LeMans
VX8 Carrier Board Programming Guide

Document Number 500-00330

Revision 2.02

September 1998

Part Number 500-00330
Revision 2.02

Customer Feedback

At Spectrum, we recognize that product documentation that is both accurate and easy to use is important in aiding you in your new product development. We appreciate hearing your comments on how our product's documentation could be improved.

If you wish to comment on any Spectrum documentation then please fax or e-mail a completed copy of this page to us.

Full Name of Document: ____

Document Number: ______________________ Version Number: _________________

If you have found a technical inaccuracy please describe it here:

____

____

____

____

____

If you particularly liked or disliked an aspect of the manual then please describe it here:

____

____

____

____

____

It may be helpful for us to call you to discuss your comments. If this would be acceptable please provide the following details:

Name: _________________________________ Telephone #: ____________________

Organization: ____

Thank you for your time,

Spectrum Signal Processing Documentation Group

Fax:	(604) 421-1764
Email:	documentation@spectrumsignal.com

Contacting Spectrum…

Spectrum’s team of dedicated Applications Engineers are available to provide technical support to you for this product. Our office hours are Monday to Friday, 8:00 AM to 5:00 PM, Pacific Standard Time.

Telephone	1-800-663-8986 or (604) 421-5422
Fax	(604) 421-1764
Email	support@spectrumsignal.com
Internet	http://www.spectrumsignal.com

When you contact us, please have the following information on hand:

- A concise description of the problem
- The name of all Spectrum hardware components
- The name and version number of all Spectrum software components
- The minimum amount of code that demonstrates the problem
- The version number of all software packages, including compilers and operating systems

Part Number 500-00330
Revision 2.02

Preface

Spectrum Signal Processing offers a complete line of DSP hardware, software and I/O products for the DSP Systems market based on the latest DSP microprocessors, bus interface standards, I/O standards and software development environments. By delivering quality products, and DSP expertise tailored to specific application requirements, Spectrum can consistently exceed the expectations of our customers. We pride ourselves in providing unrivaled pre and post sales support from our team of application engineers. Spectrum has excellent relationships with third party vendors which allows us to provide our customers with a more diverse and top quality product offering.

Spectrum achieved ISO 9001 quality certification in 1994.

As Spectrum’s hardware products are static sensitive, please take precautions when handling and make sure they are protected against static discharge.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1. Purpose of This Manual

This manual provides the information you need to develop VXIbus system applications using Spectrum's VX8 VXIbus TIM-40 Carrier Board. The manual describes the host and DSP libraries used to program and interface to the VX8s in your system. Helpful programming methods, tips, and software examples are also provided.

A second manual, the *VX8 Carrier Board Technical Reference Manual* (TRM), is the primary hardware reference. You must be familiar with this manual in order to develop system architecture and data flow paths. The TRM is also the primary reference for modifying or extending the functionality of the driver.

Caution: The hardware interfaces of the VX8 Carrier Board are extremely complex and interrelated. You are strongly urged to make use of the supplied C40 software control libraries to initialize and transfer data to the hardware interfaces.

1.2. Reference Documents

This guide is meant to be used in conjunction with the following documents:

- ***VX8 Carrier Board Technical Reference Manual*** available from Spectrum
- ***VX8 Carrier Board Installation Guide*** available from Spectrum
- *VMEbus Extensions for Instrumentation (VXIbus) VXI-1* Revision 1.4. Authored by the VXIbus Consortium, Inc.
- *TMS320C4x User's Guide* available from Texas Instruments
- *TMS320C4x C Source Debugger User's Guide* available from Texas Instruments
- *TMS320 Floating-Point DSP Assembly Language Tools User's Guide* available from Texas Instruments
- *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* available from Texas Instruments
- *Getting Started Guide* from Texas Instruments
- *VXIpnp documents* - VXIplug&play Systems Alliance
- *SCV64 User Manual* - *VMEbus Interface Components Manual* from Tundra Semiconductor Corporation

- *XDSC40 Board MS-DOS User Guide* available from Spectrum
- *BALLISTIC VXIbus Interface Chip Data Sheet* available from Hewlett-Packard

1.3. Conventions Used in This Manual

This guide uses the following conventions:

- *Italic* font is used to designate placeholder names, such as command parameter names, cross-references, and references to other documents. For example:

  A second manual, the *VX8 Carrier Board Technical Reference Manual* (TRM), is the primary hardware reference.

- **Bold** font is used to emphasize text within paragraphs. For example:

  If you edit the **board.cfg** file, you'll have to run **composer.exe** with the file.

- `This font` is used to designate contents of text files (source code, configuration files), filenames, directory names, text that appears on the screen, and commands that you must enter in an interactive display. For example:

  These strings should be defined in the [`common`] section.

- An "h" after a number indicates that this is hexadecimal notation (base 16). For example:

  This write-only register is located at address 8B00 0000h on the Shared DRAM Bus.

- "0x" before a number indicates that this is hexadecimal notation (base 16). For example:

  The default TIM-40 based DSP linker command file **vx8_tim.cmd** also defines the space COM_KERNEL at address 0x8000 0000 with length 0x100 (1 Kbyte).

2 Hardware Overview

2.1. Features

Spectrum's VX8 Carrier Board is a VXIbus based multiple DSP processing engine.

2.1.1. TMS320C4x Nodes

Six TIM-40 sites and two on-board 60 MHz TMS320C40 (C40) processors are incorporated onto the VX8 Carrier Board. The embedded C40s are nodes A and B, and the TIM-40 sites are nodes C to H. Each node has one buffered C4x communication port brought to the front panel.

Each embedded node (A and B) features:

- One bank of 128k x 32 SRAM on both the local and global buses for a total of 1 Mbytes per C40. (Upgradeable to 512 x 32 SRAMs at the factory;)
- One 32kx8 PEROM for booting or TIM-40 IDROM compatibility on the local bus;
- Global bus signals routed to buffers to allow for HP Local Bus DMA controlled data writes to global SRAM;
- The capability to write to the HP Local Bus output FIFO or to access the shared global DRAM through the global bus connector; and
- The capability to access the SCV64 IC to act as a VXIbus master.

Node A has an additional 32k x 8 PEROM used for the board's boot kernel.

Node B has a DUART equipped with RS-232 drivers brought to the front panel Dual RS-232 asynchronous serial ports.

2.1.2. Bus Interfaces

The VX8 has a register based VXIbus interface incorporating an optional Hewlett-Packard (HP) local bus interface. The HP Local Bus interface uses a high speed BALLISTIC interface chip and an intelligent DMA controller.

The VX8 can function as either a Master or as a Slave module on the VXIbus. VXIbus D32, D16, and D08E0 data access is supported in the following address modes:

Address Mode	Master	Slave
A32	Yes	Yes
A24	Yes	No
A16	Yes	VXIbus registers only

2.1.3. Diagnostic Support

Diagnostic and debugging support for the VX8 is provided through the following features:

- C language source symbolic debugger through front panel JTAG in and out connectors to an XDS510 or a DBC3040
- On board Test Bus Controller for C language source symbolic debugger with WIN95/NT Intel VXIbus slot 0 controllers

2.2. Board Layout

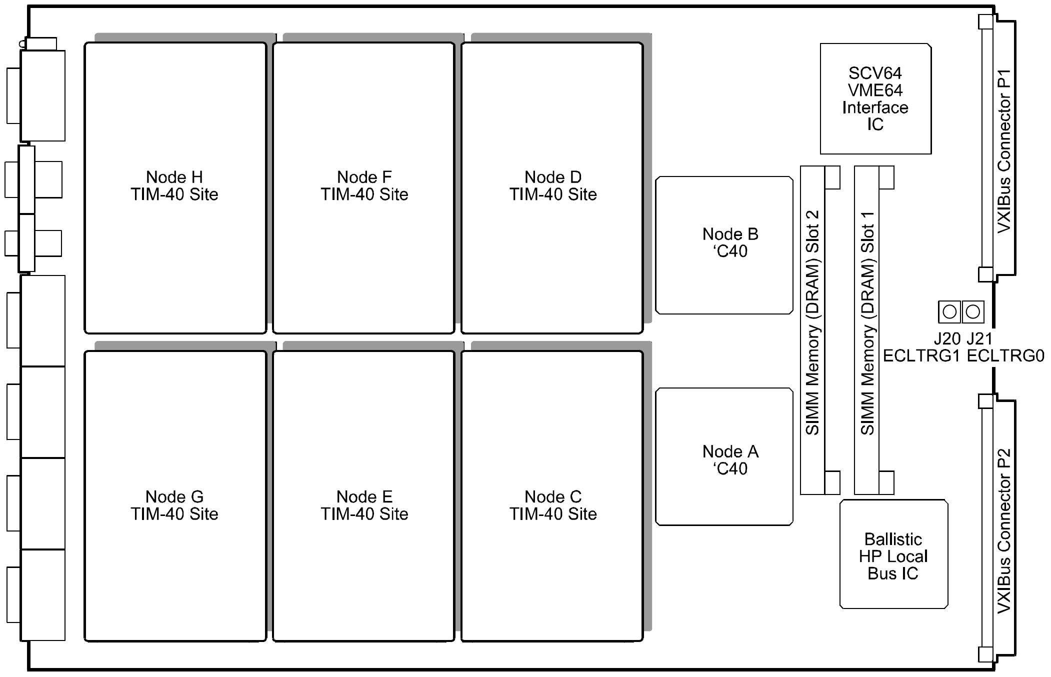


Figure 1 VX8 Carrier Board Layout

2.3. Front Panel

The front panel of the VX8 has a variety of connectors and status LEDs as shown in the following illustration. The pinouts for the connectors and the LEDs are described in the *VX8 Carrier Board Technical Reference Manual*.

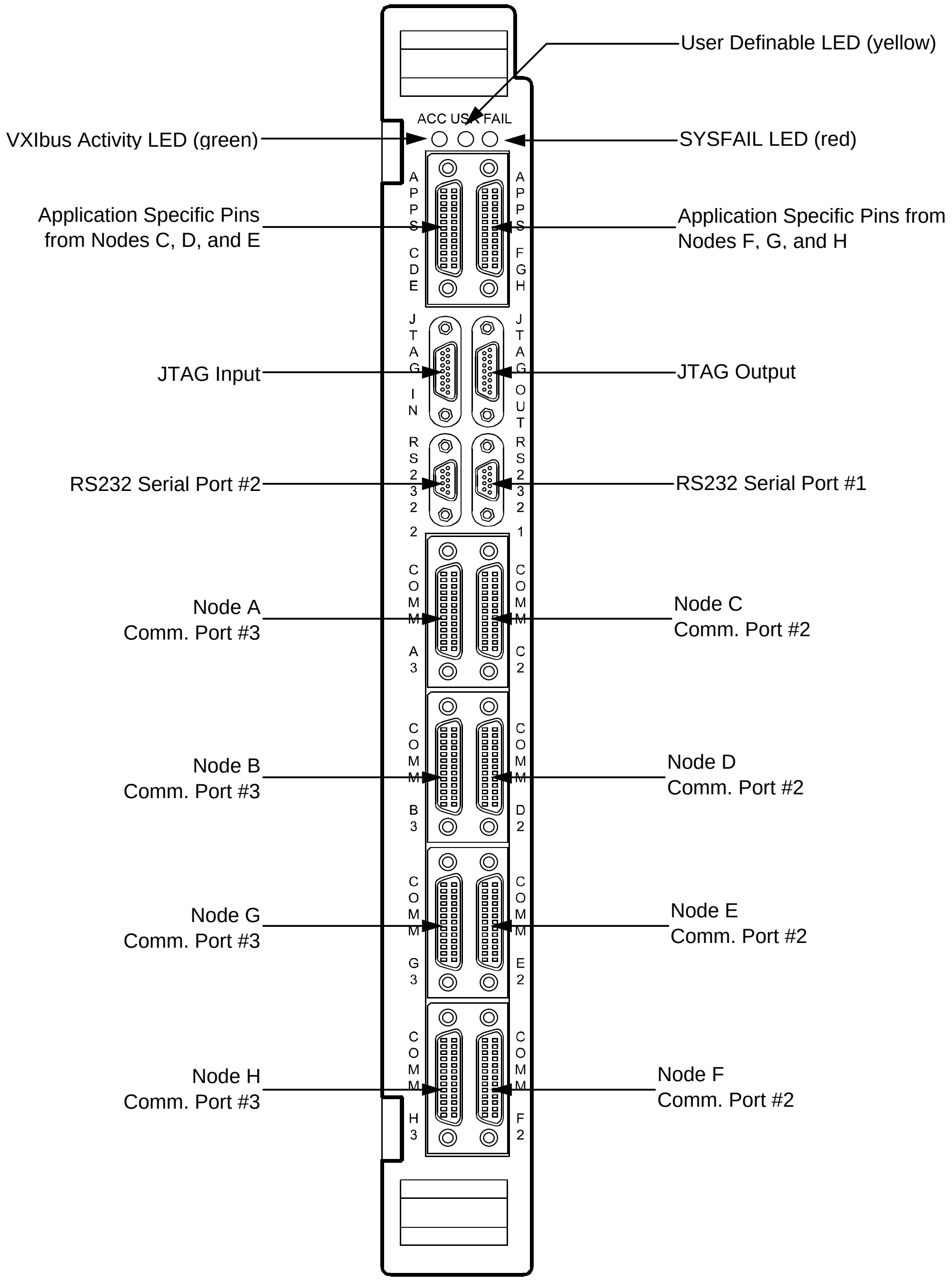


Figure 2 VX8 Front Panel

2.3.1. Status LEDs

LED	Color	Description
ACC	Green	VXIbus Activity LED. ON when there is activity between the VXIbus and the VX8 Carrier Board.
USR	Yellow	User Definable LED. ON when bit 0 (D0) of the LED register is set to “0”. This write-only register is located at address 8B00 0000h on the Shared DRAM Bus.
FAIL	Red	VXIbus SYSFAIL LED. The LED is driven when the SCV64 IC drives the VXIbus SYSFAIL line.

2.3.2. Connectors

JTAG IN	Texas Instruments’ XDS510 or Spectrum’s DBC3040 can be connected to the JTAG IN connector for use with a debug monitor.
JTAG OUT	The JTAG OUT connector allows the VX8 to be part of a multi-module JTAG path.
RS232 Serial Ports	Two RS232 Serial ports are supported by the Node B embedded ‘C40 DSP.
Communication Ports	One communication port from each of the Nodes is brought to the front panel via these connectors.

2.4. C4x Communication Port Architecture

The C4x Communications ports provide high speed parallel interface communications (~20 Mbytes/sec) to other DSPs and I/O sources. The communication is inherently bi-directional and point to point so there is no latency for access and a single COMM port can be used for half duplex communication between two devices.

The TMS320C40 provides 6 COMM Ports and the TMS320C44 provides 4 COMM Ports. COMM Port routing on the VX8 Carrier Board accounts for fewer COMM Ports on a C44 by ensuring that the front panel connections are valid for Spectrum’s C40 and C44 based TIM-40 Modules.

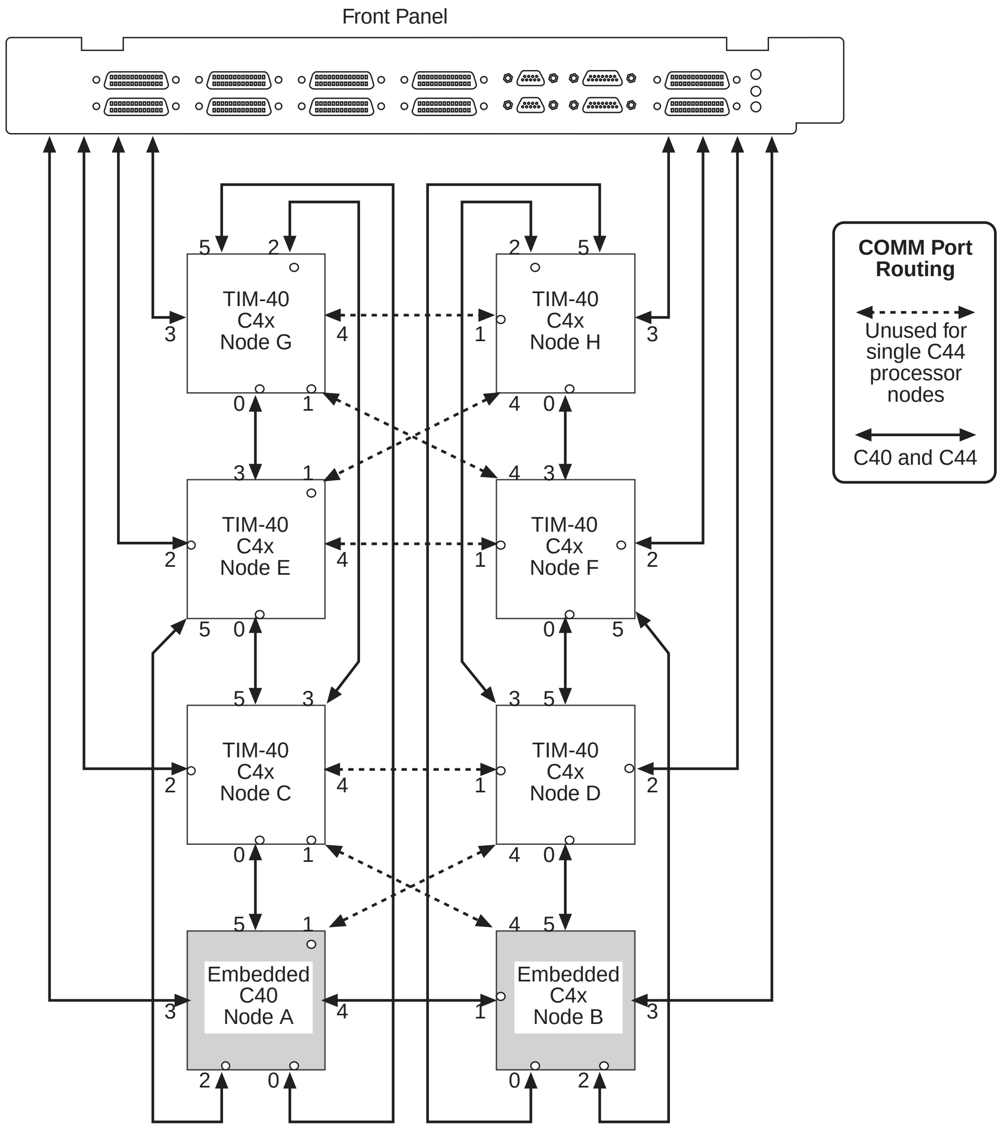


Figure 3 Communication Port Routing

Although the C44 does not use COMM ports 0 and 3, Spectrum's C44 based TIM-40 modules route COMM ports 1 and 4 to COMM ports 0 and 3 for compatibility with existing motherboard designs. As a result, COMM Ports 1 and 4 of single C44 based TIM-40 modules are not available. The COMM Port layout shown in *Figure 3* ensures that the front panel COMM ports are valid for all current and planned Spectrum C40 and C44 based TIM-40 Modules.

Refer to the *TMS320C4x User's Guide* for further information on the C4x COMM Ports.

2.5. Bus Architecture

Several different communication buses are used on the VX8 Carrier board to connect the C40 processors, TIM-40 sites, memory devices, and interface circuitry. Although the buses are not the only way that devices are interconnected on the VX8, they are the primary means of data transfer between devices.

Local Bus The Local Bus address range is specific to a single C4x DSP, and is therefore not shared with other processors or nodes. It is a private memory bus of a particular C4x.

Near Global Bus The Near Global Bus of the VX8 refers to the Global Bus of each TIM-40 site and the embedded C40 nodes. The SRAM located on these Global Buses is zero wait state from the DSP that owns it, but can be accessed by other DSPs, the HP Local Bus DMA Controller, and the VXIbus Slave Interface via the Global Shared Bus.

Global Shared Bus The Global Shared Bus interconnects the:

- Buffered Global Buses of each TIM-40 site via the Global Connectors;
- Buffered Global Buses of the embedded C40 nodes A and B;
- DRAM Shared Bus; and
- HP-Local Bus Interface and registers.

32-bit buffers isolate the Global Shared Bus from all these areas except for the HP-Local Bus Interface, which is connected to the bus through a 2 x 1k x 32-bit FIFO.

DRAM Shared Bus The DRAM Shared Bus enables the VXIbus slave interface to access the DRAM, Test Bus Controller, and the Global Shared Bus. It also allows a C4x DSP to access the DRAM, control / status registers, and the SCV64 as a VXIbus master. Two 72-pin SIMM sites allow expansion of the global shared DRAM using standard PC DRAM memory modules (see Note below).

Note: We recommend that DRAM memory be purchased from Spectrum, since this memory has been tested and is guaranteed to be compatible with the VX8 board.

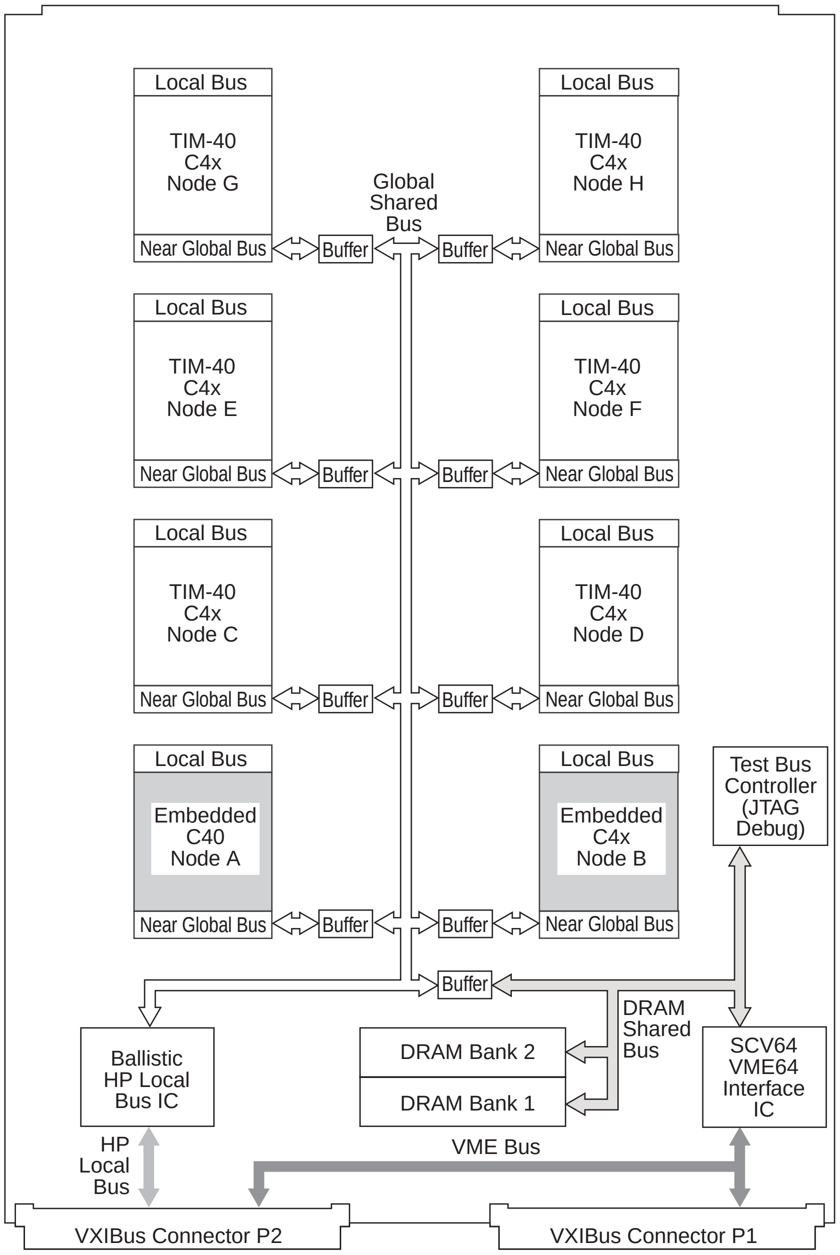


Figure 4 Bus Architecture

2.6. C4x Interrupt Architecture

The four configurable IIOF lines from each ‘C4x are used for interrupts

- Between other C4x nodes on the board
- From the SCV64 VXIbus interface chip
- From the HP Local Bus interface
- From the Dual 16550 UART (DUART) (Node B only)

- From the VXIbus A16 Interface (Node A only)

The following figure shows the VX8 interrupt architecture.

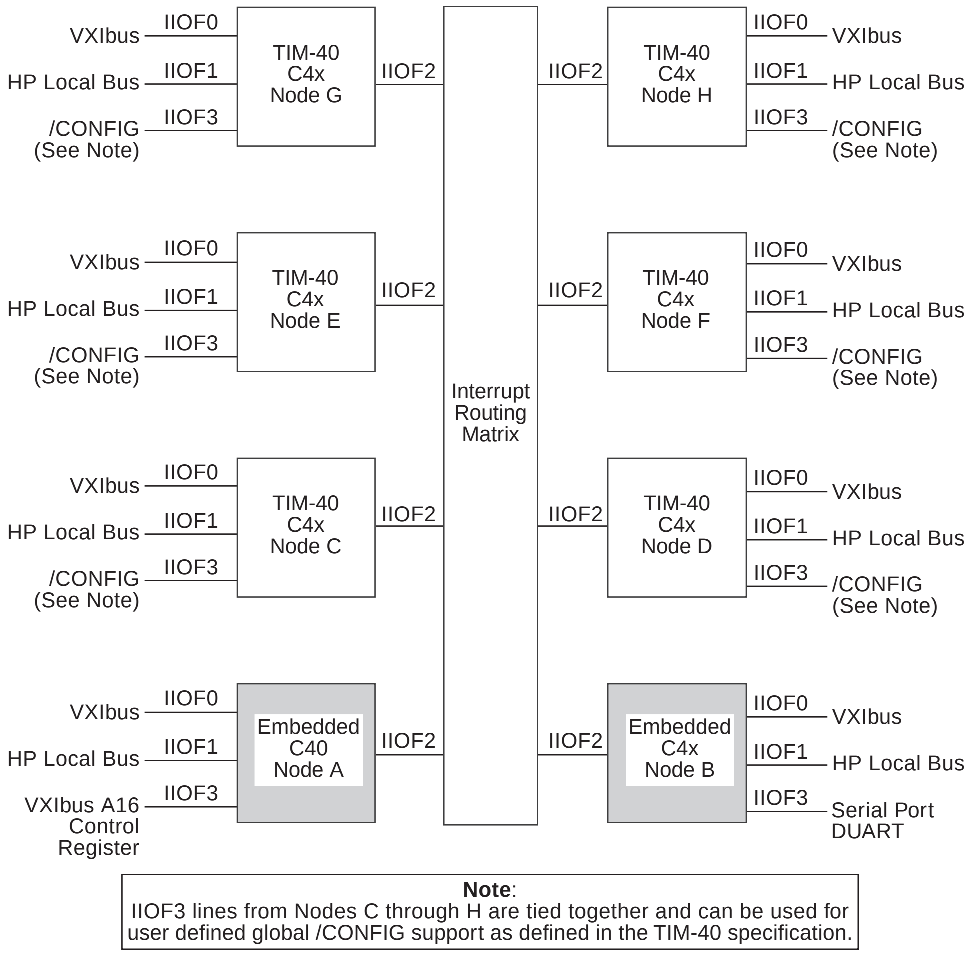


Figure 5 VX8 Interrupt Architecture

2.6.1. VXIbus Interrupts (IIOF0)

Interrupts from the SCV64 VXIbus interface chip are mapped to IIOF0 of all the C4x nodes. These indicates that either a VXIbus interrupt has occurred or that one of the on-board SCV64 interrupt sources has occurred (SCV64 DMA done, for example). Because the interrupt can have several sources, it must be level-triggered so that the source of the interrupt can be identified. VXIbus interrupts can be enabled or disabled based on their interrupt level through registers in the SCV64 VXIbus interface chip. This allows software selectable receipt of interrupts of different priorities. Interrupts must be enabled before they can be generated. The interrupt vector received is latched for reading by the ‘C4x servicing the interrupt through the IACK space in the shared memory map. The ‘C4x’s interrupt service routine must produce the IACK cycle through the SCV64.

To determine the source and initiate clearing of the interrupt, an interrupt acknowledge (IACK) cycle must be performed. VX8 software functions are available to configure and acknowledge the interrupts for this.

Vectored interrupts can also be generated from any node to the VXIbus using the internal SCV64 register set.

2.6.2. HP Local Bus Interrupts (IIOF1)

Three interrupts from the HP Local Bus interface are mapped to IIOF1 of all ‘C4x nodes:

- End of Block (EOB)
- Write FIFO Almost Empty (WAE) sent once per transition of the WAE flag
- Write FIFO Almost Full (WAF)

These interrupts are ORed together onto the IIOF1 line. Use level-triggered interrupts to identify the source of the HP-Bus interrupt. The HINTENABLE, HINTSTAT, and HINTCLR registers on the Global Shared Bus are used to enable, identify, and clear the interrupts. Refer to the *HP Local Bus Interface* section of the *VX8 Carrier Board Technical Reference Manual* for more information on these interrupts.

2.6.3. Interrupt Routing Matrix (IIOF2)

The IIOF2 lines from each node can be used as an interrupt or as a general purpose I/O to signal other ‘C4x nodes. This interrupt scheme is under software control, allowing IIOF2 lines to be tied together in any combination through the Interrupt Routing Matrix. The Interrupt Routing Matrix is configured by a register located on node A’s local bus. For further information on the IIOF2 Interrupt Routing Matrix see the *Embedded C40 Node A* section of the *VX8 Carrier Board Technical Reference Manual*.

2.6.4. IIOF3 Interrupts

The IIOF3 line is used for three different purposes on the VX8 depending on which node it belongs to.

Node	Usage	Description
A	VXIbus A16 interrupt	Whenever the host writes to the VXIbus A16 Interface Control Register an interrupt is sent to the IIOF3 line of the Node A C40. For further information on the VXIbus A16 interface see the *VX8 Carrier Board Technical Reference Manual* .

Node	Usage	Description
B	DUART interrupt	Node B uses IIOF3 as the interrupt from the serial port DUART. The interrupt lines from Channel 0 and Channel 1 UARTs are ORed together and routed to Node B's IIOF3. For further information on the DUART interrupt see the *Embedded C40 Node B* section of the *VX8 Carrier Board Technical Reference Manual* .
C to H	TIM-40 /CONFIG	The IIOF3 lines from the TIM-40 modules (nodes C through H) are not brought off the carrier board. The lines are tied together on the VX8 board, allowing them to be used for the /CONFIG line.

2.7. JTAG Debugging

A JTAG IN connector is provided on the front panel of the board for connection to an XDS510 from Texas Instruments or a DBC3040 from Spectrum. This allows use of the Texas Instruments' standard TMS320C4x debug monitor or third party debug monitors such as GO DSP's Code Maestro from an external PC or SUN workstation.

A JTAG OUT connector allows the VX8 to be part of a multi-module JTAG path. The open collector /CONFIG and /GRESET signals are bussed between boards via the JTAG connectors. The JTAG cable allows multi-board resetting (required if the front panel COMM ports are connected between boards) and /CONFIG to be bussed between devices.

Note: Because of the directional reset state of C4x COMM ports, all VX8s connected together via COMM port MUST be connected by their front panel JTAG connectors. The front panel JTAG connectors will reset JTAG connected VX8s together, thereby damage to COMM ports on reset will be avoided.

Each of the C4x processors has a JTAG interface for debugging purposes. The JTAG chain is controlled by the JTAG PAL, which routes the data lines to each available C40 node in turn. If a TIM site is not occupied then the JTAG chain bypasses that node. The full JTAG sequence is JTAG IN, Node A, C, E, G, B, D, F, H, JTAG OUT. For multiple processor TIM-40 modules, refer to the TIM-40 module documentation for information on the order in which the processors are connected in the JTAG scan path. When an external debugger is connected to the JTAG IN connector of a board, the on-board TBC is disabled.

For further details on the JTAG interface, refer to *the VX8 Carrier Board Technical Reference Manual*.

2.7.1. JTAG Connection

The JTAG cable from your external debugger should be connected only to the JTAG IN of board 1. For multiple boards, connect the JTAG OUT of board 1 to the JTAG IN of board 2, etc..

Note: Ensure that your hardware is powered down before connecting the JTAG cable and setting up the JTAG chain.

2.7.2. JTAG Software Setup

The following describes the software setup required when using Texas Instruments' standard TMS320C4x debug monitor. If you're using a third party debug monitor, refer to that product's documentation for specific software setup instructions.

A configuration file (**board.cfg)** is required to tell the debugger how many C4x processors there are in a JTAG scan chain on your target system and their order in the chain. A sample **board.cfg** has been provided on the *VX8 C4x Support Software* disk and can be found in the *examples\debug* directory. This sample **board.cfg** file includes a setup for three VX8 boards and defines which processors are present in the JTAG scan chain. Sites which are populated are uncommented. In this example, only nodes A and B of board 1 are populated; other processors are commented out.

```
;"CPU_H3"         TI320C4X                ; Module in site H
;"CPU_F3"         TI320C4X                ; Module in site F
;"CPU_D3"         TI320C4X                ; Module in site D
;"CPU_B3"         TI320C4X                ; Module in site B

;"CPU_G3"         TI320C4X                ; Module in site G
;"CPU_E3"         TI320C4X                ; Module in site E
;"CPU_C3"         TI320C4X                ; Module in site C
;"CPU_A3"         TI320C4X                ; Module in site A

;"CPU_H2"         TI320C4X                ; Module in site H
;"CPU_F2"         TI320C4X                ; Module in site F
;"CPU_D2"         TI320C4X                ; Module in site D
;"CPU_B2"         TI320C4X                ; Module in site B

;"CPU_G2"         TI320C4X                ; Module in site G
;"CPU_E2"         TI320C4X                ; Module in site E
;"CPU_C2"         TI320C4X                ; Module in site C
;"CPU_A2"         TI320C4X                ; Module in site A

;"CPU_H1"         TI320C4X                ; Module in site H
;"CPU_F1"         TI320C4X                ; Module in site F
```

```
;"CPU_D1"         TI320C4X               ; Module in site D
"CPU_B1"         TI320C4X               ; Module in site B

;"CPU_G1"         TI320C4X               ; Module in site G
;"CPU_E1"         TI320C4X               ; Module in site E
;"CPU_C1"         TI320C4X               ; Module in site C
"CPU_A1"         TI320C4X               ; Module in site A
```

The JTAG cable from your external debugger should be connected only to the JTAG IN of board 1. For multiple boards, uncomment the appropriate sites and connect the JTAG OUT of board 1 to the JTAG IN of board 2, etc..

Note: Processors listed in the **board.cfg** file are listed in reverse order than they actually occur in the JTAG chain. The last board listed in the **board.cfg** file should be the first board in the JTAG chain.

If you edit the **board.cfg** file, you'll have to run **composer.exe** with the file in order for your changes to take effect.

A memory descriptor file (**emuinit.cmd**) is also required to tell the debugger which areas of memory it can and cannot access. A sample **emuinit.cmd** file has been provided on the *VX8 C4x Support Software* disk and can be found in the *examples\debug* directory. Using a text editor, edit this file to properly reflect the VX8's memory mapping (refer to the *VX8 Carrier Board Technical Reference Manual* for details). If you're setting bus control registers in the **emuinit.cmd** file, refer to the *VX8 Carrier Board Technical Reference Manual* and the TIM module documentation for the correct values to calculate.

Note: If you're loading your application via JTAG, the Global Memory Control Register (GMCR) and Local Memory Control Register (LMCR) have to be set in the **emuinit.cmd** file in order for your program to load, and for the processors to access their memory correctly. You'll need a separate **emuinit.cmd** file for each type of TIM module. Refer to the documentation provided with your TIM modules for the correct GMCR and LMCR values.

Refer to the *TMS320C4x C Source Debugger User's Guide* for further details and if you're using Spectrum's DBC3040, also refer to the *XDSC40 Board MS-DOS User Guide*.

Note: If using Spectrum's DBC3040, do **not** try to integrate C4x programs with MS-DOS applications and do **not** use the debugger to write C4x programs (use the information provided in this manual instead) as described in the *XDSC40 Board MS-DOS User Guide.*

2.7.3. Debugging tips

When debugging, keep in mind that memory windows to lockable regions (Near Global SRAMs, HP registers, SCV registers, etc.) may not refresh correctly or be visible if other entities (other DSPs, SCV64, or BALLISTIC chip) are using global resources as well.

For information on accessing shared resources while debugging, see *section 6.5.5*.

3 Software Overview

The VX8 Support Software product provides hardware initialization, hardware control, host communications, DSP library functions, and examples. The VX8 Support Software primarily consists of two libraries: the C4x DSP Library (VX8 C4x Support Software Library) and the Host Library (the VX8 Instrument Driver).

The C4x Support Software Library contains TMS320C4x functions which perform common tasks (initialization and control) and data transfers required by all C4x processors on the VX8 Carrier Board.

The VX8 Instrument Driver provides a host API for performing configuration, control, and communications with a VX8 system. The primary functions of the VX8 Instrument Driver are to initialize and communicate with a VX8 system.

The VX8 board does not ship with resident application software. The VX8 requires C40 and host software development, using the VX8 Support Software, in order to integrate a VX8 DSP subsystem into a total VXIbus solution. The VX8's overall functionality depends on its hardware configuration and the host and C4x application software through which the VX8 will be able to process data and communicate with other devices.

Note: The VX8 Carrier Board is not a typical VXIbus Instrument due to its configurable nature. The functionality and behavior of a VX8 is completely defined by its application software and its hardware configuration. The VX8 devices must be loaded with a DSP application before they become functional instruments.

3.1. Software Environment

Creating a VX8 application can be broken up into two interrelated tasks: Host software development and DSP software development. The VX8 Support Software product gives you both host and DSP routines to help you develop your application in less time without the need of having detailed knowledge about the inner workings of the VX8. *Figure 6* shows a high level view of the various components in the VX8 Software environment.

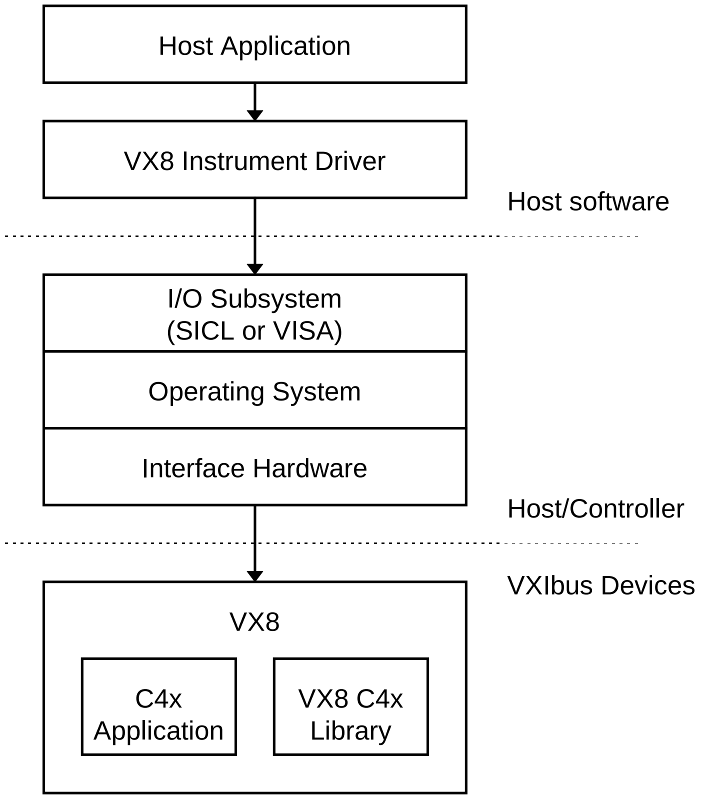


Figure 6 VX8 Software Environment

3.1.1. Host Software Environment

There are many VXIbus host controllers and software configurations available from manufacturers. Fortunately, developing VXIbus host applications on these development platforms is greatly simplified by the standard I/O library present in most VXIbus systems.

The I/O library provides host software with an API (Application Program Interface) to routines for VXIbus communications and control. The I/O library provides numerous functions ranging from low level I/O to higher level interactions like message based communication. The I/O library simplifies host software development by isolating the host software from the underlying host operating system (OS) and interface hardware. Together, the VX8 specific functions contained in the VX8 Instrument Driver and the I/O library provide application developers with a standard, easy to use software interface for developing the host based component of your VX8 application.

The I/O libraries supported by the VX8 Instrument Driver are the VXIpnp Alliance defined **VISA** (Virtual Instrument Software Architecture) and **SICL** (Hewlett-Packard Standard Instrument Control Library).

SICL is available from Hewlett-Packard for many of its VXIbus products. VISA is an open software standard defined by the VXIpnp Systems Alliance. VISA is available on a variety of VXIbus products from Hewlett-Packard and National Instruments, among others.

The VXIpnp components of the VX8 Instrument Driver (the VXIpnp module) **do not** apply to systems using the SICL I/O subsystem.

3.1.2. DSP Software Environment

The development environment for the TMS320C4x DSP software is relatively straight forward. You can develop C4x DSP application code for the VX8 on any platform which supports Texas Instruments' TMS320C4x tools.

Currently, you may use DOS, WIN95, WIN NT, or Sun / HP-UX based environments to author and compile DSP software. TI Tools version 5.0 is required under Windows NT.

JTAG based debugging can only be performed on platforms which have C4x JTAG support (DOS, WIN95, WIN NT, SunOS/Solaris). JTAG based debugging is currently **unavailable** for HP-UX based computers.

In addition to the C4x tools available from TI, there are a number of tools available from third party vendors which can help in your application development and debugging.

The VX8 C4x Support Software Library provides VX8 DSP applications with a simple API for initialization, configuration, control, and I/O routines tailored for the VX8 hardware. These routines reduce the amount of in depth information VX8 developers require to generate their applications.

Using the VX8 C4x Support Software Library is a simple chore. To gain access to the calls in the library, simply include the required include files and link in the library as you would with any other static library when you build your DSP software. Specific information about the VX8C4xSS will be detailed in later chapters.

3.2. VX8 Support Software

The VX8 Support Software provides host and DSP initialization, communications, and control functions for the VX8 Carrier Board. The VX8 Support Software consists of several major components:

- SICL/VISA Instrument Driver
- VX8 C4x Support Software Library
- Example programs

3.2.1. SICL/VISA VX8 Instrument Driver

The VX8 Instrument Driver provides a host API for your application to initialize, control, and communicate with a VX8 system. The device driver consists of ANSI-C host code which will function with either VISA or SICL I/O libraries.

Source code is provided for the VX8 Instrument Driver.

3.2.2. VX8 C4x Support Software Library

The VX8 C4x Support Software Library provides your DSP application with routines to configure and communicate with the various VX8 interfaces.

The C4x library can be separated into the following components:

- **Global Bus Interface Module** supporting accesses from a DSP to the global shared or DRAM shared buses
- **VXIbus Interface Module** supporting direct mastering of the VXIbus from a DSP, SCV64 DMA initialization, and VXIbus interrupt support
- **HP Local Bus Interface Module** supporting setup, initialization, and data transfer for the HP Local Bus Interface
- **DUART Module** providing initialization, data transfer and interrupt support for the dual UART on Node B’s Local Bus
- **Node A Initialization Kernel Module** supporting VXIbus A16 register access from the host and VX8 initialization after a board reset. The Node A C40 DSP must service any requests made by the host through the A16 configuration registers. Users must link the supplied interrupt service routine for IIOF3 in with their Node A DSP application code and enable this interrupt. This is taken care of by the **boota.asm** boot initialization routine

Source code is provided for the C4x Support Software Library.

3.2.3. Example Programs

The VX8 Support Software provides several example programs which demonstrate how to use the various interfaces on the VX8 as well as how to combine host and DSP software to form an application. Some of the examples require additional hardware to run. Please refer to the example program notes in the *Chapter 11* of this manual for system requirements.

The following table lists the symbolic constants used in the example programs.

Table 1 Symbolic Constants Used in the Example Programs

Constant	Example program	Description and/or Value
_FROM_DRAM	vx8mult.c, vx8mult.h	0
ADDR_INC	vx8mult.c, vx8mult.h	Address increment; is 4 or 1 depending on whether the target is byte or long word addressed. It will be 1 for C4x, 4 for VXIbus devices
blockSize	vx8hpgen.c	0x10
BOARD2_BASE_ADDR	vx8vxi.c	0x28000000
CH1	vx8duart.c	channel number; 1
CH2	vx8duart.c	channel number; 2
DEST	vx8vxi.c	BOARD2_BASE_ADDR + VX8_A32_SLV_DRAM_OFFSET
HP_CONSUME_ISR	vx8hpcon.c	rename the ISR; c_int04
LENGTH	vx8duart.c	ramp length; 10
LENGTH	vx8vxi.c	0x1000
MULT_CTRL_REG, MULT1_REG, MULT2_REG, MULT_RESULT_REG	vx8mult.c, vx8mult.h	mult example registers
MULT_LDF_FILE	mult.c	"mult.ldf"
MULT_REG_OFFSET	vx8mult.c, vx8mult.h	offset into the DRAM space to have registers
MULT_SDF_FILE	mult.c	"mult.sdf"
SCV_ISR	vx8lm.c, VX8DMA.C, vx8vxi.c	rename the ISR; c_int03
UART_ISR	Vx8duart.c	rename the ISR; c_int06
VX8_A32_SLV_DRAM_OFFSET	vx8vxi.c	0x04000000
VX8_BOARD240	mult.c	"vxi,240" or "VXI0::240::INSTR"
VX8_MULT_UNINIT, VX8_MULT_LOADED, VX8_MULT_START, VX8_MULT_DONE, VX8_MULT_END	vx8mult.c, vx8mult.h	MULT control register defined values

3.3. Data Types

Data types used in VX8 host and DSP code are defined in these header files:

- ssvx8typ.h defines data types used in VX8 host code
- type_c3x.h defines data types used in VX8 DSP code

The following base type modifiers are used:

U	unsigned
R	register
P	pointer
V	volatile

Note that “volatile” generally applies to everything dereferenced from the base type. For example, PVPVUINT16 == PVPUINT16.

Table 2 Data Types

Data type	Description
STATUS	unsigned long; error code return value for functions
PVOID	pointer to void
Device handles	
VXIDEV_HANDLE	typedef ViSession (see VISA documentation)
PVXIDEV_HANDLE	typedef ViPSession (see VISA documentation)
INT8 types	
INT8	char
PINT8	pointer to a char
VINT8	volatile char
PVINT8	pointer to a volatile char
VPINT8	volatile pointer to a (volatile) char
PVPINT8	pointer to a volatile pointer to a char
UINT8 types	
UINT8	unsigned char
VUINT8	volatile unsigned char
PUINT8	pointer to an unsigned char
PVUINT8	pointer to a volatile unsigned char
VPUINT8	volatile pointer to a (volatile) unsigned char
PVPUINT8	pointer to a volatile pointer to a (volatile) unsigned char
INT16 types	
INT16	short
VINT16	volatile short
PINT16	pointer to a short
PVINT16	pointer to a volatile short
VPINT16	volatile pointer to a (volatile) short
PVPINT16	pointer to a volatile pointer to a short
UINT16 types	
UINT16	unsigned short
VUINT16	volatile unsigned short
PUINT16	pointer to an unsigned short
PVUINT16	pointer to a volatile unsigned short
VPUINT16	volatile pointer to a (volatile) unsigned short
PVPUINT16	pointer to a volatile pointer to (volatile) unsigned short
INT32 types	
INT32	long
VINT32	volatile long

Data type	Description
INT32	pointer to a long
PVINT32	pointer to a volatile long
VPINT32	volatile pointer to a (volatile) long
PVPINT32	pointer to a volatile pointer to a long
UINT32 types	
UINT32	unsigned long
VUINT32	volatile unsigned long
PUINT32	pointer to an unsigned long
PVUINT32	pointer to a volatile unsigned long
VPUINT32	volatile pointer to a (volatile) unsigned long
PVPUINT32	pointer to a volatile pointer to (volatile) unsigned long
floating point (32-bit) types	
FLOAT32	float
VFLOAT32	volatile float
PFLOAT32	pointer to an float
PVFLOAT32	pointer to a volatile float
VPFLOAT32	volatile pointer to a (volatile) float
PVPFLOAT32	pointer to a volatile pointer to a (volatile) float
floating point (64-bit) types	**(defined in ssvisa.h)**
FLOAT64	float
PFLOAT64	pointer to an float
VFLOAT64	volatile float
PVFLOAT64	pointer to a volatile float
string types	
STRING	char
STRING256[MAX_STR_LEN]	string of 256 characters
Boolean types	
BOOLEAN	unsigned long

3.4. Hardware and Software Requirements

VX8 application developers should be familiar with VISA and/or SICL software development. A knowledge of ANSI C software development on VME/VXIbus systems, and TMS320C4x software development in ANSI C/assembly is vital.

VX8 Support Software for HP-UX 9.X/SICL was developed on an HP V743 Embedded controller with HP-UX 9.05 and SICL C.03.09. This version of VX8 Support Software is to be used with the following hardware and software configurations:

Hardware	Software
• VXIbus mainframe chassis with a minimum of 2 slots (for slot 0 controller and a single VX8 card) • VXIbus slot 0 controller (HP V743 VXIbus Embedded Controller for example) • VX8 Carrier Board • DOS/WIN 95 PC and an external JTAG interface for TMS320C4x DSP software development	• ANSI-C compiler • DOS based TI TMS320C4x development tools • SICL C.03.09 • HP-UX 9.05

The VX8 VISA Windows 95 and Windows NT Instrument driver was developed under Windows NT version 4.0 using Microsoft Visual C compiler version 5.0 with a National Instruments VXI-MXI2 extender. It supports the following hardware and software configurations:

Hardware	Software
• VXIbus mainframe chassis with a minimum of 2 slots (for slot 0 controller and a single VX8 card) • VX8 Carrier Board • DOS or Windows 95 PC with an external JTAG interface for TMS320C4x DSP software development. This can be the same PC as the host if running Windows 95. TI Tools version 5.0 and Go DSP Code Composer are required for ‘C4x development on Windows NT.	• Microsoft Visual C compiler version 5.0 • TI TMS320C4x development tools (version 5.0 required for Windows NT) • VISA (version 1.1 or later) • Windows 95 or Windows NT

4 Reset Conditions and Initialization

4.1. Reset

The VX8 board can be reset from a number of sources, all of which will generate either a hard or soft reset condition.

4.1.1. Hard Reset

A hard reset signifies the resetting of the entire system. This includes the Slot 0 controller, other VXIbus devices, and all VX8 boards. This condition resets all devices on the VX8 board, and the VXIbus Resource Manager configures the VX8 board via the A16 space. These writes to A16 cause the Node A IIOF3 ISR to be triggered. Sources of hard resets include:

- **Power On Reset (PORST) -** Entire VXIbus chassis is initialized after power is applied.
- **SYSRST -** The SYSRST* line on the VXIbus backplane is pulled low by another VXIbus device. SYSRST* is functionally equivalent to a PORST without the disruption of power from the backplane.

4.1.2. Soft Reset

A soft reset signifies the resetting of only VX8s in a system. This can be a single VX8 board or a number of VX8 boards connected by their front panel JTAG connectors. Since the state of the Slot 0 controller is not affected by this action, the Resource Manager will not re-configure nor re-initialize the devices via their A16 configuration registers. Therefore the previous A16 configuration information is still valid (offset value and A32 enable state). To retain this previous state across soft resets, the relevant A16 registers are not reset by a soft reset condition, however the READY and PASSED bits in the A16 Status Register are cleared. Sources of soft reset include:

- **VXIbus A16 Control Register Reset -** If the RESET bit in the A16 VXIbus control register is asserted, the VX8 board will generate a local reset to the entire board with the exception of the A16 Register Set. It will also cause the /GRESET line on the front panel JTAG connector to be asserted to ensure that other C4x DSPs in the system are not driving their COMM Ports in the wrong direction after a DSP is reset.
- **Front Panel JTAG -** If the /GRESET line on the front panel JTAG connector is asserted, the VX8 board will generate a local reset to the board.

Note: The SYSFAIL LED will light if a board is reset by its A16 control register. Boards reset via front panel JTAG will not light their SYSFAIL LEDs.

4.2. Boot Kernel Initialization

The Node A Embedded C40 Boot Kernel is responsible for initializing the VX8 on power-up and interacts with several instrument driver functions. The initialization must be performed by Node A since it is the only DSP on the VX8 with access to the VXIbus defined A16 registers.

The A16 Interrupt Service Routine (ISR) portion of the Node A boot kernel must remain resident on Node A to perform actions triggered by writes to the A16 VXIbus registers by the host. This is accomplished by linking in the ISR for IIOF3 on Node A with your DSP application code for Node A.

The following table illustrates the sequence of events that takes place after a reset:

Table 3 Boot Kernel Initialization

Host	Node A DSP	Other DSPs
Asserts, then releases /RESET	Boots kernel code from the Boot Kernel PEROM and runs board self test	Configured to boot from COMM Port, so they are waiting for instruction
Writes to A16 Configuration Registers to setup A32 offset and sets A32 Enable bit	Receives an interrupt on IIOF3, reads the A16 Configuration Registers, initializes SCV64, and enables A32 slave image	Still waiting
Optional: Sets the bottom two 32-bit words in Node A Near Global to “uninitialized” values. Writes to A16 Control Register requesting Firmware Revision and Self Test Status	Receives an interrupt on IIOF3, reads the A16 Configuration Registers, writes self test results to address 0x8000 0001, writes firmware revision to address 0x8000 0000, writes firmware revision and self test status bits to A16 Status Register	Still waiting
Optional: Polls on Firmware and Self Test locations in Node A Near Global for the values to change from the “uninitialized” value then clears the Firmware Rev and Self Test Query bits in the A16 Control Register.	No Action	Still waiting
Host starts code download by loading the intermediate boot kernel to all DSPs through Node A.	Node A receives command and downloads intermediate boot kernel to other nodes via COMM Port.	Receives Intermediate Boot Kernel from Node A via COMM Port. Not all DSPs are physically connected to Node A via COMM port, so this download may take several ‘hops’.
Host continues code download by sending application code to the furthest DSP in the COMM Port chain.	Passes code data along.	Target DSP receives user’s application code and starts executing it.
Last DSP to be downloaded is Node A	Node A receives application code and starts executing. The Node A boot kernel is now gone.	All DSPs are now running user’s application code.

Since there is a Boot Kernel running on Node A and intermediate boot loaders are downloaded to each DSP at initialization, there are some DSP resources that are not available to you at various times. The next section describes what these resources are and how contention is avoided.

For information on the SCV64 configuration after a reset, refer to the *SCV64 Default Configuration* section of the *VX8 Carrier Board Technical Reference Manual*.

5 VX8 C4x Software System Description

5.1. Introduction

A library of DSP function calls is provided that allow you to start developing application code faster by not having to fully understand the complexities and register level functionality of the hardware interfaces. These function calls provide initialization and control functionality for all hardware interfaces, as well as optimized data transfer routines for moving data between the various memory banks on the board. Complete source code for the C4x Support Software is provided allowing you to expand on the functionality provided or to further optimize the provided functions.

5.2. Fundamentals of C4x Code Development

5.2.1. Procedure for Getting a DSP C Program Running

To get a C program running:

1. Write the C Source Code for the DSP using the examples provided as a guideline.
2. Compile and assemble the program using the supplied TI Floating Point compiler batch files.
3. Link the resulting object file with the C startup (or boot) file, the appropriate VX8 C4x Support Software Library, the TI run-time support library RTS40.LIB and the linker command file (describing memory types and memory maps). Any additional libraries, such as PRTS40.LIB (the parallel Run-Time Support Library from TI) must also be linked with your application code. See *Figure 7* which follows.
4. Download and run the resulting **.out** file using the debug monitor (XDSC40 if using a Spectrum supplied debugger) and debug the code using breakpoints, watch windows, etc.

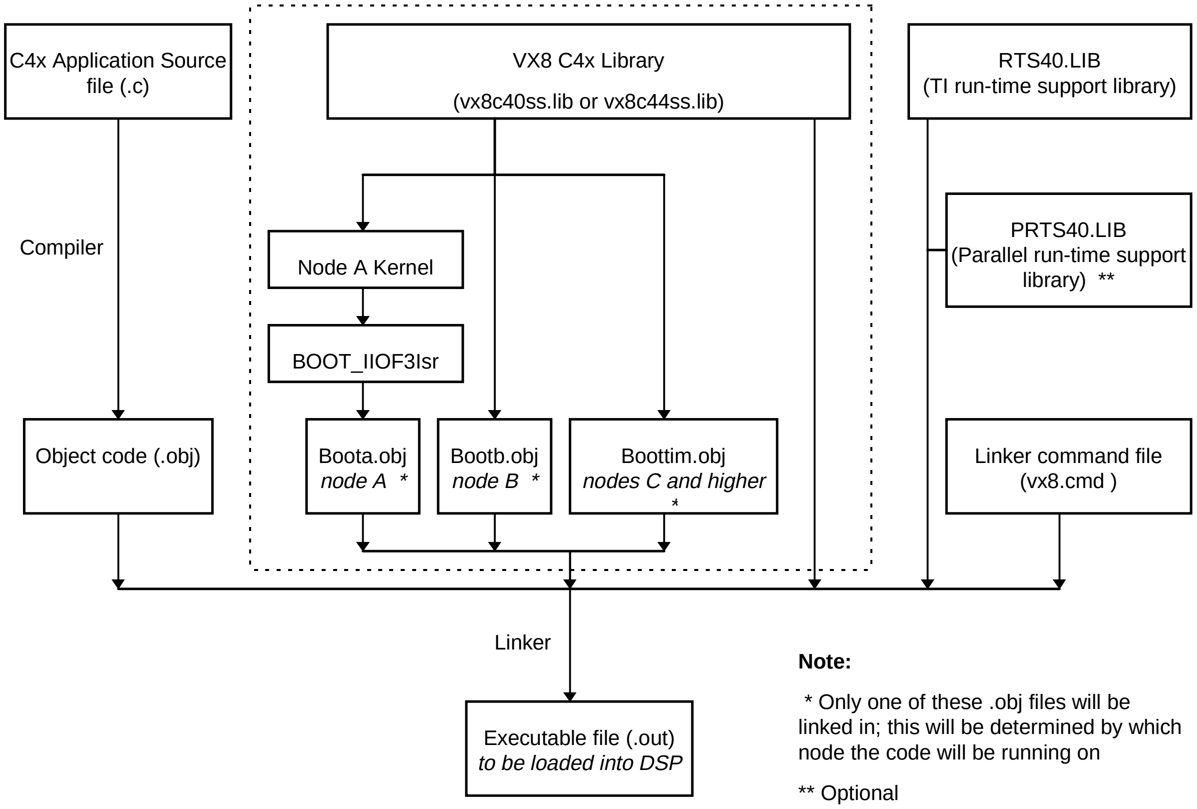


Figure 7 Building an Executable File

Notes:

- The VX8 C4x Support Software that is shipped to you is built to support stack passing of variables and the small memory model. You’ll need to rebuild the libraries if a different memory model or variable passing scheme is required.
- If possible, run your DSP program and frequently accessed data out of internal or Local Bus memories. This will prevent the DSPs from stalling when the Near Global Memories are accessed by the SCV64, HP Bus Interface, or other C4x’s.
- For a complete description of the compiler switches, compiler link sections, and linker command files refer to the Texas Instruments *TMS320 Floating-Point DSP Optimizing C Compiler User’s Guide* and the *TMS320 Floating-Point DSP Assembly Language Tools User’s Guide*.
- Two compilations of the **vx8c4xss** are supplied: **vx8c40ss.lib** and **vx8c44ss.lib**. **vx8c40ss.lib** should be linked with DSP application software which will be loaded onto ‘C40 based TIM-40 modules, including the Node A and Node B embedded processors. **vx8c44ss.lib** should be linked with DSP application software destined for ‘C44 based TIM-40 modules. See *section 5.2.2* for more information.

- The VX8 C4x Support Software has been compiled using TI Floating Point Tools version 4.70. The C4x Support Software library may require re-compiling if you are developing with a different version of the TI Floating Point Tools.
- When building code with version 5.0 of the TI Floating Point Tools, use the COFF version 0 flag, -v0.
- The batch files supplied with the VX8 C4x Support Software are written for DOS/Win95/WinNT environments. To operate under UNIX, these batch files require some modification.
- You will be required to set up your Texas Instruments TMS320C4x tools environment in order to build your code. Refer to the *Getting Started Guide* supplied with the TI tools for instructions on installing the supplied TI C4x tools and setting up the environment variables (path, A_DIR, C_DIR, C_OPTION, and TMP) used by the tools.

5.2.2. Compiler Batch Files

Compiler batch files for the TI Floating Point Tools are supplied with the VX8 Carrier Board C4x Support Software Disk.

Note: Several batch files are provided for the various processing nodes on the VX8 Carrier Board because different boot routines and linker command files are needed for the specific processing nodes.

The following batch file is named **ticompa.bat** and should be used for compiling and linking C DSP code for the Node A embedded C40.

```
call cl30 -v40 -g boot_a.asm %1 -z -x -cr -o %1.out
vx8a.cmd -l vx8c40ss.lib -l rts40.lib -l prts40.lib -m
%1.map
```

The following batch file is named **ticompb.bat** and should be used for compiling and linking C programs for the Node B embedded C40.

```
call cl30 -v40 -g boot_b.asm %1 -z -x -cr -o %1.out
vx8b.cmd -l vx8c40ss.lib -l rts40.lib -l prts40.lib -m
%1.map
```

The following batch file is named **ticompt.bat** and should be used for compiling and linking C programs for MDC40Sxx TIM modules. This batch file uses an example linker command file. Please refer to the *User Manual* for the specific TIM-40 module that you are using, as it will highlight any particular hardware requirements for the linker command file or the boot file.

```
call cl30 -v40 -g boot_tim.asm %1 -z -x -cr -o %1.out
vx8_tim.cmd -l vx8c40ss.lib -l rts40.lib -l prts40.lib -m
%1.map
```

When building DSP code for MDC44ST ‘C44 processors, link the **vx8c44ss.lib** instead of **vx8c40ss.lib**. The -v44 flag should be used except for in the optimizer; use the -v40 flag. This is due to a bug in the TI tools version 4.70 and 4.60.

5.2.3. Boot (Startup) Files

The C boot initialization routine performs the C runtime environment initialization. This routine consists of an assembly language function called **c_int00** which performs the following operations:

1. Defines the code entry point to be **c_int00**
2. Allocates and initializes the system stack and sets up the frame pointer
3. Auto-initializes global variables by copying data from the initialization tables in **.cinit** to the storage allocated for the variables in **.bss**. This is only done when the ROM memory model is used
4. Performs a C4x IACK instruction causing the VX8 to release interrupts for general use
5. Releases the /CONFIG line indicating the user’s application code is now running. On Nodes A&B /CONFIG is accessed through a register. On most TIM-40 modules, /CONFIG is connected to IIOF3

   Note: If you want to use this functionality on TIM modules that do not use the IIOF3 I/O pin as CONFIG*, you will have to modify the **boot_tim.asm** initialization file.

6. Calls the function **main()** to begin executing the user’s C program
7. Goes into an infinite loop if control returns from **main()**

The COFF Loader provided with the VX8 Carrier Board Support Software will ensure that the entry point to the user’s application code is **c_int00**.

Three boot files are provided with the VX8 Carrier Board: **boot_a.asm**, **boot_b.asm**, and **boot_tim.asm**. These files are for use with Node A embedded C40, Node B embedded C40, and with standard MDC40Sxx TIM-40 modules, respectively.

5.2.4. Linker Command Files

Linker command files are ASCII files that contain one or more of the following:

- **Input filenames -** can be object files, archive libraries, or other command files;
- **Linker options -** can be used in the command file in the same manner that they are used on the command line;

- **Linker Directives -** either MEMORY, which specifies the target memory configuration, or SECTIONS, which controls how code sections are built and allocated; and
- **Assignment statements -** define and assign values to global symbols.

Three linker command files are provided with the VX8 Carrier Board: **vx8a.cmd**, **vx8b.cmd**, and **vx8_tim.cmd**. These files are for use with Node A embedded C40, Node B embedded C40, and with standard MDC40Sxx TIM-40 modules, respectively.

5.2.5. C4x DSP Local and Global Memory Maps

Refer to the *VX8 Carrier Board Technical Reference Manual* for detailed processor memory maps.

5.3. Resources Required by the VX8 C4x Support Software Library

There are some system resources that are not available to you at various stages of initialization, booting, and at runtime. This section describes what resources are required by the VX8 C4x Support Software Library, when they are required, and what the limitations are to your code.

5.3.1. Node A TMS320C40

Memory Node A requires external memory for executing the Node A Boot Kernel and it requires external memory as storage for data passed from the host. You cannot place any code (**.text** section) or initialized variables (**.cinit** section) from your application code in the memory space allocated for the Node A Boot Kernel.

After loading the application code, this memory space is free for use by your application. The default Node A linker command file **vx8a.cmd** defines the space KERNEL at address 0x8000 0000 with length 0x800 (8 Kbytes). This prevents the TI Floating Point Tools from placing your application code within this space.

Table 4 Node A Memory Restrictions

C40 Address	Description
8000 0000h 8000 0001h	Used for firmware revision and self test results reporting to the host. Can be modified at any time by the A16 Control Register ISR.
8000 0002h 8000 07FFh	Reserved for Node A Boot Kernel during loading and is free for use after loading.

Note: Addresses 0x8000 0000 and 0x8000 0001 of Node A’s Near Global SRAM are used by the A16 Control Interrupt Service Routine to return Firmware Revision and Self Test Results back to the host when queried. If the host needs to access this information at runtime, you should avoid using these memory locations in your application.

Lock Stack The lock stack is defined in the **boot_*.asm** code as an uninitialized section **.lstack** of size LOCK_STACK_SIZE. The .lstack section can be relocated to other regions of the C4x memory map in the corresponding **vx8*.cmd** linker command file. No other memory is required for support of the VX8 C4x Support Software Library, and no other memory is reserved after booting of your application code.

Interrupts The external interrupt IIOF3 on Node A is always allocated to VXIbus A16 Register Support. The interrupt service routine for this interrupt is provided, **BOOT_IIOF3Isr()** (an alias for c_int06, the required nomenclature for ISRs), with the VX8 C4x Support Software Library and must be linked with your application code. No other interrupts are required for support of the VX8 C4x Support Software Library.

COMM Ports Ensure that COMM Port transfers are not initiated until the loading process is complete. This will prevent DSPs that are attempting to boot via COMM port from being corrupted. We recommend having the host use a flag to signal the DSP to continue code execution once loading is complete.

5.3.2. Node B TMS320C40 and TIM-40 Based DSPs

Memory The Node B TMS320C40 and all TIM-40 Based C4x DSPs require external memory for executing the intermediate COMM Port Boot Kernel. You cannot place any code (**.text** section) or initialized variables (**.cinit** section) from your application code in the memory space allocated for the Intermediate Boot Kernel. After loading of your application code is complete, this memory space is now free for use as uninitialized variables, working data memory, or as a target for HP Local Bus DMA transfers. The default Node B linker command file **vx8b.cmd** defines the space COM_KERNEL at address 0x8000 0000 with length 0x100 (1 Kbyte). The default TIM-40 based DSP linker command file **vx8_tim.cmd** also defines the space COM_KERNEL at address 0x8000 0000 with length 0x100 (1 Kbyte). These space definitions prevent the TI Floating Point Tools from placing the user’s application code in contention with the Intermediate Boot Kernel.

Lock Stack The lock stack is defined in the **boot_*.asm** code as an uninitialized section **.lstack** of size LOCK_STACK_SIZE. The .lstack section can be relocated to other regions of the C4x memory map in the corresponding **vx8*.cmd** linker command file. No other memory is required for support of the VX8 C4x Support Software Library, and no other memory is reserved after booting of your application code.

Interrupts No Interrupts are required for support of the VX8 C4x Support Software Library on Node B or on TIM-40 based DSPs.

COMM Ports Ensure that COMM Port transfers are not initiated until the loading process is complete. This will prevent DSPs that are attempting to boot via COMM port from being corrupted. We recommend having the host use a flag to signal the DSP to continue code execution once loading is complete.

6 VX8 C4x Support Software

The VX8 C4x Support Software Library is divided into 5 distinct modules: **A16 Control, Global Bus, VXIbus, HP Local Bus**, and **DUART**. These modules are discussed in the sections that follow.

Use of the function calls in your application code will result in a typical function call series of events, where the relevant registers are pushed onto the stack and the function is called. Return from the function pops the registers off the stack to restore the environment of the calling function. The TI Floating Point Tools decide at compile time which registers can be used.

The prefixes used in the VX8 C4x Support Software functions indicate their usage, as shown in the following table:

Table 5 Prefixes Used in the VX8 C4x Support Software

Prefix	Module
BOOT	Node A routines
DUART	DUART functions
HPBUS	HP Local Bus functions
SCV64	SCV64 functions
VX8	Global Shared Bus functions

6.1. A16 Control Module

6.1.1. Description

The A16 Control Module sets up the VXI A16 interface of the VX8, for use by any of Spectrum's supplied host interface libraries.

The A16 Control Module is part of a larger group of functions that make up the Node A Boot Kernel PEROM code.

Caution: Do not reprogram or modify the code in the Node A Boot PEROM except when installing Spectrum supplied firmware updates.

Some functionality from the Boot Kernel Module must be present at runtime to support the VXIbus A16 Configuration Registers. This is accomplished by linking in the VXIbus A16 Interrupt Service Routine (BOOT_IIOF3Isr) with your application code (see *Figure 7* in *section 5.2.1*). For information on resources required by this module, see *section 5.3.1*.

6.1.2. A16 Control Function

Table 6 A16 Control Function

Function	Description
BOOT_IIOF3Isr	Handles VXIbus requests via the A16 control and offset registers.

6.2. Global Bus Interface Module

6.2.1. Description

These functions are used for all transactions between a C4x node and an external resource on the Global Shared and DRAM Shared buses. These external resources include memory, registers, other Node’s Near Global SRAM, DRAM, the HP Local Bus, and the VXIbus. The transfers require the global bus to be locked by the calling C4x (by using VX8_Lock).

Transfers local to the calling C4x (internal RAM, local SRAM, near global SRAM) can be performed through direct read/write accesses to increase throughput and simplify code design. C4x DMA to the Global Shared Bus is not recommended.

Functions specific to the operation of the HP Local Bus and the VXIbus are outlined in their respective sections.

6.2.2. Global Bus Functions

Table 7 Global Shared and DRAM Shared Bus Functions

Function Name	Function Description
VX8_FastTransfer	Performs single cycle 32-bit data block transfers between the calling processor's memory space and another memory location. Since this function does **not** lock the DSP to a target, it **must** be preceded by a call to **VX8_Lock** and followed by a call to **VX8_Unlock**.
VX8_Lock	Locks the global bus for access by the calling C4x. The previous location on the lock stack is unlocked and the new global bus address location is locked.
VX8_Read	Sets up and reads a block of data from a defined address space on the global bus or the VXIbus.
VX8_ReadAsyncReg	Reads a single memory location from an asynchronous register on the VX8 board. This function reads the memory location until two consecutive reads return identical values.
VX8_ReadBit	Reads the value of a particular bit from a memory location on the global bus and returns the bit value.
VX8_ReadReg	Reads a 32-bit value from a single memory location on the global bus. This routine should not be used when referencing a DSP's own local or near global memories.
VX8_SetUserLED	Sets the state of the front panel user-defined LED.
VX8_UnLock	Unlocks the current global bus resource, and locks the previous global bus address location according to the lock stack.
VX8_Write	Sets up and writes a block of data to a defined address space on the global bus, or the VXIbus.
VX8_WriteBit	Sets the state of individual bits in a memory location on the board. Should not be used for transfers to a DSP's own RAM.
VX8_WriteReg	Writes a 32-bit value to a single memory location on the global bus. This function should not be used for transfers to a DSP's own RAM.

6.2.3. Shared Bus Transfers

Shared bus accesses to another C4x's Near Global SRAM, the Global Shared DRAM, the LED register, the SCV64, the HP Bus registers, or the VSTATUS/VCONTROL register can be done with the generic global bus functions defined in this module.

For information on Bus Locking, see *section 6.5.*

6.2.4. The Global Shared Bus Lock Stack

Since the interrupt service routines or multi-tasking operating systems may require access to the Shared Buses, a Lock Stack has been implemented to support seamless /LOCK operation.

Note: Use of the Global Shared Bus Lock Stack is seamless if the VX8 C4x Support Software Library functions are used to access the Shared Buses. This section is provided for reference purposes only. If the VX8 C4x Support Software Library functions are not used then the issues described here must be dealt with directly by your application code.

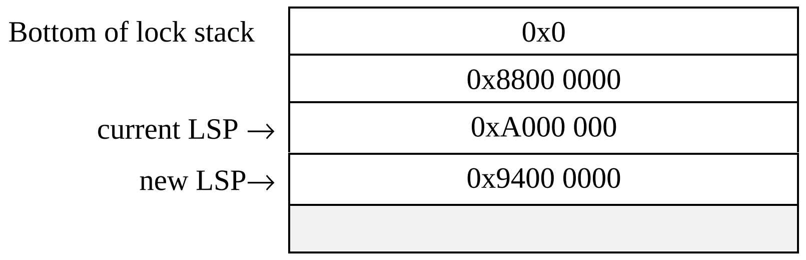
The global bus lock stack is a 16 deep global buffer on each C4x that holds current and previous global bus addresses. The stack is initialized by **c_int00** and manipulated by the functions: **VX8_Lock** and **VX8_UnLock**. All other functions manipulate the lock stack through calls to these base functions.

Note: The size of the Global Shared Bus Lock Stack is set in **boot_a.asm**, **boot_b.asm**, and **boot_tim.asm** by the "LOCK_STACK_SIZE .set" line near the top of each of these files. Modify this line and reassemble your code to change the size of the lock stack. You will also have to adjust your linker command file to allocate more memory for the lock stack.

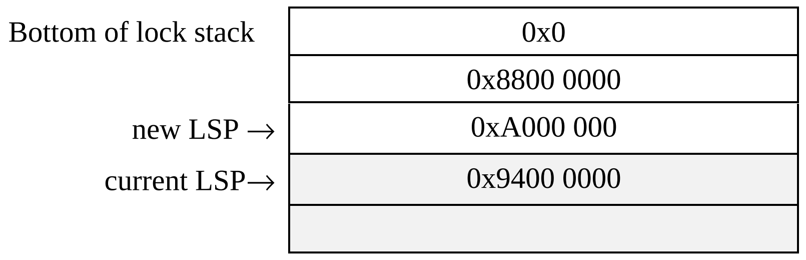
The lock stack pointer is used to push and pop lock addresses onto the lock stack. This variable is a global variable, and is initialized to zero (unlocked) by **c_int00** at boot time and before **main()** in your code.

When a call to a lock stack function is made, interrupts on the C4x are disabled, the unlock operation is performed, the lock stack pointer is updated, a lock operation is performed, and the global interrupts are enabled.

For example, if the global bus is currently locked to Node C (0xA000 000), and the C4x is locking to DRAM (0x9400 0000), the Node C address will be unlocked (current LSP), the LSP moved up in memory, and the bus locked to DRAM (new LSP).



When unlocking from DRAM, the DRAM address is unlocked (current LSP), the LSP moved down in memory, and the bus locked to Node C (new LSP). The old DRAM lock address still exists on the lock stack, but is considered invalid.



This system allows for multiple nested calls to **VX8_Lock** and **VX8_Unlock** as long as they are symmetrical, as can be seen in the following example. This is useful for modular function development because you don’t have to worry about whether an address region has been locked by a calling function.

```
void function1 (void)
{
  VX8_Lock(0x88000000);    /* lock to SCV64 Register Set */
     :
     :         /* manipulate DRAM */
     :
  function2();

  VX8_UnLock();
}
void function2 (void)
{
  VX8_Lock(0xA000000);          /* lock to DRAM */
     :
     :         /* manipulate DRAM */
     :
  function3();

  VX8_UnLock();
}
void function3 (void)
{
  VX8_Lock(0x94000000);  /* lock to Node C */
     :
     :         /* manipulate Node C */
     :
  VX8_UnLock();
  return;
}
```

Functions that perform single address read or writes, such as **VX8_ReadReg** and **VX8_WriteReg** perform a lock and an unlock internally, but do not change the lock stack pointer.

6.3. VXIbus Interface Module

6.3.1. Description

The VXIbus Interface Module provides the capability for the C4x DSP nodes to write to the SCV64 for initialization, setup of VXIbus Mastering, VXIbus Interrupt Generation and Handling, and setup of the SCV64 DMA Controller.

Note: C4x DSP Nodes use only 32-bit long word addresses, so special attention must be given to D08 and D16 reads and writes. The software overhead for the SCV64 and byte-lane manipulation increase the number of cycles required for SRAM to VXIbus transfers. D32 transfers between SRAM and the VXIbus are preferred over D08 and D16 transfers where possible.

D16 and D08EO slave cycles to Near Global SRAM are not recommended.

6.3.2. VXIbus Functions

Table 8 VXIbus Related Functions

Function Name	Description
VX8_SCV64AckInterrupt	Generates a VXIbus /IACK cycle - used after an interrupt is received from the SCV64 on IIOF0.
VX8_SCV64DisableInterrupt	Disables an interrupt line from the VXIbus.
VX8_SCV64DMATransfer	Initiates DMA transfers between the VXIbus and DRAM/SRAM.
VX8_SCV64EnableInterrupt	Enables an interrupt line from the VXIbus.
VX8_SCV64GenerateInterrupt	Generates a VXIbus interrupt.
VX8_SCV64SetSysFail	Asserts or de-asserts the SYSFAIL* line on the VXIbus from the SCV64.
VX8_SCV64SetVXIBusReqRel	Sets VXIbus request and release parameters.

VXIbus transfers can be performed via **VX8_Read**, **VX8_Write**, or **VX8_SCV64DMATransfer** routines. All VX8_SCV64 prefixed functions lock to the SCV64 using the mailbox registers.

See *Chapter 11* for VXIbus software examples.

6.4. HP Local Bus Interface Module

6.4.1. Description

The Local Bus defined in VXIbus systems is a daisy chained bus that connects adjacent modules through the VXIbus mainframe. Pins on row C of connector P2 send data to pins on row A of the next module's P2 connector along the VXIbus mainframe as shown in the following diagram.

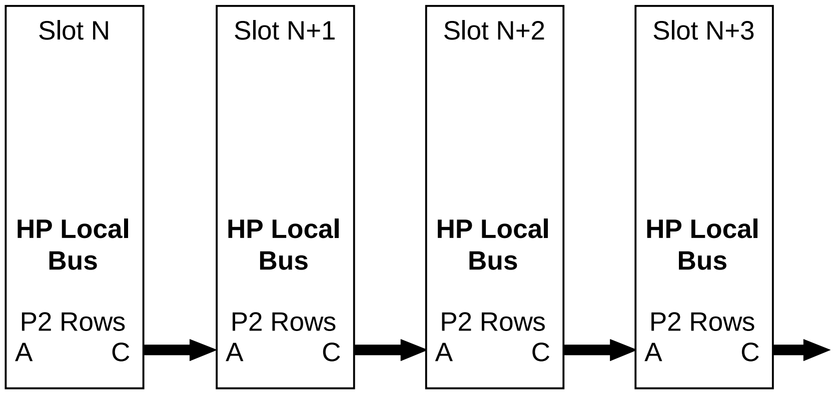


Figure 8 HP Local Bus Operation

The VXIbus HP local bus is a one-way bus that passes data from slot N to slot N+1.

The HP Local Bus interface on the VX8 Carrier Board combines the high speed data interface between VXIbus I/O modules with a flexible DMA Controller to provide a powerful I/O solution. Hewlett-Packard has a number of multi-channel analog I/O cards that communicate using the HP Local Bus.

Note: HP Local Bus **read**s (consuming data from the HP Local Bus) are only performed by the DMA Controller. The C4x DSPs cannot directly read data from the HP Local Bus Interface.

HP Local Bus **write**s (generating data to the HP Local Bus) are only performed by a C4x DSP (embedded or TIM-40 module based) writing directly to the output FIFO.

HP Local Bus transfers are done by either a collection of HP DMA Controller Initialization functions for reads, or a call to VX8_HPWriteFIFO for writes. Locks to the HP Local Bus DMA controller and WRITE FIFO are done via the HSTATUS register. Refer to the HP Local Bus CONSUME Mode example for DMA controller operation, and refer to the HP Local Bus GENERATE Mode example for an example of a DSP generating data to the HP Local Bus through writes to the FIFO.

6.4.2. HP Local Bus Interface Module Functions & Macros

Caution: Before attempting to program the HP Local Bus Interface, you should be familiar with the hardware implementation of the interface on the VX8 Carrier Board. These details are fully documented in the *VX8 Carrier Board Technical Reference Manual* .

Table 9 HP Local Bus Initialization/Status Functions

Function Name	Description
VX8_HPCheckInterrupt	Determines the source of an interrupt from the HP Local Bus.
VX8_HPClearInterrupt	Clears a specified HP Local Bus interrupt.
VX8_HPDisableInterrupt	Disables an HP Local Bus interrupt source for IIOF1.
VX8_HPEnableInterrupt	Enables an HP Local Bus interrupt source for IIOF1.
VX8_HPGetMode	Gets the mode of the HP BALLISTIC IC.
VX8_HPReset	Resets and sets up the DMA controller, HP BALLISTIC IC, and the HP Local Bus read and write FIFOs.
VX8_HPSetMode	Sets the mode, continue, and strip bits on the HP BALLISTIC IC.

Table 10 HP Local Bus CONSUME Functions

Function Name	Description
VX8_HPGetDMABuffer	Gets the value of the DMA_BUFFER bit in the HSTATUS register. Indicates which address buffer will be written to on the next write cycle.
VX8_HPGetDMAEnable	Gets the value of the DMA_ON bit in the HCONTROL register. This indicates a read DMA transfers from the HP Local Bus FIFO are enabled.
VX8_HPGetDMAIncrement	Gets the value of the DMA_INCR bit in the HCONTROL register. This value indicates whether DMA transfers are to consecutive addresses or to a port (consecutive writes to a single address).
VX8_HPGetFrameBit	Gets the value of the FRAME bit in the HWRITEEOB register.
VX8_HPSetDMAEnable	Sets the value of the DMA_ON bit in the HCONTROL register. This enables read DMA transfers from the HP Local Bus FIFO to one or more TIM-40 module(s) Near Global SRAM.
VX8_HPSetDMAIncrement	Sets the value of the DMA_INCR bit in the HCONTROL register.
VX8_HPSetDMATarget	Sets up address information for read DMA transfers from the HP Local Bus FIFO to a TIM-40 module’s SRAM.
VX8_HPSuspendDMA	Enables or disables the /SUSPEND line to the HP Local Bus DMA controller. This function can be called by Node A only.

Table 11 HP Local Bus GENERATE Functions

Function Name	Description
VX8_HPSetFrameBit	Sets the value of the FRAME bit in the HWRITEEOB register.
VX8_HPSetWriteBlockSize	Writes the HP Local Bus outbound block size to the HBLOCK register.
VX8_HPWriteFIFO	Single cycle 32-bit data block transfers to the HP Local Bus output FIFO.

Caution: The VX8 C4x Support Software Library provides functions that can be used to extend the basic functionality and modes of the BALLISTIC interface chip. The user must be familiar with the BALLISTIC HP Local Bus interface chip and the reference information described in the *VX8 Carrier Board Technical Reference Manual* before attempting to use extended HP Local Bus functionality.

Table 12 HP Local Bus Extended Functionality

Function Name	Description
VX8_HPGetContBit	Gets the value of the CONT bit in the HCONTROL register.
VX8_HPGetPauseBit	Gets the value of the PAUSE bit in the HSTATUS register. Indicates that the HP BALLISTIC IC is in a paused state.
VX8_HPGetRAEBit	Gets the value of the Read FIFO Almost Empty bit in the HSTATUS register.
VX8_HPGetRAFBit	Gets the value of the Read FIFO Almost Full bit in the HSTATUS register.
VX8_HPGetReadDone	Gets the value of the READDONE bit in the HREADEOB register.
VX8_HPGetREBit	Gets the value of the Read FIFO Empty bit in the HSTATUS register.
VX8_HPGetRFBit	Gets the value of the Read FIFO Full bit in the HSTATUS register.
VX8_HPGetStripBit	Gets the value of the STRIP bit in the HCONTROL register.
VX8_HPGetWAEBit	Gets the value of the Write FIFO Almost Empty bit in the HSTATUS register.
VX8_HPGetWAFBit	Gets the value of the Write FIFO Almost Full bit in the HSTATUS register.
VX8_HPGetWEBit	Gets the value of the Write FIFO Empty bit in the HSTATUS register.
VX8_HPGetWFBit	Gets the value of the Write FIFO Full bit in the HSTATUS register.
VX8_HPGetWriteDoneBit	Gets the value of the WRITEDONE bit in the HWRITEEOB register.
VX8_HPReadECL	Reads the value of the ECL trigger line.
VX8_HPReadTTL	Reads an 8-bit value from the TTL trigger lines.
VX8_HPRestart	Restarts the HP BALLISTIC IC when in a paused state.
VX8_HPSetContBit	Sets the value of the CONT bit in the HCONTROL register.
VX8_HPSetReadDone	Sets the value of the READDONE bit in the HREADEOB register.
VX8_HPSetStripBit	Sets the value of the STRIP bit in the HCONTROL register.
VX8_HPSetWriteDoneBit	Sets the value of the WRITEDONE bit in the HWRITEEOB register.
VX8_HPWriteTTL	Writes an 8-bit value to the TTL trigger lines.
VX8_HPWriteECL	Asserts or de-asserts the ECL trigger line.

6.5. Bus Locking

6.5.1. Why is Bus Locking Required?

All processing nodes on the VX8 Carrier Board have access to the Global Shared Bus and the DRAM Shared Bus. These shared buses are the gateway to system resources such as the HP Local Bus Interface, VXIbus Interface (Master), SCV64 (VXIbus Interrupts), front panel LED, DRAM, and other Processing Nodes' Near Global SRAMs. As a result, the global bus may be in demand by the HP Local Bus DMA controller, the SCV64 DMA controller, VXIbus masters, and DSPs on populated nodes.

To arbitrate shared bus requests from all sources, a bus locking mechanism is used in conjunction with the C4x interlock operations.

When a device requests the Global Shared Bus, it is granted the bus according to a certain priority (see section *Shared Bus Arbitration* in the *VX8 Carrier Board Technical Reference Manual* for more information).

6.5.2. Automatic Bus Locking

Generally, the provided libraries handle bus locking transparently to the user. Exceptions to this are **VX8_FastTransfer** (see *section 6.2.2 Global Bus Functions* and also *chapter 7*) and direct access from the user's programs.

6.5.3. Manual Bus Locking

For a DSP to access a resource on the Global Shared Bus or on the DRAM Shared Bus, the DSP must first be locked to that target by using the **VX8_Lock** function call. When it has finished its access, the DSP must be unlocked from that target by using the **VX8_Unlock** function call.

Nested locks/unlocks are supported by a special Lock Stack (see *section 6.2.4*).

6.5.4. How to Lock and Unlock the Global Shared Bus

A DSP must LOCK to a specific target. For example, Node A could lock to Node B's Near Global SRAM, however Node A can only communicate with Node B's Near Global SRAM, and cannot access another node's Near Global SRAM or the DRAM, HP Local Bus, or VXIbus. A separate UNLOCK (from the first target), LOCK (to the second target) sequence must be performed to switch targets.

Because of this memory addressing constraint when locking the global bus, the lock address is retained by the **VX8_Lock** and **VX8_Unlock** functions so that the memory space can be unlocked successfully.

A Global Shared Bus Lock Stack (see *section 6.2.4*) has been implemented to support seamless /LOCK operation for such nested locks/unlocks and also to support the interrupt service routines or multi-tasking operating systems which may require access to the Shared Buses.

6.5.5. Debugging Code With Bus Locking

The debugger will not be able to access any of the shared resources (for example, for displaying memory), unless the code running on the C4x is at a place that physically locks to that resource. To view a shared resource, if this is not the case, do the following:

1. In your code, place a lock to the shared resource, immediately followed by an unlock.
2. Set a breakpoint on the unlock instruction.
3. Run the code to the unlock instruction.
4. View the shared resource.

If the shared resource is visible from the VXI bus (for example, DRAM), use low-level peek and poke routines (usually supplied with the host) to non-intrusively view the shared resource.

Alternatively, you could write small C4x programs to lock to various areas and load and run them from your debugger as required. Since this is intrusive, you will need to reload and run your application.

6.5.6. Some Tips on Bus Locking

Avoid starving processors; don’t deny them access to the Global Shared Bus.

When locking to the VXIbus, perform the lock (**VX8_Lock**) to the mailbox registers on the SCV64 as this operation will be much faster than performing a dummy read from a VXIbus address.

When performing VXIbus master cycles from a C4x, globally disable the C4x interrupts (by calling a RTS40 library function **Int_Disable**). When finished, re-enable the interrupts. Disabling the interrupts is necessary, if there is any chance of the SCV64 mode or /VCONTROL register being altered by any other processors on the board.

The VXIbus interface can access the Shared DRAM Bus without stalling an HP Local Bus DMA block transfer.

Table 13 Accesses to Shared Resources Affect Other Resources

Resource being accessed	Accessed by what	What could be affected	Things to consider
Global Shared Bus	VXI bus transfers	HP Local Bus transfers	If the VXI interface attempts a slave access of the shared bus during a HP Local Bus DMA transfer on the HP Local Bus, the DMA transfer will stall.
a node's memory space	Host	The C4x node; HP Local Bus transfers	Accessing a C4x's global memory from the host is intrusive, especially if your DSP application uses the Global Shared Bus extensively. Using the shared DRAM to buffer data will isolate the host from colliding with C4x and HP BALLISTIC traffic on the global shared bus. Depending on your host interface, it may be more efficient to use DMA from the host or through the SCV64 DMA controller to perform data transfers.
Global Shared Bus	VXI bus transfers; HP Local Bus transfers	All C4x nodes	All C4x DSPs will be held off until the bus is released.
Global Shared Bus	Node A issuing a /SUSPEND signal	HP Local Bus transfers	For more details, see *VX8_HPSuspendDMA* in *chapter 7.*
Global Shared Bus	C4x node	Other C4x nodes	C4x nodes cannot preempt one another on the Global Shared Bus. All other C4x DSPs will be held off until the bus is released. Design your software in a way that processors do not starve one another.
Global Shared Bus	C4x DMA		C4x DMA to the Global Shared Bus is not recommended.
C4x Near Global memory	SCV64 DMA	All C4x nodes; HP Local Bus transfers	SCV64 DMA to C4x Near Global memories should be performed with caution as you can easily monopolize the Global Shared Bus and starve C4x and HP Local Bus accesses.

C4x node	HP Local Bus;
	SCV64 transfer;
	Another C4x node
		The C4x node's Near Global SRAM	A processor will not have access to its Near Global SRAM while the transaction is in progress.

6.6. DUART Module

6.6.1. Description

A 2 channel 16C550 style UART is provided for use on the VX8 Carrier Board. Only Node B has access to the DUART on its C4x Local Bus. Functions are provided in the VX8 C4x Support Software Library that can be used to initialize the device, configure interrupts, and transfer data between the DSP and the DUART. Both channels are brought to the front panel for connection with other standard RS-232 asynchronous interfaces in the VXIbus system. Use Cable VX8-60, rev 2.0, to connect the DUART with other interfaces. This cable can be ordered from Spectrum (part #: 00203629).

Note: The VX8 C4x Support Software Library does not provide a DUART Interrupt Service Routine because the functionality is typically very application specific. Refer to the DUART example software for a demonstration of how to configure and use interrupts for Asynchronous data transfer.

6.6.2. DUART Functions

Table 14 DUART Functions

Function	Description
VX8_DUARTCheckInterrupt	Checks which DUART interrupt was asserted on IIOF3 on Node B.
VX8_DUARTEnableInterrupt	Enables a Node B DUART interrupt source for IIOF3.
VX8_DUARTDisableInterrupt	Disables a Node B DUART interrupt source for IIOF3.
VX8_DUARTSetBaudRate	Sets the baud rates on the channels of the Node B DUART.

Table 15 DUART Macros

Macro Name	Macro Description
VX8_DUARTInByte	Reads an 8-bit value from the receive register of a DUART channel.
VX8_DUARTOutByte	Writes an 8-bit value to the transmit register of a DUART channel.

7 VX8 C4x Support Software Functions

This chapter presents the “C” language functions available in the VX8 C4x Support Software Library. The functions are listed in alphabetical order.

BOOT_IIOF3Isr

Function Handles VXIbus requests via the A16 control/offset register.

Include File vx8.h

Syntax void BOOT_IIOF3Isr (void);

Parameters None

Returned Value None

Remarks This function performs VX8 initialization on power up and is used in the servicing of the IIOF3 interrupt on Node A. Note that **BOOT_IIOF3Isr()** is a macro for **c_int06** (the syntax **c_intXX** prefix is required for ISR functions by the TMS320C4x compiler).

This function performs the following A16 control register actions:

- Sets the VX8 base address whenever the A16 Offset register is written.
- Enables or disables the A32 slave interface to the VX8 depending on the state of the A32 enable bit in the A16 Control Register.
- Writes the firmware revision number to the bottom of Node A’s near global memory (0x80000000 from the DSP. Base + 0x800000 from VXIbus A32).
- Writes the selftest results to Node A’s near global memory (0x80000001 from the DSP. Base + 0x800004 from VXIbus A32).

This function can be found in **a16ctrl.c**. This function is called in the Node A Boot Initialization kernel and must be linked with Node A’s DSP application software (see *Figure 7* in *section 5.2.1*).

This function **should not** be called by Node A’s DSP application code.

VX8_DUARTCheckInterrupt

Function Determines which event generated the interrupt from the DUART.

The associated DUART channel will freeze all interrupts and only indicate the highest priority interrupt pending.

Include File vx8.h

Interface UINT8 VX8_DUARTCheckInterrupt(UINT8 *channel*);

Parameters

channel	1 = check interrupts on channel 1 2 = check interrupts on channel 2

Returned Value

int_level	Interrupt pending on channel

Interrupt	Value	Description
DUART_INT_NONE	0x01	No interrupt pending on the DUART channel.
DUART_INT_LINE	0x06	“Receiver Line Status” interrupt pending on the DUART channel.
DUART_INT_RX	0x04	“Receiver Data Available” interrupt pending on the DUART channel.
DUART_INT_TIMEOUT	0x0C	“Character Timeout Indication” interrupt pending on the DUART channel.
DUART_INT_TX	0x02	“Transmitter Holding Register Empty” interrupt pending on the DUART channel.
DUART_INT_MODEM	0x00	“MODEM Status” interrupt pending on the DUART channel.

Remarks Upon receiving an IIOF3 interrupt from the DUART on Node B the source of the interrupt is unknown. This function reads the interrupt identification (IIR) register of the selected channel of the DUART.

This function should only be called in Node B DSP application software.

VX8_DUARTDisableInterrupt

Function Disables one of the interrupt sources on channel 1 and/or channel 2 of the DUART.

Include File vx8.h

Interface STATUS VX8_DUARTDisableInterrupt (UINT8 *channel*, UINT8 *int_level*);

Parameters

channel	0 = program both channels 1 = program channel 1 2 = program channel 2
int_level	Interrupt(s) to disable

Interrupt	Value	Description
DUART_INT_LINE	0x06	“Receiver Line Status” interrupt on the DUART channel.
DUART_INT_RX	0x04	“Receiver Data Available” interrupt on the DUART channel.
DUART_INT_TIMEOUT	0x0C	“Character Timeout Indication” interrupt on the DUART channel.
DUART_INT_TX	0x02	“Transmitter Holding Register Empty” interrupt on the DUART channel.
DUART_INT_MODEM	0x00	“MODEM Status” interrupt on the DUART channel.

Returned Value

DUART_SUCCESS	The interrupt(s) were disabled.
DUART_ERROR_CHANNEL_VALUE	Unknown channel specified.

Remarks Multiple interrupts can be disabled simultaneously. This is accomplished by ORing the desired interrupt levels together.

This function should only be called in Node B DSP application software.

VX8_DUARTEnableInterrupt

Function Enables one of the interrupts sources on channel 1 and/or channel 2 of the DUART.

Include File vx8.h

Syntax STATUS VX8_DUARTEnableInterrupt (UINT8 *channel*, UINT8 *int_level*);

Parameters

channel	0 = program both channels 1 = program channel 1 2 = program channel 2
int_level	Interrupt(s) to enable

Interrupt	Value	Description
DUART_INT_LINE	0x06	“Receiver Line Status” interrupt on the DUART channel.
DUART_INT_RX	0x04	“Receiver Data Available” interrupt on the DUART channel.
DUART_INT_TIMEOUT	0x0C	“Character Timeout Indication” interrupt on the DUART channel.
DUART_INT_TX	0x02	“Transmitter Holding Register Empty” interrupt on the DUART channel.
DUART_INT_MODEM	0x00	“MODEM Status” interrupt on the DUART channel.

Returned Value

DUART_SUCCESS	The interrupt(s) were enabled.
DUART_ERROR_CHANNEL_VALUE	Unknown channel specified.

Remarks Multiple interrupts can be enabled simultaneously. This is accomplished by ORing the desired interrupt levels together.

This function should only be called in Node B DSP application software.

VX8_DUARTInByte

Function	Reads an 8-bit value from the Receiver Buffer Register (RBR) of a DUART channel.	
Include File	vx8.h	
Syntax (macro)	UINT8 VX8_DUARTInByte (UINT8 *channel*);	
Parameters	*channel*	1 = read from channel 1 2 = read from channel 2
Returned Value	byte	8-bit value from the Receiver Buffer Register
Remarks	Ensure that your code does not read an empty Receiver Buffer Register. This function should only be called in Node B DSP application software.	

VX8_DUARTOutByte

Function Writes an 8-bit value to the Transmit Holding Register (THR) of a DUART channel.

Include File vx8.h

Syntax (macro) void VX8_DUARTOutByte (UINT8 *channel*, UINT8 *byte*);

Parameters

channel	1 = write to channel 1 2 = write to channel 2
byte	value to write to the Transmit Holding Register

Returned Value None

Remarks Ensure that your code does not write to the Transmit Holding Register when it’s full.

This function should only be called in Node B DSP application software.

VX8_DUARTSetBaudRate

Function Programs channel 1 and/or channel 2 of the DUART to operate at a given baud rate and data format.

Include File vx8.h

Syntax void VX8_DUARTSetBaudRate (UINT8 *channel*, UINT32 *div_value*, UINT32 *format*);

Parameters

channel	0 = program both channels 1 = program channel 1 2 = program channel 2
div_value	Baud rate divisor to program into channel(s). See **Remarks**
format	Data format for the channel(s)

Data Format Flag	Value	Description
DATA_5	0x00	Sets the number of data bits to five.
DATA_6	0x01	Sets the number of data bits to six.
DATA_7	0x02	Sets the number of data bits to seven.
DATA_8	0x03	Sets the number of data bits to eight.
STOP_BIT_1	0x00	Sets the number of stop bits to one.
STOP_BIT_2	0x04	If 5-bit data length, 1.5 stop bits are generated. For 6, 7, or 8-bit data lengths, 2 stop bits are generated.
NO_PARITY	0x00	No parity bit is generated.
ODD_PARITY	0x08	An odd parity bit is added to the serial data unit.
EVEN_PARITY	0x18	An even parity bit is added to the serial data unit.
MARK_PARITY	0x28	A “1” parity bit is added to the serial data unit.
SPACE_PARITY	0x38	A “0” parity bit is added to the serial data unit.
BREAK	0x40	Sets serial output to “0”.

Returned Value

DUART_ERROR_CHANNEL_VALUE	Unknown channel number.
VX8_SUCCESS	DUART baud rate(s) set.

Remarks The data format is specified by ORing the desired data format flags. For example, N81 can be specified by the following:

(DATA_8|STOP_BIT_1|NO_PARITY).

The DUART is clocked from the BAUDCLK of the SCV64 (32 MHz / 13 or 2.461538 MHz) and provides asynchronous communications up to 153.85 kbps. The highest standard data rate provided is 38.4 kbps. The 16C550 uses a standard divide by 16 on the master clock to generate its baud rate clock. A second divisor register is then used to set the input and output baud rate according to the following formula:

Baud Rate = 32 MHz / (13 * 16 * Divisor Register)

A divisor register value of 1 will give 153.85 kbps, a divisor register value of 4 will give 38.46 kbps (38.4 kbps with an error of 0.16%). A divisor of zero is not recommended.

This function should only be called in Node B DSP application software.

VX8_FastTransfer

Function Copies 32-bit data from a source address to a destination address using parallel load and store operations.

Include File vx8.h

Syntax void VX8_FastTransfer (PVUINT32 *dest*, PVUINT32 *src*, UINT32 *length*);

Parameters

dest	Volatile pointer to destination address
src	Volatile pointer to source address
length	Block length of transfer

Returned Value None

Remarks These transfers have no restrictions on the address space. You must, however, ensure that any required setup for the data transfer has been performed prior to calling this function, including a call to **VX8_Lock** to lock the shared bus to a specific target.

This function is intended for high speed data transfers between memory blocks on the VX8 Carrier Board (i.e. transferring results from Near Global SRAM to shared DRAM). VX8_FastTransfer can be used to transfer data to or from the VXIbus. Keep in mind that if the Global Shared Bus is active, the transfer may take a long time. The processor performing the VX8_FastTransfer will also starve other DSPs from accessing the Global Shared bus.

VX8_SCV64DMATransfer should be used instead of VX8_FastTransfer to perform large transfers from VX8 memories (preferably from DRAM) to the VXIbus.

You'll want to use VX8_FastTransfer instead of SCV64 DMA if, for example, low bandwidth data from a C4x's Near Global SRAM is needed by the host in the presence of high priority data from the HP Local Bus. The VX8_FastTransfer would yield the Global Shared Bus to the HP Local Bus read DMA when in progress.

VX8_HPCheckInterrupt

Function	Determines which source or event generated an interrupt from the HP Local Bus.
Include File	vx8.h
Syntax	UINT8 VX8_HPCheckInterrupt(void);
Parameters	None

Returned Values HP Local Bus interrupt sources

0x10 ⇒EOB

0x20 ⇒WAE

0x40 ⇒WAF

Remarks Upon receiving an IIOF2 interrupt from the HP Local Bus controller, the source of the interrupt is unknown. This function can be used to determine the source. However, this function does not clear the source. The three interrupt sources are EOB (end of block), WAE (write FIFO almost empty), and WAF (write FIFO almost full).

Note: A WAE or a WAF interrupt from the HP Local Bus only indicates a FIFO level transition to the WAE or WAF levels. Before acting on the interrupt, your code should verify the current FIFO level by checking the FIFO flag levels in the HSTATUS Register. The VX8 C4x Support Software has several functions (VX8_HPGet*Bit) to read the FIFO level bits in the HSTATUS Register.

Note: One or more of the HP Local Bus interrupts may occur simultaneously. This condition is indicated by the ORing of the returned flags.

VX8_HPClearInterrupt

Function Clears an HP Local Bus interrupt source or event.

Include File vx8.h

Syntax STATUS VX8_HPClearInterrupt (UINT8 *hp_int*);

Parameters

hp_int	HP Local Bus interrupt sources to be cleared
	0x10 ⇒EOB
	0x20 ⇒WAE
	0x40 ⇒WAF

Returned Values

HPBUS_SUCCESS	Interrupt(s) acknowledged.
HPBUS_ERROR_INT_NOT_SET	No interrupt source was set.

Remarks The three interrupt sources are EOB (end of block), WAE (write FIFO almost empty), and WAF (write FIFO almost full). This function will trigger the next DMA transfer if the data is ready.

Note: Multiple HP Local Bus interrupts can be acknowledged simultaneously. This is performed by ORing the *hp_int* values together.

VX8_HPDisableInterrupt

Function Disables an HP Local Bus interrupt source or event.

Include File vx8.h

Syntax (macro) STATUS VX8_HPDisableInterrupt(UINT8 *hp_ints*);

Parameters

hp_ints	HP Local Bus interrupt sources to be disabled
	0x10 ⇒EOB
	0x20 ⇒WAE
	0x40 ⇒WAF

Returned Values

VX8_SUCCESS	The interrupts were disabled.

Remarks The three interrupt sources are EOB (end of block), WAE (write FIFO almost empty), and WAF (write FIFO almost full).

Note: Multiple HP Local Bus interrupts can be disabled simultaneously. This is performed by ORing the *hp_ints* values together.

VX8_HPEnableInterrupt

Function Enables an HP Local Bus interrupt source or event.

Include File vx8.h

Syntax (macro) STATUS VX8_HPEnableInterrupt(UINT8 *hp_ints*);

Parameters

hp_ints	HP Local Bus interrupt sources to be enabled
	0x10 ⇒ EOB
	0x20 ⇒ WAE
	0x40 ⇒ WAF

Returned Values

VX8_SUCCESS	The interrupts were enabled.

Remarks The three interrupt sources are EOB (end of block), WAE (write FIFO almost empty), and WAF (write FIFO almost full).

Note: Multiple HP Local Bus interrupts can be enabled simultaneously. This is performed by ORing the *hp_ints* values together.

VX8_HPGetContBit

Function	Gets the state of the CONT line to the BALLISTIC HP Local Bus interface chip.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetContBit (void);
Parameters	None
Returned Values	State of the HP BALLISTIC CONT line indicated by bit D2 of the HCONTROL register.

VX8_HPGetDMABuffer

Function Gets the value of the DMA_BUFFER bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetDMABuffer (void);

Parameters None

Returned Values HP Local Bus Buffer state

- 0 = the next write cycle will be to buffer 0 on the target node
- 1 = the next write cycle will be to buffer 1 on the target node

Remarks Typically, this macro should be called by the HP Local Bus end of block interrupt service routine. If this macro is called while a DMA is in progress, it will indicate the current address buffer.

VX8_HPGetDMAEnable

Function Gets the value of the DMA_ON bit in the HCONTROL register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetDMAEnable (void);

Parameters None

Returned Values The read HP Local Bus DMA enable state

- 0 = Read DMA's are disabled.
- 1 = Read DMA's are enabled.

VX8_HPGetDMAIncrement

Function	Gets the value of the DMA_INCR bit in the HCONTROL register.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetDMAIncrement (void);
Parameters	None
Returned Values	The HP Local Bus DMA increment • 0 = Read DMA transfers are to a single address. • 1 = Read DMA transfers are to consecutive addresses.

VX8_HPGetFrameBit

Function	Gets the state of the FRAME bit in the HWRITEEOB register.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetFrameBit (void);
Parameters	None
Returned Values	State of the HP BALLISTIC FRAME line indicated by bit D0 of the HWRITEEOB register.

VX8_HPGetMode

Function	Gets the mode of the BALLISTIC HP Local Bus interface chip.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetMode (void);
Parameters	None
Returned Values	Value of the BALLISTIC MODE[3..0] bits. The BALLISTIC MODE bits correspond to bits D7-D4 in the HCONTROL register.
Remarks	See **VX8_HPSetMode** or the *VX8 Carrier Board Technical Reference Manual* for a description of the HP BALLISTIC IC modes.

VX8_HPGetPauseBit

Function	Indicates whether the BALLISTIC HP Local Bus interface chip is in a paused state.
Include File	vx8.h
Interface (macro)	UINT8 VX8_HPGetPauseBit (void);
Parameters	None
Returned Values	State of the HP BALLISTIC PAUSE line indicated by bit D3 in the HSTATUS register. • 0 = The BALLISTIC is not paused. • 1 = The BALLISTIC is paused.

VX8_HPGetRAEBit

Function Gets the value of the Read FIFO Almost Empty bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetRAEBit (void);

Parameters None

Returned Values State of the HP Local Bus Read FIFO Almost Empty (RAE) flag. RAE is indicated by bit D6 in the HSTATUS register.

- 0 = Read FIFO is below the almost empty flag level.
- 1 = Read FIFO is above the almost empty flag level.

VX8_HPGetRAFBit

Function Gets the value of the Read FIFO Almost Full bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetRAFBit (void);

Parameters None

Returned Values State of the HP Local Bus Read FIFO almost full (RAF) flag. RAF is indicated by bit D5 in the HSTATUS register.

- 0 = Read FIFO is above the almost full flag level.
- 1 = Read FIFO is below the almost full flag level.

VX8_HPGetReadDone

Function	Gets the value of the READDONE bit in the HREADEOB register.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetReadDone (void);
Parameters	None
Returned Values	State of the HP BALLISTIC READDONE line indicated by bit D0 in the HREADEOB register.

VX8_HPGetREBit

Function Gets the value of the Read FIFO Empty bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetREBit (void);

Parameters None

Returned Values State of the HP Local Bus Read FIFO empty (RE) flag. RE is indicated by bit D7 in the HSTATUS register.

- 0 = Read FIFO is empty.
- 1 = Read FIFO contains data.

VX8_HPGetRFBit

Function Gets the value of the Read FIFO Full bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetRFBit (void);

Parameters None

Returned Values State of the HP Local Bus Read FIFO full (RF) flag. RF is bit D4 in the HSTATUS register.

- 0 = Read FIFO is full.
- 1 = Read FIFO is not full.

VX8_HPGetStripBit

Function	Gets the state of the STRIP input to the BALLISTIC HP Local Bus interface chip.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetStripBit (void);
Parameters	None
Returned Values	State of the HP BALLISTIC STRIP line indicated by bit D3 in the HCONTROL register.

VX8_HPGetWAEBit

Function Gets the state of the Write FIFO Almost Empty bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetWAEBit (void);

Parameters None

Returned Values State of the HP Local Bus Write FIFO almost empty (HP_INT_WAE) flag. HP_INT_WAE is indicated by bit D10 in the HSTATUS register.

- 0 = Write FIFO is below the almost empty flag level.
- 1 = Write FIFO is above the almost empty flag level.

VX8_HPGetWAFBit

Function	Gets the state of the Write FIFO Almost Full bit in the HSTATUS register.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetWAFBit (void);
Parameters	None
Returned Values	State of the HP Local Bus Write FIFO almost full (HP_INT_WAF) flag. HP_INT_WAF is indicated by bit D9 in the HSTATUS register. • 0 = Write FIFO is above the almost full flag level. • 1 = Write FIFO is below the almost full flag level.

VX8_HPGetWEBit

Function	Gets the state of the Write FIFO Empty bit in the HSTATUS register.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetWEBit (void);
Parameters	None
Returned Values	State of the HP Local Bus Write FIFO empty (WE) flag. WE is indicated by bit D11 in the HSTATUS register. • 0 = Write FIFO is empty. • 1 = Write FIFO contains data.

VX8_HPGetWFBit

Function Gets the state of the Write FIFO Full bit in the HSTATUS register.

Include File vx8.h

Syntax (macro) UINT8 VX8_HPGetWFBit (void);

Parameters None

Returned Values State of the HP Local Bus Write FIFO Full (WF) flag. WF is indicated by bit D8 in the HSTATUS register.

- 0 = Write FIFO is full.
- 1 = Write FIFO is not full.

VX8_HPGetWriteDoneBit

Function	Gets the state of the WRITEDONE bit in the HWRITEEOB register.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPGetWriteDoneBit (void);
Parameters	None
Returned Values	State of the HP BALLISTIC WRITEDONE line. WRITEDONE is indicated by bit D1 in the HWRITEEOB register.

VX8_HPReadECL

Function	Reads the value of the ECL trigger line ECLTRG0.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPReadECL (void);
Parameters	None
Returned Values	State of the P2 ECLTRG0 line. ECLTRG0 is indicated by bit D1 in the ECLTRG register.

VX8_HPReadTTL

Function	Reads an 8-bit value from the TTL trigger lines (TTLTRG0 through TTLTRG7) on P2.
Include File	vx8.h
Syntax (macro)	UINT8 VX8_HPReadTTL (void);
Parameters	None
Returned Values	State of the 8-bit TTL lines. TTLTRG[7..0] correspond to bits D7-D0 in the TTLTRG register.

VX8_HPReset

Function Resets the HP Local Bus DMA controller, read/write FIFOs, and BALLISTIC HP Local Bus IC. This function also sets the flag levels of the read and write FIFOs.

Include File vx8.h

Syntax STATUS VX8_HPReset (UINT32 *wae*, UINT32 *waf*, UINT32 *rae*, UINT32 *raf*);

Parameters

wae	Level to be programmed into the write FIFOs almost empty flag.
waf	Level to be programmed into the write FIFOs almost full flag.
rae	Level to be programmed into the read FIFOs almost empty flag.
raf	Level to be programmed into the read FIFOs almost full flag.

Note: Flag level settings can range from 1 to 1020. Both almost empty and almost full flag levels are measured from empty.

Returned Values

HPBUS_SUCCESS	The HP Local Bus peripherals were reset and programmed successfully.
HPBUS_ERROR_FIFO_FLAG_VALUE	A FIFO flag level is out of range.
HPBUS_ERROR_FIFO_INIT	Error in programming the FIFO flag levels. The FIFOs are in an unknown state.

Remarks FIFO levels are set following a reset to the FIFO IC and cannot be changed until the next FIFO/HP BALLISTIC reset.

For proper operation of DMA transfers from the HP Local Bus, the following rule must be followed:

- RAF must be between 1 and 1020 (inclusive) **AND**
- RAF must be smaller than the block size by at least 4.

For example, the following is fine: RAF size of 1020 and block size of 1024. The following is not recommended: RAF size of 1020, block size of 1023.

VX8_HPRestart

Function	Restarts the BALLISTIC HP IC when in a paused state.
Include File	vx8.h
Syntax (macro)	STATUS VX8_HPRestart (void);
Parameters	None
Returned Values	VX8_SUCCESS The BALLISTIC HP IC was restarted.
Remarks	The HP BALLISTIC IC MODE bits are latched on the falling edge of restart. This means that the HP BALLISTIC IC MODE must be set before restart is indicated.

VX8_HPSetContBit

Function	Sets or clears the CONT line to the HP Local Bus BALLISTIC interface chip.	
Include File	vx8.h	
Syntax (macro)	STATUS VX8_HPSetContBit(UINT8 *state*);	
Parameters	*state*	State of the CONT bit in the HCONTROL register 0 = pause between modes 1 = do not pause between mode switches
Returned Values	VX8_SUCCESS	The CONT bit was written.

Remarks This macro allows the HP BALLISTIC interface chip to switch between modes in a continuous manner. The continuous mode overrides the default paused/restart handshake mode of operation. With the CONT bit set, the BALLISTIC IC will immediately transition out of its paused state.

Setting the CONT bit will restart the BALLISTIC IC from a paused state and the current HP BALLISTIC mode will be assumed.

The CONT bit is typically set after the HP Local Bus interface is reset (with **VX8_HPReset**) and the HP BALLISTIC mode is set (with **VX8_HPSetMode**).

VX8_HPSetDMAEnable

Function	Sets the value of the DMA_ON bit in the HCONTROL register, and initiates a read DMA transfer from the HP Local Bus FIFO to a C40's Near Global SRAM.	
Include File	vx8.h	
Syntax (macro)	STATUS VX8_HPSetDMAEnable (UINT8 *state*);	
Parameters	*state*	Value of the DMA_ON bit in the HCONTROL register 1 = enable DMA transfers 0 = disable DMA transfers
Returned Values	VX8_SUCCESS	An HP Local Bus Read DMA was initiated (VX8 is CONSUME'ing data).

VX8_HPSetDMAIncrement

Function	Sets the value of the DMA_INCR bit in the HCONTROL register.
Include File	vx8.h
Syntax (macro)	STATUS VX8_HPSetDMAIncrement (UINT8 *state*);
Parameters	*state* — 0 = no address incrementing (for DMA transfers to a Port) 1 = increment DMA address
Returned Values	VX8_SUCCESS — The DMA increment bit was written.
Remarks	This function allows the DMA transfers to write to a single address, or successive memory locations.

VX8_HPSetDMATarget

Function Writes the target of an HP Local Bus read DMA transfer to the HP DMA controller, and sets the buffer addresses of the target and the wait states of the buffer memory.

Include File vx8.h

Syntax STATUS VX8_HPSetDMATarget (PVUINT32 *buf_0*, PVUINT32 *buf_1*,
UINT8 *wait*, UINT32 *flags*);

Parameters

buf_0	Volatile pointer to global memory. Must be on a 1kword (0x400) boundary. This is an offset from the base of the Global Shared memory of the targets specified by HTARGET and **not** an absolute address.
buf_1	Volatile pointer to global memory. Must be on a 1kword (0x400) boundary. This is an offset from the base of the Global Shared memory of the targets specified by HTARGET and **not** an absolute address.
wait	Number of wait states for the DMA transfers. Valid values are 0 - 7. See the following note.
flags	Target modifier flags (listed on the next page)

Note: A 60 MHz MDC40Sxx TIM-40 module cannot operate at 0 wait states. Only the embedded C40 DSPs can operate at 0 wait states for 80 MBytes/sec operation. TIM-40 modules are limited to 1 wait state for 60 MBytes/sec operation.

Returned Values

HPBUS_SUCCESS	HP Local Bus DMA target was successfully programmed.
HPBUS_ERROR_WAIT_STATE_VALUE	Wait state value out of range
HPBUS_ERROR_BUF_ADDR_VALUE	Invalid destination address
HPBUS_ERROR_FLAG_VALUE	The transfer flags are invalid.

Remarks The mode of the DMA transfers are not set by this function and the DMA transfers are not engaged. See **VX8_HPSetMode().**

Target modifier flags are listed on the next page.

Target Modifier Flag	Description
HP_NODE_A	Node A is the target of the HP Local Bus read DMA transfer.
HP_NODE_B	Node B is the target of the HP Local Bus read DMA transfer.
HP_NODE_C	Node C is the target of the HP Local Bus read DMA transfer.
HP_NODE_D	Node D is the target of the HP Local Bus read DMA transfer.
HP_NODE_E	Node E is the target of the HP Local Bus read DMA transfer.
HP_NODE_F	Node F is the target of the HP Local Bus read DMA transfer.
HP_NODE_G	Node G is the target of the HP Local Bus read DMA transfer.
HP_NODE_H	Node H is the target of the HP Local Bus read DMA transfer.
HP_NODE_ALL	All nodes are the target of the HP Local Bus read DMA transfer.
HP_NODE_TIM	All TIM-40 nodes (C, D, E, F, G, H) are the target of the HP Local Bus read DMA transfer.
HP_NODE_CDGH	Only TIM-40 nodes C, D, G, H are the target of the HP Local Bus read DMA transfer.
HP_NODE_EFGH	Only TIM-40 nodes E, F, G, H are the target of the HP Local Bus read DMA transfer.
HP_NODE_CD	Only TIM-40 nodes C and D are the target of the HP Local Bus read DMA transfer.
HP_ADDR_MODE	Set HP Local Bus DMA to increment addresses.
HP_PORT_MODE	Set HP Local Bus DMA not to increment addresses (that is, writing to a Port.)

VX8_HPSetFrameBit

Function	Set or clears the FRAME bit in the HWRITEEOB register.	
Include File	vx8.h	
Syntax (macro)	STATUS VX8_HPSetFrameBit(UINT8 *state*);	
Parameters	*state*	State of the FRAME bit in the HWRITEEOB register 0 = do not append the current block with a frame bit 1 = append the current block with a frame bit
Returned Values	VX8_STATUS	The FRAME bit was written.
Remarks	A FRAME bit will be appended to the end of the current block being written to the Write FIFO.	

VX8_HPSetMode

Function Sets the operating mode of the HP Local Bus BALLISTIC interface chip, the value for the continue (CONT) bit, and the value for the STRIP bit.

Include File vx8.h

Syntax STATUS VX8_HPSetMode (UINT8 *mode*, UINT8 *cont*, UINT8 *strip*);

Parameters

mode	Sets the operating mode of the BALLISTIC IC. Valid inputs are 0 - 15. See the next page for a listing of modes
cont	0 = set the state of the BALLISTIC IC to be PAUSED, after the current reads or writes are completed (as indicated by READDONE and WRITEDONE)
	1 = the BALLISTIC IC will perform a transition out of a PAUSED state and will continuously operate in its current mode.
strip	0 = the FRAME bit will **not** be stripped off any data being piped through
	1 = the FRAME bit will be stripped off any data being piped through

Returned Values

HPBUS_SUCCESS	The operating mode of the BALLISTIC IC was successfully programmed.
HPBUS_ERROR_MODE_VALUE	The mode to be programmed is out of range.
HPBUS_ERROR_NOT_PAUSED	The BALLISTIC IC must be in a paused state to be programmed.

Remarks The BALLISTIC chip must be in a paused state before the mode, CONT, and STRIP bits can be written.

MODE[3..0] correspond to bits D7-D4 in the HCONTROL register. The HP BALLISTIC modes are given in the table on the next page.

If the CONT bit is set to 1, the HP BALLISTIC IC will immediately begin transmitting or receiving data, depending on which mode it’s currently in. That is, a call to **VX8_HPRestart** will not be required.

Please refer to the *VX8 Carrier Board Technical Reference Manual* for details on the operation of the HP Local Bus.

Table 16 BALLISTIC Modes

Mode	Comments
HP_MODE_PIPE_EOB	Pipe
HP_MODE_CONSUME	Consume
HP_MODE_EAVEDROP	Eavesdrop
HP_MODE_GENERATE	Generate
HP_MODE_TRANSFORM	Transform
HP_MODE_PC	See *BALLISTIC Data Sheet*
HP_MODE_PE	See *BALLISTIC Data Sheet*
HP_MODE_PG	Append
HP_MODE_PT	See *BALLISTIC Data Sheet*
HP_MODE_CP	Strip
HP_MODE_EP	See *BALLISTIC Data Sheet*
HP_MODE_GP	Insert
HP_MODE_TP	See *BALLISTIC Data Sheet*
HP_MODE_CPT	See *BALLISTIC Data Sheet*
HP_MODE_CPTP	See *BALLISTIC Data Sheet*

VX8_HPSetReadDone

Function	Sets or clears the READDONE line to the BALLISTIC HP Local Bus interface chip.
Include File	vx8.h
Syntax (macro)	STATUS VX8_HPSetReadDone (UINT8 *state*);
Parameters	*state* State of READDONE (bit D0) in the HREADEOB register.
Returned Values	VX8_SUCCESS The READDONE bit was written.
Remarks	The next end of block (HP_INT_EOB) received will trigger the end of transfer.

VX8_HPSetStripBit

Function	Sets or clears the STRIP input to the BALLISTIC HP Local Bus interface chip.
Include File	vx8.h
Syntax (macro)	STATUS VX8_HPSetStripBit(UINT8 *state*);
Parameters	*state* — State of STRIP (bit D3) in the HCONTROL register 0 = pipe the FRAME bit through 1 = strip off the FRAME bit
Returned Values	VX8_SUCCESS — The STRIP bit was written.
Remarks	The STRIP bit will force the FRAME bit to be stripped off any data being piped through. STRIP is latched by the BALLISTIC IC on the falling edge of RESTART. This means that STRIP will not take affect until the next RESTART. STRIP should only be set when the BALLISTIC IC is in a PAUSED state.

VX8_HPSetWriteBlockSize

Function	Writes the HP Local Bus outbound block size to the HBLOCK register.
Include File	vx8.h
Syntax (macro)	STATUS VX8_HPSetWriteBlockSize (UINT32 *block_size*);

Parameters		
	block_size	The block size of outgoing HPbus transfers in long words

Returned Values		
	VX8_SUCCESS	The block size was written.

VX8_HPSetWriteDoneBit

Function	Sets or clears the write done line to the BALLISTIC HP Local Bus interface chip.	
Include File	vx8.h	
Syntax (macro)	STATUS VX8_HPSetWriteDoneBit (UINT8 *state*);	
Parameters	*state*	State of WRITEDONE (bit D1) in the HWRITEEOB register.
Returned Values	VX8_SUCCESS	The WRITEDONE bit was written.
Remarks	The next end of block (HP_INT_EOB) output to the BALLISTIC IC will trigger the end of transfer.	

VX8_HPSuspendDMA

Function	Enables or disables the SUSPEND* line to the HP Local Bus DMA controller.
Include File	vx8.h
Syntax (macro)	void VX8_HPSuspendDMA (UINT32 *state*);
Parameters	*state* — 0 = de-assert the /SUSPEND line 1 = assert the /SUSPEND line
Returned Values	None
Remarks	This function can be called by Node A only. The functionality of the /SUSPEND line is to allow a processor to stop the HP Local Bus transfer. On the VX8, the /SUSPEND line gives Node A the ability to acquire the Global Shared Bus when HP Local Bus transfers are occurring. This is particularly useful for Node A to service time critical interrupts.

VX8_HPWriteECL

Function	Asserts or de-asserts the data bit for the ECL trigger line ECLTRG0.
Include File	vx8.h
Syntax (macro)	STATUS VX8_HPWriteECL (UINT8 *state*);
Parameters	*state* — 0 = de-assert the ECL out line 1 = assert the ECL out line
Returned Values	VX8_SUCCESS — The ECLTRG0 was enabled.
Remarks	A write to the ECLTRG register latches the data indicated by *state* to bit D0. The data is provided to P2 through Emitter Coupled Logic (ECL) drivers. The output from this latch is then re-synchronized to the ECLTRG1 clock signal, so if there is no clock present on ECLTRG1, then the value of ECLTRG0 will not be clocked out to the P2 connector.

VX8_HPWriteFIFO

Function Performs back-to-back write cycles to the write FIFO port on the HP Local Bus.

Include File vx8.h

Syntax STATUS VX8_HPWriteFIFO (PVUINT32 *src*, UINT32 *length*);

Parameters

src	Volatile pointer to the source address. Can be internal RAM, local SRAM, or near global SRAM.
length	Length of block to transfer. Maximum value is 1024.

Returns

VX8_SUCCESS	Transfer complete
VX8_ERROR_INVALID_ADDRRESS	Invalid address, addresses cannot be on the global bus. This includes DRAM, and the VXIbus.
VX8_ERROR_INVALID_BLOCK_SIZE	The block size is greater than 1024.

Remarks This routine is written in assembly language for high speed transfer to the HP Local Bus. To achieve maximum performance, the source of the data transfer should be in Internal or C40 Local Bus SRAM. If not, then a pipeline conflict plus the crossing of a page boundary on the global bus will limit the throughput.

VX8_HPWriteTTL

Function Writes an 8-bit value to the TTL trigger lines (TTLTRG0 through TTLTRG7 on P2).

Include File vx8.h

Syntax (macro) STATUS VX8_HPWriteTTL (UINT8 *byte*);

Parameters

byte	8-bit value to write to the TTL trigger lines.
	0x01 = Set TTLTRG0
	0x02 = Set TTLTRG1
	0x04 = Set TTLTRG2
	0x08 = Set TTLTRG3
	0x10 = Set TTLTRG4
	0x20 = Set TTLTRG5
	0x40 = Set TTLTRG6
	0x80 = Set TTLTRG7

Returned Values

VX8_SUCCESS	The TTL triggers were written.

Remarks TTLTRG* lines are open collector TTL lines used for intermodule communication. Any module may drive these lines and receive information on these lines. They are general purpose lines that may be used for trigger, handshake, clock, or logic state transmission.

Note: Multiple TTLTRG lines can be set or cleared simultaneously. This is performed by ORing the appropriate *byte* values together to assert or de-assert the respective TTLTRG lines.

VX8_Lock

Function Locks the Global Shared Bus for access by the calling C4x processor.

Include File vx8.h

Syntax STATUS VX8_Lock (PVUINT32 *dest*);

Parameters

dest	Volatile pointer to destination address. This pointer can reside in the calling C4x's internal, local, or Global SRAM.

Returned Values

VX8_SUCCESS	The Global Shared Bus was locked successfully.
VX8_ERROR_INVALID_ADDRESS	The destination address was not in the Global Shared Bus, the DRAM Shared Bus, nor the VXIbus address space.

Remarks This function internally unlocks the previous lock location and locks the new destination address.

Generally, the provided libraries handle bus locking transparently to the user. For more information, see sections *6.5.1* and *6.5.2*.

Lock addresses are confined to the Global Shared Bus (i.e. Locks to Near Global, Local, and internal are not allowed).

This function must occur with a corresponding **VX8_Unlock** call to release the Global Shared Bus.

While the Global Shared Bus is locked by a C4x, all other C4x DSPs will be held off until the bus is released.

The VX8 Carrier Board uses a read operation to lock the Global Shared Bus. When locking to a port, use a different address in the same lock space which is not offended by read operations.

VX8_Read

Function Reads a block of memory locations from the VXIbus, DRAM, or some other node's Near Global SRAM into the calling node's local SRAM, global SRAM or internal memory.

Include File vx8.h

Interface STATUS VX8_Read (PVUINT32 *dest*, PVUINT32 *src*, UINT32 *length*,
VX8_TRANSFER_TYPE *mode*);

Parameters

dest	Volatile pointer to destination address. Must be in the calling C4x's SRAM or internal RAM space.
src	Volatile pointer to source address. Can be SRAM, DRAM, or the VXIbus. VXIbus source addresses are byte-addressed - all others are 32-bit word addressed.
length	Block length of transfer.
mode	Transfer mode flag that determines the type of transfer from the source to the calling DSP's memory. Transfer Flags are listed on the next page.

Returned Values

VX8_SUCCESS	Memory block read successfully.
VX8_ERROR_INVALID_ADDRESS	Source or destination address is not compatible with the transfer mode.
VX8_ERROR_INVALID_PARAMETER	Invalid transfer mode flag.

Remarks Data widths less than 32-bit are right-justified in the destination address space. That is, D16 and D08E0 reads will place data in the destination space "unpacked". Therefore, for two consecutive D16 reads, the data will appear in the bottom 16 bits in two consecutive 32-bit memory locations.

This routine performs locking and unlocking of the global shared bus. The destination address must be local to the DSP (internal, local, or near global SRAM).

'C44 based TIM-40 module writes must remain within a 64 Mbyte page.

Transfer Flag	Description
VX8_D32	32-bit data transfer to on-board SRAM or DRAM
VXI_A32_D32_PRIV	32-bit supervisory data transfer to A32 space on the VXIbus
VXI_A32_D16_PRIV	16-bit supervisory data transfer to A32 space on the VXIbus
VXI_A32_D08_PRIV	8-bit supervisory data transfer to A32 space on the VXIbus
VXI_A32_D32_NONPRIV	32-bit user data transfer to A32 space on the VXIbus
VXI_A32_D16_NONPRIV	16-bit user data transfer to A32 space on the VXIbus
VXI_A32_D08_NONPRIV	8-bit user data transfer to A32 space on the VXIbus
VXI_A24_D32_PRIV	32-bit supervisory data transfer to A24 space on the VXIbus
VXI_A24_D16_PRIV	16-bit supervisory data transfer to A24 space on the VXIbus
VXI_A24_D08_PRIV	8-bit supervisory data transfer to A24 space on the VXIbus
VXI_A24_D32_NONPRIV	32-bit user data transfer to A24 space on the VXIbus
VXI_A24_D16_NONPRIV	16-bit user data transfer to A24 space on the VXIbus
VXI_A24_D08_NONPRIV	8-bit user data transfer to A24 space on the VXIbus
VXI_A16_D16_PRIV	16-bit supervisory data transfer to A16 space on the VXIbus
VXI_A16_D08_PRIV	8-bit supervisory data transfer to A16 space on the VXIbus
VXI_A16_D16_NONPRIV	16-bit user data transfer to A16 space on the VXIbus
VXI_A16_D08_NONPRIV	8-bit user data transfer to A16 space on the VXIbus

VX8_ReadAsyncReg

Function Reads a single memory location from an asynchronous register on the VX8 board.

Include File vx8.h

Syntax UINT32 VX8_ReadAsyncReg (`PVUINT32` *src*, UINT32 *bitmask*);

Parameters

src	Volatile pointer to source address. Can be in the calling C4x's local or global address space.
bitmask	Mask used for and-ing with the result. A mask of 0x0000 0001 should be used to read bit D0.

Returned Values

result	Contents of the source address. Accesses to the VXIbus address space return a value of zero.

Remarks Several registers (VSTATUS, A16 offset, A16 Control) could be modified by another source at the same instant that a DSP reads them. As a result, they must be read twice with the same result to ensure that the value read is valid. 8-bit registers should use a mask of 0x0000 00FF, as the upper data bits are generally not driven by the hardware. If the upper data bits are included in the mask, the software may hang trying to read the same result twice.

This routine performs locking and unlocking of the global shared bus. Unnecessary operations will be performed if the register in question is local to the DSP (internal, local, or near global).

VX8_ReadBit

Function Reads the value of a particular bit from a memory location on the global bus.

Include File vx8.h

Syntax (macro) UINT32 VX8_ReadBit(PVUINT32 *dest*, UINT32 *mask*);

Parameters

dest	Volatile pointer to a memory location on the global bus.
Mask	Mask used for and-ing with the result. A mask of 0x0000 0001 should be used to read bit D0.

Returned Values

result	Contents of the source address. Accesses to the VXIbus address space return a value of zero.

Remarks This function can be used to confirm the state of more than one bit.

This routine performs locking and unlocking of the global shared bus. Unnecessary operations will be performed if the register in question is local to the DSP (internal, local, or near global).

VX8_ReadReg

Function	Reads a single memory location from the VX8 board peripherals, from other DSP's Near Global SRAM, or from DRAM.	
Include File	vx8.h	
Syntax	UINT32 VX8_ReadReg (PVUINT32 *src*);	
Parameters	*src*	Volatile pointer to source address. The pointer can be stored in the calling C4x's local, global, internal, and VXIbus address space.
Returned Values	result	Contents of the source address.
Remarks	This routine performs locking and unlocking of the global shared bus. Unnecessary operations will be performed if the register in question is local to the DSP (internal, local, or near global memories).	

VX8_SCV64AckInterrupt

Function Generates the correct acknowledge cycle for the SCV64, determines the interrupt source, and returns an interrupt vector, if valid. This function will return the value read from the LMFIFO if the location monitor caused the interrupt.

Include File vx8.h

Syntax

```
STATUS VX8_SCV64AckInterrupt(   PVUINT32 int_value,
                                PVUINT32 vector,
                                PUINT32 lm_val );
```

Parameters

int_value	Volatile pointer to the interrupt source number. 0 = invalid interrupt number 1-7 = VXIbus interrupt 8 = local timer interrupt 9 = local location monitor interrupt 10 = local DMA complete (DONE) interrupt 11 = local DMA Configuration Error (CERR) interrupt 12 = local DMA Local Bus Error (DLBER) interrupt 13 = local Local Bus Error (LBERR) interrupt 14 = local VXI Bus Error (VBERR) interrupt
vector	Volatile pointer to 8-bit vector number. This vector is generated by VXIbus interrupts only and is invalid (set to zero) for all other interrupt sources.
lm_val	Pointer to a 32-bit location. If the interrupt was caused by the Location Monitor, the value read from the LMFIFO will be returned.

Returned Values

SCV64_SUCCESS	Interrupt source acknowledged successfully.
SCV64_ERROR_LOCAL_ACCESS	An access error has occurred.
SCV64_ERROR_BAD_IACK	An access error occurred while running the IACK cycle.
SCV64_ERROR_BUS_INT_LEVEL_VALUE	A bad local interrupt occurred.

Remarks Upon receiving an IIOF0 interrupt from the SCV64 the source of the interrupt is unknown since all of the SCV64 interrupts share this one line. IIOF0 should be configured as a level triggered interrupt on the DSPs which are handling the interrupts from the SCV64. This function will return the source of the interrupt.

If a VXIbus interrupt is indicated, an IACK cycle will be performed. This function performs all of the necessary cycles (reads and writes) and associated registers to handle the VXIbus interrupts. Your application must enable the desired SCV64 interrupts that the VX8 is to handle. Interrupt enabling and disabling are taken care of by **VX8_SCV64EnableInterrupt** and **VX8_SCV64DisableInterrupt.**

For VXIbus interrupts, in addition to enabling the interrupts through the **VX8_SCV64EnableInterrupt** function, your VXIbus system must be configured so that only one device is set up to handle any one VXIbus interrupt line. Ensure that no two VXIbus devices are handling the same VXIbus interrupt level. Refer to the documentation supplied with the other VXIbus devices in your system.

The local timer interrupt is generated by a /KBERR condition on the SCV64. The expiring of the SCV64 local bus timer means that a local transaction failed to complete in the allotted time.

The Location Monitor interrupt (/LMINT) is asserted when the SCV64 Location Monitor FIFO (LMFIFO) contains data. The LMFIFO is a 31 long word deep message queue accessible from the VXIbus or from the DSPs by writing to the top longword of the VX8s A32 slave image. If written by a DSP, the LMFIFO write cycle is internally handled by the SCV64 and does not actually go out onto the VXIbus. Writing to a full LFIFO will cause a bus error. If the interrupt was a Location Monitor Interrupt, the LMFIFO will be read and returned via the *lm_val* parameter.

For additional information regarding local interrupt sources see *Section 2.3.2* of the *SCV64 User Manual - VMEbus Interface Components Manual*.

VX8_SCV64DisableInterrupt

Function Disables a SCV64 interrupt.

Include File vx8.h

Syntax STATUS VX8_SCV64DisableInterrupt(UINT8 *int_level*);

Parameters

int_level	VXIbus or local SCV64 interrupt number to disable 0 - 7 = VXIbus interrupt from level 0 to 7 8 = local timer interrupt 9 = local location monitor interrupt

Returned Values

SCV64_SUCCESS	The interrupt was disabled on the SCV64.
SCV64_ERROR_INT_LEVEL_VALUE	The interrupt number was out of range.

Remarks Interrupts sources can be either local to the SCV64 or from the VXIbus. VXIbus interrupts can be any interrupt level between 0 and 7. Local SCV64 interrupts include the location monitor interrupt and the timer interrupt.

The local timer interrupt should not be disabled. Disabling the local timer interrupt will disable the VX8 hardware’s ability to notify your software of a local bus timeout.

VX8_SCV64DisableInterrupt does not allow the disabling of the SCV64 local interrupt which handles DMA transfers and errors (LIRQ5). This is because the SCV64 VME/VXIbus error is indicated by the same interrupt conditions as the SCV64 DMA done and errors. Applications should always be made aware of exceptions and fatal errors.

See **VX8_SCV64AckInterrupt** for more information about the various interrupts.

VX8_SCV64DMATransfer

Function Initiates DMA transfers between the VXIbus and Shared DRAM/ Near Global SRAM.

Include File vx8.h

Syntax STATUS VX8_SCV64DMATransfer(PVUINT32 *local_addr*,
PVUINT32 *vxi_addr*,
UINT32 *direction*,
UINT32 *length*,
DMA_TRANSFER_TYPE *transfer_mode*);

Parameters

local_addr	Volatile pointer to local DRAM or Global SRAM address
vxi_addr	Volatile pointer to byte-addressed VXIbus physical address
direction	0 = DMA_READ for data transferred from VXIbus to Near Global SRAM or shared DRAM 1 = DMA_WRITE for data transferred from Near Global SRAM or shared DRAM to VXIbus
length	Block length or transfer
transfer_mode	Transfer Flags listed below

Transfer Flag	Description
A32_D64_NONPRIV	32-bit supervisory data transfer to A32 space on the VXIbus
A32_D32_PRIV	32-bit supervisory data transfer to A32 space on the VXIbus
A32_D16_PRIV	16-bit supervisory data transfer to A32 space on the VXIbus
A32_D32_NONPRIV	32-bit user data transfer to A32 space on the VXIbus
A32_D16_NONPRIV	16-bit user data transfer to A32 space on the VXIbus
A24_D32_PRIV	32-bit supervisory data transfer to A24 space on the VXIbus
A24_D16_PRIV	16-bit supervisory data transfer to A24 space on the VXIbus
A24_D32_NONPRIV	32-bit user data transfer to A24 space on the VXIbus
A24_D16_NONPRIV	16-bit user data transfer to A24 space on the VXIbus

Returned Values

SCV64_SUCCESS	DMA setup and initiated
SCV64_ERROR_DMA_ACTIVE	VXIbus DMA transfer currently in progress.
SCV64_ERROR_UNALIGNED_TRANSFER	Invalid VXIbus address. Must be word or longword aligned.

Remarks

The SCV64 DMA controller is configured for DMA transfers between the VXIbus and DRAM/SRAM. DMAs can only be word, longword transfers, or D64 transfers.

When determining the SCV64 DMA source/destination, keep in mind that the SCV64 IC views the VX8 as being byte addressable. That is, the SCV64 DMA address registers are in terms of bytes, not long words. Both the *local_addr* and *vxi_addr* parameters to VX8_SCVDMATransfer are byte addresses, as the function does not modify the parameters before writing them to the SCV64 DMA registers.

D64 DMA transfers will only work when communicating with other D64 capable devices. D64 transfer addresses must be double long word aligned. The D64 DMA length is expressed in long words like the D32 modes. However, the length must be a multiple of double long words. D2, D1, and D0 of the DMA transfer count are 0 for D64 (MBLT) DMA transfers.

The SCV64 counter is setup by the Node A Kernel to be 20 bits. The SCV64 counter can be reduced to a 12-bit counter by clearing the DTCISIZ (bit 31 in the SCV64 Mode Control Register). With DTCISIZ set, the maximum length of a SCV64 DMA transfer in a D16 transfer mode is 2Mb (each count represents a word.) In a similar manner, the maximum length of a transfer in a D32 or a MBLT D64 transfer mode is 4Mb.

SCV64 DMA to C4x Near Global memories should be performed with caution as you can easily monopolize the Global Shared Bus and starve C4x and HP Local Bus accesses.

VX8_SCV64EnableInterrupt

Function Enables a SCV64 interrupt.

Include File vx8.h

Syntax STATUS VX8_SCV64EnableInterrupt(UINT8 *int_level*);

Parameters

int_level	VXIbus / Local SCV64 interrupt number to enable.
	0-7 = VXIbus interrupt from level 0 to 7
	8 = local timer interrupt
	9 = local location monitor interrupt

Returned Values

SCV64_SUCCESS	The interrupt was enabled on the SCV64.
SCV64_ERROR_INT_LEVEL_VALUE	The interrupt number was out of range.

Remarks Interrupts sources can be either local to the SCV64 or from the VXIbus. VXIbus interrupts can be any interrupt level between 0 and 7. Local SCV64 interrupts include the location monitor interrupt, or the timer interrupt.

The SCV64 local interrupt to handle DMA transfers and errors (LIRQ5) is enabled in the Node A boot kernel. Their is no option to enable or disable the SCV64 local interrupt which handles DMA transfers and errors. This is because the SCV64 VME/VXIbus error is indicated by the same interrupt conditions as the SCV64 DMA done and errors. Applications should always be made aware of exceptions and fatal errors.

In addition to enabling VXIbus interrupts through this function, your VXIbus system must be configured so that only one device is set up to handle any one VXIbus interrupt line. Ensure that no two VXIbus devices are handling the same VXIbus interrupt level. Refer to the documentation supplied with the other VXIbus devices in your system.

See **VX8_SCV64AckInterrupt** for more information about the various interrupts.

VX8_SCV64GenerateInterrupt

Function Generates a VXIbus interrupt at a given interrupt level and vector.

Include File vx8.h

Syntax STATUS VX8_SCV64GenerateInterrupt(UINT8 *int_level*, UINT8 *vector*);

Parameters

int_level	0 - 7 = interrupt level to generate on the VXIbus
vector	8-bit vector value to be placed on VXIbus

Returned Values

SCV64_SUCCESS	Interrupt successfully generated on VXIbus.
SCV64_ERROR_INT_ACTIVE	The interrupt level is already active.
SCV64_ERROR_INT_LEVEL_VALUE	The interrupt level was out of range.

VX8_SCV64SetSysFail

Function Asserts or de-asserts the SYSFAIL* line on the VXIbus from the SCV64.

Include File vx8.h

Syntax void VX8_SCV64SetSysFail (UINT8 *state*);

Parameters

state	0 = de-assert the SYSFAIL* line
	1 = assert the SYSFAIL* line

Returned Values None

Remarks The state of the SYSFAIL* line (visible by other devices on the VXIbus backplane) is dependent upon the state of the SYSFAIL_INHIBIT bit in the A16 Control register. If the SYSFAIL_INHIBIT bit is set, toggling the SYSFAIL bit on the SCV64 has no affect on the state of the SYSFAIL* line on the VXIbus.

VX8_SetUserLED

Function Sets or clears the state of the front panel user LED.

Include File vx8.h

Syntax void VX8_SetUserLED (UINT8 *state*);

Parameters

state	0 = turn the user LED off
	1 = turn the user LED on

Returned Values None

Remarks The front panel user LED is accessible from all C4x Node sites.

VX8_SCV64SetVXIBusReqRel

Function Sets the request and release parameters for the VXIbus.

Include File vx8.h

Syntax

```
STATUS VX8_SCV64SetVXIBusReqRel(UINT32 request_mode,
                                UINT32 bus_level,
                                UINT32 bus_clr,
                                UINT32 release_mode,
                                UINT32 timeout_enable,
                                UINT32 timeout);
```

Parameters

request_mode	0 = fair mode nonzero = demand mode
bus_level	Valid bus request levels are 0-3 (0 is the highest priority)
bus_clr	0 = ignore BCLR nonzero = release bus if BCLR is active
release_mode	0 = release on request nonzero = release when done
timeout_enable	0 = disable nonzero = enable
timeout	0 = 0 µs 1 = 2 µs 2 = 4 µs 3 = 8 µs

Returned Values

SCV64_SUCCESS	The release/request parameters were successfully initialized.
SCV64_ERROR_BUS_LEVEL_VALUE	The bus request level was out of range.
SCV64_ERROR_BUS_TIMEOUT_VALUE	The ownership timeout value was out of range.

Remarks The default SCV64 bus request settings set by the Node A kernel are given in the following table:

request_mode	0 = fair mode
bus_level	bus request level = 3 (highest level)
bus_clr	nonzero = release bus if BCLR is active
release_mode	nonzero = release when done
timeout_enable	nonzero = enable
timeout	3 = 8 µs

Note that changing SCV64 modes can cause inbound slave cycles to be corrupted or cause bus errors. If your application requires changing the SCV64 setup from the default configuration, only configure the SCV64 when there is no chance of A32 slave accesses to the board.

A description of VXIbus request and release modes can be found in *Section 2.2.*2 and *2.2.3* of the *SCV64 User Manual -VMEbus Interface Components Manual*.

VX8_UnLock

Function Unlocks the current Global Shared Bus and restores the previously locked context.

Include File vx8.h

Syntax STATUS VX8_UnLock (void);

Parameters None

Returned Values

VX8_SUCCESS	The global bus was unlocked successfully. The unlock target is taken from the lock stack.
VX8_ERROR_UNLOCK_MISMATCH	A call to **VX8_UnLock** was made without a successful matching call to **VX8_Lock**.

Remarks The current lock location is unlocked and the previous destination address is locked. See *Section 6.2.4 The Global Shared Bus Lock Stack* for reference.

This function should occur with a corresponding **VX8_Lock** call to properly maintain the lock stack. Keep in mind that while a DSP is locking to the Global Shared Bus, no other DSP can gain access to the bus.

VX8_Write

Function Writes a block of memory locations from the calling node's local SRAM, global SRAM or internal memory to the VXIbus, DRAM, or other node's SRAM.

Include File vx8.h

Syntax STATUS VX8_Write (PVUINT32 *dest*, PVUINT32 *src*, UINT32 *len*,
VX8_TRANSFER_TYPE *mode*);

Parameters

dest	Volatile pointer to destination address. Can be SRAM, DRAM, or the VXIbus. VXIbus source addresses are byte-addressed.
src	Volatile pointer to source address. Must be in the calling C4x's SRAM or internal RAM space.
len	Block length of transfer. VXIbus transfers are byte addressed - all others are 32 bit word addressed.
mode	Transfer flags listed on the next page of this manual.

Returned Values

VX8_SUCCESS	Memory block written successfully.
VX8_ERROR_INVALID_ADDRESS	Source or destination address is not compatible with the transfer mode.
VX8_ERROR_INVALID_PARAMETER	Invalid transfer mode flag.

Remarks Data widths less than 32-bit are assumed to be right-justified in the source address space. That is, D16 and D08E0 writes assume data to be in the source space "unpacked". So, for two contiguous D16 writes, the data should appear in the bottom 16 bits in two consecutive 32-bit memory locations visible from the DSP.

This routine performs locking and unlocking of the global shared bus. The source address must be local to the DSP (internal, local, or near global SRAM).

'C44 based TIM-40 module writes must remain within a 64 Mbyte page.

Transfer Flag	Description
VX8_D32	32-bit data transfer to on-board SRAM or DRAM.
VXI_A32_D32_PRIV	32-bit supervisory data transfer to A32 space on the VXIbus.
VXI_A32_D16_PRIV	16-bit supervisory data transfer to A32 space on the VXIbus.
VXI_A32_D08_PRIV	8-bit supervisory data transfer to A32 space on the VXIbus.
VXI_A32_D32_NONPRIV	32-bit user data transfer to A32 space on the VXIbus.
VXI_A32_D16_NONPRIV	16-bit user data transfer to A32 space on the VXIbus.
VXI_A32_D08_NONPRIV	8-bit user data transfer to A32 space on the VXIbus.
VXI_A24_D32_PRIV	32-bit supervisory data transfer to A24 space on the VXIbus.
VXI_A24_D16_PRIV	16-bit supervisory data transfer to A24 space on the VXIbus.
VXI_A24_D08_PRIV	8-bit supervisory data transfer to A24 space on the VXIbus.
VXI_A24_D32_NONPRIV	32-bit user data transfer to A24 space on the VXIbus.
VXI_A24_D16_NONPRIV	16-bit user data transfer to A24 space on the VXIbus.
VXI_A24_D08_NONPRIV	8-bit user data transfer to A24 space on the VXIbus.
VXI_A16_D16_PRIV	16-bit supervisory data transfer to A16 space on the VXIbus.
VXI_A16_D08_PRIV	8-bit supervisory data transfer to A16 space on the VXIbus.
VXI_A16_D16_NONPRIV	16-bit user data transfer to A16 space on the VXIbus.
VXI_A16_D08_NONPRIV	8-bit user data transfer to A16 space on the VXIbus.

VX8_WriteBit

Function Sets or clears a single bit in a longword memory location on a VX8 board peripherals, SRAM, or DRAM.

Include File vx8.h

Syntax STATUS VX8_WriteBit (PVUINT32 *dest*, UINT32 *mask*, UINT8 *state*);

Parameters

dest	Volatile pointer to destination address. Can be in the calling C4x's local or global address space.
Mask	Register mask for bits to be set or cleared. A mask of 0x0000 0003 should be used to set bits D0 and D1.
State	Value to set the masked bits. (0 = clear, 1 = set)

Returned Values

VX8_SUCCESS	The value was written successfully.

Remarks This routine performs locking and unlocking of the global shared bus. Unnecessary operations will be performed if the register in question is local to the DSP (internal, local, or near global).

VX8_WriteReg

Function Write a single value to a VX8 board peripheral, SRAM, or DRAM.

Include File vx8.h

Syntax void VX8_WriteReg (PVUINT32 *dest*, UINT32 *value*);

Parameters

dest	Volatile pointer to destination address. Can be in the calling C4x's local or global address space.
Value	Value to write to the destination address.

Returned Values void

Remarks This routine performs locking and unlocking of the Global Shared Bus. Unnecessary lock and unlock operations will be performed if the register in question is local to the DSP (internal, local, or near global).

When accessing a processor's Near Global, Local, or internal memories, direct register access is recommended.

8 VX8 Host Software System Description

8.1. Introduction

The VX8 Instrument Driver provides the host component of your VX8 application with functions to perform system initialization, DSP code download, control, and data transfer.

This instrument driver is built on top of either SICL or VISA I/O libraries which provide the lower level routines used to access the backplane. The VX8 instrument driver provides function calls required to interface to the VX8 Carrier Board and satisfies VXIpnp requirements in the VISA version of the instrument driver software.

The host accesses a VX8 Carrier Board through the VXIbus A16 and A32 address spaces. When accessing the VX8 directly using I/O library routines, care must be taken when performing transfers other than D32 to VX8 memory. Specifically, non-D32 transfers to C4x Near Global Memory requires byte swapping and manipulation due to:

- the long word addressing of the C4x;
- endian differences between the host and VX8; and
- the byte laning performed on the VXIbus.

Wherever possible, use D32 transfers.

8.2. Host Software Development

VX8 applications require both host and DSP software development. The VX8 Instrument Driver will allow you to easily configure your VX8 system and get your DSP software loaded and running on the system.

The host software development environment may differ depending on your host computer, operating system, I/O library, and compiler. Refer to your SICL or VISA documentation that has been supplied with your VXIbus controller/interface card for information on compiling and linking with the I/O libraries.

The VX8 Instrument Driver is written in ANSI C and source code is provided allowing you to expand its functionality or to further optimize the provided functions. The only component of the VX8 Instrument Driver that is not supplied as source code is the System Definition Module (described in *Section 9.4.2*).

9 VX8 SICL/VISA Instrument Driver

The instrument driver provides host applications with a software interface to basic routines for configuring, controlling, and communicating with the VX8. Two versions of the instrument driver are supplied, one for SICL I/O libraries and one for VISA I/O libraries. Both versions are written in standard ANSI C, both versions support multiple VX8s in a system, and both versions allow multiple sessions on the host to open communications to any single VX8.

9.1. SICL Instrument Driver

The following diagram illustrates the SICL Instrument Driver software hierarchy.

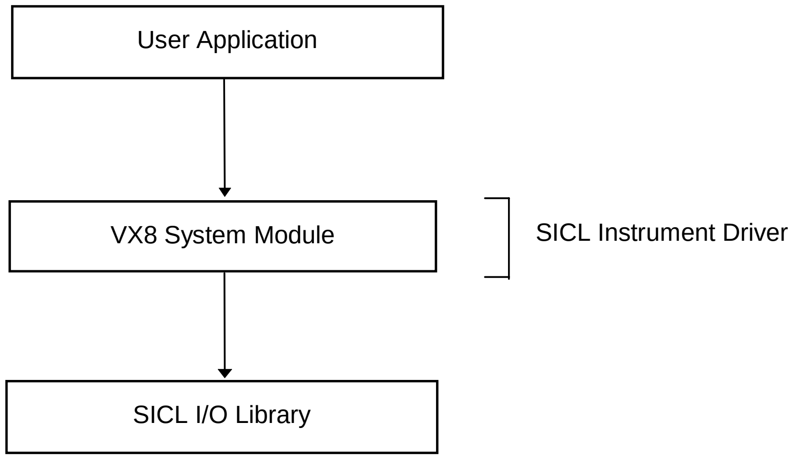


Figure 9 SICL Instrument Driver Program Flow

In certain instances, you may wish to bypass the instrument driver and call the I/O library functions directly, for example to optimize data transfer rates to the host. Refer to your SICL documentation if you're going to access the I/O library functions directly.

Caution: Take extreme care when directly calling I/O library routines as your calls can easily affect components outside of the VX8 System.

See *Table 18 System Module Functions* for a listing of the System Module functions.

If you are building code in the HPUX SICL environment, the compiler environment on the host should be set in the following manner.

Additional INCLUDE Directories	/opt/hpux/include /opt/hpux/ssVX8/host/include
Additional Library Paths	/opt/hpux/lib

9.2. VISA Instrument Driver

Note: The VISA Transition Library (VTL) is not supported because it lacks several 32-bit I/O functions required by the VX8.

Although a VXIpnp (VXI Plug and Play) module has been supplied in order to meet VXIpnp compliance, the VXIpnp instrument driver specification does not support all of the functionality required by the VX8. C4x considerations dictate that multiple VX8s be treated as a single system, which the VXIpnp module specification does not accommodate. The effect is that your application must make calls to the VX8 System Module level of the instrument driver.

If you want your VX8 application to access the VX8 system using a VXIpnp compliant instrument driver interface, your application should call the VXIpnp Module functions to open, close and check error messages. See the following *VXIpnp Module* section for more details. All other calls should be made to System Module functions or directly to the I/O library functions.

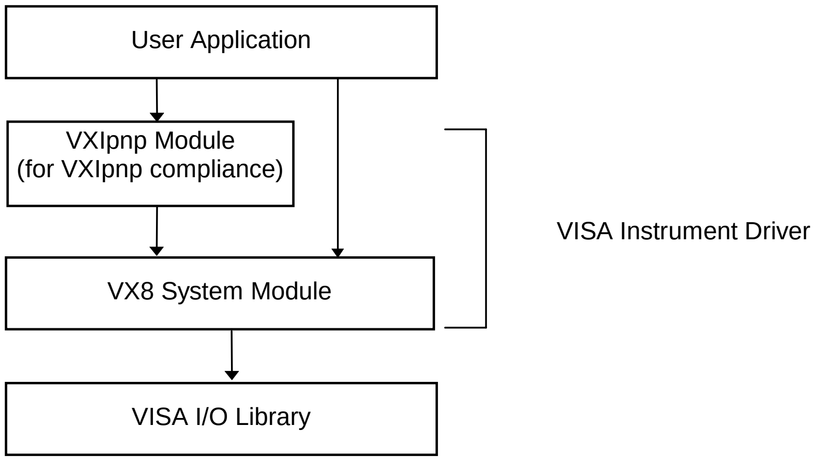


Figure 10 VISA Instrument Driver Program Flow

In certain instances, you may wish to bypass the instrument driver and call the I/O library functions directly, for example to optimize data transfer rates to the host. Refer to your VISA documentation, if you're going to access the I/O library functions directly.

Caution: Take extreme care when directly calling I/O library routines as your calls can easily affect components outside of the VX8 System.

See *Table 17 VXIpnp Module Functions* for a list of the VXIpnp Module functions and *Table 18 System Module Functions* for a list of the System Module functions.

The driver was compiled as a Win32 DLL under Microsoft Visual C version 5.0. If you are building VISA code under Windows 95 or Windows NT, the compiler environment on the host should be set in the following manner.

Additional INCLUDE Directories	Windows NT	[VXIPNPPATH]\WinNT\include [VXIPNPPATH]\WinNT\ssVX8\include
	Windows 95	[VXIPNPPATH]\Win95\include [VXIPNPPATH]\Win95\ssVX8\include
Additional Preprocessor Definitions	Windows NT Windows 95	_VISA
Additional Library Paths	Windows NT	[VXIPNPPATH]\WinNT\lib\msc
	Windows 95	[VXIPNPPATH]\Win95\lib\msc

9.3. VXIpnp Module

To meet VXIpnp (VXI Plug and Play) compliance, the VXIpnp Module has been included as part of the VX8 instrument driver for use with the VISA I/O libraries. This module contains the function calls required by the VXIpnp instrument driver specifications.

Note: The VXIpnp module is only applicable to systems using the **VISA** I/O library (that is, not with SICL).

The VXIpnp instrument driver specification stipulates that all compliant instrument drivers supplied for a VXIbus instrument must possess a minimum set of function calls which will provide a means to initialize, initiate, and terminate communications with the VXIbus device.

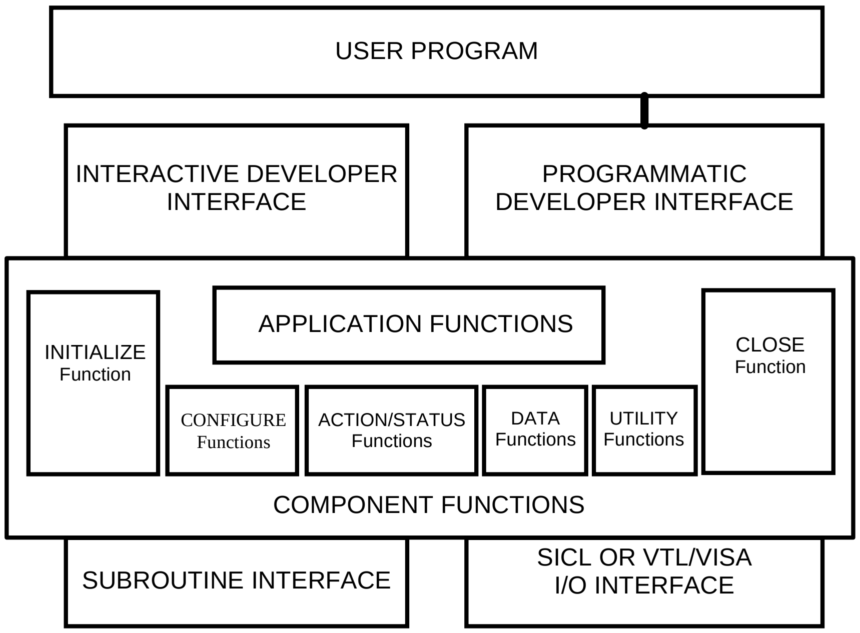


Figure 11 VXIpnp Instrument Driver Internal Design Model

Of the eight supplied VXIpnp functions, only the **ssVX8_init**, **ssVX8_open**, **ssVX8_revision_query**, **ssVX8_error_message**, and **ssVX8_close** are functional. To meet VXIpnp compliance, these functions should be used to open, close and check error messages (see *VXIpnp Module Usage*). The remaining functions are stubbed and return a NOT SUPPORTED warning when called.

Table 17 VXIpnp Module Functions

Function Name	Function Description
ssVX8_close	Closes the VX8 system referenced by the user-defined attribute of the inputted device session. The inputted device session is then closed.
ssVX8_error_message	Returns a text explanation for a given error code.
ssVX8_error_query	Returns VI_WARN_NSUP_ERROR_QUERY.
ssVX8_init	Opens communications with a VX8 device. This function returns an I/O library handle to a device. In the VX8 context, this function merely exists to satisfy VXIpnp requirements. A call to ssVX8_open is required following a call to ssVX8_init to open and initialize the VX8 system.
ssVX8_open	This function calls ssVX8_SystemOpen to open and initialize a system of VX8 devices. The VX8 system handle returned by ssVX8_SystemOpen is stored in the user-defined attribute of the session opened by the preceding call to ssVX8_init.
ssVX8_reset	This function will return VI_WARN_NSUP_RESET.
ssVX8_revision_query	This function will return the firmware and driver revision.
ssVX8_self_test	This function will return VI_WARN_NSUP_SELF_TEST.

VXIpnp Module Usage

The five operational functions of the VXIpnp Module are:

- ssVX8_init
- ssVx8_open
- ssVX8_error_message
- ssVX8_revision_query
- ssVX8_close

When using the VXIpnp module, you must first call the required **ssVX8_init** function to open communications with a VX8 device, then you'll immediately call a second function, **ssVX8_open.** The **ssVX8_open** function calls the System module functions of the VX8 instrument driver, **ssVX8_SystemOpen**, which has parameters that the **ssVX8_init** function does not allow. The **ssVX8_open** function then stores the system resource handle generated by **ssVX8_SystemOpen** in the user-defined attribute of the board handle obtained from the Resource Manager (RM) from the preceding **ssVX8_init** call.

The **ssVX8_close** function extracts the system handle from the user attribute and calls **ssVX8_SystemClose** (a System module function) to deallocate the resources taken by the prior **ssVX8_SystemOpen** call. *Figure 12 Program Flow VXIpnp vs. SICL* illustrates the program flow.

The third operational function in the VXIpnp module is **ssVX8_error_message.** This function returns a text string describing all error return values generated by the VX8 instrument driver software.

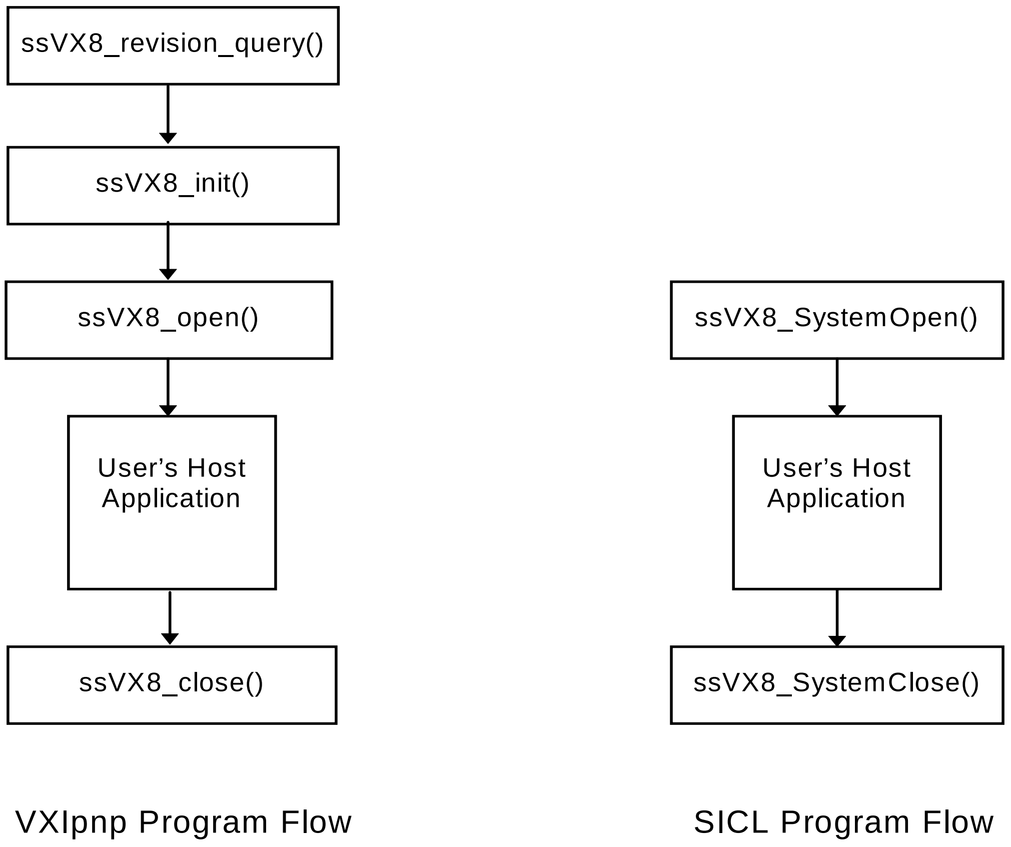


Figure 12 Program Flow VXIpnp vs. SICL

Note: The **ssVX8_init** function should only be called once for an entire system of VX8 devices.

Note: You should not use the handle returned by the **ssVX8_init** function to directly access the device. You must subsequently call **ssVX8_open** to initialize the VX8 system, and then use the I/O routines in the System module to communicate with the VX8 devices.

9.4. System Module

The System module represents the top level routines in the VX8 Instrument Driver which applications will invoke to perform VX8 system initialization, control, loading, and communications.

When interconnected by the front panel COMM ports and JTAG connectors, VX8s form a multi-board network of DSPs and TIM-40 modules, or a “VX8 System”. The Instrument Driver was written to communicate to VX8s as a System rather than on an individual board by board basis. Viewing VX8s as a System is required for the following two reasons:

- The directional power-on state of the TMS320C4x COMM ports requires that interconnected DSPs be reset together; and
- A general DSP application loader for TIM-40 modules requires loading via COMM ports. Not all TIM-40 modules or C4x processors on TIM-40 modules have global bus connectivity. Outside of loading from ROM, COMM port loading is the only viable alternative.

The reset requirements and COMM port loading scheme together require that (without a resident OS to maintain communication links through COMM ports) all processors in a system be loaded together.

The functions in the System Module perform their actions on VX8s as a System comprised of multiple boards, each with multiple processors. A VX8 System is defined by two configuration files:

- System Definition File (SDF); and
- Load Definition File (LDF).

The SDF defines the hardware configuration of a VX8 system: the VX8 boards, TIM-40 modules, DSPs, COMM port connections, etc. The LDF defines the DSP application software that is to be loaded onto a VX8 system defined by an SDF.

The VX8 Instrument Driver parses these files to generate, internally, an image of the VX8 System. The System Module routines are then able to act on the VX8s as a system based on the SDF description.

The System Module functions are listed in the following table.

Table 18 System Module Functions

Function Name	Function Description
ssVX8_Deref	This function will return a valid open device session to the device indicated by the resource name inputted. The application can use the device session to directly call I/O library functions.
ssVX8_SystemCheckConfig	Determines if any VX8 devices in the system have CONFIG* asserted. This indicates that at least one processor in the system has not been loaded with DSP application code.
ssVX8_SystemClose	Closes and frees up resources associated with an opened VX8 system.
ssVX8_SystemErrorMessage	Returns a text explanation for a given error code.
ssVX8_SystemRevisionQuery	Reads the firmware revisions of all the VX8 devices present in the system structure and stores the results in the system structure.
ssVX8_SystemGetDriverRev	Returns a text string indicating the driver revision.
ssVX8_SystemLoadCode	Loads the DSP application code specified by the Load Definition File onto the VX8 system. Loading DSP software will reset the VX8 system.
ssVX8_SystemOpen	Opens and initializes a VX8 system defined by the inputted System Definition File. This function will optionally reset and/or load DSP software onto the system. The software to be loaded is indicated by the Load Definition File inputted. This function returns a system handle which is used by all System Module functions.
ssVX8_SystemRead	Reads a block of data from the specified VX8.
ssVX8_SystemReset	Resets all VX8 devices in the system.
ssVX8_SystemWrite	Writes a block of data to the specified VX8.

9.4.1. Opening a VX8 System

Opening a VX8 System is performed by the **ssVX8_SystemOpen** call. For VXIpnp compliant applications, a call must be made to **ssVX8_init**, followed by a call to **ssVX8_Open (**see *section 9.3 VXIpnp Module* for details.) The program flow for opening a VX8 System is shown in the following diagram. Opening a system involves parsing an SDF and LDF, opening device sessions to each VX8, resetting the VX8s, and finally, loading the DSP applications onto the processors. Once loaded with the DSP code, the VX8 System will be "operational" to the extent defined by your DSP software.

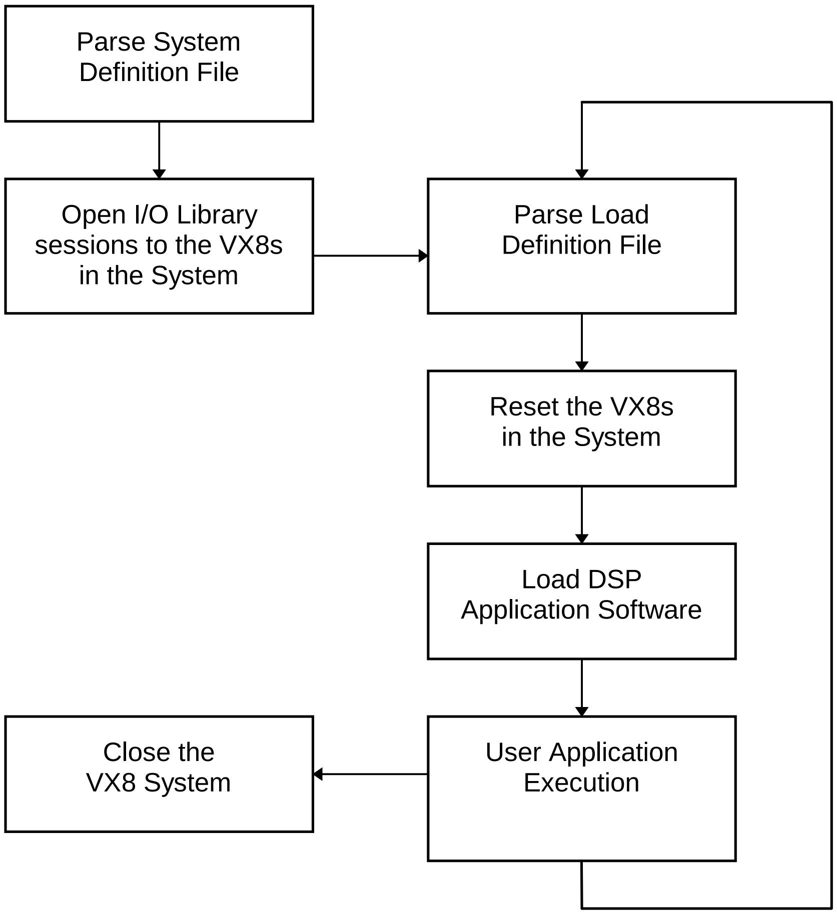

Parse System Definition File
Open I/O Library sessions to the VX8s in the System
Parse Load Definition File
Reset the VX8s in the System
Load DSP Application Software
User Application Execution
Close the VX8 System

Figure 13 Instrument Driver Program Flow

Host applications can open multiple sessions to a VX8 device by calling **ssVX8_SystemOpen** with the same SDF multiple times or by directly calling I/O library routines. It will be necessary for the host application to be aware of which invocation has the duty of initializing the VX8. Host applications should also be handling the possibility of another host session having placed a VX8 into reset.

If your host program needs to reset a device on the HP Local Bus, the host program first has to retrieve the ID for that device and then do a reset of that device by specifying its ID. The reset order of devices is determined by their ID sequence, not their physical slot sequence.

Accessing the VX8 through multiple sessions will be non-exclusive since the host task switching is OS dependent. It is left to the host applications to implement software constructs (semaphores) which will provide exclusive VX8 accesses between contending host sessions. Allowing multiple sessions to obtain resource handles to a single VX8 will provide the most flexibility for developers at a cost of enabling sessions to contend for VX8 resources.

9.4.2. System Definition File (SDF) Description

The SDF is a text based file that defines the hardware configuration of your VX8 System. The SDF modularly defines the hardware components in a logical manner. From top down, a VX8 System is composed of boards and COMM connections. Boards are comprised of TIM-40 modules, COMM connections, and a base address. TIM-40 modules are comprised of processors and their COMM connections. Processors are comprised of attributes only.

Note: Currently, there is no SDF editor available; you'll be required to use a text editor to modify the SDF used by your application. However, an SDF editor will be available in future product releases.

Sections

The SDF is broken up into sections which are identifiable by a name enclosed in square braces, for example [`common`] and [`TestSystem`]. Sections identify TIM-40 modules, VX8 boards, VX8 systems, processors, and boot processors. Each section can contain several attributes describing the section. For example, TIM-40 module and VX8 board sections contain Component and COMMConnection attributes identifying processors and COMM port connections. Section names cannot contain spaces and are case sensitive.

The parsing software does not perform multiple passes over the SDF. Sections must be defined before they are referenced by other sections. SDFs should be defined with the [`common`] section at the top, followed by processor sections, TIM-40 sections, Board sections, the System section, and finally the Boot Processor section.

Reserved Section Names:

- common (symbol table)
- BootProc
- End

Types and Models

The type of each section, except for the reserved sections, is identified by the Type attribute. Type is equated to one of several keywords.

Type Keywords:

- Processor
- TIM
- Board
- Resource
- System
- Prototype

Sections may also have a Model attribute, which is simply a text identifier that can be used to provide information about the component. The Model attribute **does not** affect the SDF and can be anything you want.

Components

Depending on the type of section, the Components attribute identifies the components belonging to that specific section.

Section names are used in the `Components` attribute of TIM-40 module, Board, and System sections to specify which section the components belong to. However, the sections must be defined before they are referenced in Components attributes.

Components Syntax:

```
Components = { (<identifier>, <# COMM ports>, <defining section name>),
               (<identifier>, <# COMM ports>, <defining section name>)
             }
```

The following is an example of a Components attribute declaration of a TIM-40 module with two processors:

```
Components = {(Proc0, 6,C44_PROC_ST1), (Proc1, 6, C44_PROC_ST2)}
```

COMMConnections

The COMM port connections for TIM-40 modules and VX8 boards and systems are identified by the COMMConnections attribute. There can only be one entry for each bi-directional COMM port connection.

COMMConnections Syntax:

```
COMMConnections = {(<source identifier>, <COMM #>, <dest identifier>, <COMM #>),
                   (<source identifier>, <COMM #>, <dest identifier>, <COMM #>)
                  }
```

The following is an example of a COMMConnections attribute of a TIM-40 module:

```
COMMConnections = {  // Src,     ID,    Dst,     ID
                        (Proc0,   0,    TIM,     0),    // TIM module connector
                        (Proc0,   1,    TIM,     1),    //  "
                        (Proc0,   2,    TIM,     2),    //  "
                        (TIM,     3,    Proc0,   3),    //  "
                        (TIM,     4,    Proc0,   4),    //  "
                        (TIM,     5,    Proc0,   5)     //  "
```

Note that for a given section, the <*source identifier*> and <*dest identifier*> can be that section’s “Type”. That is, for a TIM definition section, the keyword “TIM” can be used as a source or destination identifier in the COMMConnections attribute. Likewise, in a board definition section, the keyword “Board” can be used. Such assignments connect the COMM ports of the components to a logical connector on the actual TIM or board. These associations are then used to connect the logical TIM COMM ports to the logical “Site” COMM ports in the Board Definition.

Symbol Table

The Symbol Table section is identified by the reserved section name “common”. Define any string constants, such as the VX8 board names (I/O library resource identifiers), in this section. Entries in the symbol table can be referenced in other sections.

The following is an example of a Symbol Table section identifying SICL base addresses (I/O resource descriptors) to be used in board definition sections:

```
[common]
   VX8_BASE_ADDR_1 = vxi,240;
   VX8_BASE_ADDR_2 = vxi,241;
```

The following is an example of a Symbol Table section identifying VISA base addresses (I/O resource descriptors) to be used in board definition sections:

```
[common]
   VX8_BASE_ADDR_1 = VXI0::240::INSTR;
   VX8_BASE_ADDR_2 = VXI0::241::INSTR;
```

Processor Definitions

Processors have several attributes, of which, only a few are used by the VX8 Instrument Driver. The processor attributes which are of importance are “BootPort”, “LMCR”, “GMCR”, and of course “Type = Processor;”.

The following is an example of a processor definition section:

```
[C40_PROC_VX8EMBED]
   Type = Processor;
   Model = C40;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60  ;
   LMCR = 0x3e840000;
   GMCR = 0x3a4c0000;
```

BootPort This processor attribute is used to identify which COMM ports the processor is capable of booting from. Two keywords (ALL and NONE) and any number in the specified range for the processor are valid. “ALL” indicates to the SDF parser that any COMM port is suitable for loading. “NONE” indicates that the TIM is not to be loaded, but exists in the VX8 System Structure. A number indicates to the SDF software to attempt loading using the specified COMM port only. If a processor is not reachable for loading by COMM port, the SDF software will produce an error.

LMCR and GMCR The LMCR and GMCR values are used during loading to set the Local Memory Control Register (LMCR) and the Global Memory Control Register (GMCR) on the C4x DSP. The LMCR and GMCR values should be set to the values indicated by your TIM documentation.

Note: TIM-40 module processors that are connected to the Global Bus must configure themselves to have STRB0 and STRB1 controlled by external RDY signal.

TIM-40 Module Definition

TIM sections define a TIM-40 module by its processors and their COMM port connections. The processors are defined in the `Components` attribute. The COMM port connections are specified in the COMMConnections attribute.

The following is an example definition of a TIM-40 module with two processors:

```
[MDC44ST]
   Type = TIM;
   Model = MDC44ST;
   Components = {(Proc0, 6, C44_PROC_STA), (Proc1, 6, C44_PROC_STB)}

   COMMConnections = {// Src,     ID,   Dst,     ID
                        (Proc0,   1,    Proc1,   4),    // On TIM connection

                        (Proc0,   2,    TIM,     2),    // TIM module connector
                        (Proc0,   4,    TIM,     3),    //  "
                        (Proc0,   5,    TIM,     4),    //  "

                        (Proc1,   1,    TIM,     0),    //  "
                        (Proc1,   2,    TIM,     1),    //  "
                        (Proc1,   5,    TIM,     5)     //  "
                     }
```

Board Definition

Boards are comprised of TIM-40 modules, COMM connections, and attributes. The `Components` and `COMMConnections` attribute syntax are the same as for the TIM-40 Module definition. Boards have an additional attribute `BaseAddress` which equates to an I/O resource descriptor like “vxi,240” for SICL or “VXI0::240::INST” for VISA. These strings should be defined in the [`common`] section.

The DRAM configuration of your VX8 is not specified in the SDF. Bounds checking for I/O to VX8 DRAM is the responsibility of your application software. The following is an example of a board definition section:

```
[Custom_Board2]
   Type = Board;
   Model = VX8;

   BaseAddress = VX8_BASE_ADDR_2;    // Addr is defined above

   Components = {
         (General,0,BOARD1_GENERAL),
         (SiteA, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteB, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteC, 6, GENERAL_SITE),                  // Empty Site
         (SiteD, 6, GENERAL_SITE),                  // Empty Site
         (SiteE, 6, GENERAL_SITE),                  // Empty Site
         (SiteF, 6, GENERAL_SITE),                  // Empty Site
         (SiteG, 6, GENERAL_SITE),                  // Empty Site
         (SiteH, 6, GENERAL_SITE)                   // Empty Site
   }


   COMMConnections = {
         (SiteA,   0,    SiteG,   5),
         (SiteA,   1,    SiteD,   4),
         (SiteA,   2,    SiteE,   5),
         (SiteA,   3,    Board,   1),
         (SiteA,   4,    SiteB,   1),
         (SiteA,   5,    SiteC,   0),

         (SiteB,   0,    SiteH,   5),
         (SiteB,   2,    SiteF,   5),
         (SiteB,   3,    Board,   3),
         (SiteB,   4,    SiteC,   1),
         (SiteB,   5,    SiteD,   0),

         (SiteC,   2,    Board,   0),
         (SiteC,   3,    SiteG,   2),
         (SiteC,   4,    SiteD,   1),
         (SiteC,   5,    SiteE,   0),

         (SiteD,   2,    Board,   2),
         (SiteD,   3,    SiteH,   2),
         (SiteD,   5,    SiteF,   0),

         (SiteE,   1,    SiteH,   4),
         (SiteE,   2,    Board,   4),
         (SiteE,   3,    SiteG,   0),
         (SiteE,   4,    SiteF,   1),

         (SiteF,   2,    Board,   6),
         (SiteF,   3,    SiteH,   0),
         (SiteF,   4,    SiteG,   1),
        (SiteG,   3,    Board,   5),
        (SiteG,   4,    SiteH,   1),

        (SiteH,   3,    Board,   7)
   }
```

Similar to the TIM-40 definition, the `<source identifier>` `can be that sections "Type". At the Board Definition level, the COMMConnection associations map logical TIM COMM port connections to front panel COMM ports for that board. The following table provides the COMM port mappings from site to front panel in the SDF example.`

Site	Site Physical COMM Port #	Board Logical COMM Port #
A	3	1
B	3	3
C	2	0
D	2	2
E	2	4
F	2	6
G	3	5
H	3	7

System Definition

The VX8 System is composed of the boards COMM connections. The Components and COMMConnections attribute syntaxes for a System definition are similar to those of a TIM-40 Module definition. The COMMConnections section in a System definition section specifies the front panel COMM port connections in your system.

The following is an example of a System definition section:

```
[TestSystem]
   Type = System;
   Model = Development;
   Components =
   {
         (Board1, 8,  CUSTOM_Board1),    // [Board1] section defines sites.
         (Board2, 8,  CUSTOM_Board2)     // Standard base board no sites filled
   }

   COMMConnections = {  // Src,     ID,      Dst,     ID
         (Board1,  1,  Board2,  4)  // Connection of board front panels
         (Board1,  2,  Board2,  1)  //      "     "    "     "     "
   }
```

BootProc

The BootProc section defines the processors which are responsible for loading software onto the other processors in the system.

BootProc Syntax:

<*Board Name*>:<*Site Name*>:<*Boot Processor Name*> = <*board base address*>;

The following is an example of a BootProc section:

```
[BootProc]
   Board1:SiteA:Proc0 = VX8_BASE_ADDR1;
   Board2:SiteA:Proc0 = VX8_BASE_ADDR2;
```

9.4.3. System Definition File (SDF) Examples

To meet your system requirements, Spectrum has provided three different SDFs for the following hardware configurations:

- a single VX8 board with a base configuration
- a single VX8 board populated with additional TIM-40 modules
- a multi-board VX8 system

The following three sections contain the example SDF files.

SDF Example 1 : Base VX8 Configuration

The following is a sample SICL System Definition File (SDF) containing a single VX8 board with the base configuration at a logical address of 240. Note that the board base addresses use SICL nomenclature. These descriptors must be modified if you using VISA.

```
[common]
   VX8_BASE_ADDR_1 = vxi,240;

[C40_PROC_VX8EMBED]
   Type = Processor;
   Model = C40;           // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;        // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3e840000;
   GMCR = 0x3a4c0000;

[VX8_EMBEDDED_SITE]
   Type = TIM;
   Model = VX8EMBED;
   Components = {(Proc0, 6, C40_PROC_VX8EMBED)}
   COMMConnections = {  // Src,     ID,   Dst,     ID
                        (Proc0,   0,    TIM,     0),    // TIM module connector
                        (Proc0,   1,    TIM,     1),    //  "
                        (Proc0,   2,    TIM,     2),    //  "
                        (TIM,     3,    Proc0,   3),    //  "
                        (TIM,     4,    Proc0,   4),    //  "
                        (TIM,     5,    Proc0,   5)     //  "
                  }

[GENERAL_SITE]
   Type = SITE;

[Custom_Board1]
   Type = Board;
   Model = VX8;     // one of [V8, VX8, ...]

   BaseAddress = VX8_BASE_ADDR_1;    // Addr is defined above

   Components = {
         (SiteA, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteB, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteC, 6, GENERAL_SITE),                  // Empty Site
         (SiteD, 6, GENERAL_SITE),                  // Empty Site
         (SiteE, 6, GENERAL_SITE),                  // Empty Site
         (SiteF, 6, GENERAL_SITE),                  // Empty Site
         (SiteG, 6, GENERAL_SITE),                  // Empty Site
         (SiteH, 6, GENERAL_SITE)                   // Empty Site
   }

   COMMConnections = {
         (SiteA,   0,    SiteG,   5),
         (SiteA,   1,    SiteD,   4),
         (SiteA,   2,    SiteE,   5),
         (SiteA,   3,    Board,   1),
         (SiteA,   4,    SiteB,   1),
         (SiteA,   5,    SiteC,   0),

         (SiteB,   0,    SiteH,   5),
         (SiteB,   2,    SiteF,   5),
         (SiteB,   3,    Board,   3),
         (SiteB,   4,    SiteC,   1),
         (SiteB,   5,    SiteD,   0),

         (SiteC,   2,    Board,   0),
         (SiteC,   3,    SiteG,   2),
         (SiteC,   4,    SiteD,   1),
         (SiteC,   5,    SiteE,   0),

         (SiteD,   2,    Board,   2),
         (SiteD,   3,    SiteH,   2),
         (SiteD,   5,    SiteF,   0),

         (SiteE,   1,    SiteH,   4),
         (SiteE,   2,    Board,   4),
         (SiteE,   3,    SiteG,   0),
         (SiteE,   4,    SiteF,   1),

         (SiteF,   2,    Board,   6),
         (SiteF,   3,    SiteH,   0),
         (SiteF,   4,    SiteG,   1),

         (SiteG,   3,    Board,   5),
         (SiteG,   4,    SiteH,   1),

         (SiteH,   3,    Board,   7)

   }


[TestSystem]
   Type = System;
   Model = Development;
   Components =
   {
         (Board1, 8,  CUSTOM_Board1)    // [Board1] section defines sites.
   }

[BootProc]
   Board1:SiteA:Proc0 = VX8_BASE_ADDR1;

[End]
```

SDF Example 2 : A Single VX8 Populated with TIM-40 Modules

The following is a sample SICL System Definition File (SDF) containing a single VX8 board populated with TIM-40 modules. Note that the board base addresses use SICL nomenclature. These descriptors must be modified if you are using VISA.

Notice how the C44 processors in the ST TIM-40 module are specified as having 6 COMM ports. The missing COMM ports on the TMS320C44 (0 and 3) are not included in the COMMConnections declaration in the ST TIM-40 section. The ST TIM-40 module is also specified as having 2 processors in the Components declaration.

Front panel COMM port connections are made between Nodes A and C, B and D, E and G, and between F and H.

```
[common]
   VX8_BASE_ADDR_1 = vxi,240;

[C40_PROC_VX8EMBED]
   Type = Processor;
   Model = C40;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3e840000;
   GMCR = 0x3a4c0000;

[C40_PROC_SS]
   Type = Processor;
   Model = C40;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3dec2050;
   GMCR = 0x3a4c0000;

[C44_PROC_STA]
   Type = Processor;
   Model = C44;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3d74a850;
   GMCR = 0x3a4c8000;

[C44_PROC_STB]
   Type = Processor;
   Model = C44;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3d74a850;
   GMCR = 0x3a4c8000;


[VX8_EMBEDDED_SITE]
   Type = TIM;
   Model = VX8EMBED;
   Components = {(Proc0, 6, C40_PROC_VX8EMBED)}
   COMMConnections = {  // Src,     ID,   Dst,     ID
                        (Proc0,   0,    TIM,     0),    // TIM module connector
                        (Proc0,   1,    TIM,     1),    //  "
                        (Proc0,   2,    TIM,     2),    //  "
                        (TIM,     3,    Proc0,   3),    //  "
                        (TIM,     4,    Proc0,   4),    //  "
                        (TIM,     5,    Proc0,   5)     //  "
                  }
```

```
[MDC40SS]
   Type = TIM;
   Model = MDC40SS;
   Components = {(Proc0, 6, C40_PROC_SS)}
   COMMConnections = {  // Src,     ID,   Dst,     ID
                        (Proc0,   0,    TIM,     0),    // TIM module connector
                        (Proc0,   1,    TIM,     1),    //  "
                        (Proc0,   2,    TIM,     2),    //  "
                        (TIM,     3,    Proc0,   3),    //  "
                        (TIM,     4,    Proc0,   4),    //  "
                        (TIM,     5,    Proc0,   5)     //  "
                  }



[MDC44ST]
   Type = TIM;
   Model = MDC44ST;
   Components = {(Proc0, 6, C44_PROC_STA), (Proc1, 6, C44_PROC_STB)}

   COMMConnections = {// Src,     ID,   Dst,     ID
                        (Proc0,   1,    Proc1,   4),    // On TIM connection

                        (Proc0,   2,    TIM,     2),    // TIM module connector
                        (Proc0,   4,    TIM,     3),    //  "
                        (Proc0,   5,    TIM,     4),    //  "

                        (Proc1,   1,    TIM,     0),    //  "
                        (Proc1,   2,    TIM,     1),    //  "
                        (Proc1,   5,    TIM,     5)     //  "
                     }


[GENERAL_SITE]
   Type = SITE;

[Custom_Board1]
   Type = Board;
   Model = VX8;     // one of [V8, VX8, ...]

   BaseAddress = VX8_BASE_ADDR_1;    // Addr is defined above

   Components = {
         (SiteA, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteB, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteC, 6, MDC40SS),                       // Single processor module
         (SiteD, 6, MDC40SS),                       // Single processor module
         (SiteE, 6, MDC40SS),                       // Single processor module
         (SiteF, 6, MDC40SS),                       // Single processor module
         (SiteG, 6, MDC44ST),                       // Dual processor module
         (SiteH, 6, MDC44ST)                        // Dual processor module
   }

   COMMConnections = {
         (SiteA,   0,    SiteG,   5),
         (SiteA,   1,    SiteD,   4),
         (SiteA,   2,    SiteE,   5),
         (SiteA,   3,    Board,   1),
         (SiteA,   4,    SiteB,   1),
         (SiteA,   5,    SiteC,   0),

         (SiteB,   0,    SiteH,   5),
         (SiteB,   2,    SiteF,   5),
         (SiteB,   3,    Board,   3),
         (SiteB,   4,    SiteC,   1),
         (SiteB,   5,    SiteD,   0),

         (SiteC,   2,    Board,   0),
         (SiteC,   3,    SiteG,   2),
         (SiteC,   4,    SiteD,   1),
         (SiteC,   5,    SiteE,   0),

         (SiteD,   2,    Board,   2),
         (SiteD,   3,    SiteH,   2),
         (SiteD,   5,    SiteF,   0),
         (SiteE,   1,    SiteH,   4),
         (SiteE,   2,    Board,   4),
         (SiteE,   3,    SiteG,   0),
         (SiteE,   4,    SiteF,   1),

         (SiteF,   2,    Board,   6),
         (SiteF,   3,    SiteH,   0),
         (SiteF,   4,    SiteG,   1),

         (SiteG,   3,    Board,   5),
         (SiteG,   4,    SiteH,   1),

         (SiteH,   3,    Board,   7)

   }


[TestSystem]
   Type = System;
   Model = Development;
   Components =
   {
         (Board1, 8,  CUSTOM_Board1)    // [Board1] section defines sites.
   }

   COMMConnections = {  // Src,     ID,      Dst,     ID
                        (Board1,  0,  Board1,  1),  // front panel C2 to A3
                        (Board1,  2,  Board1,  3),  // front panel D2 to B3
                        (Board1,  4,  Board1,  5),  // front panel E2 to G3
                        (Board1,  6,  Board1,  7)   // front panel F2 to H3
                     }

[BootProc]
   Board1:SiteA:Proc0 = VX8_BASE_ADDR1;

[End]
```

SDF Example 3 : Multiple VX8 Configuration

The following is a sample SICL System Definition File (SDF) containing two VX8 boards with TIM-40 modules. Notice how the multiple board base address (resource names) are defined in the common section and also referenced in the individual board definition sections. Note that the board base addresses use SICL nomenclature. These descriptors must be modified if you are using VISA.

```
[common]
   VX8_BASE_ADDR_1 = vxi,240;
   VX8_BASE_ADDR_2 = vxi,241;

[C40_PROC_VX8EMBED]
   Type = Processor;
   Model = C40;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3e840000;
   GMCR = 0x3a4c0000;

[C40_PROC_SS]
   Type = Processor;
   Model = C40;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3dec2050;
   GMCR = 0x3a4c0000;

[C44_PROC_STA]
   Type = Processor;
   Model = C44;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3d74a850;
   GMCR = 0x3a4c8000;

[C44_PROC_STB]
   Type = Processor;
   Model = C44;          // one of [C44, C40, S62, S60, ...]
   BootPort = ALL;       // Can boot from any COMM port
   Speed = 60;
   LMCR = 0x3d74a850;
   GMCR = 0x3a4c8000;


[VX8_EMBEDDED_SITE]
   Type = TIM;
   Model = VX8EMBED;
   Components = {(Proc0, 6, C40_PROC_VX8EMBED)}
   COMMConnections = {  // Src,     ID,   Dst,     ID
                        (Proc0,   0,    TIM,     0),    // TIM module connector
                        (Proc0,   1,    TIM,     1),    //  "
                        (Proc0,   2,    TIM,     2),    //  "
                        (TIM,     3,    Proc0,   3),    //  "
                        (TIM,     4,    Proc0,   4),    //  "
                        (TIM,     5,    Proc0,   5)     //  "
                  }


[MDC40SS]
   Type = TIM;
   Model = MDC40SS;
   Components = {(Proc0, 6, C40_PROC_SS)}
   COMMConnections = {  // Src,     ID,   Dst,     ID
                        (Proc0,   0,    TIM,     0),    // TIM module connector
                        (Proc0,   1,    TIM,     1),    //  "
                        (Proc0,   2,    TIM,     2),    //  "
                        (TIM,     3,    Proc0,   3),    //  "
                        (TIM,     4,    Proc0,   4),    //  "
                        (TIM,     5,    Proc0,   5)     //  "
                  }
```

```
[MDC44ST]
   Type = TIM;
   Model = MDC44ST;
   Components = {(Proc0, 6, C44_PROC_STA), (Proc1, 6, C44_PROC_STB)}

   COMMConnections = {// Src,     ID,   Dst,     ID
                        (Proc0,   1,    Proc1,   4),    // On TIM connection

                        (Proc0,   2,    TIM,     2),    // TIM module connector
                        (Proc0,   4,    TIM,     3),    //  "
                        (Proc0,   5,    TIM,     4),    //  "

                        (Proc1,   1,    TIM,     0),    //  "
                        (Proc1,   2,    TIM,     1),    //  "
                        (Proc1,   5,    TIM,     5)     //  "
                     }


[GENERAL_SITE]
   Type = SITE;

[Custom_Board1]
   Type = Board;
   Model = VX8;     // one of [V8, VX8, ...]

   BaseAddress = VX8_BASE_ADDR_1;    // Addr is defined above

   Components = {
         (SiteA, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteB, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteC, 6, MDC40SS),                       // Single processor module
         (SiteD, 6, MDC40SS),                       // Single processor module
         (SiteE, 6, MDC40SS),                       // Single processor module
         (SiteF, 6, MDC40SS),                       // Single processor module
         (SiteG, 6, MDC44ST),                       // Dual processor module
         (SiteH, 6, MDC44ST)                        // Dual processor module
   }

   COMMConnections = {
         (SiteA,   0,    SiteG,   5),
         (SiteA,   1,    SiteD,   4),
         (SiteA,   2,    SiteE,   5),
         (SiteA,   3,    Board,   1),
         (SiteA,   4,    SiteB,   1),
         (SiteA,   5,    SiteC,   0),

         (SiteB,   0,    SiteH,   5),
         (SiteB,   2,    SiteF,   5),
         (SiteB,   3,    Board,   3),
         (SiteB,   4,    SiteC,   1),
         (SiteB,   5,    SiteD,   0),

         (SiteC,   2,    Board,   0),
         (SiteC,   3,    SiteG,   2),
         (SiteC,   4,    SiteD,   1),
         (SiteC,   5,    SiteE,   0),

         (SiteD,   2,    Board,   2),
         (SiteD,   3,    SiteH,   2),
         (SiteD,   5,    SiteF,   0),

         (SiteE,   1,    SiteH,   4),
         (SiteE,   2,    Board,   4),
         (SiteE,   3,    SiteG,   0),
         (SiteE,   4,    SiteF,   1),

         (SiteF,   2,    Board,   6),
         (SiteF,   3,    SiteH,   0),
         (SiteF,   4,    SiteG,   1),

         (SiteG,   3,    Board,   5),
         (SiteG,   4,    SiteH,   1),

         (SiteH,   3,    Board,   7)
   }
```

```
[Custom_Board2]
   Type = Board;
   Model = VX8;     // one of [V8, VX8, ...]

   BaseAddress = VX8_BASE_ADDR_2;    // Addr is defined above

   Components = {
         (SiteA, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteB, 6, VX8_EMBEDDED_SITE),             // Embedded C40
         (SiteC, 6, MDC40SS),                       // Single processor module
         (SiteD, 6, MDC40SS),                       // Single processor module
         (SiteE, 6, MDC40SS),                       // Single processor module
         (SiteF, 6, MDC40SS),                       // Single processor module
         (SiteG, 6, MDC44ST),                       // Dual processor module
         (SiteH, 6, MDC44ST)                        // Dual processor module
   }

   COMMConnections = {
         (SiteA,   0,    SiteG,   5),
         (SiteA,   1,    SiteD,   4),
         (SiteA,   2,    SiteE,   5),
         (SiteA,   3,    Board,   1),
         (SiteA,   4,    SiteB,   1),
         (SiteA,   5,    SiteC,   0),

         (SiteB,   0,    SiteH,   5),
         (SiteB,   2,    SiteF,   5),
         (SiteB,   3,    Board,   3),
         (SiteB,   4,    SiteC,   1),
         (SiteB,   5,    SiteD,   0),

         (SiteC,   2,    Board,   0),
         (SiteC,   3,    SiteG,   2),
         (SiteC,   4,    SiteD,   1),
         (SiteC,   5,    SiteE,   0),

         (SiteD,   2,    Board,   2),
         (SiteD,   3,    SiteH,   2),
         (SiteD,   5,    SiteF,   0),

         (SiteE,   1,    SiteH,   4),
         (SiteE,   2,    Board,   4),
         (SiteE,   3,    SiteG,   0),
         (SiteE,   4,    SiteF,   1),

         (SiteF,   2,    Board,   6),
         (SiteF,   3,    SiteH,   0),
         (SiteF,   4,    SiteG,   1),

         (SiteG,   3,    Board,   5),
         (SiteG,   4,    SiteH,   1),

         (SiteH,   3,    Board,   7)

   }
```

```
[TestSystem]
   Type = System;
   Model = Development;
   Components =
   {
      (Board1, 8,  CUSTOM_Board1),    // [Board1] section defines sites.
      (Board2, 8,  CUSTOM_Board2)     // [Board2] section defines sites.
   }

   COMMConnections = {  // Src,     ID,      Dst,     ID
                         (Board1,  0,  Board2,  1),  // front panel C1-2 to A2-3
                         (Board1,  2,  Board2,  3),  // front panel D1-2 to B2-3
                         (Board1,  4,  Board2,  5),  // front panel E1-2 to G2-3
                         (Board1,  6,  Board2,  7),  // front panel F1-2 to H2-3
                         (Board2,  0,  Board1,  1),  // front panel C2-2 to A1-3
                         (Board2,  2,  Board1,  3),  // front panel D2-2 to B1-3
                         (Board2,  4,  Board1,  5),  // front panel E2-2 to G1-3
                         (Board2,  6,  Board1,  7)   // front panel F2-2 to H1-3
                      }

[BootProc]
   Board1:SiteA:Proc0 = VX8_BASE_ADDR1;
   Board2:SiteA:Proc0 = VX8_BASE_ADDR2;

[End]
```

9.4.4. Load Definition File (LDF)

The LDF defines the software configuration of your VX8 system. It contains the paths and filenames for the C4x DSP executables that are to be loaded by the Instrument Driver. The LDF syntax is similar to the SDF, but is much simpler.

Note: Currently, there is no LDF editor available; you'll be required to use a text editor to modify the LDF used by your application. However, an LDF editor will be available in future product releases.

The LDF is composed of a single section called [Files]. In the Files section, simply list the processor, defined by its case sensitive resource name, and equate it to the C4x COFF file that you wish to have loaded by the Instrument Driver. The order in which the processors are listed is not important, but the processors **must** correspond to a processing node defined in the SDF file.

The resource name syntax is:

<*Board Name*>:<*Site Name*>:<*Processor Name*> = <*COFF file name*>;

The following is an example Load Definition File (LDF).

```
[Files]

       Board1:SiteA:Proc0 = c:\vx8\examples\vx8mult.out;  // Embedded TIM A

[end]
```

9.4.5. Calling I/O Library Routines Directly

You can choose to bypass the **ssVX8_SystemRead** and **ssVX8_SystemWrite** functions and directly call the I/O library routines to fine tune or increase the performance of your VX8 application.

This can be done by opening the VX8 system normally and then using **ssVX8_Deref** to obtain an I/O library session to the specified VX8. You can then directly call the I/O library read/write routines to transfer data to and from the globally accessible memories on the VX8.

You can also open a new device session to a VX8 in the usual manner indicated by the I/O library you're using (`iopen` for SICL, `viOpen` for VISA). In this case, your application code will be responsible for closing the session when done. This is particularly useful in VISA if your application needs to keep multiple regions of a VX8 mapped simultaneously. Opening additional sessions to maintain mapped regions on the VX8 is more efficient than the constant mapping and unmapping that would occur with a single session.

10 VX8 Instrument Driver Functions

This chapter presents the "C" language functions available in the VX8 Host Software Library. The functions are listed in alphabetical order.

ssVX8_close

Function	Closes a VX8 system by closing all session handles and freeing any resources used by the system.
Include File	vx8.h
Syntax	ViStatus _VI_FUNC ssVX8_close (ViSession *vi*);
Parameters	*vi* — Unique logical identifier to a session which contains the handle to a VX8 system stored in its user-defined attribute.

Returned Values

VI_SUCCESS	The system was successfully closed.
SSVX8_ERROR_SYSTEMCLOSE_FAILED	The system did not successfully close.

Remarks This function is only applicable to VISA VXIpnp applications.

This function calls **ssVX8_SystemClose** to close the system referenced by the user-defined attribute of the vi session. The vi session is then closed.

ssVX8_Deref

Function Returns an open device session handle to the indicated VX8.

Include File vx8.h

Syntax RESULT _SS_FUNC ssVX8_Deref (SYSHANDLE *VX8_System*, STRING *boardName*, PPVOID *boardHandle*);

Parameters

VX8_System	VX8 system handle
boardName	Name of a VX8 device
boardHandle	Pointer to storage for a device session handle

Returned Values

VI_SUCCESS	The system was successfully closed.
SSVX8_ERROR_SYSTEM_STRUCT_UNINITIALIZED	The system was uninitialized.
SSVX8_ERROR_NO_SSVX8_FOUND	The VX8 was not found in the system structure.

Remarks The returned handle can be used to directly call I/O library routines in order to communicate with the VX8 through its A32 slave interface.

The syntax for *boardName* will differ depending on whether your I/O library is SICL or VISA. For SICL, the syntax will resemble “vxi,240”. For VISA, the syntax will resemble “VXI0::240::INST”. Refer to the I/O library documentation for the proper resource descriptor nomenclature.

ssVX8_error_message

Function Gets a text description for a given error code.

Include File vx8.h

Syntax ViStatus _VI_FUNC ssVX8_error_message (ViSession *vi*, ViStatus *error*, ViString *message*);

Parameters

vi	Unique logical identifier to a session
error	Error number
message	String description of error

Returned Values

VI_SUCCESS	Successfully returned the error string.
SSVX8_ERROR_ERROR_MESSAGE_UNKNOWN	Unknown error code

Remarks This function is only applicable to VISA VXIpnp applications and merely calls **ssVX8_SystemErrorMessage.**

ssVX8_error_query

Function This is a “stubbed” function that returns VI_WARN_NSUP_ERROR_QUERY when called.

Include File vx8.h

Syntax ViStatus _VI_FUNC ssVX8_error_query (ViSession *vi*, ViPInt32 *error*,
ViString *error_message*);

Parameters

vi	Unique logical identifier to a session
error	Pointer to error number storage
message	String description of error

Returned Values

VI_WARN_NSUP_ERROR_QUERY	Error query not supported.

Remarks This function is only applicable to VISA VXIpnp applications. Although non-functional, this function must exist for VXIpnp compliance.

ssVX8_init

Function Opens a session.

Include File vx8.h

Syntax ViStatus _VI_FUNC ssVX8_init (ViRsrc *rsrcName*, ViBoolean *id_query*, ViBoolean *reset*, ViPSession *vi*);

Parameters

rsrcName	Unique symbolic name of a resource
id_query	Flag to query the A16 registers to confirm that the device is a VX8
reset	Flag to reset the board - not supported
vi	Unique logical identifier to a session

Returned Values

VI_SUCCESS	Successfully initialized the device.
SSVX8_ERROR_BAD_RSRC_DESCRIPTOR	Unknown *rsrcName*.
VI_ERROR_FAIL_ID_QUERY	The device *rsrcName* is not a VX8.
VI_WARN_NSUP_RESET	Soft resetting is not supported.

Remarks A subsequent call to **ssVX8_open** must be performed with the session handle returned from this function call. A call to **viOpenDefaultRM** is NOT required.

This function is only applicable to VISA VXIpnp applications.

rsrcName syntax will resemble “VXI0::240::INST”. Refer to the VISA I/O library documentation for the proper resource descriptor nomenclature.

This function will return VI_WARN_NSUP_RESET if the *reset* parameter is set.

ssVX8_open

Function Opens a VX8 system.

Include File vx8.h

Syntax

```
ViStatus _VI_FUNC ssVX8_open(ViSession vi, SYSHANDLE *VX8SystemHndlPtr,
                             ViString SDF_file, ViString LDF_file,
                             ViUInt32 flags);
```

Parameters

vi	Unique logical identifier to a session
VX8SystemHndlPtr	Pointer to a system handle storage
SDF_file	SDF filename and path
LDF_file	LDF filename and path
flags	Flags described below

Flag	Value	Description
SSVX8_SYSOPEN_NOACTION	0x0	Does not reset the boards or load DSP application.
SSVX8_SYSOPEN_LOADSYSTEM	0x1	Load the DSP applications described by the LDF onto the system.
SSVX8_SYSOPEN_RESETSYSTEM	0x2	Reset the boards in the system.

Returned Values

VI_SUCCESS	Successfully opened the VX8 system.
SSVX8_ERROR_BAD_RSRC_DESCRIPTOR	Unknown *rsrcName*.
VI_ERROR_FAIL_ID_QUERY	The device *rsrcName* is not a VX8.
VI_WARN_NSUP_RESET	Soft resetting is not supported.

Remarks This function is only applicable to VISA VXIpnp applications. A call to **ssVX8_open** must be preceded by a call to **ssVX8_init**.

ssVX8_reset

Function	This is a “stubbed” function that returns VI_WARN_NSUP_RESET when called.	
Include File	vx8.h	
Syntax	ViStatus _VI_FUNC ssVX8_reset (ViSession *vi*);	
Parameters	*vi*	Unique logical identifier to a session
Returned Values	VI_WARN_NSUP_RESET	Reset not supported.
Remarks	This function is only applicable to VISA VXIpnp applications. Although non-functional, this function must exist for VXIpnp compliance.	

ssVX8_revision_query

Function This function reports the firmware revision of the VX8 in question and the driver revision.

Include File vx8.h

Syntax ViStatus _VI_FUNC ssVX8_revision_query (ViSession *vi*, ViString *driver_rev*, ViString *instr_rev*);

Parameters

vi	Unique logical identifier to a session
driver_rev	Instrument driver revision
instr_rev	Instrument firmware revision

Returned Values

VI_SUCCESS	Successfully reported driver and instrument firmware revisions.

Remarks This function is only applicable to VISA VXIpnp applications. Although non-functional, this function must exist for VXIpnp compliance.

ssVX8_self_test

Function This is a “stubbed” function that returns VI_WARN_NSUP_SELF_TEST when called.

Include File vx8.h

Syntax ViStatus _VI_FUNC ssVX8_self_test (ViSession *vi*, ViPInt16 *test_result*,
ViString *test_message*);

Parameters

vi	Unique logical identifier to a session
test_result	Result of self test
test_message	String description of the self test results

Returned Values

VI_WARN_NSUP_SELF_TEST	Self test not supported.

Remarks This function is only applicable to VISA VXIpnp applications. Although non-functional, this function must exist for VXIpnp compliance.

ssVX8_SystemCheckConfig

Function Checks the CONFIG* status of all the boards in the VX8 system.

Include File vx8.h

Syntax RESULT _SS_FUNC ssVX8_SystemCheckConfig (SYSHANDLE *VX8_System*);

Parameters

VX8_System	VX8 system handle

Returned Values

SUCCESS	All A16 status register CONFIG* bits in the VX8 configured in the system are de-asserted.
SSVX8_ERROR_VX8_STILL_IN_CONFIG	At least one DSP in the system is still asserting CONFIG*.

Remarks The purpose of the TIM-40 CONFIG* line is to indicate that all DSPs are loaded and running. For the CONFIG* functionality to work, all DSPs in the VX8 system must be loaded with software that will de-assert their CONFIG* signals. Refer to the **boot_*.asm** C boot routines for Nodes A, B, and TIM-40 sites in *Chapter 5* of this manual.

ssVX8_SystemClose

Function Closes a VX8 system.

Include File vx8.h

Syntax RESULT _SS_FUNC ssVX8_SystemClose (SYSHANDLE *VX8_System*);

Parameters

VX8_System	VX8 system handle

Returned Values

VI_SUCCESS	Successfully closed the VX8 system.
SSVX8_ERROR_SYSTEMCLOSE_FAILED	An error occurred during the closing of the VX8 system.
SSVX8_ERROR_CLOSE_FAILED	An error occurred during the closing of a device session.

Remarks This function closes the device sessions to the VX8s in the system. It then deallocates the system structure.

For VISA VXIpnp applications, you should call **ssVX8_close** instead of **ssVX8_SystemClose**.

ssVX8_SystemErrorMessage

Function Gets a text description for a given error code.

Include File vx8.h

Syntax

```
RESULT _SS_FUNC ssVX8_SystemErrorMessage (VXIDEV_HANDLE   vi,
                                          RESULT error,
                                          STRING message);
```

Parameters

vi	VX8 IO handle
error	Return code for which to get text description
message	Text description for *error*

Returned Values

SUCCESS	Successfully reported driver and instrument firmware revisions.
SSVX8_ERROR_ERROR_MESSAGE_UNKNOWN	Unknown error value

Remarks This function only returns text descriptions for errors returned by the VX8 Instrument Driver.

ssVX8_SystemRevisionQuery

Function Obtains the firmware revision number for the specified VX8 device.

Include File vx8.h

Syntax

```
RESULT _SS_FUNC ssVX8_SystemRevisionQuery (SYSHANDLE VX8_System
                                           STRING boardName,
                                           STRING driver_rev,
                                           STRING instr_rev);
```

Parameters

VX8_System	VX8 system handle
boardName	Name of the VX8 device
driver_rev	Pointer to storage for the driver revision description
instr_rev	Pointer to storage for the instrument firmware revision description

Returned Values

SUCCESS	Successfully reported driver and instrument firmware revisions.
SSVX8_ERROR_READFIRMWAREREV_TIMEOUT	The Node A DSP on a VX8 in the system has failed to return its firmware revision number or a VX8 board in the system has failed.
SSVX8_ERROR_SYSTEM_STRUCT_UNINITIALIZED	The system structure was uninitialized.
SSVX8_ERROR_NO_SSVX8_FOUND	The VX8 specified by boardName was not found.

Remarks The instrument driver revision is copied into *driver_rev* and the firmware revision of the VX8 indicated by boardName is copied into *instr_rev*.

The firmware revision will be valid only if the application on the Node A processor has been built with the Boot_IIOF3Isr ISR installed to service the IIOF3 interrupt; a timeout will occur otherwise. Both *driver_rev* and *instr_rev* should be pointers to allocated strings of at least 8 characters.

ssVX8_SystemGetDriverRev

Function	Gets the instrument driver revision.
Include File	vx8.h
Syntax	void _SS_FUNC ssVX8_SystemGetDriverRev (PUINT8 *driver_revision*);
Parameters	*driver_revision* — Pointer to storage for the driver revision description
Returned Values	`VI_SUCCESS` — Successfully reported driver and instrument firmware revisions.
Remarks	The calling function must allocate space for the driver revision. The storage required is 8 characters.

ssVX8_SystemLoadCode

Function Loads DSP application software onto the processors in a VX8 system.

Include File vx8.h

Syntax

```
RESULT _SS_FUNC ssVX8_SystemLoadCode (SYSHANDLE VX8_System,
                                      STRING loadDefFile,
                                      UINT32  flags);
```

Parameters

VX8_System	VX8 system handle
loadDefFile	Path and filename for a Load Definition File. Specify “” if no override is required
flags	Flags described below

Flag	Value	Description
SSVX8_SYSLOADCODE_NOCHECKCONFIG	0x0	Does not check /CONFIG line status after loading.
SSVX8_SYSLOADCODE_CHECKCONFIG	0x2	Performs a **ssVX8_SystemCheckConfig** call after loading the system.

Returned Values

VI_SUCCESS	Successfully loaded the system.
SSVX8_ERROR_NODEA_TIMEOUT	An error occurred in loading (Node A is no longer responding).
SSVX8_ERROR_PROCLOADCODE_COFFPARSE_FAILED	An error occurred during the parsing of a C4x COFF file.
SSVX8_ERROR_VX8_STILL_IN_CONFIG	At least one DSP in the system is still asserting CONFIG*.
SSVX8_ERROR_LDFOPENLDF_FAILED	An error occurred during the parsing of the LDF.
SSVX8_ERROR_SYSTEM_RESET_TIMEOUT	One of the VX8 devices in the system failed to initialize after being reset.
SSVX8_ERROR_READFIRMWAREREV_TIMEOUT	Timed out waiting to read a VX8 firmware revision.
SSVX8_ERROR_READSELFTESTRES_TIMEOUT	Timed out waiting to read a VX8 self test result.
SSVX8_ERROR_NO_VX8_IN_SYSTEM_STRUCT	No VX8 devices were found in the system definition.

Remarks The software to be loaded is specified in a Load Definition File (LDF). If no LDF is specified here, the LDF that was previously parsed by **ssVX8_SystemOpen** will be used. If an LDF is specified, it will override the previously parsed LDF and load the DSP applications specified by this new LDF. The new LDF will also serve as the new software definition if the ssVX8_SystemLoadCode is invoked again without an LDF defined.

If an LDF was **not** specified in the previous call to **ssVX8_SystemOpen**, an LDF must be specified here. See *Section 9.4.4* for more information about the LDF format.

Invoking this function resets the VX8 system. This will abruptly terminate all applications currently running on the system. If your application requires to be “gracefully” terminated, you should implement a scheme to halt activities on the processors on the system before calling the system load code function.

In order to for the CONFIG* functionality to work, all processors must be loaded with code which will at least de-assert the CONFIG* line on the particular site. See **ssVX8_SystemCheckConfig** for details.

The loading mechanism for the VX8 uses COMM ports to transfer the program information to the DSPs in the system. Node A will be running the Boot Initialization kernel. All processors will be loaded with a small kernel which will redirect load information through the COMM ports to reach a target processor. Refer to *Section 5.3* for the resources used by the loading mechanism on the C4x DSPs. It is imperative that all C4x DSP applications do not initiate COMM port communications until the applications are completely loaded.

The COMM port information in the SDF is used to determine the COMM port path by which each DSP in the system will be loaded. The SDF COMM port information for processors, TIM-40 modules, front panel connections, and the carrier board must be correct for the loading mechanism to correctly function.

Processors which are not to be loaded must be defined in the SDF as having the attribute `BootPort = NONE;` so that these processors are not included during the generation of boot path information. It is not sufficient to simply omit the processor from the LDF as the loading algorithm may use the processors to load others via COMM port. This is required for processor-less TIM-40 modules or for processors booting from ROM.

ssVX8_SystemOpen

Function Opens a VX8 System and optionally resets the system or loads DSP software onto the system.

Include File vx8.h

Syntax RESULT _SS_FUNC ssVX8_SystemOpen (SYSHANDLE **VX8SystemHndlPtr*,
STRING *SysDefFile*,
STRING *LoadDefFile*,
UINT32 *flags*);

Parameters

VX8SystemHndlPtr	Pointer to VX8 system handle storage
SysDefFile	Path and filename for the System Definition File
LoadDefFile	Path and filename for a Load Definition File. Specify “” if you are specifying an LDF in a subsequent call to **ssVX8_SystemLoadCode**
flags	Flags described below

Flag	Value	Description
SSVX8_SYSOPEN_NOACTION	0x0	Does not reset or load code on to the system.
SSVX8_SYSOPEN_LOADSYSTEM	0x1	Performs an **ssVX8_SystemLoadCode** call.
SSVX8_SYSOPEN_RESETSYSTEM	0x2	Resets the boards in the system.

Returned Values

VI_SUCCESS	Successfully opened the VX8 system.
SSVX8_ERROR_NODEA_TIMEOUT	An error occurred during loading (Node A is no longer responding).
SSVX8_ERROR_PROCLOADCODE_COFFPARSE_FAILED	An error occurred during the parsing of a C4x COFF file.
SSVX8_ERROR_VX8_STILL_IN_CONFIG	At least one DSP in the system is still asserting CONFIG*.
SSVX8_ERROR_LDFOPENLDF_FAILED	An error occurred during the parsing of the LDF.
SSVX8_ERROR_SDFOPENSDF_FAILED	An error occurred during the parsing of the SDF.
SSVX8_ERROR_SYSTEM_RESET_TIMEOUT	One of the VX8 devices in the system failed to initialize after being reset.
SSVX8_ERROR_READFIRMWAREREV_TIMEOUT	Timed out waiting to read a VX8 firmware revision.
SSVX8_ERROR_READSELFTESTRES_TIMEOUT	Timed out waiting to read a VX8 self test result.
SSVX8_ERROR_NO_VX8_IN_SYSTEM_STRUCT	No VX8 devices were found in the system definition.
SSVX8_ERROR_BAD_RSRC_DESCRIPTOR	A VX8 in the SDF does not exist in the chassis (check your LA settings).
SSVX8_ERROR_NO_SSVX8_FOUND	A device in the SDF is NOT a VX8.
SSVX8_ERROR_MEMORY_ALLOCATION	Failed to map memory through the I/O library.

Remarks This is the main initialization routine for the VX8 system. The VXIpnp applications should call **ssVX8_open** (after calling **ssVX8_init**) instead of **ssVX8_SystemOpen**.

If the SSVX8_SYSOPEN_LOADSYSTEM flag is indicated, the **LoadDefFile** parameter must indicate a valid LDF and **ssVX8_SystemLoadCode** will be called internally by this function. The VX8 System will be reset before the DSP application software is loaded.

If the SSVX8_SYSOPEN_LOADSYSTEM flag is not indicated, the VX8 system will not be loaded. This option should only be used if you plan on loading the system at a later time with a call to **ssVX8_SystemLoadCode** or are opening a system which has already been loaded.

If the SSVX8_SYSOPEN_RESETSYSTEM flag is indicated, **ssVX8_SystemReset** will be called internally by this function. If SSVX8_SYSOPEN_LOADSYSTEM is also indicated, the reset will occur before the DSP software loading.

The calling function must allocate memory for the system handle.

A device session to each VX8 system is open and stored within the system structure which pointed to by VX8SystemHndlPtr. Use **ssVX8_Deref** to obtain the device session for a particular VX8 in the system. This handle can then be used to call I/O library functions to access the device.

Note that there is a mapping behavior difference between the SICL and VISA device session handles. VISA session handles only allow a single mapping and any attempts to map an already mapped session will block until the previous mapping is removed. SICL session handles allow multiple mappings to different address spaces with a single session handle.

It is possible for another host session to open the same system with the same SDF. This will allow multiple host applications to communicate to the VX8s in the system. Only one application can reset or load the VX8 system and other applications must not attempt to access the devices during the time when the VX8 system is being reset or loaded.

Refer to sections *9.4.2* and *9.4.3* for information on the SDF and LDF formats.

ssVX8_SystemRead

Function Reads a block of data from a specified VX8 device into memory on the host. The data format is in long words (D32).

Include File vx8.h

Syntax

```
RESULT _SS_FUNC ssVX8_SystemRead (SYSHANDLE VX8_System,
                                  STRING board_name,
                                  PUINT32 dst,
                                  PUINT32 src,
                                  UINT32 length,
                                  UINT32 flags);
```

Parameters

VX8_System	VX8 system handle
board_name	Unique symbolic name of a resource
dst	Destination address on the host
src	Source offset (relative to flags)
length	Length of block in long words (D32)
flags	Flags described on the next page

Returned Values

VI_SUCCESS	Success
SSVX8_ERROR_RW_IO_FAILED	I/O library read/write error
SSVX8_ERROR_RW_BAD_VX8_ADDRESS_RANGE	Bad DSP address or length
SSVX8_ERROR_MEMORY_ALLOCATION	Mapping error
SSVX8_WARN_RW_ZERO_LENGTH	Length is zero

Remarks Refer to the *VX8 Carrier Board Technical Reference Manual* for the VX8's A32 slave memory map.

Always keep in mind that the host views the VX8 as being byte-addressed. The Host code must increment the address by 4 (bytes) for consecutive memory locations on the VX8. All DSP defined offsets must be multiplied by 4 (or left shift by 2) to be used as respective offsets on the host.

The syntax for *board_name* will differ depending on whether your I/O library is SICL or VISA. For SICL, the syntax will resemble "vxi,240". For VISA, the syntax will resemble "VXI0::240::INST". Refer to the I/O library documentation for the proper resource descriptor nomenclature.

The SICL implementation by default maps 64k at a time. ssVX8_SystemRead maps and unmaps the memory window for transfers which span 64k pages. The Instrument Driver map window size is set at compile time by the SSVX8_RW_WIN_NUM_64K_PAGES define in **ssvx8brd.c**. If your SICL application performs transfers of more than 64k, the instrument driver may be more efficient if the number of 64k pages mapped is increased. This will require you to modify this define and recompile the instrument driver.

Accessing C4x global memories from the host is intrusive, especially if your DSP application uses the Global Shared Bus extensively. Using the shared DRAM to buffer data will isolate the host from colliding with C4x and HP BALLISTIC traffic on the global shared bus.

Depending on your host interface, it may be more efficient to use DMA from the host or through the SCV64 DMA controller to perform data transfers.

Note that reads to any broadcast areas will return invalid data.

The following table lists the flags that can be specified.

Flag	Value	Description
SSVX8_RW_NO_OFFSET	0x00000000	src relative to the A32 base address
SSVX8_RW_TBC	0x01000000	src relative to the test bus controller base
SSVX8_RW_NODEA	0x02000000	src relative to Node A Global base
SSVX8_RW_NODEB	0x03000000	src relative to Node B Global base
SSVX8_RW_NODEC	0x04000000	src relative to Node C Global base
SSVX8_RW_NODED	0x05000000	src relative to Node D Global base
SSVX8_RW_NODEE	0x06000000	src relative to Node E Global base
SSVX8_RW_NODEF	0x07000000	src relative to Node F Global base
SSVX8_RW_NODEG	0x08000000	src relative to Node G Global base
SSVX8_RW_NODEH	0x09000000	src relative to Node H Global base
SSVX8_RW_DRAM	0x0F000000	src relative to DRAM base

ssVX8_SystemReset

Function	This function will reset all of the VX8 devices in a system..
Include File	vx8.h
Syntax	RESULT _SS_FUNC ssVX8_SystemReset (SYSHANDLE *VX8_System*);
Parameters	*VX8_Sytem* VX8 system handle
Returned Values	VI_SUCCESS Successfully reset the VX8 system

Remarks

Invoking this function resets the VX8 system. This will abruptly terminate all applications currently running on the system. If your application requires to be “gracefully” terminated, you should implement a scheme to halt activities on the processors on the system before calling this function. A call to **ssVX8_SystemLoadCode** will be required to reload DSP applications.

When a VX8 system is reset, all VX8 devices defined in the SDF that was opened with the **ssVX8_SystemOpen** call are reset whether they are connected via the front panel COMM port/JTAG or not.

ssVX8_SystemWrite

Function Writes a block of data from memory on the host to a specified VX8 device.

Include File vx8.h

Syntax

```
RESULT _SS_FUNC ssVX8_SystemWrite (SYSHANDLE VX8_System,
                                   STRING board_name,
                                   PUINT32 dst, PUINT32 src,
                                   UINT32 length, UINT32 flags);
```

Parameters

VX8_System	VX8 system handle
board_name	Unique symbolic name of a resource
dst	Destination offset (relative to flags)
src	Source address of data on the host
length	Length of block in long words (D32)
flags	Flags described below

Flag	Value	Description
SSVX8_RW_NO_OFFSET	0x00000000	dst relative to the A32 base address
SSVX8_RW_TBC	0x01000000	dst relative to the test bus controller base
SSVX8_RW_NODEA	0x02000000	dst relative to Node A Global base
SSVX8_RW_NODEB	0x03000000	dst relative to Node B Global base
SSVX8_RW_NODEC	0x04000000	dst relative to Node C Global base
SSVX8_RW_NODED	0x05000000	dst relative to Node D Global base
SSVX8_RW_NODEE	0x06000000	dst relative to Node E Global base
SSVX8_RW_NODEF	0x07000000	dst relative to Node F Global base
SSVX8_RW_NODEG	0x08000000	dst relative to Node G Global base
SSVX8_RW_NODEH	0x09000000	dst relative to Node H Global base
SSVX8_RW_ALLNODES	0x0A000000	dst relative to Broadcast ALL base
SSVX8_RW_CDEFGH	0x0B000000	dst relative to Broadcast TIM sites base
SSVX8_RW_CDGH	0x0C000000	dst relative to Broadcast C, D, G, & H base
SSVX8_RW_EFGH	0x0D000000	dst relative to Broadcast E, F, G, & H base
SSVX8_RW_CD	0x0E000000	dst relative to Broadcast C & D base
SSVX8_RW_DRAM	0x0F000000	dst relative to DRAM base

Returned Values

VI_SUCCESS	Success
SSVX8_ERROR_RW_IO_FAILED	I/O library read/write error
SSVX8_ERROR_RW_BAD_VX8_ADDRESS_RANGE	Bad DSP address or length
SSVX8_ERROR_MEMORY_ALLOCATION	Mapping error
SSVX8_WARN_RW_ZERO_LENGTH	Length is zero

Remarks Refer to the *VX8 Carrier Board Technical Reference Manual* for the VX8's A32 slave memory map.

Keep in mind that the host system views the VX8 as being byte-addressed. Therefore, the Host code must increment the address by 4 (bytes) for consecutive memory locations on the VX8. All DSP defined offsets must be multiplied by 4 (or left shift by 2) to be used as respective offsets on the host.

The syntax for board_name differs depending on whether your I/O library is SICL or VISA. For SICL, the syntax resembles “vxi,240”. For VISA, the syntax will resemble “VXI0::240::INST”. Refer to the I/O library documentation for the proper resource descriptor nomenclature.

The SICL implementation by default maps 64k at a time. ssVX8_SystemRead maps and unmaps the memory window for transfers which span 64k pages. The Instrument Driver map window size is set at compile time by the SSVX8_RW_WIN_NUM_64K_PAGES define in **ssvx8brd.c**. If your SICL application performs transfers of more than 64k, the instrument driver may be more efficient if the number of 64k pages mapped is increased. This will require you to modify this define and recompile the instrument driver.

Accessing C4x global memories from the host is intrusive, especially if your DSP application uses the Global Shared Bus extensively. Using the shared DRAM to buffer data will isolate the host from colliding with C4x and HP BALLISTIC traffic on the global shared bus.

Depending on your host interface, it may be more efficient to use DMA from the host or through the SCV64 DMA controller to perform data transfers.

11 Example Software

This chapter presents examples of specific routines that can be used to program the VX8 Carrier Board. The examples demonstrate how to perform interrupt handling and data transfers (+DMA) on the VXIbus, HP Local Bus, and the RS-232 DUART.

The example software can be found in the *examples* directory of the *C4x Support Software* directory. The mult.c example can also be found in the *examples* directory of the *VX8 Instrument Driver Software* directory. Refer to chapters 3 and 4 of the *VX8 Carrier Board Installation Guide* for information on how to install and locate the example programs

Note that for HP-UX 9.0x SICL, you can use the **ivxisc** utility to obtain information about your VXIbus system configuration. If you're not using HP-UX 9.0x SICL, check your controller's documentation for a utility that displays your VXIbus system's configuration.

MDC44ST 'C44-based examples are contained in the **examples.c44** directory of the C4X Support Software directory. The contents of this directory are similar to the files in the directory for the 'C40 DSPs. The only difference lies in the DSP software building process in which the **vx8c44ss.lib** file is linked instead of the **vx8c40ss.lib** file used in the examples directory. This is specified in the **vx8tim.cmd** linker command file in **examples.c44**. The -v44 flag is also used by the compiler shell batch file **ticompt.bat** in **examples.c44**. All of the software support functions and instrument driver functions in these examples use a header file called **vx8.h**.

11.1. Floating Point Multiplication Example

The **mult.c** example demonstrates how the top level instrument driver function calls are used. Specifically, how they are used to open a VX8 system, download code, and get data back from a DSP application. The two floating point arguments that you provide in **mult.c** are multiplied by the VX8, and the resulting value is then returned back to the host in order to be displayed.

The C4x component of the floating point multiplication example, **vx8mult.c,** demonstrates the basics of building a TMS320C4x application using the VX8 C4x Support Software Library. The **VX8_Read** and **VX8_Write** calls are used in the **vx8mult.c** example to move data on the VX8's Global Shared Bus.

vx8mult.c and **mult.c** together demonstrate the basics of using the instrument driver and the C4x library in your VX8 application. The multiplication example uses only the basic functionality of the VX8. To see how you can use other interfaces on the VX8, refer to the other examples provided with the VX8 Support Software.

The floating point and control data are passed through the VX8's memory interface through SRAM (or through DRAM with a #define change in **vx8mult.h**). The file

vx8mult.h demonstrates how to construct cross platform or common header files for host and DSP application interactions included by both the DSP and host software.

System Configuration

VXIbus Configuration

Controller	HP E1497A - v743
Operating System	HP-UX version 9.0
I/O Library	HP SICL version 3.0
Mainframe	HP 75000 Series C
VX8 Logical Address	240
VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32 No TIM-40 modules DRAM optional

PC Configuration

PC	At least 4Mbytes of memory (16Mbytes recommended) Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
Operating System	MS-DOS or Win95

Running the Program

The **mult.c** example should be executable as is. If you encounter problems executing **mult.c**, you may have to recompile the host **mult.c** code. If the SDF and ssvx8 libraries are not in your C environment, you will have to include the directory with the "**-**L<*pathname*>" option.

On an HP-UX host, compile the **mult.c** host application by entering the following command:

```
cc -o mult -Aa -g -v mult.c -D_HPUX -D_SICL -lssvx8 -lsdf -lsicl
```

The operation is performed by the batch file **mmult**, which is a UNIX script.

The C4x code, **vx8mult.c**, has been compiled and linked to generate the COFF file VX8MULT.OUT. To rebuild **vx8mult.c**, simply run **makemult.bat** in your C4x development environment. This batch file calls **ticompa.bat**, which compiles **vx8mult.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to include the respective directories in the VX8 C4x software.

The System Definition File (SDF), **mult.sdf**, specifies the VX8 system configuration of a single VX8, with no TIM-40 modules, no front panel COMM port connections, and a "vxi,240" (logical address is 240) I/O resource descriptor.

The Load Definition file (LDF), **mult.ldf**, specifies that VX8MULT.OUT is to be loaded onto Node A of Board1 defined in **mult.sdf**.

Program Description

The multiplication program example **mult.c** performs the following actions:

- Opens the VX8 System
- Initializes the MULT control register
- Loads DSP software onto the VX8
- Asks you to input 2 floating point numbers
- Writes the numbers to the VX8 and asks the DSP to perform the multiplication
- The VX8 performs the multiplication (with the necessary IEEE to TI floating point conversions), places the result in the result memory location, and signals the host that the transaction is done.
- The host reads and displays the result, then asks for a key hit to continue.

11.2. Node B DUART and C4x ISR Example

The **VX8duart.c** example is a simple C4x program that shows you how to set up the dual UARTs (DUARTs) on the VX8 Node B processor and how to install an interrupt service routine to handle incoming UART data. This example program is meant to be run **only** on Node B and should be loaded via JTAG so that you can watch the program as it executes.

The front panel RS-232 connectors must be connected using a NULL modem cable.

The main program of the **VX8duart.c** example writes a ramp of data to channel 1 of the DUART and the data is received on channel 2 via the NULL modem cable. An ISR is set up to receive the data on channel 2. The ISR simply generates a sum of all data received through the UART channel 2.

This is done by first installing an ISR for the UART interrupt line. The UART interrupt will trigger when any data is received. The ISR simply adds the data in the block to the current sum. The static variable "verified" in the UART ISR should be equal to the block length "LENGTH". At the end of the ISR, the DUART interrupt is disabled once the entire block is received.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM optional
		Front panel RS-323 connectors connected by a NULL modem cable (required)
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The DUART example should be run from JTAG on Node B. The C4x code, **vx8duart.c**, has been compiled and linked to generate the COFF file VX8DUART.OUT. To rebuild **vx8duart.c**, simply call **makeuart.bat**. This batch file calls **ticompb.bat**, which compiles **duart.c** and links it with the C boot routine **boot_b.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8duart.c**, start the debugger and type:

```
> load vx8duart.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

Put the variable sum in the watch window using the command

```
> wa sum
```

After the execution, the value “sum” should be 45.

Program Description

The DUART program example **vx8duart.c** performs the following actions:

- Sets up the baud rate and data formats for channel 1 and channel 2 on the DUART.
- Disables interrupts, installs the UART_ISR to service Node B's IIOF3 interrupt, and then re-enables the interrupts.
- Writes Data out to channel 1 on the DUART
- When the ISR is triggered, the data is read from channel 2 and summed.

11.3. C4x HP Local Bus Consume Mode and ISR Example

The **vx8hpcon.c** example demonstrates how to set up the VX8 in order for it to consume data from the HP Local Bus, and how to use the HP Local Bus interrupt on IIOF1 to process the data.

A second HP Local Bus device is required to be placed upstream (to the left) of the VX8 that's running this example code. The block size of the data should match the block length specified in this example. A second VX8 running the **vx8hpgen.c** example (see next example) can be used to serve as an HP Local Bus data source.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM optional
	2nd HP Local Bus board	Another HP Local Bus board set up to generate data. Block size should match the length set in this example. A second VX8 running the vx8hpgen.c example can be used to serve as an HP Local Bus data source.
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The **vx8hpcon.c** example is to be run from JTAG. It is compiled with the appropriate C initialization routines for Node A. The C4x code, **vx8hpcon.c**, has been compiled and linked to generate the COFF file VX8HPCON.OUT. To rebuild **vx8hpcon.c**, simply call **makecons.bat**. This batch file calls **ticompa.bat**, which compiles **vx8hpcon.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8hpcon.c**, start the debugger and type:

```
> load vx8hpcon.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

Put the variable sum in the watch window using the command

```
> wa sum
```

After the execution, the value “sum” should be 240.

Program Description

The example first installs the ISR for the HP interrupt line (II0F1) and enables End of Block (EOB) interrupt from the BALLISTIC. The ISR simply performs a sum of all the data received.

The HP BALLISTIC IC is reset and configured to be in consume mode. The HP Local Bus Read DMA is then configured by setting the block size and DMA targets. The HP Local Bus Read DMA is then enabled and the BALLISTIC is restarted. When a block is received (determined by an EOB), it generates an interrupt and the ISR will sum the data in the buffer just written. The HP Local Bus interrupt is then cleared at the end of the ISR.

11.4. C4x HP Local Bus Generate Mode Example

The **vx8hpgen.c** example demonstrates how to set up the VX8 in order for it to write data (generate) to the HP Local Bus. This example uses the **VX8_HPWriteFIFO** function to move the data from Node A’s memory to the HP Local Bus.

A second HP Local Bus device is required to be placed downstream (to the right) of the VX8 that’s running this example code. The block size of the data should match the block length specified in this example. A second VX8 running the **vx8hpcon.c** example (see previous example) can be used to serve as an HP Local Bus data sink.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM optional
	2nd HP Local Bus board	Another HP Local Bus board set up to consume data. Block size should match the length set in this example. A second VX8 running the vx8hpcon.c example can be used to serve as an HP Local Bus data sink.
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The **vx8hpgen.c** example is to be run from JTAG. It is compiled with the appropriate C initialization routines for Node A. The C4x code, **vx8hpgen.c**, has been compiled and linked to generate the COFF file VX8HPGEN.OUT. To rebuild **vx8hpgen.c**, simply call **makegen.bat**. This batch file calls **ticompa.bat**, which compiles **vx8hpgen.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8hpgen.c**, start the debugger and type:

```
> load vx8hpgen.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

Program Description

The HP BALLISTIC IC is reset then configured. The BALLISTIC is set to GENERATE mode and the block size is set. **vx8hpgen.c** then generates 2 blocks of 16 long words onto the HP Local bus.

It is possible to implement and set up the VX8 to have an HP Local Bus interrupt handler to continually write data out to the HP Local Bus. The Write FIFO Almost Full (WAF) and Write FIFO Almost Empty (WAE) interrupts can be enabled to trigger the HP Local Bus interrupt.

11.5. C4x IIOF2 Interrupt Matrix Example

The **vx8iiof2.c** example shows you how to use the IIOF2 interrupt matrix accessible from the Node A processor. The IIOF2 matrix is set up by Node A so that all node IIOF2 lines are connected on matrix line number 7.

It is left to developers to write and install ISRs on IIOF2 of other nodes. You can use JTAG to examine the state of the IIOF2 lines from the other nodes by extending the IIOF2 0 assertion delay in this program.

Caution: Nodes that are not running the sample code **must not** have their IIOF2 lines set as GPIO outputs. Damage can be caused to the C4x DSPs if two outputs are driving one another.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Logical Address	240
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM optional
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The **vx8iiof2.c** example should be run from JTAG. It is compiled with the appropriate C initialization routines for Node A. The C4x code, **vx8iiof2.c**, has been compiled and linked to generate the COFF file VX8IIOF2.OUT. To rebuild **vx8iiof2.c**, simply call **makeint.bat**. This batch file calls **ticompa.bat**, which compiles **vx8iiof2.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8iiof2.c**, start the debugger and type:

```
> load vx8iifo2.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

After the execution, all other nodes should have their interrupt line asserted. This can be verified by observing bit 10 of the IIF register via JTAG; a “1” indicates that the interrupt is asserted.

Program Description

The program first disables the IIOF2 interrupt matrix. The IIOF2 pin on Node A is then set up as a GPIO output and asserted (1). Then the IIOF2 interrupt matrix is enabled with all sites connected on line 7. Then the IIOF2 line is toggled low (assert interrupt).

11.6. SCV64 Interrupt and Location Monitor Example

This example demonstrates how to use the **VX8_SCV64AckInterrupt** function to handle interrupts from the SCV64 (specifically, the SCV64 Location Monitor (LM) interrupt.)

The Location Monitor (LM) is a 31 stage deep FIFO that can be accessed by writing to the top memory location of the A32 space configured on the VX8. The LM can be written to from either the VXIbus or locally from the DSPs by writing to this upper long word address on the board’s A32 space. Refer to the *SCV64 User Manual - VMEbus Interface Components Manual* and the *VX8 Carrier Board Technical Reference Manual* for more information about the LM.

The **vx8lm.c** example sets up and enables the Location Monitor interrupt on the SCV64, writes to the Location Monitor in the A32 address space, and then waits to count the number of interrupts received.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Logical Address	240
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM optional
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The **vx8lm.c** example should be run from JTAG. It is compiled with the appropriate C initialization routines for Node A. The C4x code, **vx8lm.c**, has been compiled and linked to generate the COFF file VX8LM.OUT. To rebuild **vx8lm.c**, simply call **makelm.bat**. This batch file calls **ticompa.bat**, which compiles **vx8lm.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8lm.c**, start the debugger and type:

```
> load vx8lm.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

Put the variable hit in the watch window using the command

```
> wa hit
```

After the execution, the value "hit" should be 1.

Program Description

The SCV_ISR is installed to handle the LM interrupts and the location monitor interrupt is enabled on the SCV64. The main program will then wait forever for LM interrupts.

When a location monitor interrupt is received, the ISR acknowledges the interrupt by reading the location monitor FIFO and incrementing the number of location monitor interrupts received.

Again, this example requires another device to write the location monitor on the SCV64 (the uppermost long word in the VX8's A32 memory).

11.7. VXIbus (SCV64) DMA Example

The **vx8dma.c** example demonstrates how to perform SCV64 DMA memory transfers. The example sets up a single SCV64 A32/D32 DMA to another VX8.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Board 1 Logical Address	240
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM (minimum of 8 Mbytes)
	VX8 Board 2 Logical Address	241
	2nd VX8 board	Another VX8 at a base address of 0x28000000
		No TIM-40 modules
		DRAM (minimum of 8 Mbytes)
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The **vx8dma.c** example should be run from JTAG. It is compiled with the appropriate C initialization routines for Node A. The C4x code, **vx8dma.c**, has been compiled and linked to generate the COFF file VX8DMA.OUT. To rebuild **vx8dma.c**, simply call **makedma.bat**. This batch file calls **ticompa.bat**, which compiles **vx8dma.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The

TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8dma.c**, start the debugger and type:

```
> load vx8dma.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

Put the variables NbInt and NbErr in the watch window using the commands

```
> wa NbInt
> wa NbErr
```

After the execution, the value “NbInt” should be 2 and the value “NbErr” should be 0.

Program Description

The **vx8dma.c** example is set up to perform a single SCV64 A32D32 DMA transfer. The source address, destination address, direction, and length are defined as global variables.

An ISR is set up to handle the SCVDMA interrupt that will trigger when the transfer is complete. The example also measures the amount of time the transfer takes to complete and calculates the transfer rate in Mbytes/sec. Then the program:

- Starts a DMA write to the second board
- Waits for the DMA to finish its transfers
- Starts a DMA read from the second board

When the second transfer is done, the ISR compares the sent and received data.

This example requires another A32 VXIbus device to write to. The example is set up to write to shared DRAM on a second VX8 board at a base address of 0x28000000.

Below is the calculation used to determine the DRAM address on the second VX8 board assumed to be at a base address of 0x28000000.

```
        0x28000000     ( offset of the second board in A32 address space )
    +   0x04000000     ( address of DRAM in the VXIbus slave memory map )
===============
        0x2c000000
```

Note: The VXIbus addresses used for SCV64 DMA transfers differ from the addresses required to directly master the VXIbus. Direct VXIbus mastering requires converting the desired VXIbus address to a long word address which resides in the C4x DSP global memory map.

The SCV64 DMA VXIbus (VMEbus) address is simply the desired VXIbus address with no required translation. The SCV64 DMA Local address is based on the VX8 A32 slave image, not the C4x global memory map.

The **VX8_SCV64DMATransfer** source address parameter is the VXIbus slave address offset of shared DRAM with respect to the base address of the VX8. Shared DRAM starts at an offset of 0x04000000. Refer to the *VX8 Technical Reference manual* for a complete VX8 VXIbus A32 slave memory map.

11.8. VX8 VXIbus Master Example

The **vx8vxi.c** example demonstrates how to use the C4x to directly master the VXIbus. Mastering the VXIbus is accomplished by a simple call to **VX8_Write** or **VX8_Read**. Data is written from a data buffer in the DSP's RAM to the target VX8's DRAM and read back into a second buffer. The two buffers are then compared.

This example requires an additional VX8 to serve as an A32 target. The software has been set up for the second VX8 to be at a base address of 0x28000000 in A32 and configured with at least 8 Mbytes of DRAM.

System Configuration

VXIbus Configuration	Controller	HP E1497A - v743
	Operating System	HP-UX version 9.0
	I/O Library	HP SICL version 3.0
	Mainframe	HP 75000 Series C
	VX8 Logical Address	240
	VX8 Configuration	Embedded near global memory 128k X 32 or 512k X 32
		No TIM-40 modules
		DRAM optional
	2nd VX8 Logical Address	241
	2nd VX8 board	Another VX8 at a base address of 0x28000000
		No TIM-40 modules
		DRAM (minimum of 8 Mbytes)
PC Configuration	PC	At least 4Mbytes of memory (16Mbytes recommended)
		Texas Instruments XDS510 or Spectrum's XDSC40 for C4x debugging.
	Operating System	MS-DOS or Win95

Running the Program

The **vx8vxi.c** example should be run from JTAG. It is compiled with the appropriate C initialization routines for Node A. The C4x code, **vx8vxi.c**, has been compiled and linked to generate the COFF file VX8VXI.OUT. To rebuild **vx8vxi.c**, simply call **makevxi.bat**. This batch file calls **ticompa.bat**, which compiles **vx8vxi.c** and links it with the C boot routine **boot_a.asm** and the VX8 C4x Support Software library. The TMS320C4x compiler environment may have to be set to point to the respective directories in the VX8 C4x software.

To run **vx8vxi.c**, start the debugger and type:

```
> load vx8vxi.out
```

You may be required to change directories or set up the debugger environment variables to load the code.

Put the variable "errors" in the watch window using the command

```
> wa errors
```

After the execution, the value "errors" should be 0.

Program Description

The example first installs the SCV64 ISR to handle the unlikely occurrence of bus errors from the SCV64. It then initializes both data buffers to different values. The data from buf1 is then written to the other board with a call to **VX8_Write** and read back into buf2 with a call to **VX8_Read**. The two buffers are compared for differences and the number of errors is summed.

Pay particular attention to how the address of the second board is calculated. The VXIbus address that you want to access must be divided by 4 and then added to the base of the VXIbus image at 0xC0000000 in order to generate the appropriate long word address suitable for use by the DSP.

The address of the DRAM on the second VX8 has been computed in the following manner:

```
          0xC0000000    (  beginning of VXIbus space )
+    (    0x28000000    (  offset of the second board in the A32 address space )
          0x04000000    (  address of DRAM in the VXIbus slave memory map )
     ) / 4
===============
          0xCB000000
```

Note: The VXIbus addresses used for SCV64 DMA transfers differ from the addresses required to directly master the VXIbus. Direct VXIbus mastering requires converting the desired VXIbus address to a long word address which resides in the C4x DSP global memory map.

The SCV64 DMA VXIbus (VMEbus) address is simply the desired VXIbus address with no translation required. The SCV64 DMA Local address is based on the VX8 A32 slave image, not the C4x global memory map.

Appendix A: Status Codes

This appendix lists the possible VX8 status codes. *Table 19 Status Codes for DSP functions* lists the status codes for the C4x Support Software functions; *Table 20 Status codes for Host functions* lists the status codes for the Host Software functions.

Table 19 Status Codes for DSP functions

Error Number (Base 16)	Status Code	Description
0	DUART_SUCCESS	The interrupt(s) were successfully enabled or disabled.
0	HPBUS_SUCCESS	The HP Local Bus peripheral(s) was reset and/or programmed successfully. For VX8_HPClearInterrupt, the interrupt was successfully cleared.
0	SCV64_SUCCESS	The SCV64 operation completed successfully.
0	VX8_SUCCESS	The operation completed successfully.
65	VX8_ERROR_INVALID_PARAMETER	Invalid parameter.
66	VX8_ERROR_INVALID_ADDRESS	Invalid source or destination address.
67	VX8_ERROR_UNLOCK_MISMATCH	A call to VX8_Unlock was made without a successful matching call to VX8_Lock.
C9	SCV64_ERROR_INIT_DELAY	SCV64 delay line calibration failed.
CA	SCV64_ERROR_OFFSET_VALUE	Bad A32 slave address.
CB	SCV64_ERROR_BUS_LEVEL_VALUE	The bus request level was out of range.
CC	SCV64_ERROR_BUS_TIMEOUT_VALUE	The ownership timeout value was out of range.
CD	SCV64_ERROR_LOCAL_ACCESS	An access error has occurred.
CE	SCV64_ERROR_BAD_IACK	An access error occurred while running the IACK cycle.
CF	SCV64_ERROR_BUS_INT_ACTIVE	VXIbus interrupt is already active.
D0	SCV64_ERROR_BUS_INT_LEVEL_VALUE	A bad local interrupt occurred.

Error Number (Base 16)	Status Code	Description
D1	SCV64_ERROR_DMA_ACTIVE	VXIbus DMA transfer currently in progress.
D2	SCV64_ERROR_UNALIGNED_DMA_TRANSFER	SCV64 DMA does not support unaligned transfers.
012D	HPBUS_ERROR_INVALID_BLOCK_SIZE	The block size is greater than 1024.
012E	HPBUS_ERROR_FIFO_FLAG_VALUE	A FIFO flag level is out of range.
012F	HPBUS_ERROR_FIFO_INIT	Error in programming the FIFO flag levels. The FIFOs are in an unknown state.
0130	HPBUS_ERROR_WAIT_STATE_VALUE	Wait state value out of range.
0131	HPBUS_ERROR_BUF_ADDR_VALUE	Invalid destination address.
0132	HPBUS_ERROR_FLAG_VALUE	The transfer flags are invalid.
0133	HPBUS_ERROR_MODE_VALUE	The mode to be programmed is out of range.
0134	HPBUS_ERROR_NOT_PAUSED	The BALLISTIC IC must be in a paused state to be programmed.
0135	HPBUS_ERROR_INT_NOT_SET	No interrupt source was set.
0191	DUART_ERROR_CHANNEL_VALUE	Unknown channel number.

Table 20 Status codes for Host functions

Error Number (Base 16)	Status Code	Description
BFFC 0800	SSVX8_ERROR_ERROR_MESSAGE_UNKNOWN	Unknown error return value.
BFFC 0801	SSVX8_ERROR_MEMORY_ALLOCATION	Error in memory allocation.
BFFC 0802	SSVX8_ERROR_IO_UNMAP_FAILED	Error in memory deallocation.
BFFC 0803	SSVX8_ERROR_SELF_TEST_FAILED	VX8 device failed its self test.
BFFC 0804	SSVX8_ERROR_SELF_TEST_UNKNOWN_RESULT	VX8 device reported an invalid self-test result.
BFFC 080A	SSVX8_ERROR_CLOSE_FAILED	An error occurred during the closing of an I/O handle.
BFFC 080B	SSVX8_ERROR_VX8_ALREADY_IN_RESET	Before the VX8 system reset operation, a VX8 was already in reset mode. This can be caused by the VX8 belonging to multiple active SDF's or a failed VX8.
BFFC 080C	SSVX8_ERROR_VX8_RESET_TIMEOUT	VX8 device failed to indicate passed and ready in the allotted time.
BFFC 080D	SSVX8_ERROR_RESET_FAILED	A VX8 failed reset.
BFFC 080E	SSVX8_ERROR_SYSTEM_RESET_TIMEOUT	One of the VX8 devices in the system failed to reset.
BFFC 0810	SSVX8_ERROR_NO_SSVX8_FOUND	The board referenced by the I/O handle is not a VX8.
BFFC 0811	SSVX8_ERROR_BAD_RSRC_DESCRIPTOR	A bad resource descriptor.
BFFC 0812	SSVX8_ERROR_NO_VX8_IN_SYSTEM_STRUCT	A bad System Definition File (SDF). The SDF must contain at least one VX8 device.
BFFC 0813	SSVX8_ERROR_SDFOPENSDF_FAILED	An error occurred during the parsing of the System Definition File (SDF).
BFFC 0814	SSVX8_ERROR_LDFOPENLDF_FAILED	An error occurred during the parsing of the Load Definition File (LDF).
BFFC 0815	SSVX8_ERROR_SYSTEMCLOSE_FAILED	An error occurred during the closing of the VX8 system.

Error Number (Base 16)	Status Code	Description
BFFC 0816	SSVX8_ERROR_SYSREAD_BOARD_NOT_FOUND	ssVX8_SystemRead could not find a device with the indicated resource descriptor.
BFFC 0817	SSVX8_ERROR_SYSWRITE_BOARD_NOT_FOUND	ssVX8_SystemWrite could not find a device with the indicated resource descriptor.
BFFC 0818	SSVX8_ERROR_VX8_STILL_IN_CONFIG	A VX8 device is still asserting CONFIG*.
BFFC 081B	SSVX8_ERROR_RW_BAD_VX8_ADDRESS_RANGE	ssVX8_BoardRead/Write bad address.
BFFC 081D	SSVX8_ERROR_RW_IO_FAILED	ssVX8_BoardRead/Write I/O call failed.
BFFC 081E	SSVX8_WARN_RW_ZERO_LENGTH	ssVX8_BoardRead/Write zero length specified.
BFFC 081F	SSVX8_ERROR_SYSTEM_STRUCT_UNINITIALIZED	The system structure was uninitialized.
BFFC 0820	SSVX8_ERROR_SETHPUXMAPSPACE_FAILED	The map space parameter was unsuccessfully set.
BFFC 0821	SSVX8_ERROR_SETIMAPA32BA_FAILED	An error occurred during the determination of a VX8 A32 base address.
BFFC 0822	SSVX8_ERROR_NAK_BUFFER_TIMEOUT	The Node A kernel has stopped responding during loading.
BFFC 0823	SSVX8_ERROR_PROCLOADCODE_COFFPARSE_FAILED	An error occurred during the parsing of a COFF file.
BFFC 0824	SSVX8_ERROR_READFIRMWAREREV_TIMEOUT	Node A failed to report the firmware revision number in the allotted time.
BFFC 0825	SSVX8_ERROR_READSELFTESTRES_TIMEOUT	Node A failed to report the self test results in the allotted time.

Appendix B: Definitions and Acronyms

API	Application Program Interface
C4x	Abbreviation for the Texas Instruments Family of TMS320C4x DSPs.
C40	Abbreviation for the Texas Instruments TMS320C40 DSP.
C44	Abbreviation for the Texas Instruments TMS320C44 DSP. The C44 has only 4 COMM ports compared to 6 COMM ports on the C40.
COFF	**Common Object File Format -** a binary file format generated by linkers which contains code, loading information, and debugging information. Refer to the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*.
DMA	**Direct Memory Access -** a term indicating that data is moved from one location to another without the direct intervention of the processor. Your application code typically must configure the DMA Controller, but the code does not directly move the data.
DMA Controller	There are a number of Direct Memory Access Controllers that you can program on the VX8 Carrier Board: • The HP Local Bus Interface DMA Controller transfers data from HP Local Bus to the Near Global SRAM of one or more C4x DSPs. This is the only method of consuming data from the HP Local Bus. • The SCV64 has an internal DMA Controller that can be used to master the VXIbus and transfer between the VX8 Carrier Board and some other VXIbus slave (the host or another VX8 Board, for example). • The TMS320C4x DSP has internal DMA Controllers that can be used to transfer data between the DSP and COMM Ports or memory buses.
DRAM Shared Bus	One of two Shared Buses on the VX8. The VXIbus slave interface can access the DRAM, Test Bus Controller, and the Global Shared Bus via the DRAM Shared Bus. A C4x DSP can access the DRAM, control / status registers, and the SCV64 as a VXIbus master via the DRAM Shared Bus.
DSP	Digital Signal Processor

DUART	**Dual Universal Asynchronous Receiver/Transmitter** - National Semiconductor’s NS16C552 or compatible) device.
endian	The ordering of bytes in a multi-byte number. This affects D16 and DE08 data transfers to the VX8.
FIFO	**First In First Out** buffer. There are several FIFOs on the VX8 including the HP BALLISTIC Read FIFO, SCV64 Transmit and Receive FIFOs, SCV64 Location Monitor FIFO, and the C4x built-in COMM port FIFOs.
Global Shared Bus	The Global Shared Bus interconnects all 8 DSP Nodes, the HP-Local Bus Interface, and the DRAM Shared Bus.
ISR	Interrupt Service Routine
LDF	**Load Definition File** - defines the software configuration of your VX8 system.
LED	**Light-Emitting Diode**. There are three LEDs on the VX8: red = sysfail, green = VXIbus access, and yellow is user definable.
LM	Location Monitor
Lock	To access the Global Shared Bus, a C4x DSP must lock to the bus. See the **VX8_Lock** and **VX8_Unlock** functions.
Near Global SRAM	The SRAM located on the Global Bus of any C4x DSP in the system. Near Global SRAM is zero wait state from the DSP that owns it, but can be accessed by other DSPs, the HP Local Bus DMA Controller, and the VXIbus Slave Interface via the Global Shared Bus.
PDI	**Programmatic Developer Interface** - the VXIpnp name for the instrument driver interface to an application program.
PEROM	Programmable and Erasable Read Only Memory - an ATMEL acronym for FLASH devices.
RAE	**Read FIFO Almost Empty** - a condition which occurs when the Read FIFO level reaches the programmed almost empty level. This causes an IIOF1 interrupt on the DSPs if enabled.
RAF	**Read FIFO Almost Full** - a condition which occurs when the Read FIFO level reaches the programmed almost full level. This causes an IIOF1 interrupt on the DSPs if enabled.
RM	VXIbus **Resource Manager** which is responsible for configuring VXIbus devices through their A16 configuration registers.

SCV64	VME Interface IC from Tundra Semiconductor that is used on the VX8 Carrier Board to provide A32 / D32, D16, D08(EO) transfers to the board.
SDF	**System Definition File -** a text based file which defines the hardware configuration of your VX8 System.
SICL	HP's Standard Instrument Control Library
Slot 0 Device	A required VXIbus device which is responsible for driving several VXIbus backplane signals. In the base hardware configuration, the Slot 0 device is the HPV743.
VISA	Virtual Instrument Software Architecture
VME	**Versa Module Eurocard -** a bus standard defined in IEEE-1014. VME is the standard which the VXIbus specification has expanded on.
VX8	Spectrum Signal Processing's TMS320C40 VXIbus board which features 2 embedded C40's, 6 TIM-40 sites, and HP local bus.
VXI	**VME eXtensions for Instrumentation -** a bus standard defined in IEEE 1155
VXI Controller	A VXIbus device which is responsible for controlling the VXIbus mainframe. In the base hardware configuration, the controller is the HP V743.
VXIpnp	**VXI Plug and Play - Systems Alliance**. This group has defined an instrument driver specification that simplifies the integration of VXIbus systems by standardizing the software interface to VXIbus instruments. This group has also defined the VISA I/O library specification.
VXIpnp Module	A component of the VX8 instrument driver which contains the required functions specified by the VXIpnp instrument driver specifications.
WAE	**Write FIFO Almost Empty -** a condition that occurs when the Write FIFO level reaches the programmed Almost Empty level. This causes an IIOF1 interrupt on the DSPs if enabled.
WAF	**Write FIFO Almost Full -** a condition that occurs when the Write FIFO level reaches the programmed Almost Full level. This causes an IIOF1 interrupt on the DSPs if enabled.