc = WSII_DmaMemAlloc (BoardHandle,
                      &DmaHandle,
                      0x10000,
                      &pBuffer);

if (rc==WSII_SUCCESS && pBuffer)
{
    printf("Allocation was successful, initializing memory\n");
    memset(pBuffer, 0x0, 0x10000);
    rc = WSII_DmaMemFree (BoardHandle, DmaHandle);
}
```

2.3.10.5 WSII_DmaMemFree

```
WSII_RetCode  
WSII_DmaMemFree ( WSII_BOARD      hBoard,  
                  WSII_DMA_HANDLE DmaHandle )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
DmaHandle	WSII_DMA_HANDLE that is to be freed.

Returns:

WSII_SUCCESS, if successful, rc will be a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

For a more thorough overall discussion of DMA, see the beginning of this section. This call releases a buffer allocated with WSII_DmaMemAlloc. Note that the DmaHandle must not be “bound” for this call to succeed. If it is, call WSII_DmaUnbind before calling this function.

Example:

```
WSII_RetCode    rc;  
DWORD          *pBuffer;  
WSII_DMA_HANDLE DmaHandle;  
  
rc = WSII_DmaMemAlloc (BoardHandle,  
                      &DmaHandle,  
                      0x10000,  
                      &pBuffer);  
  
if (rc==WSII_SUCCESS && pBuffer)  
{  
    printf("Allocation was successful, initializing memory\n");  
    memset(pBuffer, 0x0, 0x10000);  
    rc = WSII_DmaMemFree (BoardHandle, DmaHandle);  
}
```

2.3.10.6 WSII_DmaBind

```
WSII_RetCode
WSII_DmaBind( WSII_BOARD          hBoard,
              WSII_DMA_HANDLE     DmaHandle,
              DWORD                *pBuffer)
              DWORD                dRequestedLengthDwords,
              DWORD                *pGrantedLengthDwords,
              DWORD                *HardwareAddress )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
DmaHandle	WSII_DMA_HANDLE that is to be bound
pBuffer	Address of the pointer that will point to the user's buffer.
dRequestedLengthDwords	Number of DWORDs to "bind"
pGrantedLengthDwords	Number of DWORDs that the API could bind, may be less than the request.
HardwareAddress	Address that the application will give to the Processing Element for the DMA operation.

Returns:

WSII_SUCCESS, if successful; otherwise, rc will be a valid WSII_RetCode.
See *wsii.h* for a list of possible errors.

Remarks:

For a more thorough overall discussion of DMA, see the beginning of this section. On return, `HardwareAddress` will contain a 32-bit value that the host can write to the PE. The PE will use this address (or an offset from this address) to initiate DMA operations. See the WILDSTAR™-II manual for more information on how the PE uses this value.



CAUTION

The WILDSTAR™-II hardware that maps host memory between the PE's LAD bus and the PCI bus is a scatter/gather table that contains 8192 entries each referencing 4096 bytes of contiguous host memory. `WSII_DmaBind` is called to bind host memory, allocated with `WSII_DmaMemAlloc`, to this scatter/gather table. The number of entries in the scatter/gather table used to bind the host memory is equal to: $\text{ceil}(\text{mem_bytes}/4096) + 1$. The one additional entry that is added is used to terminate a binding to prevent the PE from trying to access memory beyond the bounded target memory segment. The total amount of memory that can be bound at one time is

dependant on the number of bindings, but the total is always less than 32 MB.

Example:

```
WSII_RetCode    rc;
DWORD          *pBuffer;
WSII_DMA_HANDLE DmaHandle;
DWORD          Length;
DWORD          HardwareAddress;

rc = WSII_DmaMemAlloc (BoardHandle, &DmaHandle,
                      0x10000,      &pBuffer);
if (rc==WSII_SUCCESS && pBuffer)
{
    printf("Allocation was successful, initializing memory\n");
    memset(pBuffer, 0x0, 0x10000);

    rc = WSII_DmaBind( BoardHandle, DmaHandle, pBuffer,
                      0x10000, &Length, &HardwareAddress );
    if (rc!= WSII_SUCCESS)
    {
        printf("DMA Bind failed!\n");
        printf("rc=%d; %s\n",rc,WSII_GetErrorString(rc));
        WSII_DmaMemFree (BoardHandle, DmaHandle);
        return(rc);
    }
    else if (Length != 0x10000)
    {
        printf("DMA Bind was unable to bind full amount!\n");
    }

    /* Write the Target Address into the PE */
    WSII_WriteReg_32( BoardHandle, WSII_PE0,
                      DMA_TAR,      HardwareAddress);
    /* Write the Transfer Count into the PE */
    WSII_WriteReg_32(BoardHandle, WSII_PE0, DMA_TCR, Length);
    /* Write the PE control register to
     * tell the PE it can now safely do DMA */
    WSII_WriteReg_32(BoardHandle, WSII_PE0, DMA_CSR, 0x1);

    /* ... additional code here, perhaps waiting for a
     * PE interrupt signaling DMA completion, etc... */

    /* tell the PE it may no longer do DMA */
    WSII_WriteReg_32(BoardHandle, WSII_PE0, DMA_CSR, 0x0);

    WSII_DmaUnbind(BoardHandle, DmaHandle);
    WSII_DmaMemFree (BoardHandle, DmaHandle);
}
```

2.3.10.7 WSII_DmaUnbind

```
WSII_RetCode  
WSII_DmaUnbind (  WSII_BOARD      hBoard,  
                  WSII_DMA_HANDLE DmaHandle )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
DmaHandle	The WSII_DMA_HANDLE that is to be freed

Returns:

WSII_SUCCESS, if successful; otherwise, pBuffer will be NULL and rc will be a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

For a more thorough overall discussion of DMA, see the beginning of this section. On return, the DmaHandle will be “unbound”; that is, the mapping between the host buffer that the DmaHandle describes and the WILDSTAR™-II board will be released and the PE *must no longer attempt to perform DMA operations* using the address that was returned from WSII_DmaBind. Note that the host buffer is **not** released by this call, and the user may re-bind the DmaHandle at any time.

2.3.10.8 WSII_DmaErrorOp

```
WSII_RetCode  
WSII_DmaErrorOp ( WSII_BOARD          hBoard,  
                  DWORD              ChannelMaskToClear,  
                  WSII_DMA_ERRORS    *pErrors)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
ChannelMaskToClear	Bit-mask of channels to be cleared
pErrors	Contains information about DMA errors

Returns:

WSII_SUCCESS, if successful.

Remarks:

For a more thorough overall discussion of DMA, see the beginning of this section. ChannelMaskToClear specifies a mask of the channels on which to clear errors. The bits are defined as follows:

Bit 0: DMA to main board PE0	Bit 5: DMA from main board PE2
Bit 1: DMA from main board PE0	Bit 6: DMA to External I/O board 0
Bit 2: DMA to main board PE1	Bit 7: DMA from External I/O board 0
Bit 3: DMA from main board PE1	Bit 8: DMA to External I/O board 1
Bit 4: DMA to main board PE2	Bit 9: DMA from External I/O board 1

If ChannelMaskToClear is specified as 0x0, no errors are cleared; the function call will only fill in the WSII_DMA_ERRORS structure pointed to by pErrors. If pErrors is NULL, then the call will not return error information; it will only clear the errors indicated by ChannelMaskToClear. If ChannelMaskToClear is 0x0 and pErrors is NULL, then the call does nothing.

If pErrors is not NULL then the (user-allocated) structure pointed to by pErrors will be filled in with any errors found on all channels. The structure member ChannelErrors is an array of DWORDs. Each array member's index corresponds to its channel number. Each DWORD contains a bit mask of all errors found:

- Bit0: General Error
- Bit1: Page Error
- Bit2: Target Abort
- Bit3: Master Abort
- Bit4: Split Completion Error (PCI-X only)
- Bit5: Unexpected Split Completion Error (PCI-X only)

Example:

The following example illustrates a query of DMA errors, followed by checks for errors on any channel. It prints out information for each channel's errors, then finally clears all errors for every channel on which any error was found.

```
WSII_DMA_ERRORS   DmaErrors
DWORD             ChannelMaskToClear;
DWORD             i;

ChannelMaskToClear = 0x0;

    /* Query the state of all errors, clearing nothing*/
WSII_DmaErrorOp (hBoard, ChannelMaskToClear, &DmaErrors);

for (i=0; i < WSII_NUM_DMA_CHANNELS; i++)
{
    if (DmaErrors.ChannelErrors[i])
    {
        if (BitTst(DmaErrors.ChannelErrors[i],
WSII_DMA_GENERAL_ERROR))
            printf("Found a general error on channel %d\n",i);
        if (BitTst(DmaErrors.ChannelErrors[i],
WSII_DMA_PAGE_ERROR))
            printf("Found a page error on channel %d\n",i);
        if (BitTst(DmaErrors.ChannelErrors[i],
WSII_DMA_TARGET_ABORT))
            printf("Found a Target Abort error on channel %d\n",i);
        if (BitTst(DmaErrors.ChannelErrors[i],
WSII_DMA_MASTER_ABORT))
            printf("Found a Master Abort error on channel %d\n",i);

        ChannelMaskToClear = BitSet(ChannelMaskToClear,i);
    }
}

    /* Clear down any errors found above */
WSII_DmaErrorOp (hBoard, ChannelMaskToClear, &DmaErrors);
```

2.3.10.9 DMA Under Solaris™

Below are DMA API calls specific to the Solaris™ operating system.

2.3.10.10 WSII_DmaMemAllocWithId

```
WSII_RetCode  
WSII_DmaMemAllocWithId (WSII_BOARD hBoard,  
                        WSII_DMA_HANDLE *pDmaHandle,  
                        DWORD dNumDwords,  
                        DWORD **ppBuffer,  
                        DWORD dBufferId)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
pDmaHandle	Address of user's WSII_DMA_HANDLE
dNumDwords	Number of DWORDs (32-bit values) to allocate
ppBuffer	Address of the pointer that will point to the user's buffer
dBufferId	Buffer ID (non-zero positive integer) associated with this allocation

Returns:

WSII_SUCCESS, if successful, otherwise pBuffer will be NULL and rc will be a valid WSII_RetCode. See wsii.h for a list of possible errors.

Remarks:

This API is similar to the WSII_DmaMemAlloc() API. The difference is that this API allows the user to associate a buffer Id with the allocation, which can later be used by other processes to obtain a view of this memory with a call to the API, *WSII_DmaGetMap()*.

2.3.10.11 WSII_DmaGetMap

```
WSII_RetCode  
WSII_DmaGetMap (WSII_BOARD hBoard,  
                DWORD dBufferId,  
                DWORD dNumDwords,  
                DWORD **ppBuffer)
```

Parameters:

hBoard	Board handle returned by WSII_Open()
dBufferId	Buffer ID (non-zero positive integer) associated with the allocation of interest
dNumDwords	Number of DWORDs (32-bit values) to map into process space
ppBuffer	Address of the pointer that will point to the mapping

Returns:

WSII_SUCCESS, if successful; otherwise, pBuffer will be NULL and rc will be a valid WSII_RetCode. See wsii.h for a list of possible errors.

Remarks:

This API allows a process to obtain a mapping to an allocation that was previously allocated with WSII_DmaMemAllocWithId().

2.3.10.12 WSII_DmaSync

```
WSII_RetCode  
WSII_DmaSync (  WSII_BOARD          hBoard,  
                WSII_DMA_HANDLE     DmaHandle,  
                WSII_DMA_SYNC       View )
```

Parameters:

hBoard	Board handle returned by WSII_Open()
DmaHandle	The WSII_DMA_HANDLE to be synchronized
View	Type of memory view to be synchronized

Returns:

WSII_SUCCESS, if successful, rc will be a valid WSII_RetCode. See *wsii.h* for a list of possible errors.

Remarks:

This call is used to synchronize the caching between the Solaris host processor and the WILDSTAR™-II board. The Solaris host should call this API with the `view` parameter set to, `WSII_SYNC_FOR_CPU` before it views a DMA buffer that was modified by a DMA operation performed by the WILDSTAR™-II board. Similarly, the Solaris host should call this API with the `view` parameter set to, `WSII_SYNC_FOR_DEV` before it initiates DMA operations on the WILDSTAR™-II board, where the Solaris host processor made modifications to the DMA buffer to be transferred to the WILDSTAR™-II board.



INDEX

Block RAM, 1-3
CLB, 1-3
Conventions, 1-2
Defines, 2-1
DMA, 2-3, 2-13, 2-16, 2-61, 2-62, 2-73, 2-74, 2-75,
2-76, 2-77, 2-78, 2-79, 2-80, 2-81, 2-82, 2-83,
2-84, 2-85, 2-86, 2-87, 2-89, I
Driver, 1-3
Enumerations, 2-1
Euro I/O Card, 1-3
External I/O, 1-3
External I/O Cards, 1-3
FPGA, 1-3
Icons, 1-2
KCLK, 1-3
LAD Bus, 1-3
MCLK, 1-3
PCI, 1-4
PCLK, 1-4
PE. See Processing Element
PE0, 1-4
PE1, 1-4
PE2, 1-4
Processing Element, 1-4
Solaris, 2-86
Structures, 2-3
UCLK, 1-4
VIRTEX™ FPGA, 1-4
VIRTEX™-E FPGA, 1-4
VME, 1-4
WILDSTAR™ II, 2-48
WSII_BOARD_SITE, 2-1
WSII_BoardReset, 2-11
WSII_CLOCK_SRC, 2-1
WSII_DAISSY_CHAIN_BUFFER, 2-3
WSII_DISPLAY_LEVEL, 2-2
WSII_DMA_ERRORS, 2-3
WSII_DmaBind, 2-82
WSII_DmaErrorOp, 2-85
WSII_DmaMemAlloc, 2-80
WSII_DmaMemAllocWithId, 2-87
WSII_DmaMemFree, 2-81
WSII_DmaUnbind, 2-84
WSII_FAN_MODE, 2-2
WSII_FLAGS, 2-1
WSII_GetClockFrequency, 2-35
WSII_GetErrorString, 2-70
WSII_GetInformation, 2-12
WSII_GetLastError, 2-69
WSII_GetPower, 2-57, 2-59, 2-60
WSII_GetSlotNum, 2-4, 2-5, 2-6
WSII_GetStartupFrequency, 2-38
WSII_GetTemperature, 2-48
WSII_GetTemperatureThresh, 2-50
WSII_INFO_REQ, 2-2
WSII_InterruptQueryStatus, 2-
61
WSII_InterruptReset, 2-62
WSII_InterruptWait, 2-63, 2-64, 2-65, 2-66
WSII_LimitTemperatureThresh, 2-52
WSII_MEM_INFO, 2-2, 2-3
WSII_Open, 2-7
WSII_PE_NUM, 2-1, 2-2
WSII_PeDeprogram, 2-29
WSII_PeProgram, 2-23
WSII_PeProgramDaisyChain, 2-26
WSII_PeProgramOnStartup, 2-28
WSII_PeReset, 2-30, 2-31
WSII_PeSetFanMode, 2-56
WSII_POWER_SRC, 2-2
WSII_QueryTemperatureEvents, 2-54
WSII_ReadReg_32, 2-19
WSII_ReadRegs_32, 2-21
WSII_RESET_OP, 2-2
WSII_ResetTemperatureEvents, 2-55
WSII_SetClockFrequency, 2-36, 2-40, 2-41, 2-42,
2-43
WSII_SetDisplayLevel, 2-72
WSII_SetErrorCallback, 2-67
WSII_SetStartupFrequency, 2-39
WSII_SetTemperatureThresh, 2-51
WSII_TEMP_THRESHOLD, 2-2
WSII_TEMPERATURE_SRC, 2-2
WSII_UpdateDisplay, 2-71
WSII_WriteReg_32, 2-20
WSII_WriteRegs_32, 2-22
ZBT, 1-5

