ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *National Instruments GPIB-VXI/C* at our website: ***Click HERE***

GPIB-VXI/CP
Software Reference Manual



December 1993 Edition

Part Number 320405-01

© Copyright 1992, 1994 National Instruments Corporation.
All Rights Reserved.

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices:
Australia (03) 879 9422, Austria (0662) 435986, Belgium 02/757.00.20, Canada (Ontario) (519) 622-9310,
Canada (Québec) (514) 694-8521, Denmark 45 76 26 00, Finland (90) 527 2321, France (1) 48 14 24 24,
Germany 089/741 31 30, Italy 02/48301892, Japan (03) 3788-1921, Netherlands 03480-33466, Norway 32-848400,
Spain (91) 640 0085, Sweden 08-730 49 70, Switzerland 056/20 51 51, U.K. 0635 523545

Limited Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

TIC™ is a trademark of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

Warning Regarding Medical and Clinical Use of National Instruments Products

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual xi
Organization of This Manual xi
Conventions Used in This Manual xii
Related Manual xii
Related Documentation xiii
Customer Communication xiii

Chapter 1
Introduction 1-1
What Your Kit Should Contain 1-1
GPIB-VXI/C Operation without CIs 1-1
CI Operation 1-3
CI Characteristics 1-4
Communication with the Local Command Parser 1-5
Summary 1-5
Differences Between CIs and Physical VXI Message-Based Devices 1-6
Communication Protocol Implementation 1-6
The VXIbus System's View of a CI 1-6
Downloaded CIs and EPROMed CIs 1-6
Resident CIs 1-7
Code Instrument Development Overview 1-7
Development System Configuration 1-7
Compiler, Assembler and Linker Requirements 1-8
Position Independent Code Versus Static Code 1-8

Chapter 2
Code Instrument Structure 2-1
CI Memory Structures 2-1
Static CI Memory Structure 2-1
Static DCI Memory Structure 2-3
Static ECI Memory Structure 2-5
PI CI Memory Structure 2-6
PI DCI Memory Structure 2-7
PI ECI Memory Structure 2-8
CI Header and Pre-Initialization Segment Structures 2-9
CI Initialization Routine and Process Structures 2-9
The Pre-init Routine 2-9
The Asynch Process 2-10
Hardware Interrupts 2-11
The Worker Process 2-11

Chapter 3
Designing a Code Instrument 3-1
The CI Design Process 3-1
Using the 852 Adapter CI as a Shell 3-3
The init.c File 3-4

The asynch.c File 3-5
Handling Word Serial Commands and Queries 3-5
Handling Physical Word Serial Commands and Queries 3-7
Handling Event/Response Signals/Interrupts 3-7
Handling Word Serial Data In Notices 3-8
Handling DOR Requests 3-8
Handling Control Register Writes 3-8
The worker.c File 3-9
Handling the VXI Startup Sequence 3-9
Handling Word Serial Messages 3-10
Accessing Nonvolatile Configuration Information 3-10
Using the 68881 Math Coprocessor 3-10
Implementing a CI Function Library 3-11

Chapter 4
Creating a Downloaded Code Instrument 4-1
Static and Position Independent ECI Compatibility Issues 4-1
Segment Ordering 4-1
Initialization 4-2
Static DCI Creation Sequence 4-3
Creating a DCI Memory Map 4-3
DCI Memory Image Format 4-4
PI DCI Creation Sequence 4-4

Chapter 5
Creating an EPROMed Code Instrument 5-1
Creating A Static ECI Memory Map 5-2
Creating a PI ECI Memory Map 5-2
ECI Memory Image Format 5-2

Chapter 6
C Function Calls 6-1
Interface to National Instruments-Supplied Functions 6-2
Interface to User-Defined Functions/Libraries 6-2
Word Serial Protocol Functions 6-4
CIAbortNormalOperation 6-5
CIEndNormalOperation 6-5
CIReleaseDevice 6-6
ControlOut 6-6
CreateBuf 6-7
DCIwsClear 6-7
DuplicateCommonBuf 6-8
HWwsClear 6-8
SendDORrequest 6-9
SignalOut 6-9
VXI Register Access Functions 6-10
SafeReadByte 6-11
SafeReadWord 6-11
SafeWriteByte 6-12
SafeWriteWord 6-12
VXIregBase 6-13

GPIB-VXI/C Resource Access Functions 6-14
ClearParser 6-16
DCIParserDeq 6-16
DCIParserEnq 6-17
DCIParserQuery 6-18
Disable68881 6-18
DMAmove 6-19
Enable68881 6-20
NVconf 6-20
ReenableHandler 6-21
ReserveHandler 6-21
GPIB-VXI/C Trigger Functions 6-22
AcknowledgeTrig 6-23
DisableTrigSense 6-24
EnableTrigSense 6-25
GetTrigHandler 6-26
MapTrigToTrig 6-27
SetTrigHandler 6-29
SrcTrig 6-30
TrigAssertConfig 6-32
TrigCntrConfig 6-34
TrigExtConfig 6-35
TrigTickConfig 6-37
UnMapTrigToTrig 6-39
WaitForTrig 6-40

Chapter 7
Word Serial Drivers 7-1
Buffer-Based Communication 7-1
GPIB Buffer Transfers 7-2
Handling a Clear Command 7-2
Handling an Unrecognized Command Event 7-3
Word Serial Communication Examples 7-3
Sending a Command 7-3
Sending a Response 7-4
Receiving a Buffer 7-4
Sending a Buffer 7-5
Word Serial Commander Device Driver Calls 7-6
d_ctrl(DEV_WS, . .) 7-7
d_read(DEV_WS, . .) 7-8
d_write(DEV_WS, . .) 7-9
Word Serial Slave (Servant) Driver Calls 7-10
d_ctrl(DEV_WSsl, . .) 7-11
d_read(DEV_WSsl, . .) 7-12
d_write(DEV_WSsl, . .) 7-13

Chapter 8
VXI pROBE 8-1
Entering and Exiting VXI pROBE 8-1
VXI pROBE Menu 8-2
VXI pROBE Commands 8-4
Notation and Syntax 8-4

Help Screen Command 8-5
HELP 8-5
Mode Control Commands 8-6
BOOT 8-7
CONF 8-7
DIAG 8-7
ECI 8-8
IN 8-9
ME 8-9
RBO 8-9
GPIB Port Control Commands 8-10
GD 8-10
GDO 8-10
GE 8-11
GEO 8-11
GP 8-11
GPS 8-12
pROBE Output Routing Control Commands 8-13
OMD 8-13
OME 8-13
Macro and Script Commands 8-14
MA 8-15
MD 8-15
MI 8-16
MS 8-16
Hardware Access Commands 8-17
MO 8-17
P 8-18
VR 8-19
VXI Device Information Commands 8-20
FI 8-20
FV 8-21

Appendix A
Code Instruments Source Code A-1

Appendix B
GPIB-VXI/C VXI Trigger Support B-1

Appendix C
Customer Communication C-1

Glossary Glossary-1

Index Index-1

Figures

Figure 1-1. GPIB-VXI/C Operation Without Code Instruments....1-2
Figure 1-2. Code Instrument Operation....1-3

Figure 2-1. Static CI Local Memory Map (with Global CI RAM Allocation) 2-2
Figure 2-2. Static DCI Memory Structure....2-3
Figure 2-3. Static ECI Memory Structure 2-5
Figure 2-4. Position Independent CI Local Memory Map (with All RAM for Dynamic Use) 2-6
Figure 2-5. Position Independent DCI Memory Structure 2-7
Figure 2-6. Position Independent ECI Memory Structure....2-8

Figure 5-1. Static ECI EPROM Image Versus PI ECI EPROM Image 5-1

Figure B-1. GPIB-VXI/C GPIO Connections....B-1

Tables

Table 3-1. Code Instrument Capabilities 3-2

Table 4-1. DCI Segment Ordering and Content....4-1
Table 4-2. ECI Segment Ordering and Content 4-2

Table 6-1. VME A16 and A24 Space 6-10

About This Manual

This manual contains information you will need to develop Code Instruments, which are custom software modules that run directly on the GPIB-VXI/C. This manual assumes that you are familiar with the VXIbus specification, as well as 68000 Assembly and C programming languages.

Organization of This Manual

The *GPIB-VXI/CP Software Reference Manual* is organized as follows:

- Chapter 1, *Introduction*, lists the contents of your GPIB-VXI/C kit, illustrates GPIB-VXI/C operation with and without software modules known as Code Instruments (CIs), and introduces the functions, applications, and implementations of CIs.
- Chapter 2, *Code Instrument Structure*, describes the Code Instrument memory and process structures.
- Chapter 3, *Designing a Code Instrument*, contains information that will help you make design decisions about your Code Instruments.
- Chapter 4, *Creating a Downloaded Code Instrument*, contains information you will need to create a Downloaded Code Instrument.
- Chapter 5, *Creating an EPROMed Code Instrument*, describes how to convert a Downloaded Code Instrument to the form of an EPROMed Code Instrument.
- Chapter 6, *C Function Calls*, contains descriptions of C language function calls for the GPIB-VXI/C firmware. This chapter also discusses interfaces to the National Instruments-supplied functions and to your user-defined functions and libraries.
- Chapter 7, *Word Serial Drivers*, contains information about the Word Serial device drivers, including descriptions of the driver calls, their purpose, and examples of their use.
- Chapter 8, *VXI pROBE*, describes the pROBE debugging tool for the VXI environment, including its functions, menus, and VXI commands.
- Appendix A, *Code Instruments Source Code*, contains a summary of the contents of the C and assembly source code files that you need in order to implement Code Instruments on the GPIB-VXI/C. These files can be found in machine-readable form on the GPIB-VXI/C Distribution Disk Code Instrument Examples/Shell Files that you received from National Instruments as part of this kit.
- Appendix B, *GPIB-VXI/C VXI Trigger Support*, contains an overview of the VXI triggering capabilities of the GPIB-VXI/C and GPIB-VXI/CP.
- Appendix C, *Customer Communication*, contains forms for you to complete to facilitate communication with National Instruments concerning our products.

- The *Glossary* contains an alphabetical list of terms used in this manual and a description of each.
- The *Index* contains an alphabetical list of key terms and topics used in this manual, including the page where each one can be found.

Conventions Used in This Manual

Throughout this manual, the following conventions are used to distinguish elements of text:

italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept. In this manual, italics are also used to denote Word Serial commands and queries.
`monospace`	Text in this font denotes text or characters that are to be literally input from the keyboard, sections of code, command or query syntax, console responses, and syntax examples. This font is also used for the names of all commands and queries used in the GPIB-VXI/C local command set.
<CR>	Angle brackets enclosing a term in Times font represent a keystroke on the keyboard.
`<address>`	Angle brackets enclosing a term in monospace denote a parameter to a console (VXI pROBE) command.

Numbers in this manual are base 10 unless noted as follows:

- Binary numbers are indicated by a -b suffix (for example, 11010101b)
- Octal numbers are indicated by an -o suffix (for example, 325o),
- Hexadecimal numbers are indicated by an -h suffix (for example, D5h)
- ASCII character and string values are indicated by double quotation marks (for example, "This is a string").

In this manual, the symbol <CR> is used to indicate the ASCII carriage return character.

OR stands for Boolean logical OR, denoted as | in C language code.

Terminology that is specific to a chapter or section is defined at its first occurrence.

Related Manual

GPIB-VXI/C User Manual, part number 320404-01

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- *IEEE Standard Codes, Formats, Protocols, and Common Commands*, ANSI/IEEE Standard 488.2-1987
- *IEEE Standard Digital Interface for Programmable Instrumentation*, ANSI/IEEE Standard 488.1-1987
- *IEEE Standard for a Versatile Backplane Bus: VMEbus*, ANSI/IEEE Standard 1014-1987
- *VXIbus System Specification*, Revision 1.3, VXIbus Consortium

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix C, *Customer Communication*, at the end of this manual.

Chapter 1
Introduction

This chapter lists the contents of your GPIB-VXI/C kit, illustrates GPIB-VXI/C operation with and without software modules known as Code Instruments (CIs), and introduces the functions, applications, and implementations of CIs.

What Your Kit Should Contain

Your GPIB-VXI/C kit should contain the following components:

Kit Component	Part Number
GPIB-VXI/CP Software Reference Manual	320405-01
pSOS-68K Real-Time Multi-Tasking Operating System Kernel Manual, Software Components Group, Inc.	320201-01
pROBE-68K System Debug/Analyzer Manual, Software Components Group, Inc.	320202-01
pREP/C C Language Run-Time Environment Package Manual, Software Components Group, Inc.	320203-01
GPIB-VXI/C Distribution Disk Code Instrument Examples/ Shell Files, source (5 25 in. 360K IBM PC format)	420400-44

GPIB-VXI/C Operation without CIs

The typical Commander/Servant relationships for GPIB-VXI/C operation without CIs are illustrated in Figure 1-1. When the GPIB-VXI/C's Commander communicates directly with the GPIB-VXI/C, it is normally communicating with the local command set parser, which parses and executes the local commands described in Chapter 3, *Local Command Set*, of the *GPIB-VXI/C User Manual*. Although the GPIB and serial controllers are not Commanders of the command parser in the VXI sense, they are its *master* in the sense that it will respond to their commands as if they were its Commander. The GPIB-VXI/C maintains independent control paths to the local command set parser from the GPIB controller, the serial controller, the GPIB-VXI/C's Commander, and for each CI.

Note: Conflicts can occur when a device has multiple active Commanders. National Instruments recommends that you use only one command path to the local command set parser in your application software.

The GPIB-VXI/C has four ports for communicating with other devices. Each port consists of its electrical interface and the associated system software. The GPIB-VXI/C communicates with its Commander through the Word Serial Servant port, and with its Servants through the Word Serial Commander port. The GPIB System Controller communicates with the GPIB-VXI/C and its Message-Based Servants through the GPIB port, which maps GPIB addresses to VXI logical addresses. A serial controller can access the local command parser through the RS-232 port, and therefore can communicate with the GPIB-VXI/C's Message-Based Servants through the Word Serial communication commands.

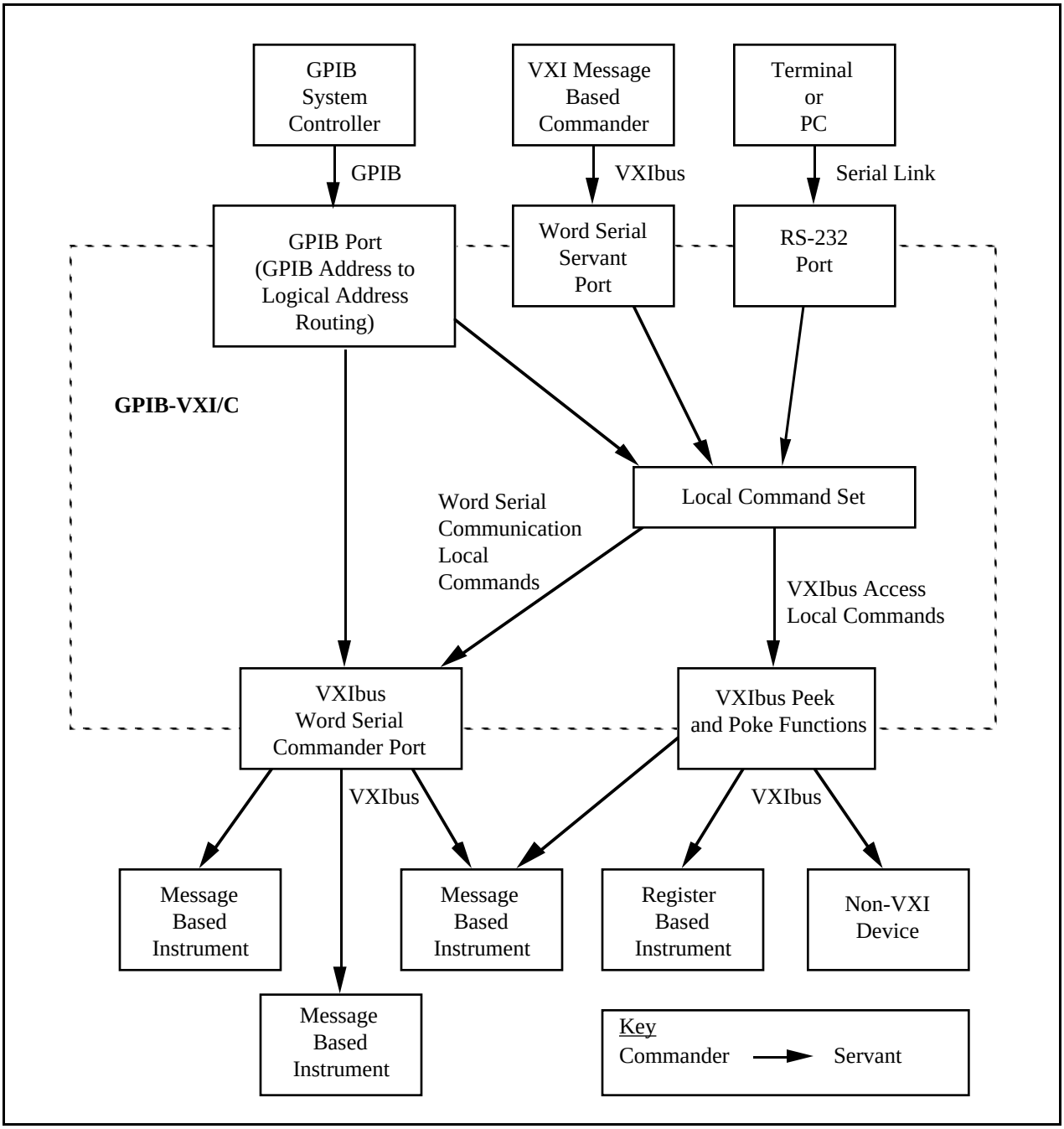
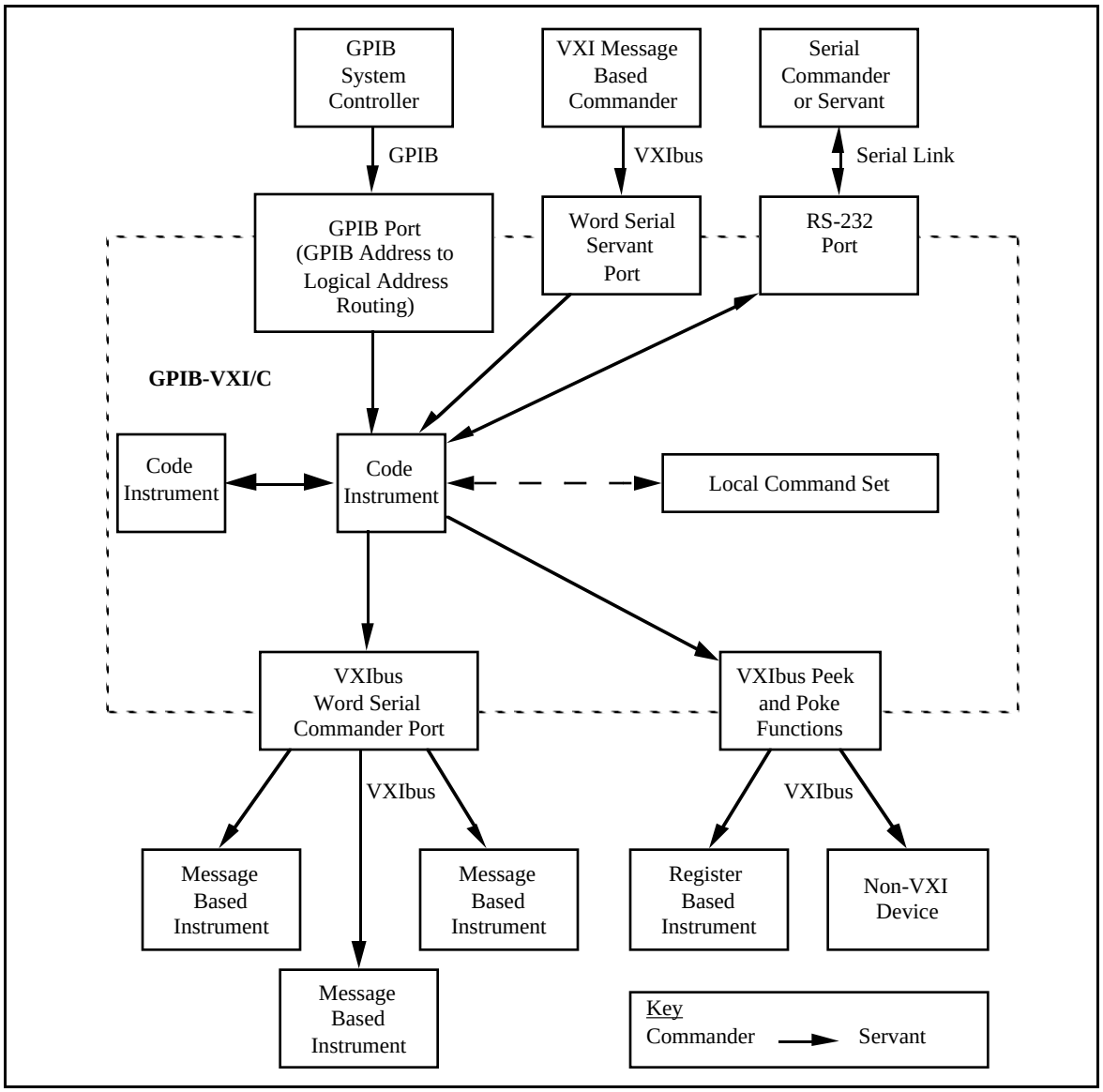


Figure 1-1. GPIB-VXI/C Operation Without Code Instruments

The GPIB and serial controllers can also directly manipulate the GPIB-VXI/C's Register-Based and non-VXI Servants through the VXIbus access local commands. This solution to the general problem of controlling Register-Based and non-VXI devices is relatively ineffective for high-performance applications, however, because of the low-level functions the System Controller must perform and the resulting heavy GPIB traffic.

CI Operation

A CI is a set of software routines that can perform the functions of a physical VXI Message-Based device. These CI capabilities are illustrated in Figure 1-2. CIs coexist with the IEEE-488 VXI translation and local command set functions shown in Figure 1-1, with the exception that the Word Serial Servant and RS-232 ports support only one Commander/Servant connection (either to the local command set parser or to a CI, but not both) at one time.



Typical CI applications include:

- Parsing and interpreting command languages
- Creating virtual (hierarchical) instruments
- Creating Message-Based interfaces for Register-Based and non-VXI devices

CIs are implemented in three forms:

- As part of the National Instruments supplied firmware (*Resident CIs*, or *RCIs*)
- As downloaded object code (*Downloaded CIs*, or *DCIs*)
- As user-defined add-on firmware (*EPROMed CIs*, or *ECIs*)

CI Characteristics

A CI has all of the capabilities of a physical Message-Based Commander. A CI can do any of the following functions:

- Set up its own set of configuration registers
- Have Servants assigned to it
- Engage in Word Serial communication with its Commander and Servants
- Receive and send VXI event/response signals
- Receive VXI interrupts from VXI and non-VXI devices (RORA and ROAK)
- Directly access the A16 or A24 registers/memory of its VXI and non-VXI Servants
- Communicate with the GPIB System Controller through a GPIB primary or secondary address
- Perform memory-to-memory DMA operations using 68070 DMA channel
- Source or accept any number of TTL (0 to 7) or ECL (0 to 1) triggers using any of the VXIbus-defined protocols
- Configure the 16-bit trigger counter or the dual 5-bit tick timers
- Configure the TTL/ECL trigger crosspoint switch/signal conditioning setup.

As with physical devices, a CI must be an immediate Servant of the GPIB-VXI/C in order to have a GPIB address. In addition to these VXIbus device capabilities, CIs can also communicate directly with the local command parser and the serial port.

The GPIB-VXI/C emulates the physical capabilities of a Message-Based device for each CI. Because a CI can be a Commander or a Servant, you can construct multilevel hierarchies of CIs and physical Message-Based devices. The only restriction is that a CI cannot be mapped out of the hierarchy of devices within the GPIB-VXI/C. In other words, a CI can be any of the following:

- A top-level Commander
- The Commander of any number of CIs and/or physical VXI devices

- A Servant of another CI
- A Servant of the GPIB-VXI/C's Commander

A CI cannot, however, be the Servant of a physical VXI device that is not the GPIB-VXI/C's Commander.

A CI appears from the point of view of the GPIB and other CIs to be an actual physical device, because it performs all of the functions that a physical Message-Based device performs. If the CI takes control of the physical Word Serial registers on the GPIB-VXI/C, it becomes a physical VXI device. This step is usually not required, however, because most CIs function as Commanders that drive other Servants in the system rather than as Servants to higher-level Commanders.

A non-VXI device does not have VXI configuration registers, so it does not appear in the VXI device hierarchy. A CI that provides a Message-Based interface for a non-VXI device (together with that non-VXI device) is viewed by the system as a single device. Typical examples of non-VXI devices include:

- VME boards (CPU, Register-Based, memory, and so on)
- Colorado Data Systems (CDS) 73A-852 adapter module

Communication with the Local Command Parser

CIs can communicate directly with the GPIB-VXI/C local command parser. This feature gives CIs a wide range of system configuration and control capabilities. For example, a CI can

- Assign itself a GPIB address
- Disconnect the local command parser from the GPIB
- Take control of the GPIB-VXI/C's serial port or physical Word Serial registers
- Set VXI interrupt handler levels
- Reconfigure the system hierarchy

Built-in function calls and device drivers for performing all of these operations are supplied as part of the GPIB-VXI/C firmware.

Summary

With the capabilities just discussed, a CI can emulate or replace any existing VXI or VME device, or extend a device's native capabilities to new levels of functionality. CIs improve the system structure because

- GPIB traffic is greatly reduced.
- Register-Based and non-VXI devices can be treated as if they were Message-Based.
  - The GPIB controller sees one type of instrument (an IEEE-488 instrument).
  - Standard IEEE-488 communication with all types of VXI/non-VXI devices is possible.

- GPIB control of a VXIbus system can be implemented uniformly at a high level.
- Application software is simplified due to uniformity of control.
- System performance is greatly increased.
  - Direct access results in a tight coupling with its Servants.
  - Distributed processing removes burden from outside controller.
  - Access to VXIbus bandwidth is accomplished without GPIB overhead.

Differences Between CIs and Physical VXI Message-Based Devices

Because the CI is a set of functions running under two processes on a VXI device, a CI is not identical to a physical Message-Based device. National Instruments has minimized the differences, which are mainly in the areas of communication protocol implementation and the VXIbus system's view of the CI.

Communication Protocol Implementation

VXIbus communication methods consist of two components: the communication protocols and the combination of hardware and software that implements these protocols. While the protocols are the same for CIs and physical devices, the implementation is different. In the case of Word Serial Protocol, for example, there is no need to implement software versions of the VXIbus communication registers. Word Serial communication with CIs can be implemented more efficiently with higher-level software protocols. The difference in implementation does not modify the intent or usage of Word Serial Protocol or restrict the capabilities of the CI in any way.

The VXIbus System's View of a CI

A VXI hierarchy containing a CI should be looked at from two perspectives, the more important of which is that of the GPIB-VXI/C. The GPIB-VXI/C sees a CI as an actual physical Message-Based device. As the communication protocols are translated at the driver level, the application software can communicate with a CI in the same manner that it would communicate with a physical Message-Based device. This same viewpoint is shared by controllers on the GPIB.

The second perspective is from the viewpoint of other VXI devices, particularly the GPIB-VXI/C's Commander and Servants. Since each CI does not have its own physical Word Serial registers, the other devices only *see* the GPIB-VXI/C in the Commander/Servant hierarchy. This implies that a CI should never perform any action that distinguishes itself from the GPIB-VXI/C from the viewpoint of other physical devices. For example, a CI should not send a physical Servant device the Word Serial command *Identify Commander* with its own logical address. It should, rather, use the GPIB-VXI/C's logical address. The GPIB-VXI/C automatically routes the appropriate signals, interrupts, and other communications to each CI. The functions in the National Instruments CI libraries and drivers handle any protocol differences for the CI. In this manner, there are no incompatibilities with the VXIbus specification from the system point of view. Other devices see the GPIB-VXI/C as simply handling many physical devices at the same time; the existence and function of CIs is transparent to them.

Downloaded CIs and EPROMed CIs

You can download CIs in the form of binary code into the GPIB-VXI/C's RAM. The downloaded modules are called *Downloaded CIs*, or *DCIs*. The CI Configuration local commands download and initialize CIs. With the DCI form you can develop CIs without programming EPROMs, or you can create disk-loadable CI applications.

The GPIB-VXI/C runtime system also has an interface for installing CIs in onboard EPROMs, including a mechanism for automatically initializing them at system startup. Code Instruments stored in the EPROMs are called *EPROMed CIs*, or *ECIs*. The ECI form allows you to create stand-alone CI applications.

Resident CIs

As part of the GPIB-VXI/C firmware, National Instruments provides a *Resident CI*, or *RCI*, that communicates with the CDS 73A-852 adapter. The 852 adapter is a non-VXI device that requires a special code module somewhere in the system with a Message-Based interface. This Message-Based interface can be controlled from the GPIB or from another CI. Appendix B, *Using the DMAmove and CDS-852 Adapter Code Instruments*, in the *GPIB-VXI/C User Manual* contains information about installing and using the 852 adapter CIs.

Source code is also supplied for the CDS 73A-852 adapter CI. This source code was designed to be easily modified and is therefore more of a *shell* CI than just an example of source code. Most of the code supplies the standard VXI and GPIB-VXI/C interface used for most CI applications. Examples are also given for every capability that a CI can have.

Code Instrument Development Overview

Development System Configuration

A minimal CI development environment includes the following components and configurations:

- A VXIbus system mainframe with
  - A GPIB-VXI/C configured with the National Instruments development firmware
  - Target instrument cards
- A host computer connected to the GPIB-VXI/C GPIB port, with
  - A C compiler/assembler/linker for the 68000 microprocessor
  - A GPIB interface card and associated driver software (available for many computers from National Instruments)
- A dumb terminal or host running a terminal emulator connected to the GPIB-VXI/C serial port. The GPIB-VXI/C 9-pin serial port connector pinouts and parameter settings are given in Chapter 2, *Configuration and Startup Procedures*, of the *GPIB-VXI/C User Manual*.

Compiler, Assembler and Linker Requirements

National Instruments develops CIs with the OASYS-Green Hills C Compiler/Assembler/Librarian/Linker package for the 68000 microprocessor family, and the Microtec MCC68K Compiler/Assembler/Librarian/Linker package for the 68000 microprocessor family. Numerous C/assembly programming environments would work for CI development, but they must satisfy the following minimum requirements:

- The compiler *must* generate pure 68000 code (as opposed to 68010, and so on).
- The compiler and assembler *must* be able to externally reference freely between assembly and C code.
- The compiler *must* be a pure C compiler. It *must not* interpret any function calls as system calls or traps. Machine-dependent libraries (for example, Macintosh system interface libraries) must be removable.
- The compiler, assembler, and linker *must* be able to segment the C and assembly object code into at least two program code sections, two uninitialized data sections, and one initialized data section. The segments must be independently relocatable at link time.
- The linker *must* generate a binary memory image file for compatibility with the GPIB-VXI/C's DCI download command. It would be helpful if it could generate Motorola S-record or other format files compatible with your EPROM programmer, if you wish to create ECIs.

Note: This list is not necessarily exhaustive, and National Instruments does not guarantee the usability of any 68000 software development environment, including OASYS and Microtec Research products.

In selecting your code development tools, you should also consider your development cycle. Low-cost disk-based compilers used for the IBM Personal Computer AT and its compatibles, for example, may have unacceptably long development cycles.

Position Independent Code Versus Static Code

Position independent (PI) code has many advantages. You do not have to map locations for the code and data in the target address space, because the object image can reside anywhere in the address map of the target machine. Also, since global variables are allocated dynamically, several processes can run a single object module. In the CI context, this means that one CI image can be used to create many running CIs (for controlling multiple sets of the same hardware). CIs that use static addresses for code and data references can only be used once and must reside at a particular address in the GPIB-VXI/C memory map. This requirement forces you to create a memory map that defines where the CIs will reside. National Instruments supports both modes of CI code generation in an attempt to be compatible with as many C development systems as possible.

If your development system satisfies all of the requirements described in the previous section, you can generate Static CIs. The OASYS-Green Hills C Compiler/Assembler/Librarian/Linker package Version 1.8.3 and earlier generates static object modules compatible with Static CI development. Version 1.8.4 can generate PI code. The Microtec MCC68K Compiler/ Assembler/Librarian/Linker package can generate static as well as PI code. In order for you to generate PI CIs, your development system must also satisfy the following requirements:

- The compiler *must* generate PC relative code that initializes global variables.
- The compiler *must* generate code that references global variables relative to a 68000 address register (A2, A3, A4, A5, A6) or a data register (D2, D3, D4, D5, D6).
- The compiler/linker *must* allow you to set the initial values of the global data register and the stack pointer (address register A7).

Chapter 2
Code Instrument Structure

This chapter describes the Code Instrument (CI) memory and process structures.

CI Memory Structures

The 68070 microprocessor has a 24-bit local address space through which it can access local resources, including up to 4 Megabytes of RAM and 1 Megabyte of EPROM. The GPIB-VXI/C's runtime system and CIs share the RAM and EPROM. The two types of CI memory structures are Static CI structure and Position Independent (PI) CI structure. Code for the two types of CIs is virtually the same; the main differences are associated with the way that the code is compiled. Depending upon the capabilities of your compiler, you will need to choose one of the two CI types.

Static CIs reference hard-coded addresses and therefore must be put in exact locations in the GPIB-VXI/C RAM and EPROM memory maps. You need to make your own memory map specifiying where your CIs will reside.

PI CIs use PC relative addresses for code, and reference global variables relative to a 68070 address register (A2, A3, A4, or A5). As a result, you can load and run the same object code anywhere in the address map of the GPIB-VXI/C. PI CIs are obviously more desirable, but few compilers generate true PI code.

You will use different sets of GPIB-VXI/C local commands to install the two types of CIs. Download Static CIs to the GPIB-VXI/C using the local commands `DCISetup?` and `DCIDownLoad`. Use the local commands `DCISetupPI?` and `DCIDownLdPI` to download PI CIs to the GPIB-VXI/C. Static EPROMed CIs (ECIs) and PI ECIs have slight differences in the pre-initialization data structure in order to differentiate between the two.

Static CI Memory Structure

Because the Static CI structure uses fixed addresses, an area of RAM must be reserved for Static CIs. This RAM is called the *global CI RAM area*. A 512-kilobyte EPROM area and a variable-size RAM area are reserved in the 68070 memory map for CI implementation. The size of the global CI RAM area depends upon the size of installed RAM, the GPIB-VXI/C nonvolatile configuration, and how you use the `CIArea` local command. The global CI RAM area is partitioned into 4-kilobyte blocks.

Figure 2-1 depicts the Static CI local memory map of the global CI memory areas.

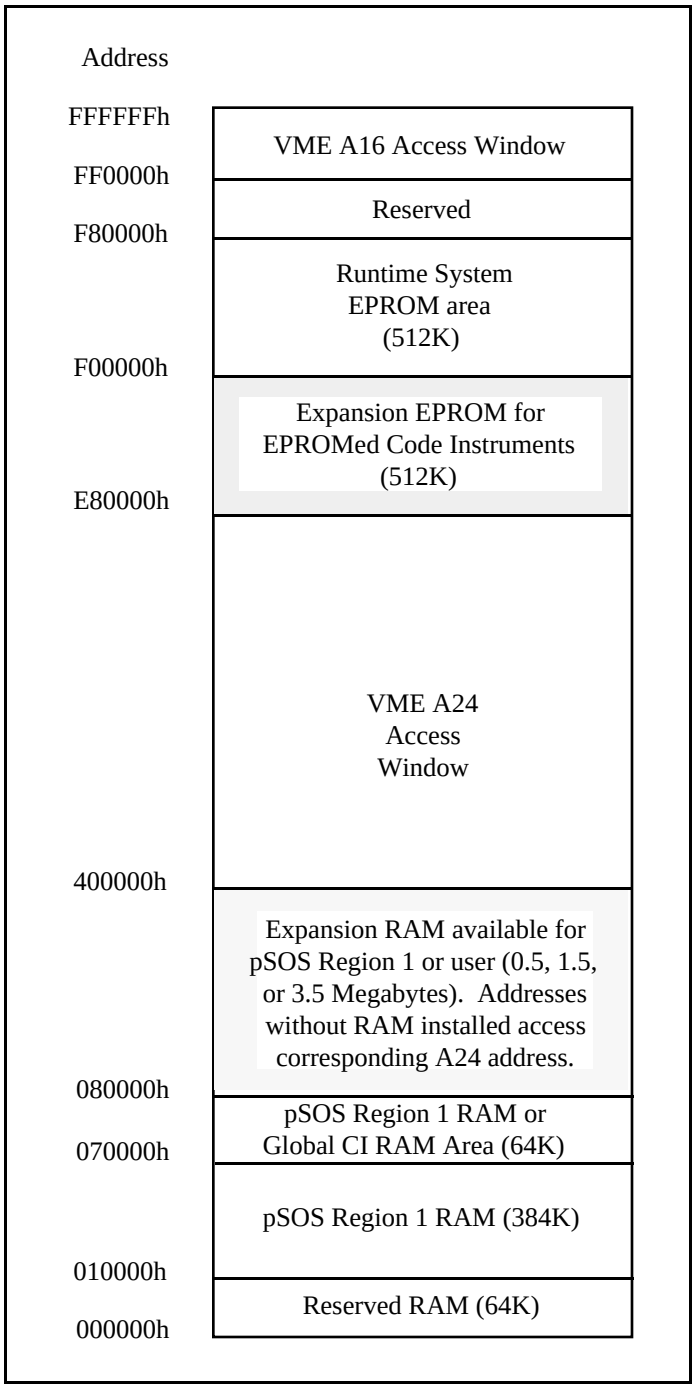


Figure 2-1. Static CI Local Memory Map (with Global CI RAM Allocation)

The minimum installed RAM configuration for the GPIB-VXI/C is 512 kilobytes located between 000000h and 07FFFFh in the GPIB-VXI/C local memory map. The first 64 kilobytes of RAM (000000h to 00FFFFh) is reserved for the National Instruments GPIB-VXI/C firmware. Normally, the rest of the 512 kilobytes of RAM is used by the pSOS operating system for global variables and for dynamic allocation (pSOS Region 1). However, you can reallocate (at most) the upper 64 kilobytes (070000h to 07FFFFh) as global CI RAM area for Static CIs by editing the nonvolatile memory parameters for pSOS and CI configuration, as described in Chapter 4, *Nonvolatile Configuration*, of the *GPIB-VXI/C User Manual*. To designate the upper 64 kilobytes as global CI RAM area, decrease the pSOS Region 1 (dynamic memory) size to 60000h, and set the CI block base to 70000h. When more than 512 kilobytes of RAM is installed on the GPIB-VXI/C, increase the pSOS dynamic RAM region size as needed. The global CI RAM area, if needed, should reside after the end of the pSOS dynamic RAM region.

Note: National Instruments strongly recommends an installed memory size of at least 1 Megabyte for running Downloaded CIs (DCIs) or multiple ECIs.

The maximum number of DCIs and ECIs that the GPIB-VXI/C can accommodate with 64 kilobytes of RAM depends upon the application, but two is usually the maximum unless you make a special effort to carefully allocate the RAM. One Megabyte of RAM gives greater flexibility with memory usage. The installed RAM is expandable from 512 kilobytes to 1, 2, or 4 Megabytes.

Static DCI Memory Structure

Figure 2-2 illustrates the Static DCI memory structure.

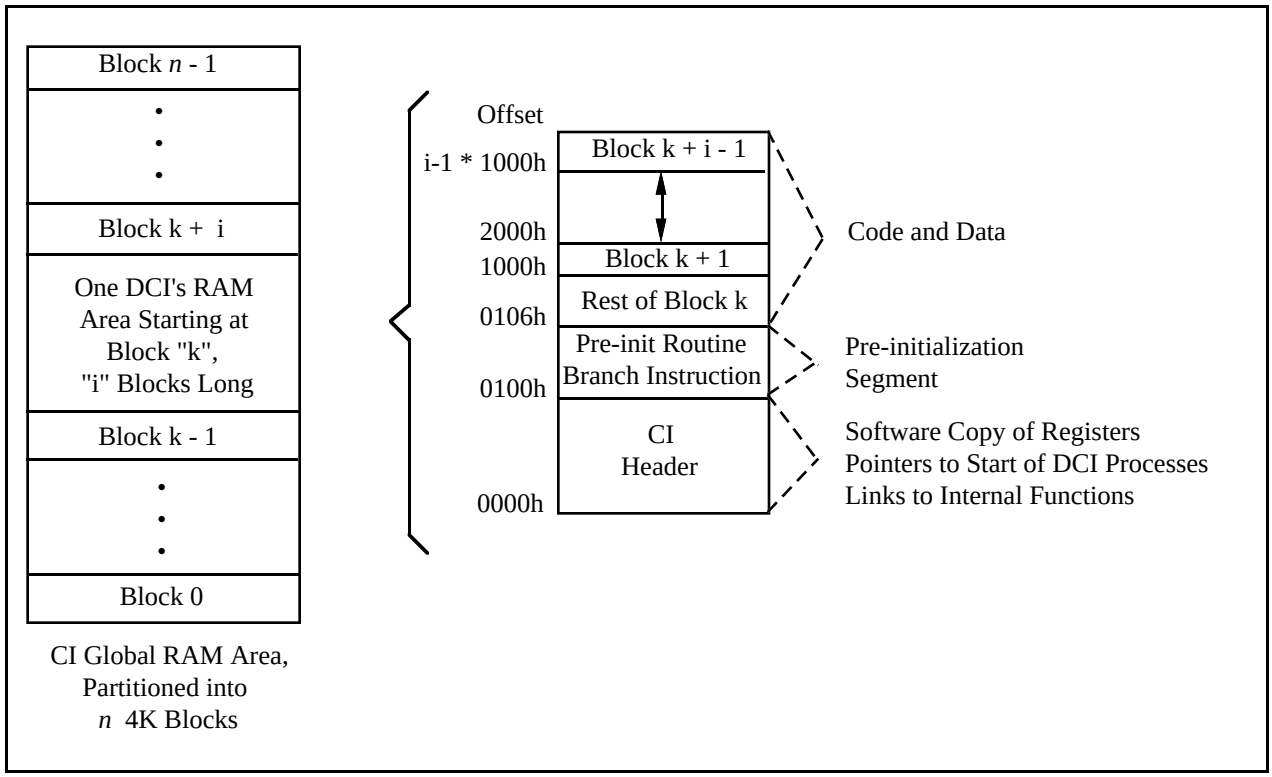


Figure 2-2. Static DCI Memory Structure

The CI global RAM area contains all memory used for the code and global data of the DCI. The `DCISetup?` local command determines the range of 4-kilobyte blocks in the global RAM area that the DCI will use. The `DCIDownLoad` local command downloads the code and data into the CI global RAM area. The first 256 (100h) bytes of the first block are reserved for the CI header. The header contains the CI's VXI software registers, pointers to the CI process entry points, and links to internally defined functions. The header links the CI into the GPIB-VXI/C runtime system.

Following the header is a variable-length pre-initialization segment in which you provide the runtime system with a standard interface to your pre-initialization routine. DCI pre-initialization segments must contain only the Pre-init routine branch instruction; however, when used for an ECI, the pre-initialization segment should also include ECI setup parameters.

The DCI's code and data are located in the remainder of the DCI's RAM area. The code image does not need to fill the requested memory area. Any memory left over can be used as a data area or common memory area for multiple CIs. As a result, CIs can share data areas.

Static ECI Memory Structure

Figure 2-3 illustrates the Static ECI memory structure.

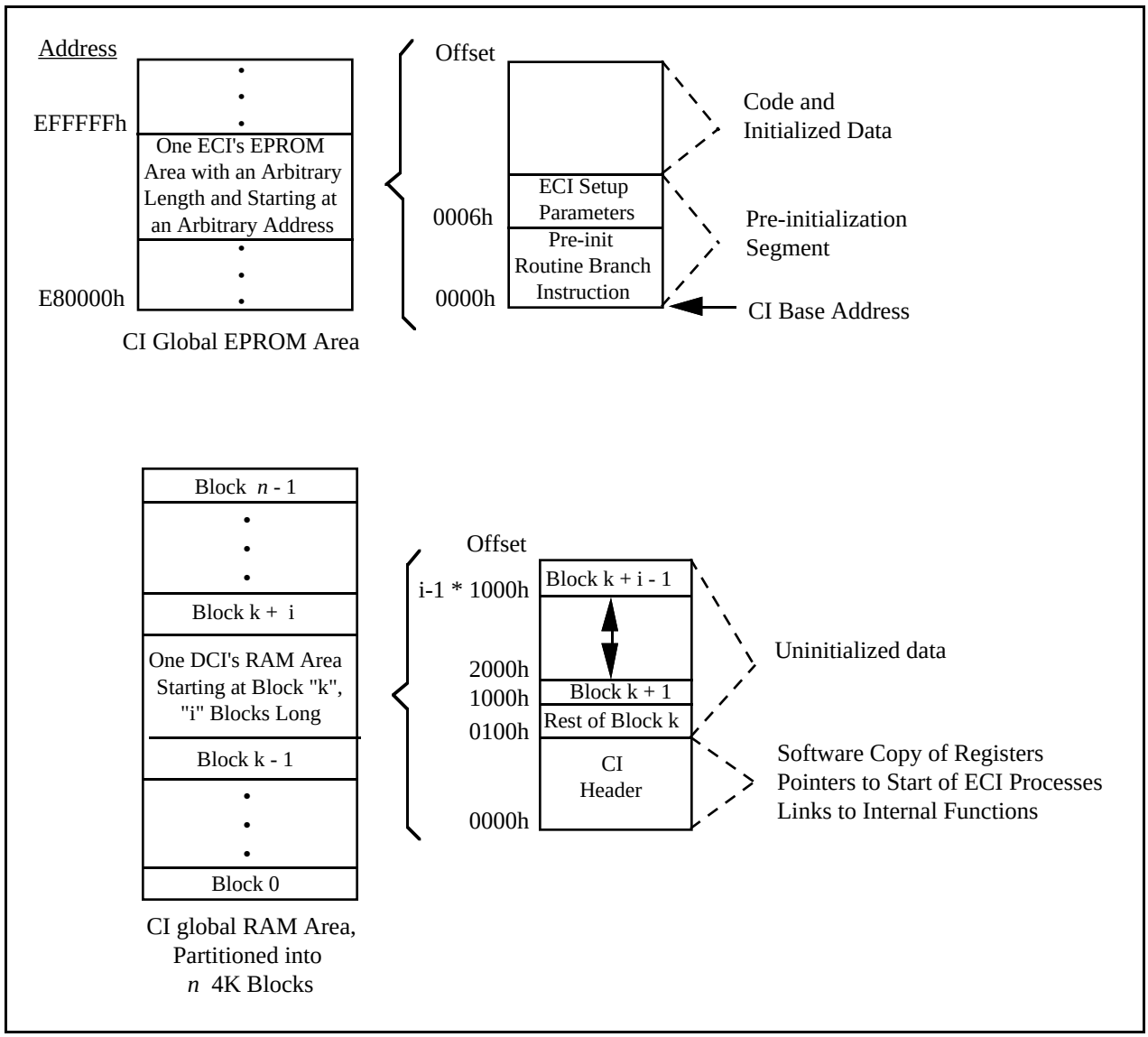


Figure 2-3. Static ECI Memory Structure

The Static ECI memory structure is similar to the Static DCI structure, except that the code, initialized data, and pre-initialization segments are located in the EPROM area. The header and uninitialized (variable) data are still located in the global CI RAM area. Linker options relocate the code and data into the memory areas.

Because ECIs are booted automatically at system startup time, the `DCISetup?` local command is not needed to specify the startup parameters. Instead, the ECI pre-initialization segment contains the CI initialization parameters (same type and size as would be sent using the `DCISetup?` command) as well as the vector to your pre-initialization routine.

PI CI Memory Structure

Because the PI CI structure has no fixed address requirements, all variables can be dynamically allocated in pSOS Region 1. Figure 2-4 illustrates the PI CI local memory map.

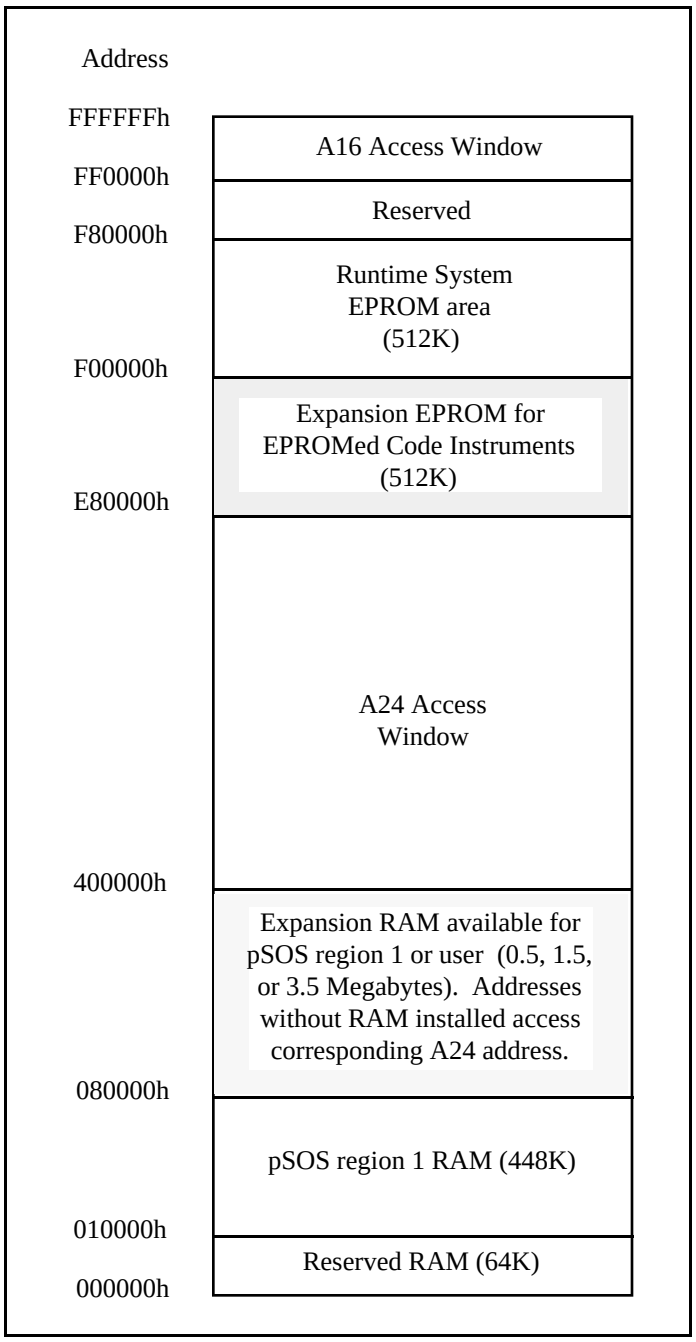


Figure 2-4. Position Independent CI Local Memory Map (With All RAM for Dynamic Use)

A 512-kilobyte EPROM area and a variable-size RAM area are available for PI CIs. The size of the RAM area depends upon the amount of installed RAM and the GPIB-VXI/C's nonvolatile configuration. The nonvolatile configuration can be used to reserve a section of high RAM for any static allocation needs, such as Shared Memory Protocol or a common static data area shared by CIs.

In most situations, PI CIs assign all of the available RAM space to pSOS dynamic RAM. Because the space needed for the PI CI is allocated directly out of this dynamic region, you can obtain a much more efficient use of the RAM. Under the Static CI model, memory is wasted because enough Static CI global RAM area must be reserved to accommodate all of the possible needs of an application. The PI CI, on the other hand, can dynamically allocate memory as it needs it. This capability reduces the PI CI's interference with the GPIB-VXI/C's runtime system memory requirements.

PI DCI Memory Structure

The structure of a PI DCI's memory areas is the same as for a Static DCI. The difference is in how the segments are allocated and located. Figure 2-5 illustrates how memory is allocated for a PI DCI.

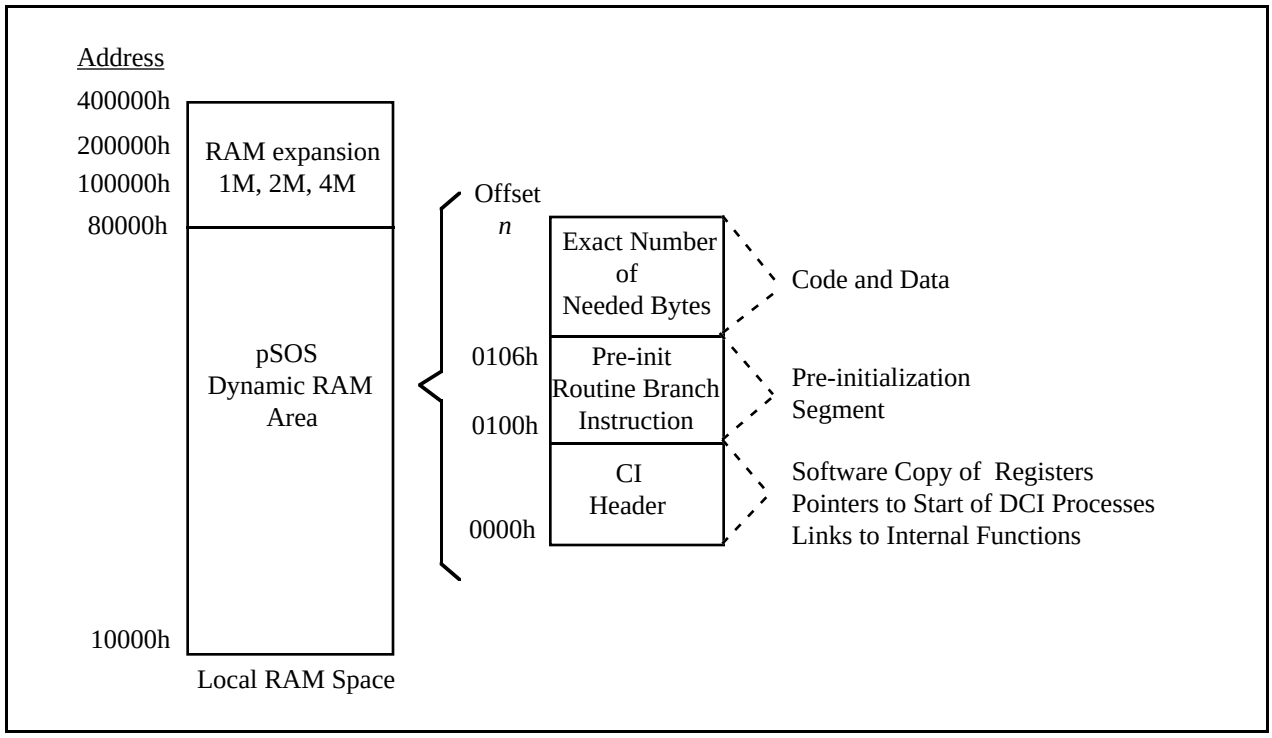


Figure 2-5. Position Independent DCI Memory Structure

Instead of using a global CI RAM space, all RAM needs are allocated out of pSOS dynamic memory. The `DCISetupPI?` local command defines how many bytes of dynamic memory the PI DCI requires. When the GPIB-VXI/C receives the `DCIDownLdPI` local command, it allocates a dynamic buffer of the size specified by the `DCISetupPI?` command parameters to

download the code and data. The code and data are assumed to be position independent. The base of the CI header is assumed to be located at the base of the dynamic buffer.

The address of the allocated segment is loaded into the 68070's address registers A2, A3, A4, A5, and A6 as well as the data registers D2, D3, D4, D5, and D6 before any code in the CI is run. The goal here is to be compatible with as many PI code-generating compilers as possible. Any other processes spawned by the CI that run PI code must set the appropriate register(s) to the base address of the header in order to access the CI's data correctly.

PI ECI Memory Structure

The PI ECI memory structure is similar to the PI DCI structure, except that the code, initialized data, and pre-initialization segments are located in the EPROM area. The header and uninitialized (variable) data are still located in the pSOS dynamic RAM area. Linker options are used to relocate the code and data into the two memory areas.

Figure 2-6 illustrates the memory structure for a PI ECI.

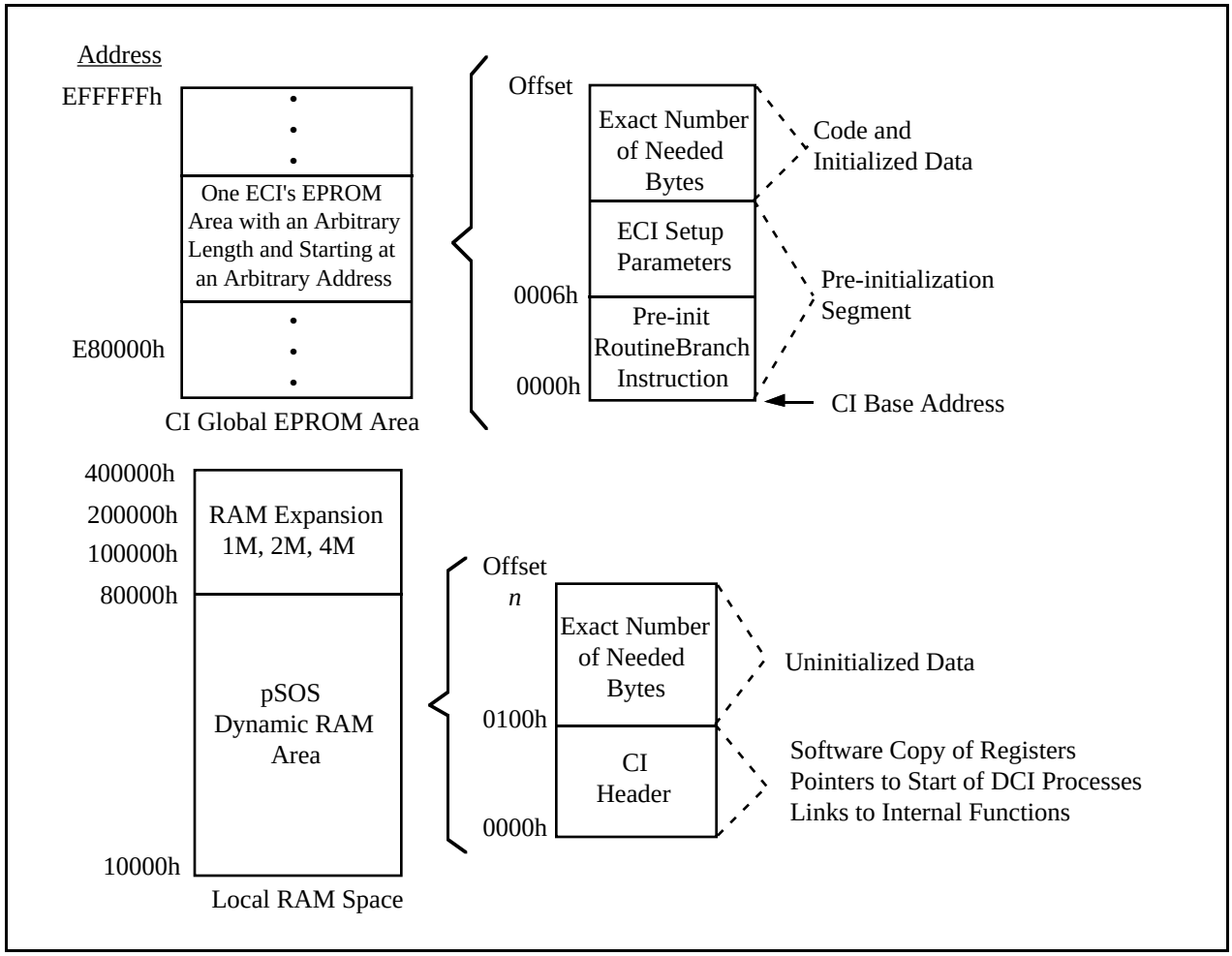


Figure 2-6. Position Independent ECI Memory Structure

Because PI ECIs are booted automatically at system startup time, the `DCISetupPI?` local command is not needed to install the startup parameters. Instead, the PI ECI pre-initialization segment contains your CI initialization parameters (same type and size as would be sent via the `DCISetupPI?` local command) as well as the standard vector to your pre-initialization routine.

CI Header and Pre-Initialization Segment Structures

The area for the CI header structure is reserved in the file `DCIheader.asm`. The header structure is defined in the file `DciStruct.h` on the GPIB-VXI/C distribution disk.

The pre-initialization segment structure definition follows the header block in the file `DCIheader.asm`. The first entry in the pre-initialization segment is a branch instruction to the CI's Pre-init routine. The Pre-init branch instruction is followed by a variable-length field containing the CI initialization (setup) parameters.

CI Initialization Routine and Process Structures

The GPIB-VXI/C runs the pSOS real-time, multitasking operating system kernel. A CI consists of a pre-initialization routine and a set of routines contained in a minimum of two pSOS processes that perform the CI physical device emulation. The two required processes are called the *Asynch* and *Worker* processes.

The following sections explain the purpose and responsibilities of the pre-initialization routine and the Asynch and Worker processes. The CDS 73A-852 adapter resident CI is used to illustrate a specific implementation of a CI.

Use the 852 adapter CI as a shell for developing your CI applications by modifying the device-dependent sections of the code. Source files are provided in text-readable source on your CI distribution disk.

The Pre-init Routine

The pre-initialization, or Pre-init routine is a function call that you use to link your CI into the runtime system. Each CI must contain a Pre-init routine.

Before it calls the Pre-init routine, the GPIB-VXI/C places all the information it knows about the CI and its environment into the CI's header. This information includes the following:

- The CI's logical addresses
- The CI's Commander's logical address
- A pointer to a list of the CI's Servants
- The GPIB-VXI/C's logical address
- The GPIB-VXI/C's Commander's logical address
- Pointers to the pSOS, pREP/C and National Instruments-supplied libraries

- The Static CI's RAM area base, number of 4-kilobyte blocks and start block number

  or

  the PI CI's dynamic RAM size required
- The Asynch message exchange ID
- The Asynch and Worker process IDs
- The CI's local command set parser ID
- The base address of GPIB-VXI/C's dual-ported RAM in the local 68070's address space
- The base address of GPIB-VXI/C's dual-ported RAM in VMEbus A24 address space

As part of the CI initialization, the GPIB-VXI/C calls the CI's Pre-init routine, which fills in header entries containing pointers to the Worker and Asynch process entry points, and the initial values of the CI's VXI registers. This information is used to link the CI into the GPIB-VXI/C's runtime environment. Executing the Pre-init routine is analogous to being in the VXI CONFIG state.

To avoid halting system execution, the Pre-init routine must not access the following:

- The local command set
- Serial port routines (printf, and so on)
- The Word Serial driver

Note: Remember that the CI is not up and running in the Pre-init routine. Neither the Asynch nor the Worker process has been spawned and many of the resources needed by the CI have yet to be created. The information supplied by the CI in the Pre-init routine is needed in order to create these resources.

The Pre-init routine method of initializing header entries is required for ECIs because the header is not downloaded into RAM; the space for it is simply allocated and the Pre-init routine must fill in the correct values. While you can initialize a DCI's header by downloading the initial values to it, National Instruments strongly recommends the Pre-init routine for uniformity and ease of migrating a DCI to an ECI. The sample shell CI for the CDS 852 adapter, located on the GPIB-VXI/C Distribution Disk Code Instrument Shell File, shows how to implement a Pre-init routine.

The Asynch Process

The purpose of the Asynch process is to simulate an interrupt service routine. In this manner, the Asynch process handles all asynchronous events for the CI. These events include the following:

- Reception of a Word Serial command from the CI's Commander
- Reception of an event/response signal/interrupt from one of the CI's Servants
- Notification of reception of a Word Serial data buffer
- A change in the value of the CI's Control register
- A request for output by the CI's Commander
- Notification that a Servant of the CI has failed

CI's do not handle hardware interrupts directly. The GPIB-VXI/C firmware automatically handles hardware interrupts and routes the necessary information to the appropriate CI in the form of an Asynch exchange message. These hardware interrupts include the following:

- VXI signals
- VXI interrupts
- SYSFAIL (device failures)
- Physical Word Serial Servant interrupts.

Hardware Interrupts

While all of the hardware interrupt conditions discussed previously are handled through the use of the Asynch process's message exchange, the GPIB-VXI/C's trigger interrupt conditions are handled directly at the interrupt service routine level. The function `SetTrigHandler` installs an interrupt service routine handler for a specified trigger line. Consult the *GPIB-VXI/C Trigger Functions* section of Chapter 6, *C Function Calls*, for further information.

The Worker Process

The Worker process initializes any resources it needs for its own operation, sets its PASSED bit, and waits for a Word Serial *Begin Normal Operation* command. The Worker process can then perform any device-dependent operations and communicate with the CI's Commander and Servants.

The GPIB-VXI/C sends the Asynch process a pSOS message when it receives a Word Serial data buffer for the CI from its Commander. The Asynch process signals the Worker process, which in turn calls the Word Serial Servant driver to read, or get the pointer to the input buffer. The Worker process can interpret the buffer either as a device-dependent command or as data. The Worker process calls the Word Serial Servant driver to output data to its Commander. If the CI's Commander is another CI, the output data is transferred (its pointer is passed) to the Commander CI. If the Commander is the GPIB-VXI/C, the output data is transferred out through the GPIB. If the Commander is the GPIB-VXI/C's Commander, the output data is transferred out through the physical Word Serial registers to the Commander of the GPIB-VXI/C. The software interface for outputting data is the same regardless of the destination.

In addition to this general functionality, the Worker process can create other pSOS processes to handle other functions, such as running the GPIB-VXI/C's serial port or performing background computations on the 68881 math coprocessor.

Chapter 3
Designing a Code Instrument

This chapter contains information that will help you make design decisions about your Code Instruments (CIs). The types and capabilities of CIs are described in detail. This chapter also explains how you can use the National Instruments-supplied source code to implement your CI applications. This information, in conjunction with the source code, assists the system analyst and the software engineer in producing CI designs at the system and implementation levels.

The CI Design Process

To design a CI you first need to define your goals. The critical design questions are:

- How many physical instruments do you want to control with CIs?
- What types of CIs are required?
- How many CIs are needed?

Use the following steps as a guide to answer these questions.

1. Categorize your CIs by functionality according to the following general categories:
    - Single Message-Based interface for Register-Based or non-VXI device
    - Single Message-Based translation interface for another Message-Based device
    - Virtual super-instrument consisting of a hierarchy of multiple Message-Based, Register-Based, and/or non-VXI devices
    - Message-Based routing device for sub-instrument hierarchies
    - CI router or sub-CIs controller
    - Any combination of the above with a CI function library
    - CI function library only
    - Any other combination or derivative of the above
2. Calculate the total number of CIs required. Each GPIB secondary address link and/or type of CI in the previous list requires one CI. If you have more than two or three CIs in your system, or if memory requirements are high (greater than a total of 64 kilobytes) upgrade the installed RAM configuration to at least 1 Megabyte (this step will give you over 512 kilobytes to work with). If you have more than five or six CIs, consider using more than one GPIB-VXI/C. You can use any number of GPIB-VXI/Cs concurrently in a system. While you can run any number of CIs on one GPIB-VXI/C, performance and memory retention degrade with an increase in the number of active CIs and secondary address links. To the 12 to 14 processes in the baseline runtime system, add one process for each secondary address link and at least two processes for each CI. The runtime system is entirely event driven, which means that it uses CPU time only when it is asked to perform a task (for example, a GPIB transfer or a local command). This implementation maximizes the CPU time available for CI processes, but there is a practical limit to the number of processes that the 68070 can

handle. If several processes are waiting for CPU time at any one time, the system will become bogged down.

3. Consider how the software and hardware support provided by the GPIB-VXI/C can help you accomplish your goals. Table 3-1 is a list of general capabilities of a CI with a short description of the method used to perform the capabilities.

Table 3-1. Code Instrument Capabilities

CI Capability	Method to Perform Capability
Access Resource Manager (RM) and system configuration information.	Access structure directly using structure pointer stored in CI header.
Receive data from the GPIB.	Call Servant Word Serial Read pSOS driver.
Put data on the GPIB.	Call Servant Word Serial Write pSOS driver.
Read/Write Word Serial to or from a Servant device.	Use pSOS local driver routines for Word Serial Commander capability.
Read/Write Word Serial to or from a Commander that is a CI or the GPIB.	Use pSOS local driver routines for Word Serial Servant capability.
Read/Write physical Word Serial registers if the CI's Commander is the physical Commander of the GPIB-VXI/C.	Use pSOS local driver routines for physical Word Serial Servant transfers.
Use serial port (UART).	Use standard C I/O calls (printf, getc, and so on).
Call basic C functions (sprintf, sscanf, strcpy, strcmp, and so on).	Use built-in pREP/C library calls.
Create pSOS exchanges/processes.	Use built-in pSOS system/library calls.
Access GPIB-VXI/C Local Command Set.	Use `DCIParserQuery`, `DCIParserEnq`, and `DCIParserDeq` function calls.
Send signal to Commander.	Call `SignalOut` function.
Receive signal from Servant.	Receive message *Signal In* on Asynch exchange.
Receive VXI interrupt Status/ID from Servant.	Receive message *Signal In* on Asynch exchange.
Write to Servant's Control register.	Call `ControlOut` function.

(continues)

Table 3-1. Code Instrument Capabilities (continued)

CI Capability	Method to Perform Capability
Get Control register change (for device-dependent changes).	Receive message *Control In* on Asynch exchange.
Access Servant's other VXI registers.	Get local base address of a device's registers from `VXIregBase` function call and access device directly.
Access A16 space.	Access directly using local base address offset for A16 window (FF0000h).
Access A24 space.	Access directly with corresponding A24 addresses staying within the amount of A24 that can be seen by the GPIB-VXI/C depending upon its RAM configuration. With 1 Megabyte of RAM installed, the window is 100000h to e7FFFFh.
Access A32 space.	Unavailable.
Perform memory-to-memory DMA (between two: A16, A24, or local).	Use `DMAmove` function for all DMA transfers.
Source a TTL or ECL trigger, start up the 16-bit counter or the dual 5-bit tick timer.	Use the function `SrcTrig`.
Configure the 16-bit counter.	Use the function `TrigCntrConfig`.
Configure the dual 5-bit tick timer.	Use the function `TrigTickConfig`.
Set up a crosspoint switch routing.	Use the function `MapTrigToTrig`.
Create a library of functions for all CIs.	Use sample to create library table and library interface for all CIs to use.

Using the 852 Adapter CI as a Shell

The 852 adapter CI is a VXI Message-Based interface for the CDS 73A-852, which is a pseudo-Message-Based non-VXI device. This CI illustrates how to directly control a device's registers over the VXIbus as well as how to put a Message-Based interface on top of a non-VXI or Register-Based device.

The main 852 adapter CI source code is located in the files `init.c`, `asynch.c`, `worker.c`, and `cds852.c`. The code is structured in such a way that you can easily modify it for your applications. In this respect, the 852 CI is a shell CI that you should use not only as a starting point for CI development but also as a continuous guide to VXI programming with CIs.

The following sections further explain the structure and operation of the 852 CI. The 852 CI code includes helpful comments and serves as a good tool for learning approaches to using some of the basic CI functionalities. While we guide you through the shell CI we recommend that you have the actual code in front of you to review. Throughout the CI development process, the actual code will always be your best reference tool.

Note: Never modify, even for your own use, any CI distribution files that this manual does not expressly define as files that can be changed. If you require any additional functions, add them by creating a new file. `DciLibCalls.asm`, which contains the functions to access the National Instruments-supplied libraries, should never be changed. Never modify or add to any of the include files, with the exception of certain sections of `S_EVENT.h`, `DEBUG.h`, and `DciStruct.h`. Create separate include files for your own purposes.

Following these simple guidelines will make it easy for you to incorporate future upgrades of the distribution files into your system.

The init.c File

The `init.c` file contains the Pre-init routine source code. The Pre-init routine contains all of the CI code needed to emulate the CONFIG state of a VXI instrument and to link the CI into the runtime system. The main functions of the Pre-init routine are as follows:

- Initializing the CI VXI registers
- Sending the GPIB-VXI/C runtime system the addresses of the Worker and the Asynch processes

If you desire, you can change the Manufacturer Identification Number and the Model Code values by editing the include file `DciStruct.h`, which `init.c` references.

Note: Use caution when modifying the Pre-init routine. The GPIB-VXI/C runtime system calls this routine directly. When the Pre-init routine is running, the CI is not yet up and running, and no CI processes have been spawned. Do *not* call any pSOS device drivers such as Word Serial or serial port I/O, nor perform any extended processing in the Pre-init routine.

You can make the following decisions in the Pre-init routine regarding the state of the CI:

- Self-assigning the CI's logical address
- Allowing the GPIB-VXI/C to dynamically assign the CI's logical address
- Aborting the CI initialization procedure

The Pre-init routine can change the logical address header entry supplied by the setup parameters if the logical address is FFh. If the CI leaves the logical address FFh in the Pre-init routine, the GPIB-VXI/C dynamically assigns the CI a logical address. Otherwise, it uses the logical address the CI assigned itself to, assuming it does not conflict with another device. If a conflict exists, the CI is assigned the next unused logical address.

The Pre-init routine should return a Boolean value. If the value is TRUE (not zero), the runtime system continues the CI initialization process. If the value is FALSE (zero), the runtime system aborts the CI initialization procedure. This feature is useful if, for example, the Pre-init routine scans the device table and finds that the CI's target Servants are not present and determines that the CI will not be able to function without them. You can use this feature to create libraries of CIs for entire sets of instruments and have the CIs run only when the appropriate devices are in the system.

The asynch.c File

The `asynch.c` file contains the main code for the Asynch process. The Asynch process is responsible for handling all of the asynchronous events for the CI, including receiving the following:

- A Word Serial command from the CI's Commander
- An event/response signal/interrupt from one of the CI's Servants
- A Word Serial data buffer input notification
- A request for output from the CI's Commander
- Notification that a Servant device has failed
- A change in the value of the CI's Control register

Note: The Asynch process is intended only to emulate an interrupt service routine, not to perform extended operations. Minimizing the Asynch processing time maximizes the CI's event handling capacity and guarantees the best possible synchronicity not only between the GPIB-VXI/C system and the CI, but also between the Asynch and Worker processes.

The runtime system sends asynchronous events to the Asynch process through the CI's pSOS Asynch message exchange. A pSOS message consists of four longwords. The first longword (four bytes) of the message is the Asynch event code and the last three are the parameters. Most Asynch event codes use only one of the parameters. Any event code above 80000000h can be used for user-specific CI intercommunication. The meaning of the event code and the parameters would, of course, be user defined.

Never use the Asynch process to call the GPIB-VXI/C parser, nor to send any Word Serial commands to a Servant that might generate events such as *Unrecognized Command*. Because the Asynch process also handles signals, the operation would time out before the Word Serial driver could see the signal.

Handling Word Serial Commands and Queries

The Asynch process calls the function `HandleWordSerialCommand` from its main loop to handle Word Serial commands and queries. This structure makes it possible to handle every possible Word Serial command, whether it is a standard Word Serial command defined by the VXIbus specification or a user-defined command. Library functions are provided to handle many VXIbus-defined commands.

The function `HandleWordSerialCommand` is a large switch statement that cases off each Word Serial command. If the command requires a response, `HandleWordSerialCommand` sends it to the CI's Commander via the Servant Word Serial driver. This function handles the general VXI requirements for Word Serial commands. You need only add device-dependent code to satisfy the VXI requirements of your particular application. The following paragraphs include more details about various VXIbus-defined Word Serial commands:

- **Read Status Byte**

  To change the CI's status byte, simply change the variable `DCIheader.StatusByte`. The Request Service (RSV) bit, bit 6 of the status byte, is automatically cleared by the handler per the IEEE-488 specification. If your CI sets RSV, it should send a Request True (REQT) event signal to its Commander to notify it of the request for service by using the `SignalOut` function, an example of which is given in code. If the CI has a GPIB secondary address, the GPIB-VXI/C will assert SRQ for the CI when the REQT event is received.

- **Clear**

  The *Clear* handler automatically aborts any Word Serial buffer transfers with both the Commander and all the Servants of the CI. This command also clears all pending operations and responses on the GPIB-VXI/C local command set parser, and propagates the *Clear* state to all of the CI's Message-Based Servants. You can supply code to reset the CI's Register-Based or non-VXI Servants to a known, idle state, which may or may not be identical to the startup state.

- **Trigger**

  The Word Serial *Trigger* command is used to synchronously trigger a CI. This function is totally device dependent. Modify the code as necessary for your purposes.

- **Release Device**

  It is impossible for a CI to *say no* to this command, so you must supply code to release the Servants. The library call National Instruments supplies for handling *Release Device* updates the device table of the GPIB-VXI/C. You must handle application-specific problems associated with losing control of the device. Because the CI may not be able to recover gracefully, you may find it necessary to take action as far as deleting the CI after the Servant device is released. A more graceful approach to this situation, though, would be to suspend CI operations (or let the CI continuously report an error) until the Servant is re-granted to the CI, if ever.

- **Read Protocols**

  This query causes the CI to report the protocols that the CI supports. You can change these freely. For instance, if you add IEEE 488.2 support to the CI, you should assert (clear to 0) the I4 bit in the response.

- **Control Response, Control Event, and Asynchronous Mode Control**

  The shell code uses the full range of these commands. The responses to these queries are recorded in the RM table as well as in the header of the CI, and are used internally by the Word Serial drivers.

- **Device-Dependent Commands/Queries**

  You are free to define and use Word Serial commands to enhance the operation of your CI. Stay within the functionality of the VXIbus specification, however. Device-dependent functionality can lead to communication problems between CIs and their Commanders or Servants (that is, both sides of the communication path must have the same interpretation of each command).

Handling Physical Word Serial Commands and Queries

The CI's GPIB and physical Word Serial paths can be made or broken independently of one another. You can use the local command `LaSaddr` to assign a GPIB address to your CI or to remove the GPIB address link from any Message-Based device, CI, or the GPIB-VXI/C local command set. You can also use the `WordSerEna` command to assign control of the physical Word Serial path to your CI. Used together, these commands make your CI capable of having two mutually exclusive Word Serial paths active at once. First, the CI has its own logical address, which is hooked up to the GPIB. Secondly, when the GPIB-VXI/C is disconnected from its physical Word Serial path, the CI is emulating the GPIB-VXI/C's logical address. Two sets of VXI registers–one physical and one CI software version–and two communication paths are active, one for each set of registers.

If your CI assumes control of the GPIB-VXI/C's physical Word Serial registers, the GPIB-VXI/C connects the physical Word Serial Servant path from the GPIB-VXI/C's Commander directly to your CI through the GPIB-VXI/C's Word Serial Servant port. The GPIB-VXI/C uses a separate pSOS message code with the same parameter format as the CI Word Serial pSOS message to notify your CI that it has received a Word Serial command through the physical path. The CI can use the same method to handle commands and queries from the physical path and from the GPIB path, or it can use very different approaches. The only required difference is in the driver call for returning a response to a command, if applicable. In this case, the logical address parameter should be the GPIB-VXI/C's logical address, rather than the CI's logical address. Using this parameter maintains the perspective of the GPIB-VXI/C's Commander, in that it is communicating with the GPIB-VXI/C's logical address and not that of the CI. See the file `SampleWSsl.c` for an example of handling physical Word Serial commands and queries.

Handling Event/Response Signals/Interrupts

The GPIB-VXI/C receives an event or response signal/interrupt when another device writes to the GPIB-VXI/C's Signal register or generates a VXI interrupt on a level that the GPIB-VXI/C is configured to handle. The runtime system converts all VXI interrupts to signals and routes each signal to the Commander of the device that generated the signal. This process occurs by reading the lower eight bits of the signal to find the logical address of the sender, determining the Commander of that logical address from a table, and sending the signal to the appropriate device. If the Commander is unknown (it is neither the RM nor in the GPIB-VXI/C immediate hierarchy), the signal is written to the Signal register of Logical Address 0 (the RM). If the Commander is a CI, the runtime system sends the event/response signal to the CI in the form of a *Signal In* Asynch message.

The `HandleSignal` function handles all of the event and response signals for the CI. The shell code is based upon a signal handler for a Message-Based Servant device and includes a handler for the *Unrecognized Command* event. For this event, it aborts the Word Serial command transfer in progress, which causes the Word Serial driver to return an error code to the routine

that called the Word Serial driver (probably code in the Worker process). The most likely signal handler routine you would need to augment is the REQT event handler. This event notifies the CI that one of its Servants is requesting service. If this action causes the CI to need service, the CI may send an REQT event signal to its Commander. If the CI has a GPIB address, the GPIB-VXI/C will assert SRQ for it. The CI would then, most likely, be serial polled with a Word Serial *Read Status Byte* query.

If the signal handler needs to handle a Register-Based device (or other non-Message-Based device), the interpretation of the upper eight bits of the signal is completely device dependent. You will need to add appropriate code to handle the contents of these bits.

If the CI requires the reception of non-VXI-compatible VXI interrupt Status/ID values (for example, VME 8-bit Status/ID values), the CI can call the function `ReserveHandle` to have all interrupts for a particular GPIB-VXI/C VXI interrupt handler routed to the CI without interpretation. Both RORA and ROAK interrupters are supported.

Remember that the CI usually receives signals from its Servants. Exceptions to this rule include Shared Memory Protocol events that, when Shared Memory is fully defined, will be used for peer-to-peer as well as Commander-to-Servant communication, and device-dependent (user-defined) signals that may or may not be peer-to-peer events.

Handling Word Serial Data In Notices

The *Word Serial Data In* Asynch event message notifies the CI that the runtime system has placed a Word Serial buffer in the CI's Word Serial Input queue. The shell CI performs this process by incrementing the queue count and signaling the Worker process to read and process the buffer. This implementation results in good synchronicity and efficient use of CPU time. The buffer structure is referred to as a *CommonBuf*, and is defined in the file `CommonBuf.h`.

Handling DOR Requests

The runtime system uses the *Data Out Ready*, or *DOR Request* Asynch event to notify the CI that its Commander (CI or GPIB-VXI/C) attempted to read data from its output queue, but no data was present. A CI does not always need to handle this event, but it must do so if, for example, it has a GPIB address and it must get data from a Servant each time it is addressed to talk. If the CI has a GPIB address, it will receive *DOR Request* each time it is addressed to talk while not in IEEE-488 Serial Poll Mode State (SPMS). One example where this functionality is very useful is in controlling an instrument that has a continuous measurement mode. The CI handles *DOR Request* by signaling the Worker Process to get a data buffer from the Servant and return it to the CI's Commander.

The CI does not need to handle *DOR Request* if a CI's Commander sends it a command buffer for each data buffer it expects in return.

Handling Control Register Writes

Because a CI does not have a physical VXI Control register, the *Control In* event message is used to emulate a write to the Control register. However, because all of the VXI-defined functionality of the register is supported through other means, you need only add the code to

support your device-dependent functionality. You can use the *Control In* event to change the state or mode of your CI as you feel appropriate.

The worker.c File

The `worker.c` file contains the main code for the 852 adapter CI's Worker process. For the most part, it is independent of the 852 adapter. The specific code for performing such tasks as reads, writes, and resets of the 852 adapter is contained in the file `cds852.c`.

The Worker process manages most of a CI's functionality, including the following:

- Handling the VXI startup sequence
- Reading and processing Word Serial message buffers
- Incorporating application-specific features and behavior

The `worker.c` shell file also contains the following examples:

- Searching the device tables for device information
- Using the GPIB-VXI/C local command set
- Using the Commander Word Serial interface
- Accessing nonvolatile configuration information
- Using the onboard 68881 math coprocessor
- Performing DMA memory-to-memory operations

Note: Avoid polled I/O operations with Servant devices, or at least limit their use to an absolute minimum. The GPIB-VXI/C is a single processor system running a multitasking operating system. Its performance degrades as the number of active CIs and GPIB address links increases, especially when one or more CIs are performing polled I/O operations. Fully event-driven CI implementations will maximize your system performance.

Handling the VXI Startup Sequence

The startup sequence includes the following processes:

- Performing self-tests
- Setting the Passed bit
- Waiting for the CI's Commander to send it the *Begin Normal Operation* command as defined in the VXIbus specification

Because the GPIB-VXI/C enters the Normal Operation state before the Worker process is initialized, the CI can perform any operations it requires to test its own functionality or that of its Servants before it receives the *Begin Normal Operation* command. This testing may involve VXIbus accesses.

Handling Word Serial Messages

The runtime system sends message buffers to the CI through its Word Serial Servant path from the CI's Commander or from the GPIB. Getting a Word Serial buffer simply involves waiting for notification from the Asynch process that a buffer is present in the CI's input queue, and calling the Word Serial Servant driver with an address in which to place the pointer to the buffer. Remember that it is the responsibility of the CI to free the buffer when it is through with it. If the Worker process passes the buffer on to a Servant, it can call the Word Serial Commander driver in such a manner that the buffer is freed automatically. However, if the buffer is simply parsed or transmitted via another output mechanism, rather than back out to the Commander, the CI must free the buffer.

The procedure for processing the message buffer is, of course, application dependent. If you implement a CI with a local command set, it should parse and execute the buffer and free it when it is done. When getting data from the GPIB, the GPIB-VXI/C runtime system allocates buffers out of pSOS dynamic memory. They must be freed so that pSOS can use them again later. The 852 adapter CI implements a superset of the 852's local command set, which means that the 852 adapter CI handles certain forms of message buffers itself, and passes the rest on to the 852 module. The 852 adapter CI handles message buffers that have command strings starting with the double exclamation point (`!!`) sequence in order to avoid conflict with the possible command sets of the 852 modules. All other buffer formats are passed on to the 852 module. In either case, the buffer is freed when the operation is complete.

For response buffers, the 852 CI waits for a *DOR Request* signal from the Asynch process indicating that it has been addressed to talk on the GPIB. The CI then requests a buffer of data from the 852 module, which it sends out through its Servant Word Serial path to the GPIB or the CI's Commander.

Accessing Nonvolatile Configuration Information

A 256-byte EEPROM stores the GPIB-VXI/C nonvolatile configuration information. Half of this space is reserved for the runtime system. The other 128 bytes (32 longwords) of information can be used by CIs in any manner required by the application.

The CI can use the function `NVconf` to retrieve one of the longwords contained in this EEPROM. The GPIB-VXI/C reads the EEPROM at boot time and keeps its contents in memory. This function is very fast, as it simply returns data from an array. You can modify the EEPROM contents with the Nonvolatile Configuration Editor as described in Chapter 4, *Nonvolatile Configuration*, of the *GPIB-VXI/C User Manual*.

Using the 68881 Math Coprocessor

The 68881 math coprocessor is a peripheral device that you can access using a National Instruments-supplied interface. You can use this interface for actual inline 68881 floating point code. The floating point instructions are trapped and processed peripherally with the 68881. A separate 68881 floating point register context is kept for each process that uses the 68881 interface. Because the 68881 is not used by any of the GPIB-VXI/C runtime system processes, CIs have sole access to the 68881. Each CI process that incorporates the 68881 must call the function `Enable68881` before executing any code that uses the 68881. The 852 adapter CI has an example of the use of this function call for the Worker process and the use of floating point calls. Because the 68881 is being used as a peripheral device, simple operations such as addition

and multiplication may be only as fast as, or even slower than an optimized math library. However, transcendental functions such as cosine and log will be many times faster than the fastest math library. Also, because using the 68881 does not require any additional library code, the CIs stay very small.

Implementing a CI Function Library

The files `UserLibCalls.asm`, `UserLibTable.c`, and `UserLibCallsInit.c` comprise an example of how you can create a shared CI function library. A CI may have multiple libraries contained in it with *N* functions in each. These libraries may be accessed by the CI and by any other CIs that have the correct library interface functions. A CI library can be accessed only by CIs on the same GPIB-VXI/C.

A shared CI library is implemented in the same manner as the National Instruments-supplied libraries. One CI is designated as the library holder and the others as remote library accessors. The holder CI contains all of the functions and a table of address pointers to these functions (see the example in `UserLibTable.c`). The initialization routine in `UserLibTable.c` pokes the address of this table into the user library section of its own CI header. The accessor CI only needs to know the logical address of the holder CI, which it can determine through its knowledge of the structure or by looking it up in the device table.

For an accessor CI to freely access a library in a holder CI, you must first define an assembly code file that externally references the function names and calls the function in the holder CI (see the example in the file `UserLibCalls.asm`). Secondly, the accessor CI must call the initialization function in `UserLibCallsInit.c` using the logical address of the holder CI. This procedure will enable the accessor CI to call all of the functions directly, as if the functions were part of its own code.

For an example of an existing library, refer to the National Instruments-supplied libraries. The libraries' external definitions and non-library holder interface are contained in the file `DciLibCalls.asm`.

Chapter 4
Creating a Downloaded Code Instrument

This chapter contains information you will need to create a Downloaded Code Instrument (DCI). It addresses compatibility considerations for DCI's and EPROMed CI's (ECIs) and outlines the procedure for constructing and running a DCI.

Static and Position Independent ECI Compatibility Issues

Segment Ordering

As explained in Chapter 2, the memory structure (segment location and ordering) differs between DCIs and ECIs. In this context, a segment is a logical grouping of code, data, or both by the linker. In order to circulate freely between DCIs and ECIs, with their different memory map structures, the CI must be organized as multiple object segments so that the GPIB-VXI/C runtime system will be able to calculate the locations of the header and the pre-initialization section. Converting a DCI to an ECI will take little effort if you organize your DCI as shown in Table 4-1.

Table 4-1. DCI Segment Ordering and Content

Segment	Type	Description
1	Uninitialized Data	CI Header
2	Code	Pre-initialization Segment Assembly Code
3	Code	CI C and Assembly Code
4	Initialized Data	Strings, Static Variables, Any Other Initialized Data
5	Uninitialized Data	Modifiable Global Variables (Locals Allocated on Stack)

To produce this multiple-segment organization, your compiler should let you specify separate object segments in which to place the C code and data. You will need an assembler directive such as SECTION that can specify segment locations for assembly code and data.

In numbering schemes, many compilers use the following section assignments for C:

- Code in section 9
- Initialized data in section 13
- Uninitialized data in section 14

Any assembly code you create should be consistent with these C section assignments.

The linker automatically places the pre-initialization, code, and initialized data segments in RAM for DCIs and in EPROM for ECIs. You do not have to modify the supplied code. Table 4-2 describes the segment ordering for ECIs.

Table 4-2. ECI Segment Ordering and Content

Segment RAM	Segment EPROM	Type	Description
1		Uninitialized Data	CI Header
	1	Code	Pre-initialization Segment Assembly Code
	2	Code	C and Assembly Code
	3	Initialized Data	Strings, Static Variables, Any Other Initialized Data
2		Uninitialized Data	Modifiable Variables

Initialization

The GPIB-VXI/C must have the following information to link a CI into the runtime system:

- The CI's logical address
- The CI's Commander's logical address
- The CI's RAM requirements:
  - For Static CIs, its starting global RAM block number and number of blocks needed
  - For Position Independent (PI) CIs, the size of dynamic RAM requirements
- The size of the Worker process stack
- The logical address for each Servant
- The Pre-init routine entry point

All but the last item in the preceding list are known as the CI setup parameters. The GPIB-VXI/C finds a DCI's setup parameters through the local command `DCISetup?` (for Static DCIs) and through `DCISetupPI?` (for PI CIs). An ECI's setup parameters are located at offset 6 in the pre-initialization segment, which is located at the base of the CI's EPROM area.

The Pre-init routine entry point is located at offset 0 in the CIs pre-initialization segment (offset 100h in a DCI's RAM area, offset 0 in an ECI's EPROM area). The entry point must be a *branch word* instruction (exactly six bytes) that branches to the Pre-init routine. Notice that any other instruction would intrude into an ECI's setup parameter area.

Static DCI Creation Sequence

The following procedure illustrates how to compile, link and run a Static DCI.

1. Determine a location for your CI's RAM base (starting block number) relative to the global CI RAM area base.

2. Compile the CI's C and assembly source files to create linker object files.

3. Create the CI's binary image by linking together all of the segments in the order specified in Table 4-1, starting at the DCI's RAM area base address.

4. Determine the number of 4-kilobyte RAM blocks the DCI needs by dividing the image's size by 4096. Include all five segments and any shared data areas.

5. Use the GPIB-VXI/C local command `DCISetup?` to prepare the GPIB-VXI/C to download and run the DCI, as described in Chapter 3, *Local Command Set*, of the *GPIB-VXI/C User Manual*.

6. If `DCISetUp?` returns an error, use the error code to diagnose the problem. Recompile and relink the CI if necessary, and repeat the `DCISetup?` command.

7. Use the GPIB-VXI/C local command `DCIDownLoad` to download the DCI to the GPIB-VXI/C.

The GPIB-VXI/C will automatically initialize the DCI and link it into the runtime system.

Creating a DCI Memory Map

If you are implementing multiple Static CIs that you want to install concurrently, you must create a RAM memory map to avoid memory conflicts at download time. Create and use a Static DCI memory map using the following steps.

1. Determine the base and size of the global CI RAM area, based on the amount of installed RAM and on the anticipated RAM requirements of all DCIs. CI process stacks are allocated out of pSOS dynamic memory and do not need to be included in this calculation.

2. Allocate contiguous groups of 4-kilobyte blocks for each DCI's RAM area, based upon the DCI's RAM requirements.

3. Configure the GPIB-VXI/C's global CI RAM area as described in the *Static DCI Memory Structure* section of Chapter 2.

DCI Memory Image Format

A UNIX makefile is included with the CI distribution files as an example of how to create binary and S-record images. The makefile is compatible with the Green Hills C Compiler. The makefile selectively recompiles the C and assembly source files to create link object files. It then links the object files to create an executable image.

Use the binary memory image form when linking a Static DCI to be compatible with the `DCIDownload` command.

PI DCI Creation Sequence

The following procedure illustrates how to compile, link and run a PI DCI.

1. Compile the CI's C and assembly source files to create linker object files.

2. Create the CI's binary image by linking together all of the segments in the order specified in Table 4-1. Because the code and data are position independent, the base address is meaningless.

3. Determine the amount of dynamic RAM you will require for this PI DCI. This is the size of the final object file, which includes all of the code and data.

4. Use the GPIB-VXI/C local command `DCISetupPI?` to prepare the GPIB-VXI/C to download and run the PI DCI, as described in Chapter 3, *Local Command Set*, of the *GPIB-VXI/C User Manual*.

5. If `DCISetupPI?` returns an error, use the error code to diagnose the problem. Recompile and relink the CI if necessary, and repeat the `DCISetupPI?` command.

6. Use the GPIB-VXI/C local command `DCIDownLdPI` to download the PI DCI to the GPIB-VXI/C.

7. The GPIB-VXI/C will automatically allocate a dynamic buffer and place the code and data in it. The GPIB-VXI/C will then initialize the PI DCI and link it into the runtime system.

Because all data and code are relative for a PI DCI, no memory map is needed.

Chapter 5
Creating an EPROMed Code Instrument

This chapter describes how to convert a Downloaded Code Instrument (DCI) to the form of an EPROMed Code Instrument (ECI).

ECIs and DCIs have two basic differences.

- An ECI is contained partially in EPROM and partially in RAM, while a DCI is wholly contained in RAM.
- The runtime system must access an ECI's setup parameters in a static manner so that it can automatically initialize the ECI at system startup.

By following the DCI memory segmentation suggestion given in Chapter 4, you can handle the first difference by reordering and relocating the memory segments at link time.

The runtime system gets an ECI's setup parameters from a variable-length field located in EPROM. You can modify your CI's setup data field by editing the file. The ECI setup parameter order and field sizes are the same as the `DCISetup?` and `DCISetupPI?` command parameters. However, because not all of the fields are the same, slight changes have been made to distinguish between the two types. If offset 8, the StartBlock field for Static ECIs, contains an FFFFh, the GPIB-VXI/C runtime system assumes that the CI is a Position Independent (PI) ECI. Figure 5-1 illustrates the differences between the Static ECI and PI ECI EPROM image section.

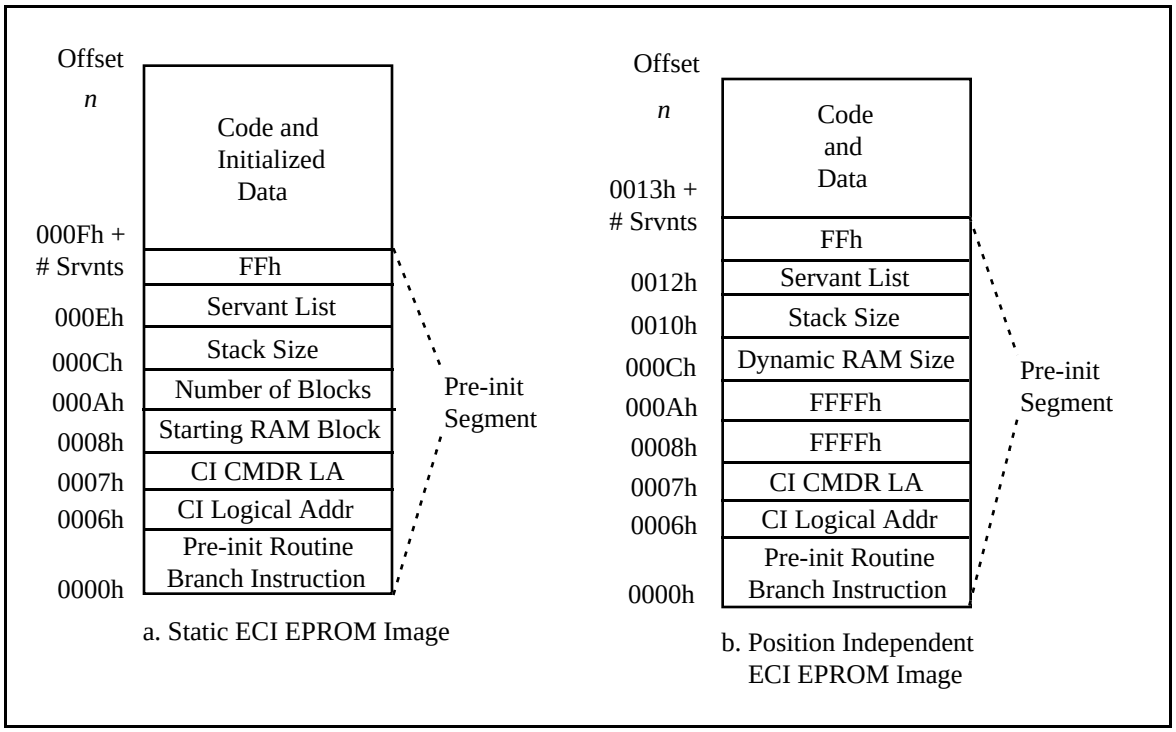


Figure 5-1. Static ECI EPROM Image Versus PI ECI EPROM Image

Creating A Static ECI Memory Map

Creating a memory map is very similar for Static ECIs and Static DCIs. Because an ECI's code and initialized data are in EPROM and its variable data are located in RAM, you must create separate memory maps for RAM and EPROM. The RAM memory map is similar to that for a Static DCI, but it needs to accommodate only each CI's header and uninitialized data segments. This tends to be a very small portion of the CI, probably needing only one global CI RAM block.

You can locate each ECI in the global EPROM area in any order you wish, because the 512-kilobyte global EPROM area (E80000h to EFFFFFh) is not segmented into blocks like the global CI RAM area. The only restriction is that each ECI's base address in EPROM must be an even address (word-aligned) as with all 68000-based programs.

Note: When creating the ECI memory map, log the address at which the base of each ECI image resides in EPROM. This address must be entered into the nonvolatile configuration section for Resident CIs (RCIs) to notify the GPIB-VXI/C runtime system to boot the ECI at system startup.

Creating a PI ECI Memory Map

If multiple PI ECIs are to reside in the same bank of EPROMs, you must create a PI ECI memory map so that code segments of different PI ECIs do not overlap. You need to ensure only that the PI ECIs are placed at mutually exclusive addresses in the EPROM space. You can locate each ECI in the global EPROM area in any order you wish, because the 512-kilobyte global EPROM area (E80000h to EFFFFFh) is not segmented into blocks as is the global RAM area. The only restriction is that each ECI's base address in EPROM must be an even address (word-aligned). Also remember when relocating a PI DCI to EPROM that the dynamic RAM requirements need to be lowered considerably because the code and initialized data no longer reside in RAM.

Note: When creating the ECI memory map, log the address at which the base of each ECI image resides in EPROM. This address must be entered into the nonvolatile configuration section for RCIs to notify the GPIB-VXI/C runtime system to boot ECI at system startup.

ECI Memory Image Format

We have found that the Motorola S-record image is the easiest form to work with when you are creating an ECI, because many EPROM programmers will accept it. The S-record image is an ASCII format that represents the CI structure in the GPIB-VXI/C's EPROM. If your programmer will not accept Motorola S-record format, use the pROBE command `ECI` to change this to a binary image. If your EPROM programmer is having problems with the S-record format that your compiler outputs, use the pROBE command `ECI` to help you reformat the S-record file. The S-record output from the pROBE command `ECI` is very clean and compact. Chapter 8, *VXI pROBE,* contains further information about the National Instruments pROBE command `ECI`.

A UNIX makefile is included with the CI distribution files as an example of how to create binary and S-record images for a Static DCI and ECI. The makefile is compatible with the Green Hills C Compiler. If you are not running UNIX, the makefile is still a good example of the actual compiler and linker instructions needed.

Chapter 6
C Function Calls

This chapter contains descriptions of C language function calls for the GPIB-VXI/C firmware. The descriptions include the purpose, definition, and return values of the function calls, and examples of their use. This chapter also discusses interfaces to the National Instruments-supplied functions and to your user-defined functions and libraries.

The GPIB-VXI/C firmware includes function calls for the following purposes:

- Word Serial Protocol communications, including
  - Creating communication buffers
  - Clearing Word Serial communications
  - Changing a Servant's Control register contents
  - Sending a signal to the Code Instrument's (CI's) Commander
  - Handling *Abort Normal Operation* and *End Normal Operation*
  - Handling a request to release device
  - Requesting data from a Servant CI
- VXI register access, including
  - Calculating local address for register locations
  - Performing protected (from BERR*) read and write operations
- GPIB-VXI/C resource access, including
  - Reading the contents of the nonvolatile memory
  - Reading the contents of the Servant area or GPIB DIP switches
  - Accessing the 68881 numeric coprocessor
  - Accessing GPIB-VXI/C local commands
  - Accessing memory-to-memory DMA
  - Sourcing TTL Triggers
  - Handling RORA and ROAK VXI Interrupters
- GPIB-VXI/C trigger functions, including
  - Sourcing TTL/ECL triggers
  - Sourcing triggers into and out of the front panel
  - Configuring the 16-bit counter
  - Configuring the dual 5-bit tick timers
  - Setting up crosspoint switch and signal conditioning modes

The GPIB-VXI/C firmware also provides a mechanism to implement user libraries for CIs to share.

Interface to National Instruments-Supplied Functions

Your CI can call pSOS, pREP/C, and the National Instruments library functions through jump tables pointed to by the `pSOSTablePointer`, `pREPcTablePointer`, and `NITablePointer` entries in the CI header. As part of the initialization process, the GPIB-VXI/C runtime system inserts these pointers into the CI's header. In this manner, CIs will be compatible with future versions of GPIB-VXI/C firmware.

The distribution file `DCILibCalls.asm` contains a set of assembly language interface routines that your CI can use to access the National Instruments-supplied functions. When the CI calls one of these functions, the interface routine finds its location in the firmware and executes a `JMP.L` (or long jump) to the appropriate function. Separate handler routines are used for each function within each of the pSOS, pREP/C and National Instruments C libraries. When the function is complete, it returns to the caller of the interface, not to the interface itself. The stack and registers are not modified by the interface code. Notice that, in this manner, any code written to call these functions can call them as if they were actually part of the final object code of the CI.

The following technical explanation of how this function interface works will be helpful if you intend to make user-defined function libraries for CIs to share.

1. Each function has an inherent offset within each library jump table according to its position in the table. This offset is loaded into D0.

2. The handler routine for each function loads the address of the appropriate jump table pointer into register A0, and then uses this address to load the address of the jump table into A0.

3. The routine then uses the address of the jump table (now in the A0 register) and the offset within the table (in the D0 register) to load the address of the function into A0.

4. Finally, the routine jumps directly to the function. Because this process does not modify the contents of the stack or the stack pointer, the parameters for the function are still in their appropriate positions. In most C compilers, A0 and D0 are scratch registers, so they need not be saved. When the function performs a *Return from Subroutine* (RTS), it will return directly to the caller of the function interface, not to the interface itself.

Interface to User-Defined Functions/Libraries

The GPIB-VXI/C Distribution Disk Code Instrument Examples/Shell Files include three files that you can use to create and access user-defined libraries. These files are `UserLibTable.c`, `UserLibCallsInit.c`, and `UserLibCalls.asm`.

`UserLibTable.c` is an example of how to create a library for a Static CI that is accessible from other CIs. This file externally references the function names according to their function type. The file also builds a library function jump table by creating an array of static, pre-initialized, unsigned longwords with the names (addresses) of the functions as the

pre-initialized values. For a position independent approach, the table should be created at runtime with the function addresses calculated relatively.

The holder CI's header can contain pointers to up to five library jump tables in the entries (`DCI_FunctionTable[0 - 4]`). The holder CI's Pre-init routine must call the `InitUserLibrary` function to poke the address of each library's function jump table into the appropriate `DCI_FunctionTable` entry. In the example code, the library pointer table entry `DCI_FunctionTable[3]` points to the library function jump table associated with the stub functions (`DCI_UserLib_Table`).

You can fill in your routines where the stub function definitions are located at the end of `UserLibTable.c` or remove the stubs and link with the object of files that contain the user library routines.

`UserLibCallsInit.c` is an example of how a CI can locate a library held by another CI. For an accessor CI, you must define the logical address of the holder CI in this file by entering the holder CI's logical address in the `#define Lib_Logical_Address` statement. The accessor CI's Pre-init routine must call the `InitLibCalls` function to get the holder CI's header base address. The `InitLibCalls` function puts the holder's header address in the global variable `Lib_DCIheader`.

`UserLibCalls.asm` contains an assembly language interface that an accessor CI uses to access the functions defined in `UserLibTable.c`. This file must define the library number and jump table offsets for each function in a manner corresponding to the contents of `UserLibTable.c`. When the accessor CI calls a function, the interface routine loads its offset within the appropriate function table into the D0 register, and executes a branch to the user library call handler routine. The handler routine loads the address of the pointer to the appropriate jump table (calculated using the `Lib_DCIheader` global) into register A0, and uses this address to load the address of the library jump table into A0. It then uses the address of the jump table (in A0) and the offset within the table (in D0) to load the address of the function into A0. It then jumps directly to the function. When the function returns, it returns to the caller of the function interface, not to the interface itself. This is a very fast and efficient method to implement a shared library. Notice that, in this manner, any code written to call these functions can call them as if they were actually part of the final object code of the CI.

Word Serial Protocol Functions

The Word Serial Protocol functions described on the following pages are used in conjunction with the Word Serial driver calls described in Chapter 7, *Word Serial Drivers*.

- `CIAbortNormalOperation`
- `CIEndNormalOperation`
- `CIReleaseDevice`
- `ControlOut`
- `CreateBuf`
- `DCIwsClear`
- `DuplicateCommonBuf`
- `HWwsClear`
- `SendDORrequest`
- `SignalOut`

CommonBuf is the name given to a buffer structure used internally by the GPIB-VXI/C for all modes of data communications (GPIB, Word Serial, CI, and local command set). A CI can use the `CreateBuf` and `DuplicateCommonBuf` functions to create or duplicate CommonBuf buffers. The advantage of this common buffer type is that the buffer can be passed from any source to any destination without modification. Only a pointer to the buffer needs to be transferred. The CommonBuf structure is defined in the include file `CommonBuf.h`.

Note: Access CommonBuf elements by structure name, rather than by offset, to accommodate future changes in the CommonBuf structure.

The functions `DCIwsClear` and `HWwsClear` clear a CI's Word Serial communications with its Commander in the event it receives a Word Serial *Device Clear* command.

A CI can use the function `ControlOut` to change the contents of a Servant's Control register. A CI can also use the `SignalOut` function to send a signal to its Commander.

The `CIAbortNormalOperation` and `CIEndNormalOperation` functions handle runtime system-related operations associated with executing the Word Serial *Abort Normal Operation* and *End Normal Operation* commands for the CI. These operations include clearing the CI's Word Serial queues and aborting pending Word Serial operations with Message-Based Servants.

The `CIReleaseDevice` function handles the runtime system-related operations associated with executing the Word Serial *Release Device* command. These operations include aborting the CI's pending Word Serial operations with the Servant (if it is Message-Based), and updating the GPIB-VXI/C's system configuration table.

The `SendDORrequest` function is used to send a pSOS Asynch Exchange Message to a DCI to request it to output data. This function is commonly used when a Servant DCI is addressed to talk on the GPIB, but has no response in its output queue. The Commander DCI can send a signal to a Servant DCI using the `SendDORrequest` function to inform it that it should receive and return a response.

CIAbortNormalOperation

Purpose: Execute Word Serial *Abort Normal Operation* command.

Definition: `CIAbortNormalOperation(CIla)`

`CIla`	`uint8`	Logical address of the calling CI

Action: Handles the runtime system-related operations associated with executing the Word Serial *Abort Normal Operation* command for the CI. These operations include clearing the CI's Word Serial queues and aborting pending Word Serial operations with Message-Based Servants.

Returns: Response according to the VXIbus specification.

Example:

```
CIAbortNormalOperation(MyLA);
```

CIEndNormalOperation

Purpose: Execute Word Serial *End Normal Operation* command.

Definition: `CIEndNormalOperation(CIla)`

`CIla`	`uint8`	Logical address of the calling CI

Action: Handles the runtime system-related operations associated with executing the Word Serial *End Normal Operation* command for the CI. These operations include clearing the CI's Word Serial queues and aborting pending Word Serial operations with Message-Based Servants.

Returns: Response according to the VXIbus specification.

Example:

```
CIEndNormalOperation(MyLA);
```

CIReleaseDevice

Purpose: Execute Word Serial *Release Device* command.

Definition: `CIReleaseDevice(CIla, sla)`

`CIla`	`uint8`	Logical address of the calling CI
`sla`	`uint8`	Logical address of the Servant to release

Action: Handles the runtime system-related operations associated with executing the Word Serial *Release Device* command. These operations include aborting the CI's pending Word Serial operations with the Servant (if it is Message-Based), and updating the GPIB-VXI/C's system configuration table.

Returns: Response according to the VXIbus specification.

Example:

```
CIReleaseDevice(MyLA, sla);
```

ControlOut

Purpose: Send a control value to a Servant's Control register.

Definition: `ControlOut(la, control)`

`la`	`int8`	Logical address of Servant to send the control to
`control`	`uint16`	Control value to send

Action: Handles writing of a Control register value to a Servant regardless of whether the Servant is a CI or a physical device.

Returns:

0:	Invalid logical address.
1:	Control value successfully sent.

Example: Send control value 1234h to Servant at Logical Address 25.

```
int8 ServantsLa;
uint16 ctrl;


ServantsLa= 25;
ctrl= 0x1234;
•
•
•
result= ControlOut(ServantsLa, ctrl);
```

CreateBuf

Purpose: Allocate a variable-length CommonBuf from pSOS dynamic memory.

Definition: `CommonBuf *CreateBuf(size, timeout)`

`size`	`uint16`	Buffer size, in 256-byte (one-quarter kilobyte) increments
`timeout`	`uint32`	Timeout count, in pSOS ticks

Action: Create CommonBuf of size 256 * `size` (0 to 16 Megabytes) and return a pointer to the buffer within `timeout` pSOS ticks. If `timeout` = 0, wait forever for buffer allocation.

Returns:

0:	Timed out before able to allocate buffer.
*CommonBuf :	Buffer successfully created.

Example: Create 32-kilobyte buffer, timeout after 60 ticks.

```
CommonBuf *Bufptr;
•
•
•
Bufptr= CreateBuf(128, 60);
```

DCIwsClear

Purpose: Clear a CI's Word Serial communication with its Commander.

Definition: `DCIwsClear(MyLa)`

`MyLa`	`int8`	Logical address of the CI

Action: Clear any pending Servant Word Serial operations. Puts the Word Serial I/O channel in the Idle state.

Returns: None.

Example: Clear this CI's (at Logical Address 9) Word Serial communication with its Commander.

```
MyLa= DCIheader.DCILogicalAddress;

DCIwsClear(MyLa);
```

DuplicateCommonBuf

Purpose: Create a copy of a CommonBuf.

Definition: `CommonBuf *DuplicateCommonBuf(Cbuf,timeout)`

`*Cbuf`	`CommonBuf`	Pointer to buffer to be copied
`timeout`	`uint32`	Timeout count, in pSOS ticks

Action: Duplicate the CommonBuf pointed to by `Cbuf`, and return a pointer to the copy within `timeout` pSOS ticks. If `timeout` = 0, wait forever for buffer allocation.

Returns:

0:	Timed out before buffer allocated.
*CommonBuf :	Buffer successfully copied.

Example: Create a copy of the CommonBuf pointed to by `DataPtr`, and return the pointer to it to `CpyPtr`; timeout after 20 ticks.

```
CommonBuf *CpyPtr,*DataPtr;
•
•
•
CpyPtr= DuplicateCommonBuf (DataPtr,20);
```

HWwsClear

Purpose: Clear Word Serial communications with a Commander on the physical Word Serial path. Used by a CI that is in control of the physical Word Serial registers.

Definition: `HWwsClear(WSb_RET_Clear)`

`WSb_RET_Clear`	A flag defined in `IncludeWS.h` that gets ORed in with the return value from any pending Word Serial operations.

Action: Clear any pending Servant WS operations. Put physical response register in known state (Not DIR, Not DOR, Not Read Ready, but Write Ready).

Returns: None.

Example: Clear CI Word Serial communication via GPIB-VXI/C physical registers.

```
HWwsClear(WSb_RET_Clear);
```

SendDORrequest

Purpose: Send *Data Out Ready Request* signal to a Servant DCI.

Definition: `SendDORrequest(la)`

`size`	`int8`	Logical address of Servant DCI

Action: Sends *DOR Request* Asynch exchange message to DCI at Logical Address `la`.

Returns:

0:	`la` was not a DCI.
1:	DOR request message sent successfully.

Example: Send *DOR Request* signal to DCI at Logical Address 34.

```
SendDORrequest(34);
```

SignalOut

Purpose: Send a signal value to the CI's Commander's Signal register.

Definition: `SignalOut(la, signal)`

`la`	`int8`	CI's Commander's logical address
`signal`	`uint16`	Signal value to send

Action: Handles sending of a signal value to a device regardless of whether it is a CI or a physical device. In the case of a physical device, the function retries the signal on Bus Error.

Returns:

0:	Invalid logical address specified.
1:	Signal sent successfully.
-1:	Bus errors occurred on multiple attempts to write Commander's signal register.

Example: Send control value 1234h to Servant at Logical Address 25.

```
int8 ServantsLa;
uint16 ctrl;


ServantsLa= 25;
ctrl= 0x1234;
•
•
•
result= ControlOut(ServantsLa, ctrl);
```

VXI Register Access Functions

The following pages contain descriptions of the VXI register access functions.

- `SafeReadByte`
- `SafeReadWord`
- `SafeWriteByte`
- `SafeWriteWord`
- `VXIregBase`

To access registers other than the Signal and Control registers, use the `VXIregBase` function to get the base address of a device's VXI registers (in local address space), and then read or write the appropriate offset relative to the base address. The `VXIregBase` function returns the local address for physical devices and CIs.

Note: All Word Serial operations performed by a CI should be implemented through the Word Serial driver, rather than through direct register manipulation.

The protected read and write functions `SafeReadByte`, `SafeWriteByte`, `SafeReadWord`, and `SafeWriteWord` support graceful recovery from exception conditions resulting from invalid memory accesses to VME A16 and A24 space. These routines provide bus error handling at the C language level for byte and word accesses.

The range of A24 addresses that the GPIB-VXI/C's CPU can access is a function of the installed RAM configuration. The location of VME A16 and A24 spaces in the GPIB-VXI/C's CPU address map are as shown in the following table.

Table 6-1. VME A16 and A24 Space

VME Space	Installed RAM	VME Address Range	Local Address
A16	any	0000 to FFFF	FF0000 to FFFFFF
A24	512K 1024K 2048K 4096K	080000 to E7FFFF 100000 to E7FFFF 200000 to E7FFFF 400000 to E7FFFF	080000 to E7FFFF 100000 to E7FFFF 200000 to E7FFFF 400000 to E7FFFF

SafeReadByte

Purpose: Read a byte in protected environment.

Definition: `SafeReadByte(address, byte)`

```
*address, *byte    int8
```

Action: Read byte at address pointed to by `address`, and store it in `byte`.

Returns:

0:	Bus error occurred during read.
1:	Read successfully completed.

Example: Read address pointed to by `addr`, and store it in location pointed to by `b`.

```
int8 *addr, b;
•
•
•
result= SafeReadByte(addr, &b);
```

SafeReadWord

Purpose: Read a word in protected environment.

Definition: `SafeReadWord(address, word)`

```
*address     uint16
*word        uint16
```

Action: Read word at address pointed to by `address`, and store it in location of `word`. Return result indicator.

Returns:

0:	Bus error occurred during read.
1:	Read successfully completed.

Example: Read address pointed to by `addr`, and store it in location pointed to by `w`.

```
uint16 *addr;
uint16 w;
•
•
•
result= SafeReadWord (addr, &w);
```

SafeWriteByte

Purpose: Write a byte in protected environment.

Definition: `SafeWriteByte(address, byte)`

```
*address    int8
byte        int8
```

Action: Write `byte` to address pointed to by `address`. Return result indicator.

Returns:

0:	Bus error occurred during write.
1:	Write successfully completed.

Example: Write 65 to address pointed to by `addr`.

```
int8 *addr;
•
•
•
result= SafeWriteByte(addr, 65);
```

SafeWriteWord

Purpose: Write a word in protected environment.

Definition: `SafeWriteWord(address, word)`

```
*address     uint16
word         uint16
```

Action: Write `word` to address pointed to by `address`. Return result indicator.

Returns:

0:	Bus error occurred during write.
1:	Write successfully completed.

Example: Write 62354 to address pointed to by `addr`.

```
uint16 *addr;
uint16 w;
•
•
•
result= SafeWriteWord(addr, 62354);
```

VXIregBase

Purpose: Get the local base address of a device's VXI registers.

Definition: `uint16 *VXIregBase(la)`

`la`	`int8`	Logical address of the device

Action: Performs the address calculations needed to generate a pointer to a device's VXI register set regardless of whether it is a CI, the GPIB-VXI/C itself, or another physical device.

Returns:

*uint16:	Local pointer to VXI register base.
0:	Invalid logical address specified.

Example: Return register base address of device at Logical Address 15.

```
uint16 *base;
.
.
.
base = VXIregBase(15);
```

GPIB-VXI/C Resource Access Functions

The following pages contain descriptions of the GPIB-VXI/C resource access functions.

- `ClearParser`
- `DCIParserDeq`
- `DCIParserEnq`
- `DCIParserQuery`
- `Disable68881`
- `DMAmove`
- `Enable68881`
- `NVconf`
- `ReenableHandler`
- `ReserveHandler`

The `NVconf` function reads the contents of the user-configurable portion of the EEPROM, which consists of 32 longword locations (128 bytes).

A CI can perform many types of 68070 memory-to-memory DMA transfers. The various transfer types possible using the `DMAmove` function include 8-bit or 16-bit transfers, 32-bit transfer counts, source or destination addresses in A16, A24, or local RAM, incrementing or static source and destination addresses, and cycle steal or burst mode transfers. Any combination of these capabilities is possible. No steps have been taken to prevent two CIs from accessing the 68070 DMA channel at the same time. A CI-dependent mutual exclusion algorithm should be used to avoid conflicts.

Using two simple C `#define` statements, contained in the file `dips.h`, a CI has access to the Servant area and GPIB DIP switch values. If nonvolatile configuration overrides these values, a CI may use them in any device-dependent manner. Notice that if the GPIB-VXI/C is Resource Manager, the Servant area DIP is not used anyway.

A CI can use the 68881 access functions to configure itself to access the 68881 numeric coprocessor. These functions determine whether or not the runtime system includes the 68881 context in context switches between processes.

The 68881 context adds significantly to the context switching time. The runtime system maintains a list of processes that use the 68881 coprocessor and the associated 68881 context for those processes. It does not perform a 68881 context switch for processes that do not use the coprocessor, thereby significantly reducing the average context switch time.

The `Enable68881` function puts the calling process in the 68881 context switch list, while the `Disable68881` function removes it from the list. Typically, a CI that uses the 68881 will call the `Enable68881` function during its initialization sequence. The `Disable68881` function is needed only if your CI only uses the 68881 periodically, and you want to maximize system performance between 68881 calculations.

The `DCIParserQuery`, `DCIParserEnq`, `DCIParserDeq`, and `ClearParser` functions create a path for your CI to send commands to the local command set parser, and to receive responses as appropriate. The local command formats and responses are described in Chapter 3, *Local Command Set*, of the *GPIB-VXI/C User Manual*.

The `DCIParserQuery` function accepts one or more commands in the form of a pointer to a C string containing the command(s). It places a buffer containing the command on the parser's input queue. `DCIParserQuery` returns a pointer to a CommonBuf containing the parser's response. If `DCIParserQuery` returns the value 0, the command(s) had no response. If it is not equal to 0, it is pointing to a buffer containing the response. If the Endflag field of the buffer is 0, your CI must dequeue additional buffer(s) containing the remainder of the command response.

The CI can call `DCIParserDeq` to dequeue a response buffer from the parser output queue. `DCIParserDeq` returns a pointer to the next response buffer, or 0, which indicates an empty queue.

The `DCIParserEnq` function queues up a buffer containing one or more local commands on the parser input queue. If your CI executes multiple buffers as a unit, it should only set Endflag on the last buffer. When the parser finishes interpreting and executing the contents of the buffer, the CI receives an `S_EVENT_PARSER` pSOS message. The CI can call `DCIParserDeq` to get the parser response buffer(s) from the parser output queue.

The CI can call `ClearParser` to remove all pending buffers on its parser input and output queues.

The maximum number of buffers on each queue is 50.

The GPIB-VXI/C can handler many configurations of VXI and non-VXI interrupts. Interrupters can be *Release on Acknowledge (ROAK)*, which is the standard VXI type interrupter, or *Release on Register Access (RORA)* interrupters. ROAK interrupters are supported both by the GPIB-VXI/C Secondary Address interface and CIs. RORA interrupters are supported only by CIs. The variable `DCIheader.RORAhandlers` prescribes which handlers are RORA handlers for a particular CI. If an interrupt (*Signal In* message) is received on the Asynch exchange for a RORA handler, the function `ReenableHandler` must be called when the interrupt processing is complete.

The `ReserveHandler` function is used to assign a particular GPIB-VXI/C Interrupt handler to route VXI Status/ID values unconditionally to a particular CI. This function is needed only when a CI is handling VME interrupters whose Status/ID values do not follow VXI restrictions.

ClearParser

Purpose: Clear parser input and output queues.

Definition: `ClearParser(ParserID)`

`ParserID uint16` ParserID assigned to DCI, found in DCI header block

Action: Removes pending data from the CI's parser input and output queues.

Returns: None.

Example:

```
uint16 ParserID;
ParserID = DCIheader.ParserID;
.
.
.
ClearParser(ParserId);
```

DCIParserDeq

Purpose: Get local command set parser response from parser output queue.

Definition: `CommonBuf *DCIParserDeq(ParserID)`

`ParserID uint16` ParserID assigned to DCI, found in DCI header block

Action: Return a pointer to a buffer containing the parser response.

Returns:

-1:	Invalid `ParserID` specified.
0:	Ouput queue empty.
*CommonBuf :	Pointer to first buffer in response.

Example: Return pointer to parser response buffer to `rspns`.

```
uint16 ParID;
CommonBuf *rspns;
•
•
•
ParID= DCIheader.ParserId;

rspns= DCIParserDeq(ParID);
```

DCIParserEnq

Purpose: Pass a CommonBuf to the local command set parser input queue.

Definition: `DCIParserEnq(ParserID,Cbuf)`

`ParserID`	`uint16`	ParserID assigned to DCI, found in DCI header block
`*Cbuf`	`CommonBuf`	Pointer to buffer to execute

Action: Execute command(s) in `Cbuf` and return a result indicator.

Returns:

-1:	Invalid `ParserID` specified.
0:	Input queue full.
1:	Buffer successfully passed to parser.

Example: Parse buffer pointed to by `cmdbfr`, and return indicator of result to `result`.

```
uint16 ParID;
CommonBuf *cmdbfr,
•
•
•
ParID= DCIheader.ParserId;

result= DCIParserEnq(ParID,cmdbfr);
```

DCIParserQuery

Purpose: Cause the local command set parser to execute a command string.

Definition: `CommonBuf *DCIParserQuery(ParserID, command)`

`ParserID`	`uint16`	ParserID assigned to DCI, found in DCI header block
`*command`	`int8`	Pointer to the command string

Action: Execute command string and return pointer to a buffer containing the parser response.

Returns:

-1:	Invalid `ParserID` specified.
0:	No response.
*CommonBuf :	Pointer to first buffer in response.

Example: Assign GPIB Address 6 to this DCI.

```
uint16 ParID;
int8 Pcom[30];
CommonBuf *rspns;
.
.
.
MyLa= DCIheader.DCILogicalAddress;
ParID= DCIheader.ParserId;

GPIBAddress= 6;

sprintf (Pcom, "LaSaddr %d,%d", MyLa, GPIBAddress);

rspns= DCIParserQuery(ParID,Pcom);
```

Disable68881

Purpose: Disable 68881 context switching for the calling process.

Definition: `Disable68881( )`

Action: Removes calling process from the 68881 context switch list.

Returns: N/A

Example: `Disable68881( );`

DMAmove

Purpose: Perform a 68070 DMA memory-to-memory transfer.

Definition: `DMAmove(source, dest, count, mode)`

`source`	`uint32`	Local address to transfer from
`dest`	`uint32`	Local address to transfer to
`count`	`uint32`	Number of bytes to transfer
`mode`	`uint32`	Bit vector for mode of transfer

Bit 0: Transfer direction
- 0 = Source to destination
- 1 = Destination to source

Bit 1: Destination size
- 0 = 16 bit
- 1 = 8 bit

Bit 2: Operand size
- 0 = 16 bit
- 1 = 8 bit

Bit 3: DMA transfer mode
- 0 = Cycle steal
- 1 = Burst

Bit 4: Source address increment
- 0 = Increment source address by operand size
- 1 = Do not increment source address

Bit 5: Destination address increment
- 0 = Increment destination address by destination size
- 1 = Do not increment destination address

Action: Perform a DMA memory-to-memory transfer using `source` and `dest` addresses to transfer `count` bytes. The types of transfer, as defined by the bit vector `mode`, include: 8 or 16 bits, A16/A24/local source or destination addresses, incrementing or static source or destination addresses, and cycle steal or burst mode transfers (mode of `0` should be most common: 16-bit source and destination, cycle steal, incrementing source and destination addresses).

Returns:

0:	Transfer was successful.
2:	DMA timing error (should never occur).
9:	Bus error occurred at source address.
10:	Bus error occurred at destination address.

Example: DMA 512 kilobytes from A24 address 0x500000 to local RAM address 0x70000 using 16-bit transfers and cycle steal transfer mode.

```
ret = DMAmove(0x500000, 0x70000, 0x80000, 0);
```

Enable68881

Purpose: Enable 68881 context switching for the calling process.

Definition: `Enable68881( )`

Action: Puts calling process in the 68881 context switch list.

Returns:
0: 68881 is not present or failed.
1: 68881 is present.

Example:

```
Enable68881( );
```

NVconf

Purpose: Get the contents of a nonvolatile CI configuration variable.

Definition: `NVconf(offset,value)`

```
offset, *value      uint32
```

where `offset` can have a value from 0 to 31.

Action: Read the CI variable in EEPROM at location `offset`, and store in `value`. Return result indicator.

Returns:
0: Invalid offset.
1: CI variable successfully returned.

Example: Read longword value at offset 22.

```
uint32 ret, confinfo;
.
.
.
ret= NVconf(22, &confinfo);
```

ReenableHandler

Purpose: Reenable a GPIB-VXI/C VXI Interrupt handler after reception of an interrupt Status/ID value from a handler designated as a RORA interrupter.

Definition:

```
ReenableHandler(handler)

handler   int8
```

Action: Reenable GPIB-VXI/C VXI Interrupt handler after receipt of a RORA interrupt.

Returns:

-1:	Interrupt successfully reenabled
1:	`handler` value out of range 1 through 3

Example: Reenable handler 3 after receipt of a RORA interrupt.

```
result= Reenable(3);
```

ReserveHandler

Purpose: Allocate a GPIB-VXI/C VXI Interrupt handler for sole use by a CI. All interrupt Status/ID values received, regardless of their values, are sent to the CI.

Definition:

```
ReserveHandler(la, handler)

la        int8
handler   int8
```

Action: Set GPIB-VXI/C VXI Interrupt handler `handler` to route all VXI interrupt Status/ID values to CI at Logical Address `la`.

Returns:

1:	Setup successfully completed
-1:	`handler` value out of range 1 through 3
-2:	`la` is not a CI or the GPIB-VXI/C's logical address

Example: Set up handler 2 to route all VXI Interrupt Status/ID values to CI at Logical Address 10.

```
result = ReserveHandler(10, 2);
```

GPIB-VXI/C Trigger Functions

The following pages contain descriptions of the GPIB-VXI/C trigger functions.

- `AcknowledgeTrig`
- `DisableTrigSense`
- `EnableTrigSense`
- `GetTrigHandler`
- `MapTrigToTrig`
- `SetTrigHandler`
- `SrcTrig`
- `TrigAssertConfig`
- `TrigCntrConfig`
- `TrigExtConfig`
- `TrigTickConfig`
- `UnMapTrigToTrig`
- `WaitForTrig`

These functions can be used to directly manipulate the VXI TTL/ECL trigger lines and the front panel trigger connectors of the GPIB-VXI/C. The trigger functions are grouped into the following four categories.

- *Source trigger functions* act as a standard interface for asserting (sourcing) TTL and ECL triggers, as well as for detecting acknowledgements from accepting devices. These functions can source any of the VXI-defined trigger protocols from the GPIB-VXI/C. The source trigger commands are `SrcTrig`, `SetTrigHandler`, and `GetTrigHandler`.
- *Acceptor trigger functions* act as a standard interface for sensing (accepting) TTL and ECL triggers, as well as for sending acknowledgements back to the sourcing device. These functions can sense any of the VXI-defined trigger protocols on the GPIB-VXI/C. The acceptor trigger functions are `EnableTrigSense`, `DisableTrigSense`, `SetTrigHandler`, and `GetTrigHandler`.
- *Map trigger functions* are configuration functions for routing and signal conditioning. You can use the `MapTrigToTrig` and `UnMapTrigToTrig` functions to configure the GPIB-VXI/C hardware to route specified source trigger locations to destination trigger locations. You can use these functions as a crosspoint switch/signal conditioning configurator.

- *Trigger configuration functions* are configuration tools for configuring not only the general settings of the trigger inputs and outputs, but also the National Instruments Trigger Interface Chip (TIC) counter and tick timers. The trigger configuration functions are `TrigAssertConfig`, `TrigExtConfig`, `TrigCntrConfig`, and `TrigTickConfig`.

AcknowledgeTrig

Purpose: Acknowledge the specified TTL/ECL or external (GPIO) trigger.

Definition: `ret = AcknowledgeTrig (controller, line)`

`controller`	`int16`	Controller on which to acknowledge trigger interrupt (`controller` parameter should always be -1)
`line`	`uint16`	TTL, ECL, or External trigger line to acknowledge:

Value	Trigger line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
40 to 49	External source/destination (GPIO 0 to 9)

Action: The TTL/ECL trigger interrupt handler is called after an TTL/ECL trigger is sensed. If the sensed protocol requires an acknowledge (ASYNC or SEMI-SYNC protocols), the application should call `AcknowledgeTrig` after performing any device-dependent operations. If a trigger line is configured, using `TrigAssertConfig`, to participate in external (GPIO) SEMI-SYNC acknowledging, `AcknowledgeTrig` may be used to manually acknowledge a pending external SEMI-SYNC trigger.

Returns:

1:	Successful, no need to acknowledge
0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line`
-4:	`line` not supported
-12:	`line` not configured for external SEMI-SYNC

Example: Acknowledge a trigger interrupt for TTL line 4 on the local CPU (or the first extended controller).

```
int16    controller;
uint16   line;
int16    ret;

ret = AcknowledgeTrig (-1, 4);
```

DisableTrigSense

Purpose: Disable the sensing of the specified TTL/ECL trigger line, counter, or tick timer that was enabled by `EnableTrigSense`.

Definition: `ret = DisableTrigSense (controller, line)`

`controller`	`int16`	Controller on which to disable sensing (`controller` parameter should always be -1)
`line`	`uint16`	Trigger line to disable sensing:

Value	Trigger line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
50	TIC counter
60	TIC TICK timers

Action: Disables the sensing of `line` that was enabled by `EnableTrigSense`.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line`
-4:	`line` not supported
-7:	`line` not supported
-12:	`line` not configured for sensing

Example: Disable sensing of TTL line 4 on the local CPU (or the first extended controller).

```
int16    ret;
int16    controller;
uint16   line;

ret = DisableTrigSense (-1, 4);
```

EnableTrigSense

Purpose: Enable the sensing of the specified TTL/ECL trigger line or starts up the counter or tick timer for the specified protocol.

Definition: `ret = EnableTrigSense (controller, line, prot)`

`controller`	`int16`	Controller on which to enable sensing (`controller` parameter should always be -1)
`line`	`uint16`	Trigger line to enable sensing: Value — Trigger line 0 to 7 — TTL trigger lines 0 to 7 8 to 9 — ECL trigger lines 0 to 1 50 — TIC counter 60 — TIC TICK timers
`prot`	`uint16`	Protocol to use 2 = START 3 = STOP 4 = SYNC 5 = SEMI-SYNC 6 = ASYNC

Action: When `prot` is sensed on `line`, the corresponding trigger interrupt handler will be invoked. In order to start up the counter or tick timers, `TrigCntrConfig` and `TrigTickConfig` must be called first, respectively.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line` or `prot`
-4:	`line` not supported
-5:	`prot` not supported
-7:	`line` already in use
-12:	`line` not configured for use in sensing
-15:	previous operation incomplete

Example: Enable sensing of TTL line 4 on the local CPU (or the first extended controller) for SEMI-SYNC protocol.

```
int16    ret;
int16    controller;
uint16   line;
uint16   prot;

ret = EnableTrigSense (-1, 4, 5);
```

GetTrigHandler

Purpose: Returns the address of the current TTL/ECL trigger, Counter, or Tick timer interrupt handler for a specified trigger source.

Definition: `func = GetTrigHandler (line)`

`line`	`uint16`	TTL, ECL trigger line or counter/tick: Value — Trigger line 0 to 7 — TTL trigger lines 0 to 7 8 to 9 — ECL trigger lines 0 to 1 50 — TIC counter 60 — TIC TICK1 tick timer

Returns:

`func:`	`void`	Pointer to the current trigger interrupt handler for a specified trigger line NULL = Invalid `line` or no hardware support

Example: Get the address of the trigger interrupt handler for TTL trigger line 4.

```
void     (*func)();
uint16   line;

func = GetTrigHandler (4);
```

MapTrigToTrig

Purpose: Map the specified TTL, ECL, Star X, Star Y, external connection (GPIO), or miscellaneous signal line to another.

Definition: `ret = MapTrigToTrig (controller, srcTrig, destTrig, mode)`

`controller` `int16` Controller on which to map signal lines (`controller` parameter should always be -1)

`srcTrig` `uint16` Source line to map to destination

`destTrig` `uint16` Destination line to map from source

Value	Source or Destination line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
40 to 49	External source/destination (GPIO 0 to 9)
40	Front panel In (connector 1)
41	Front panel Out (connector 2)
42	ECL bypass from Front panel
43	Connection to EXTCLK input pin
44 to 49	Hardware-dependent GPIOs 4 to 9
50	TIC counter pulse output (TCNTR)
51	TIC counter finished output (GCNTR)
60	TIC TICK1 tick timer output
61	TIC TICK2 tick timer output

`mode` `uint16` Signal conditioning mode (0 = no conditioning)

Bit	Conditioning Effect
0	Synchronize with next CLK edge
1	Invert signal polarity
2	Pulse stretch to one CLK minimum
3	Use EXTCLK (not CLK10) for conditioning

All other values are reserved for future expansion.

Action: Map `srcTrig` to `destTrig`. The support actually present is completely hardware-dependent and is reflected in the error status and in hardware-specific documentation.

Returns:

0:	Successful
-1:	Unsupported function, no mapping capability
-2:	Invalid `controller`
-8:	Unsupported `srcTrig`
-9:	Unsupported `destTrig`
-10:	Unsupported `mode`
-11:	Already mapped, must use `UnMapTrigToTrig`

Example: Map TTL line 4 on the local CPU (or first extended controller) to go out of the front panel with no signal conditioning.

```
int16    controller;
uint16   srcTrig;
uint16   destTrig;
uint16   mode;
int16    ret;

ret = MapTrigToTrig (-1, 4, 41, 0);
```

SetTrigHandler

Purpose: Replaces the current TTL/ECL trigger, Counter, or Tick timer interrupt handler for a specified trigger source with the specified function, `func`.

Definition: `ret = SetTrigHandler (lines, func)`

`lines`	`uint16`	Bit vector of trigger lines (1 = set, 0 = do not set): Value — Trigger line(s) to set 0 to 7 — TTL trigger lines 0 to 7 8 to 9 — ECL trigger lines 0 to 1 14 — TIC counter 15 — TIC TICK timers
`func`	`void`	Pointer to the new trigger interrupt handler NULL = `DefaultTrigHandler`

Action: Replaces the current TTL/ECL trigger, Counter, or Tick timer interrupt handler for `lines` with `func`.

Returns:
0: Successful
-1: No hardware support

Example: Set a trigger interrupt handler for TTL trigger line 4.

```
void     func (int16, uint16, uint16);
uint16   lines;
int16    ret;

ret = SetTrigHandler (16, func);
```

SrcTrig

Purpose: Source the specified protocol on a specified TTL, ECL, or External trigger line.

Definition: `ret = SrcTrig (controller, line, prot, timeout)`

Remarks: Input parameters:

`controller` `int16` — Controller on which to source trigger line (`controller` parameter should always be -1)

`line` `uint16` — Trigger line to source:

Value	Trigger line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
40 to 49	External source/destination (GPIO 0 to 9) *
50	TIC counter **
60	TIC TICK timers **

`prot` `uint16` — Protocol to use

0	ON
1	OFF
2	START
3	STOP
4	SYNC
5	SEMI-SYNC
6	ASYNC
7	SEMI-SYNC and wait for Acknowledge
8	ASYNC and wait for Acknowledge
ffffh	Abort previous Acknowledge pending (5 and 6)

`timeout` `int32` — Timeout value in milliseconds

Action: Source `prot` on `line`.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line` or `prot`
-4:	`line` not supported
-5:	`prot` not supported
-6:	Timeout occurred waiting for ACK
-7:	`line` already in use
-12:	`line` not configured for use in sourcing
-15:	previous operation incomplete
-16:	previous acknowledge still pending

* Supports ON, OFF, START, STOP, and SYNC protocols only
** Supports SYNC and SEMI-SYNC protocols only

Example: Source TTL line 4 on the local CPU (or the first extended controller) for SEMI-SYNC protocol.

```
int16    ret;
int16    controller;
uint16   line;
uint16   prot;
int32    timeout;

ret = SrcTrig (-1, 4, 5, 0L);
```

TrigAssertConfig

Purpose: Configure a specified TTL/ECL trigger line assertion method.

Definition: `ret = TrigAssertConfig (controller, line, mode)`

`controller int16` Controller on which to configure assertion mode (`controller` parameter should always be -1)

`line uint16` Trigger line to configure:

Value	Trigger line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
ffffh	General assertion configuration (all lines)

`mode uint16` Configuration mode

Bit	Specific Line Configuration Modes
0	1 = Synchronize falling edge of CLK10 0 = Synchronize rising edge of CLK10
Bit	**General Configuration Modes**
1	1 = Pass trigger through asynchronously 0 = Synchronize with next CLK10 edge
2	1 = participate in SEMI-SYNC with External trigger acknowledge protocol 0 = Do not participate

All other values are reserved for future expansion.

Action: Configure `line` with `mode`. TTL/ECL triggers may be (re)-synchronized to CLK10 on a per-line basis. It can be globally selected for all TTL/ECL trigger lines whether to synchronize to the rising or falling edge of CLK10. In addition, `line` may be specified to partake in SEMI-SYNC accepting with external acknowledge.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line` or `prot`
-4:	`line` not supported
-10:	Invalid configuration mode

Example 1: Configure all TTL/ECL trigger lines generally to synchronize to the falling edge of CLK10 (as opposed to the rising edge).

```
int16    ret;
int16    controller;
int16    line;
uint16   mode;

ret = TrigAssertConfig (-1, -1, 1);
```

Example 2: Configure TTL trigger line 4 to synchronize to CLK10 for any assertion method and do not participate in SEMI-SYNC.

```
int16    ret;
int16    controller;
int16    line;
uint16   mode;

ret = TrigAssertConfig (-1, 4, 0);
```

TrigCntrConfig

Purpose: Configure the TIC chip internal 16-bit counter.

Definition: `ret = TrigCntrConfig (controller, mode, source, count)`

`controller` `int16` Controller on which to configure TIC counter (`controller` parameter should always be -1)

`mode` `uint16` Configuration mode

Value	Configuration Mode
0	Initialize the counter
2	Reload the counter leaving enabled
3	Disable/Abort any count in progress

`source` `uint16` Trigger line to configure as input to counter:

Value	Trigger line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
70	CLK10
71	EXTCLK connection

`count` `uint16` Number of input pulses to count before terminating

Action: Configure the TIC chip internal 16-bit counter. Call `SrcTrig` or `EnableTrigSense` to actually start the counter. The input may be any trigger line, CLK10, or the EXTCLK connection. The counter has two outputs: TCNTR (one 100-nsec pulse per input edge) and GCNTR (unasserted until count goes from 1 to 0, then asserted until counter reloaded or reset). TCNTR may be mapped using `MapTrigToTrig` to any number of the TTL or ECL trigger lines. GCNTR may be mapped using `MapTrigToTrig` to any number of the External (GPIO) lines.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line` or `prot`
-10:	Invalid configuration mode
-15:	Previous count incomplete

Example: Configure the counter count 25 assertions on TTL trigger line 5 (the `prot` parameter when calling `EnableTrigSense` will determine whether the counter accepts SYNC or SEMI-SYNC assertions).

```
int16    ret;
int16    controller;
uint16   mode;
int16    source;
uint16   count;

ret = TrigCntrConfig (-1, 0, 5, 25);
```

TrigExtConfig

Action: Configure the external trigger (GPIO) lines.

Definition: `ret = TrigExtConfig (controller, extline, mode)`

`controller int16` Controller on which to configure GPIO lines (`controller` parameter should always be -1)

`extline uint16` Trigger line to configure:

Value	Trigger line
40 to 49	External source/destination (GPIO 0 to 9)
40	Front panel In (connector 1)
41	Front panel Out (connector 2)
42	ECL bypass from Front panel
43	EXTCLK
44 to 49	Hardware-dependent GPIOs 4 to 9

`mode uint16` Configuration mode

Bit	Configuration Modes
0	1 = Feedback any line mapped as input into the crosspoint switch 0 = Drive input to external (GPIO) pin
1	1 = Assert input (regardless of feedback) 0 = Leave input unconfigured
2	1 = If assertion selected, assert low 0 = If assertion selected, assert high
3	1 = Invert external input (not feedback) 0 = Pass external input unchanged

All other values are reserved for future expansion.

Action: Configures the external trigger (GPIO) lines. The external trigger lines may be feedback for use in the crosspoint switch output. The external trigger lines may be asserted high or low, or left unconfigured (tristated) for use as a crosspoint switch input. If not feedback, the external input may be inverted before mapped to a trigger line.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line` or `prot`
-10:	Invalid configuration mode

Example 1: Configure external line 41 (Front Panel Out) to not be feedback and left tristated for use as a mapped output via `MapTrigToTrig`.

```
int16    ret;
int16    controller;
int16    extline;
uint16   mode;

ret = TrigExtConfig (-1, 41, 0);
```

Example 2: Configure external line 40 (Front Panel In) to not be feedback and left tristated for use as a mapped input via `MapTrigToTrig`. Invert the Front Panel In signal.

```
int16    ret;
int16    controller;
int16    extline;
uint16   mode;

ret = TrigExtConfig (-1, 40, 8);
```

Example 3: Configure external line 48 (GPIO 8) to be feedback for use as a crosspoint switch input and output via `MapTrigToTrig.`

```
int16    ret;
int16    controller;
int16    extline;
uint16   mode;

ret = TrigExtConfig (-1, 48, 1);
```

TrigTickConfig

Purpose: Configure the TIC chip internal dual 5-bit tick timers.

Definition: `ret = TrigTickConfig (controller, mode, source, tcount1, tcount2)`

`controller` `int16` Controller on which to configure TIC tick timers (`controller` parameter should always be -1)

`mode` `uint16` Configuration mode

Value	Configuration Mode
0	Initialize the tick timers (rollover mode)
1	Initialize the tick timers (non-rollover mode)
2	Reload the tick timers leaving enabled
3	Disable/Abort any count in progress

`source` `uint16` Trigger line to configure as input to counter:

Value	Trigger line
40 to 49	External source/destination (GPIO 0 to 9)
70	CLK10
71	EXTCLK connection

t`count1` `uint16` Number of input pulses (as a power of two) to count before asserting TICK1 output (and terminating the tick timer if configured for non-rollover mode)

t`count2` `uint16` Number of input pulses (as a power of two) to count before asserting TICK2 output

Action: Configure the TIC chip internal dual 5-bit tick timers. Call `SrcTrig` or `EnableTrigSense` to actually start the tick timers. `SrcTrig` inhibits the TICK1 output from generating tick timer interrupts. `EnableTrigSense` enables the TICK1 output to generate tick timer interrupts. The input may be any external (GPIO) line, CLK10, or the EXTCLK connection. The two tick timer outputs TICK1 and TICK2 may be mapped to any number of TTL/ECL trigger lines. In addition, the TICK2 output may be mapped to any number of external (GPIO) lines.

Returns:

3:	Successful disable of the tick timers
2:	Successful reload of the tick timers
1:	Successful initialization of non-rollover mode
0:	Successful initialization of rollover mode
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line` or `prot`
-10:	Invalid configuration mode
-15:	Previous tick configured and enabled

Example 1: Configure the tick timers to interrupt every 6.55 milliseconds by dividing down CLK10 as an input. Call `EnableTrigSense` to start the tick timers and enable interrupts.

```
int16    ret;
int16    controller;
uint16   mode;
int16    source;
uint16   tcount1, tcount2;

ret = TrigTickConfig (-1, 0, 70, 16, 0);
```

Example 2: Configure the tick timers to output a continuous 9.765-kHz square wave on TICK1 output and a 1.25-MHz clock on TICK2 output by dividing down CLK10 as an input. Call `SrcTrig` to start the tick timers.

```
int16    ret;
int16    controller;
uint16   mode;
int16    source;
uint16   tcount1, tcount2;

ret = TrigTickConfig (-1, 0, 70, 10, 3);
```

UnMapTrigToTrig

Purpose: Unmap a specified TTL, ECL, Star X, Star Y, external connection (GPIO), or miscellaneous signal line that was mapped to another line using the `MapTrigToTrig` function.

Definition: `ret = UnMapTrigToTrig (controller, srcTrig, destTrig)`

`controller`	`int16`	Controller on which to unmap signal lines (`controller` parameter should always be -1)
`srcTrig`	`uint16`	Source line to unmap from destination
`destTrig`	`uint16`	Destination line mapped from source

Value	Source or Destination
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
40 to 49	External source/destination (GPIO 0 to 9)
40	Front panel In (connector 1)
41	Front panel Out (connector 2)
42	ECL bypass from Front panel
43	Connection to EXTCLK input pin
44 to 49	Hardware-dependent GPIOs 4 to 9
50	TIC counter pulse output (TCNTR)
51	TIC counter finished output (GCNTR)
60	TIC TICK1 tick timer output
61	TIC TICK2 tick timer output

Action: Unmap `srcTrig` from `destTrig`.

Returns:

0:	Successful
-1:	Unsupported function, no mapping capability
-2:	Invalid `controller`
-12:	Not previously mapped

Example: Unmap route of TTL line 4 on the local CPU (or first extended controller) to go out of the front panel as mapped by `MapTrigToTrig.`

```
int16    controller;
uint16   srcTrig;
uint16   destTrig;
int16    ret;

ret = UnMapTrigToTrig (-1, 4, 49);
```

WaitForTrig

Purpose: Wait for the specified trigger line to be sensed for the specified time.

Definition: `ret = WaitForTrig (controller, line, timeout)`

`controller`	`int16`	Controller on which to wait for trigger (`controller` parameter should always be -1)
`line`	`uint16`	Trigger line to wait on:

Value	Trigger line
0 to 7	TTL trigger lines 0 to 7
8 to 9	ECL trigger lines 0 to 1
50	TIC counter
60	TIC TICK timers

`timeout`	`int32`	Timeout value in milliseconds

Action: Wait for `line` to be sensed with a specified `timeout`. `EnableTrigSense` must be called to sensitize the hardware to the particular trigger protocol to be sensed.

Returns:

0:	Successful
-1:	Unsupportable function (no hardware support)
-2:	Invalid `controller`
-3:	Invalid `line`
-4:	`line` not supported
-6:	Timeout occurred

Example: Wait for TTL line 4 on the local CPU (or the first extended controller) to be encountered.

```
int16    ret;
int16    controller;
uint16   line;
int32    timeout;

ret = WaitForTrig (-1, 4, 10000L);
```

Chapter 7
Word Serial Drivers

This chapter contains information about the Word Serial device drivers, including descriptions of the driver calls, their purpose, and examples of their use.

The GPIB-VXI/C runtime system implements Word Serial Protocol with two pSOS device drivers. The DEV_WS driver handles communication with Servants of the GPIB-VXI/C and Code Instrument (CI), while the DEV_WSsl driver handles the Servant (slave) side communications with the Commander of the GPIB-VXI/C or CI. A CI accesses these drivers through the standard pREP/C `d_read`, `d_write`, and `d_ctrl` device driver calls.

Using the Word Serial driver calls, a CI can perform the following functions:

- Send a data buffer to its Commander or one of its Servants.
- Receive a data buffer from its Commander or one of its Servants.
- Send a Word Serial command to one of its Servants and receive a response.
- Send a response to the CI's Commander.
- Abort all pending Word Serial transfers when it receives a Word Serial *Clear* command from its Commander or an *Unrecognized Command* error from one of its Servants.

All Commanders (including CIs) must respect the Read Ready and Write Ready status of their Servants when sending the Servant any Word Serial command. In addition to the Read Ready/Write Ready status, the Commander must also respect its Servant's Data In Ready (DIR)/Data Out Ready (DOR) status when it sends the *Byte Available*, *Trigger*, or *Byte Request* commands to a Servant.

The `d_read` and `d_write` calls give a CI the capability to transfer a data buffer between itself and its Commander or Servants using the *Byte Available*/*Byte Request* protocol. A CI can send other Word Serial commands to its Servants and responses to its Commander using the `d_ctrl` call.

The runtime system includes National Instruments-supplied C function calls for performing the runtime system-related tasks associated with receiving the Word Serial *Abort Normal Operation*, *End Normal Operation*, *Release Device*, or *Clear* commands. See *Word Serial Protocol Functions* in Chapter 6, *C Function Calls*, for more information.

Buffer-Based Communication

The runtime system recognizes three structure types associated with Word Serial Protocol communication: *WScom*, *WSresp*, and *WSbuf*. *WScom* is used for sending a command, *WSresp* for sending a response, and *WSbuf* for sending or receiving a data buffer. These Word Serial driver structures are defined in the file `InclWS.h`.

The file `CommonBuf.h` defines a general-purpose structure called *CommonBuf* for buffer-based communication. CommonBufs are used for all GPIB-VXI/C I/O communication, including sending and receiving Word Serial data buffers.

GPIB Buffer Transfers

A GPIB controller can communicate only with CIs that are Servants of the GPIB-VXI/C, because a Message-Based Commander such as the GPIB-VXI/C can initiate communication only with its Servants. This restriction explains why a Message-Based device that is not a Servant of the GPIB-VXI/C cannot have a GPIB address.

If a CI is a Servant of the GPIB-VXI/C, the `d_read` and `d_write` calls it makes to the Word Serial slave driver will cause the GPIB-VXI/C to route data buffers between the CI and the GPIB port.

Handling a Clear Command

When a CI receives a Word Serial *Clear* command, it must abort any pending operations, including Word Serial transfers between itself and its Commander and Servants. It can also perform any or all of the following functions:

- Abort any other operations with its Commander and Servants.
- Clear its input and output queues, clear its SRQ status.
- Place itself in an idle parser state.
- Propagate the *Clear* command to its Message-Based Servants.

The National Instruments-supplied C function calls and Word Serial device driver calls perform all of these actions. However, aborting operation with a non-Message-Based Servant is a device-dependent action that the designer must handle.

The runtime system will terminate Word Serial transfers for a CI if it makes the following calls from its Asynch process:

- `d_ctrl(DEV_WS, la, WS_CTRL_Clear, WScom)`
  Abort Word Serial buffer transfers with Servant at Logical Address `la`.

- `DCIwsClear(MyLa)`
  Abort Word Serial buffer transfers with the CI's Commander.

- `HWwsClear(WSb_RET_Clear)`
  Abort physical Word Serial path to the GPIB-VXI/C's Commander (only if it has control of physical Word Serial registers).

A CI that has taken control of the physical Word Serial registers to become the Servant of the GPIB-VXI/C's Commander must call `HWwsClear` to clear the physical path. A CI's Word Serial path to the GPIB port can remain open after it takes over the physical path to the GPIB-VXI/C's Commander. Because the actions for the physical path and the CI path may be

independent of each other, it is up to you to decide what should happen depending upon the source of the *Clear* command.

The function `ClearServants` in `asynch.c` is an example of how a CI can propagate a *Clear* command to its Message-Based Servants.

Handling an Unrecognized Command Event (VXIbus System Specification Revision 1.2)

If a Message-Based Servant compatible with Revision 1.2 of the VXIbus specification receives a Word Serial command from a CI that it does not understand, the Servant can send the GPIB-VXI/C an *Unrecognized Command* event signal/interrupt. The GPIB-VXI/C will route it to the CI via the Asynch exchange, and the CI's Asynch process must then call the *Unrecognized Command Abort* routine in order for the call to return with the *Unrecognized Command* status. The runtime system will terminate the command in this manner for the CI if the CI's asynch process makes the following call:

```
d_ctrl(DEV_WS, la, WS_CTRL_UnRecCom, 0)
```

The file `asynch.c` already has this call in place. If all Servants of the CI are compatible with VXIbus Specification Revision 1.3 or later, handling the *Unrecognized Command* event is not necessary.

Word Serial Communication Examples

Sending a Command

An example of sending a Word Serial command to a Servant can be found in the function `SendWScommandExamples` in the file `worker.c`. In this example, the CI performs the following steps:

1. Declares a structure of type WScom (WSc in the example).

2. Sets the Command field of this structure to the command value to be sent (`WS_READ_STB`). The C language *#define* values for most Word Serial commands can be found in the file `InclWS.h`.

3. Sets the `WScom.mode` field. Because the *Read STB* command requires a response, the CI sets the `Wsc.mode` to `WS_MODE_RESPONSE`. If no response were required, the CI would set the mode field to 0.

4. Makes a `d_ctrl WS_CTRL_Command` call to the Word Serial Commander driver DEV_WS, with the logical address of the Servant and a pointer to WSc.

5. Inspects the error code returned in `WSc.Retval` following completion of the call, and the response to the command, if applicable, in `WSc.Response`.

Sending a Response

When a CI sends a response to its Commander, it performs the following steps:

1. Declares a structure for Word Serial response of type WSresp.

2. Sets the Response field of this structure to the response value.

3. Makes a `d_ctrl WSsl_CTRL_Command` call to the Word Serial slave driver, with the CI's logical address, and a pointer to the WSresp structure.

The source code might look like this:

```
WSresp WSr;                    /* Declare the WSresp structure. */

MyLa= DCIheader.DCILogicalAddress;     /* Get caller's logical
                                       address from header. */
.
.
.
WSr.Response = 0x1234;         /* Set the response value. */
d_ctrl(DEV_WSsl, MyLa, WSsl_CTRL_Response, &WSr);     /* Call
                                       the response driver. */
```

The CI should inspect `WSr.Retval` to confirm that the `d_ctrl` call was successful.

Receiving a Buffer

An example of receiving a Word Serial data buffer from the CI's Commander can be found in the `MainLoop` function in the file `worker.c`. After waiting for a `WS_DATA_IN` signal, the CI can read the buffer from its input queue. The CI performs the following steps to receive a buffer:

1. Declares a pointer to a CommonBuf, and a structure of type WSbuf. In this example, the CI declares the pointer WSBuf and the WSbuf structure WSb at the top of MainLoop.

2. Sets the mode field of the WSbuf structure. In this example, it sets the `WSb.mode` to 0 to permit sending of DIR and DOR response signals.

3. Sets the size of the read buffer by setting the `WSb.MaxReadCount` field to 512 (= 0x200). If the CI set the `WSb.Wait` field to 0, the driver would wait forever, if necessary, for the buffer to be allocated. This field is only applicable when calling the physical Word Serial driver. When a CI is hooked up to the GPIB, the GPIB Address Model automatically allocates buffers when read operations take place. The pointer to the buffer is then simply handed to the CI. However, in the case of the physical Word Serial path, you are interfacing directly with the physical Word Serial hardware, and the driver must therefore allocate its own buffer.

4. Makes a `d_read` call to the Word Serial slave driver `DEV_WSsl`, using the caller's logical address and pointers to the WSbuf structure (`&WSb`) and to the buffer pointer (`&WSBuf`). The device driver returns the pointer to the input buffer in WSBuf.

Sending a Buffer

The CI performs the following steps to send a buffer:

1. Declares a structure of type WSbuf and a pointer to a CommonBuf.

2. Creates the output buffer. The CI can receive the buffer from another source or you can create it yourself using `CreateBuf` or `DuplicateCommonBuf`. If creating the buffer yourself, fill the buffer with the data, set the `byteCount` field to the number of bytes that are to be sent (are valid in the CommonBuf data section), and set the `EndFlag` field to 1 to signify that this is the last buffer in the Word Serial transfer (equivalent to EOI on GPIB or END in Word Serial).

3. Sets the value of `WSb.mode` field and calls the DEV_WS `d_ctrl` function using the logical address of the Servant and pointers to WSb and Outbuf. With the `WSb.mode` set to 0, the CI can abort the transfer if the device is not initially DIR, and free the buffer after the transfer is complete.

Word Serial Commander Device Driver Calls

The Word Serial Commander device driver calls are described on the following pages.

- `d_ctrl`
- `d_read`
- `d_write`

A CI can access the Word Serial Commander device driver with the `d_read`, `d_write`, and `d_ctrl` pREP/C function calls. The `d_write` call is used to send a data buffer to a Message-Based Servant, while the `d_read` call is used to read a data buffer returned by a Message-Based Servant.

The `d_ctrl` call performs special Word Serial operations, such as sending a command to a Servant or aborting a previously initiated `d_write` or `d_read` operation.

d_ctrl(DEV_WS, . .)

Purpose: Perform special Word Serial operations, including sending a Word Serial command to a Servant, and aborting a pending Word Serial transfer upon reception of a *Clear* command or *Unrecognized Command* error.

The Worker process makes all `d_ctrl` calls that send commands to a Servant. Do not use the Asynch process to send commands to Servants, because no method would be available to process an *Unrecognized Command* error or a Word Serial *Clear* command during the command transfer.

The Asynch process makes all `d_ctrl` calls to abort a CI's Word Serial operations because reception of Word Serial *Clear* commands and *Unrecognized Command* errors are asynchronous events. The `asynch.c` file contains examples of handling an *Unrecognized Command* error and a *Clear* command.

Definition: `d_ctrl(DEV_WS, la, comtype, arg)`

`int8 la` — Logical address of Servant to send command to

`uint32 comtype` — Value for operation to execute:
- `WS_CTRL_Command` = Send Word Serial command
- `WS_CTRL_Trigger` = Send Word Serial *Trigger*
- `WS_CTRL_Clear` = Abort Word Serial communication with Servant at Logical Address `la`
- `WS_CTRL_UnRecCom` = Abort command transfer to Servant at Logical Address `la`.

`arg` — For sending commands (`comtype` = 0) only; a pointer to a structure of type WScom containing the mode, the return status, the command to be sent, and the response, if applicable.

Returns: For sending commands, the return status in `WSc->Retval`, and the response, if applicable, in `WSc->Response`, as defined in `IncludeWS.h`.

Example: Send Word Serial *Clear* command to a Servant at Logical Address 5.

```
WScom WSc;
int8 ServantsLa;

ServantsLa = 5;
.
.
.
WSc.Command = (uint16)WS_CLEAR;     /* Word Serial
                              command is device clear. */
WSc.mode = 0;   /* Command does not require response. */

d_ctrl(DEV_WS, ServantsLa, WS_CTRL_Command, &WSc);
```

d_read(DEV_WS, . .)

Purpose: Read a Word Serial data buffer from a Message-Based Servant. Typically called from the worker process.

Definition: `d_read(DEV_WS, la, wBuf, cBuf, unused)`

`int8 la`	Logical address of device to read from
`WSbuf *wBuf`	Structure that holds mode, max read count, and the return status
`CommonBuf **cBuf`	Address of a pointer to return the read buffer in

Returns: Return status in `wBuf->Retval`, as defined in `IncludeWS.h`, and a pointer to a pointer to the read buffer in `cBuf`.

Example: Read a buffer from a Servant at Logical Address 4.

```
WSbuf WSb;
CommonBuf *WSbuf;
int8 ServantsLa;
.
.
.
ServantsLa = 4;
.
.
.
WSb.mode= 0;                /* Abort on not DOR. */
WSb.MaxReadCount= 0x200;    /* Read maximum of 512
                            bytes. */
WSb.Wait= 0L;               /* Wait forever for buffer
                            allocation. */

d_read(DEV_WS, ServantsLa, &WSb, &WSbuf, 0);
```

d_write(DEV_WS, . .)

Purpose: Write a Word Serial buffer to a Message-Based Servant. Typically called from the Worker process.

Definition: `d_write(DEV_WS, la, wBuf, cBuf, unused)`

`int8 la`	Logical address of device to write to
`WSbuf *wBuf`	Pointer to a structure that holds mode and the return status
`CommonBuf **cBuf`	Address of a pointer to the write buffer

Returns: Return status in `wBuf->Retval`, as defined in `IncludeWS.h`.

Example: Send a data buffer to a Servant at Logical Address 6.

```
WSbuf WSb;
CommonBuf *WSBuf;
int8 ServantsLa;

ServantsLa = 6;
.
.
.
WSBuf= CreateBuf(2, 0);   /* Create and fill buffer to
                             send. */
strcpy (WSBuf->data,"This is a buffer to send");
WSBuf->byteCount= 24;     /* Set send byte count and
                             end flag fields. */
WSBuf->EndFlag= 1;

WSb.mode= 0;              /* Free buffer when done,
                             abort on not DIR. */
WSb.Wait= 0L;             /* Wait forever for buffer
                             allocation. */
d_write(DEV_WS,ServantsLa ,&WSb, &WSBuf, 0);
```

Word Serial Slave (Servant) Driver Calls

Descriptions of the Word Serial slave (Servant) driver calls are given on the following pages.

- `d_ctrl`
- `d_read`
- `d_write`

A CI can access the Word Serial slave device driver with the `d_read`, `d_write`, and `d_ctrl` pREP/C function calls. The `d_write` call is used to return a data buffer to a Message-Based Commander, while the `d_read` call is used to read a data buffer from a Message-Based Commander.

The `d_ctrl` call performs special Word Serial operations such as sending a command response to the Commander of the CI.

The slave driver calls use the caller's logical address to route data buffers and signals between the CI and its Commander. The runtime system uses the caller's logical address and its knowledge of the Commander/Servant hierarchy to determine who the caller's Commander is. A CI does not need to know who its Commander is in order to communicate with it. It only needs to know its own logical address, which is readily available in its own header.

d_ctrl(DEV_WSsl, . .)

Purpose: Send a response to a Word Serial command to the caller's Commander. Typically called from the Worker process.

Definition: `d_ctrl(DEV_WSsl, cla, comtype, WSr)`

`int8 cla`	Caller's logical address
`uint32 comtype`	Value for operation to execute: `WSsl_CTRL_Response` = Send response to caller's Commander
`WSresp *WSr`	A pointer to a structure containing the mode, the return status and the response to be sent

Returns: The return status in `WSr->Retval`, as defined in `IncludeWS.h`.

Example: Send the response 1234h to the caller's Commander.

```
WSresp WSr;
int8 MyLa;

MyLa = DCIheader.DCILogicalAddress; /* Get caller's
                              logical address from header. */
.
.
.
WSr.Response = 0x1234;           /* Response is 1234h. */

d_ctrl(DEV_WSsl, MyLa, WS_CTRL_Response, &WSr);
```

d_read(DEV_WSsl, . .)

Purpose: Read a Word Serial buffer sent to the CI by its Commander. Typically called from the Worker process.

Definition: `d_read(DEV_WSsl, cla, wBuf, cBuf, unused)`

`int8 cla`	Caller's logical address
`WSbuf *wBuf`	Structure that holds mode, max read count, and the return status
`CommonBuf **cBuf`	Address of a pointer to place the read data buffer

Setting the `Wbuf->MaxReadCount` and `Wbuf->Wait` fields is only significant when the CI is communicating with its Commander through the physical Word Serial registers; otherwise, the data buffer is allocated by the CI's Commander.

Returns: Return status in `wBuf->Retval`, as defined in `IncludeWS.h`, and a pointer to a pointer to the read buffer in `cBuf`.

Example: Read a Word Serial buffer from the CI's Commander.

```
WSbuf WSb;
CommonBuf *WSbuf;
int8 MyLa;

MyLa= DCIheader.DCILogicalAddress;  /* Get caller's
                      logical address from header. */
.
.
.
WSb.mode= 0;              /* Abort on not DOR. */
WSb.MaxReadCount= 0x200;  /* Read maximum of 512 bytes
*/
WSb.Wait= 0L;             /* Wait forever for buffer
                          allocation. */

d_read(DEV_WSsl, MyLa, &WSb, &WSbuf, 0);
```

d_write(DEV_WSsl, . .)

Purpose: Return a Word Serial data buffer to the caller's Commander. Typically called from the Worker process.

Definition: `d_write(DEV_WSsl, cla, wBuf, cBuf, unused)`

`int8 cla`	Caller's logical address
`WSbuf *wBuf`	Pointer to a structure that holds mode and the return status
`CommonBuf **cBuf`	Address of a pointer to the write buffer

Returns: Return status in `wBuf->Retval`, as defined in `IncludeWS.h`.

Example: Send a Word Serial buffer to the CI's Commander.

```
WSbuf WSb;
CommonBuf *WSBuf;
int8 MyLa;

MyLa= DCIheader.DCILogicalAddress;  /* Get caller's
                      logical address from header. */
.
.
.
WSBuf= CreateBuf(2,0);    /* Create and fill buffer to
                          send. */
strcpy (WSBuf->data,"This is a buffer to send");
WSBuf->byteCount= 24;     /* Set send byte count and end
                          flag fields. */
WSBuf->EndFlag= 1;

WSb.mode= 0;              /* Free buffer when done. */
WSb.Wait= 0L;             /* Wait forever for buffer
                          allocation. */
d_write(DEV_WSsl, MyLa, &WSb, &WSBuf, 0);
```

Chapter 8
VXI pROBE

This chapter describes the pROBE debugging tool for the VXI environment, including its functions, menus, and VXI commands.

pROBE is a system debugging and analysis tool designed for the pSOS environment by Software Components Group, Inc. You can use this tool to observe, control, and stimulate system execution. National Instruments has enhanced pROBE to include many features and commands that are useful in the GPIB-VXI/C and VXI environments, including:

- Accessing the GPIB port
- Controlling copying of the pROBE standard output to the GPIB-VXI/C's memory
- Controlling scripts and macros
- Accessing onboard and offboard hardware
- Getting information about VXI devices
- Exiting to the other GPIB-VXI/C operating modes.

While pROBE is active, the local command set is inactive and the GPIB-VXI/C's Message-Based capabilities are disabled. The `TEST` LED on the front panel indicates that the GPIB-VXI/C is in pROBE. The SYSFAIL line is released, so the GPIB-VXI/C can reside transparently within the VXIbus system.

Entering and Exiting VXI pROBE

VXI pROBE can be entered in two ways:

- At system startup, set the startup mode switches as described in the *GPIB-VXI/C Startup Mode Configuration* section in Chapter 2, *Configuration and Startup*, of the *GPIB-VXI/C User Manual*. The GPIB-VXI/C enters the VXI pROBE boot menu, from which pROBE can be initialized.

- From the local command set console, type `probe` <CR> or just `p` <CR>.

You can reinitialize pROBE with the `IN` command. The `ME` command causes the GPIB-VXI/C to enter the VXI pROBE boot menu.

The `DIAG` command causes the GPIB-VXI/C to temporarily exit pROBE and enter the diagnostics menu. The `CONF` command causes the GPIB-VXI/C to temporarily exit pROBE and enter the nonvolatile memory configuration menu. When you exit either of these utilities, the GPIB-VXI/C reenters pROBE.

The `BOOT` (or `BO`) command causes the GPIB-VXI/C to reboot pROBE silently, reboot pSOS, and enter the system mode. The `RBO` command causes the GPIB-VXI/C runtime system to cold boot into its former state.

The `ECI` command causes the GPIB-VXI/C to enter the EPROMed Code Instrument (ECI) Image Creator menu. From this menu, multiple ECI images can be combined into a single image. You can also use this menu to convert binary images to S-record images and vice versa, or to reformat S-record objects into a more compact and orderly format.

VXI pROBE Menu

If the board is configured to enter pROBE at system startup, the GPIB-VXI/C enters the VXI pROBE menu immediately, without initializing pROBE or even performing a self-test. You can use the VXI pROBE menu to download, view, modify and execute code outside the pSOS/pROBE environment, and to recover gracefully from system crashes. In this way, the GPIB-VXI/C can be used not only as a VXI development and test tool but also as a simple 68000-based CPU module.

Enter a number from 1 to 7 after the `Choice (1-7):` prompt to select a menu command. The commands are:

1. **Do a GPIB download or upload**

   This is the same as the VXI pROBE command `GP`. Prompts will request the download/upload direction and the GPIB-VXI/C memory address.

   Any host GPIB software that can address the GPIB-VXI/C to listen or talk, and that can send and receive data, can be used to download and upload the GPIB-VXI/C memory. The IBIC program available from National Instruments is an example of the type of software that can be used for this purpose. The GPIB addressing for pROBE-based download/upload is primary address only. The primary address is set by the GPIB primary address dip switch or the configuration EEPROM. The data termination method is EOI in both directions.

2. **Do a hex dump of memory**

   You can use this feature to view the GPIB-VXI/C's onboard memory without having any system software booted (pROBE or pSOS). A prompt will ask you for the start address of a 256-byte memory block to be dumped to the console. You can dump subsequent blocks by responding with a `y` to the `Dump more (y or n)?` prompt.

3. **Patch memory**

   The patch memory function is identical to the VXI pROBE `P` command. Refer to the `P` command description for details of this command's operation.

4. **Jump to execution address**

   A prompt requests the address of the first executable instruction of a code.

5. **Initialize pROBE**

   This selection cold-starts pROBE. Any existing pSOS-related information (process status, profiling information, and so on) is reinitialized.

6. Trap to pROBE

This selection reenters pROBE without destroying pSOS information. For example, you could use this feature to look at the state of all pSOS processes at the time a software or hardware bug caused the system to lock up. Or, if the reset button on the front panel is pressed and the startup DIP switches are configured for debug mode, you can use this feature to view the state of the GPIB-VXI/C before the reset. All pSOS information will reflect the state of the GPIB-VXI/C when the reset button was pressed.

7. Return to pROBE

This selection returns to pROBE from the VXI pROBE command `ME`.

VXI pROBE Commands

The following pROBE command descriptions describe National Instruments enhancements to the pROBE command set. Refer to the Software Components Group *pROBE - 68K System Debug/Analyzer Manual* for descriptions of pROBE baseline commands.

You can access pROBE commands from the RS-232 port and from the GPIB port. Both sources can be active at any time. The pROBE command line interpreter prompts you to enter a command on the RS-232 port with the following prompt:

```
pROBE>
```

Notation and Syntax

The command interpreter is case insensitive. Command parameters delimited by brackets (`[]`) are optional, while parameters delimited by braces (`{}`) are required. Parameter value choices are separated by a vertical bar (`|`). The command interpreter will prompt you to input missing parameters. The command termination character is <CR> `(ASCII 0Dh)`.

Help Screen Command

HELP

Purpose: List format and description of National Instruments pROBE commands.

Command Format: HELP or ?

Action: Displays the help screen:

```
       VXI pROBE Additional Commands   (c) 1990 National Instruments

Command/Parameters                        Description
-------------------------------------------------------------------------
 IN                              Reboot pROBE, Install pSOS config changes
 BO;  RBO                        Boot System;   Reboot System (fake Soft Reset)
 ME;  ECI                        Call DEBUG;   EPROM Code Instrument main menu
 CONF;   DIAG                    Call CONFIGURATION;  DIAGNOSTICS main menu
 GP [{D U {count}} {address}]   GPIB Binary downloading or uploading
 GPS [{D {base} U {cnt}} {MemAddr} {ImAddr}]   GPIB S-Record down/uploading
 GE;  GD;   GEO;  GDO            Enable; Disable GPIB as input/output
 OME [address];   OMD            Enable; Disable Memory as output
 MS [address]                    Initiate a memory script
 MI [name '-'A]                  Macro removal/table initialization
 MA [{M {addr} B} {name}]        Macro creation
 MD;   name [parms]              Macro displaying;   Macro calling
 P  [B W L] [R] [addr]           Patch memory
 MO {D R S {num} value}          Access Slot 0 MODID support
 FI [log addr];    FV            Find information about a; all VXI device(s)
 VR [log addr] [R W [value]] [reg name or '-'offset]
                                 Access VXI registers
 []--optional; {}--required
```

Mode Control Commands

The mode control commands are described on the following pages.

- BOOT
- CONF
- DIAG
- ECI
- IN
- ME
- RBO

The mode control commands give you the capability to exit from pROBE into the runtime system, nonvolatile configuration, and diagnostic modes. You can also reinitialize pROBE, or invoke the pROBE main menu, or the EPROMed Code Instrument (ECI) image creator menu.

BOOT

Purpose: Cold boot GPIB-VXI/C system.

Command Format: `BO` or `BOOT`

Action: Reboot pROBE silently (installing any new changes), reboot pSOS, and enter GPIB-VXI/C system mode (local command set active).

Example: `BO`

CONF

Purpose: Enter GPIB-VXI/C nonvolatile memory configuration mode.

Command Format: `CONF`

Action: Enters the nonvolatile memory configuration menu. Refer to Chapter 4, *Nonvolatile Configuration*, of the *GPIB-VXI/C User Manual*, for instructions on using the nonvolatile memory editor.

Example: `CONF`

DIAG

Purpose: Enter GPIB-VXI/C diagnostics mode.

Command Format: `DIAG`

Action: Enters diagnostics menu. Refer to Chapter 5, *Diagnostic Tests*, of the *GPIB-VXI/C User Manual* for details regarding the use of the GPIB-VXI/C self-test diagnostics.

Example: `DIAG`

ECI

Purpose: Enter GPIB-VXI/C EPROMed Code Instrument (ECI) image creator menu.

Command Format: `ECI`

Action: Enters ECI image creator menu. The following menu will be displayed:

```
GPIB-VXI EPROM Code Instrument Image Creator
      (C) 1991    National Instruments
=========================================
1). Change base of EPROM images ($E80000)
2). Clear work Image Region
3). Copy installed EPROM image to work Image
4). Download a Code Instrument to work Image
5). Split work Image into even and odd banks
6). Upload EPROM image
7). Quit configuration
```

The main purpose of the image creator menu is to make it possible to combine multiple ECI images into one single image to be downloaded to an EPROM programmer. This image creator can also be used to convert binary images to S-record images or vice versa, and to take multiple or mixed S-record or binary images and create a single clean S-record or binary image.

The ECI image creator uses a RAM buffer to emulate EPROM addresses (you must have at least 1M of RAM installed.). The default EPROM address is E80000h, which is the address of the first bank of the EPROM daughter card. Begin by clearing the work Image Region (menu selection 2). If appending to an existing set of EPROMs, copy the current image to the work Image (3). Next, download the ECI images one at a time to the work Image (4). When complete, upload the image either all at once or in separate pieces (for separate EPROM banks) using menu selections 5 and 6.

Note: In order to create a set of EPROMs, the final image must be split into even and odd banks (in a 16-bit architecture, EPROMs must be created in pairs). This function can be supplied by the host, the EPROM programmer, or menu selection 5, which splits the work Image with the even image at the first 40000h of the 512K image and the odd image at the second 40000h.

If at any time you wish to look at the current image, you can leave the ECI menu and use basic pROBE memory access commands such as `dump memory` (`dm`) to view the image. The image is located at local RAM address 80000h. Reentering the ECI menu only reinitializes the address of the psuedo-EPROM image. No image data will be corrupted.

Example: `ECI`

IN

Purpose: Initialize pROBE.

Command Format: `IN`

Action: Reboots pROBE, installing any pSOS configuration changes.

Example: `IN`

ME

Purpose: Invoke VXI pROBE menu.

Command Format: `ME`

Action: Enters VXI pROBE menu mode.

Example: `ME`

RBO

Purpose: Cold boot GPIB-VXI/C runtime system into its former state.

Command Format: `RBO`

Action: Fake a soft reset per VXIbus Specification Revision 1.2. The command will cause the GPIB-VXI/C to reinstall the previous configuration as the default configuration. No Resource Manager functions will be run.

Example: `RBO`

GPIB Port Control Commands

Descriptions of the GPIB port control commands are given on the following pages.

- GD
- GDO
- GE
- GEO
- GP
- GPS

The GPIB port control commands control pROBE I/O to the GPIB port. You can use the GE, GD, GEO, and GDO commands to independently enable and disable input and output to the GPIB port. The default settings are GPIB input enabled and GPIB output disabled.

You can use the GP and GPS commands to upload or download binary or S-record memory images between the GPIB-VXI/C and an external host via the GPIB bus.

GD

Purpose: Disable GPIB port as input.

Command Format: GD

Action: Disables GPIB port connection to pROBE command interpreter.

Example: GD

GDO

Purpose: Disable GPIB port as pROBE output.

Command Format: GDO

Action: Disables pROBE output routing to GPIB port.

Example: GDO

GE

Purpose: Enable GPIB port as input.

Command Format: `GE`

Action: Enables GPIB port connection to pROBE command interpreter.

Example: `GE`

GEO

Purpose: Enable GPIB port as pROBE output.

Command Format: `GEO`

Action: Enables pROBE output routing to GPIB port. All characters that would be printed to the console, including characters typed at the keyboard and the `pROBE>` prompt, are routed to the GPIB port as well. Characters are not buffered, so be aware that characters may need to be constantly read from the pROBE GPIB output.

Example: `GEO`

GP

Purpose: Download or upload binary object code over GPIB port.

Command Format: `GP {D | U {<count>}} {<address>}`

Action: If D/U is D, downloads data to locations starting at `<address>`, until EOI is sent. If D/U is U, uploads `<count>` bytes from locations starting at `<address>`. The last byte is sent with EOI.

`<count>` and `<address>` are hexadecimal values.

Example: Download to address 80000.

```
GP D 80000
```

GPS

Purpose: Download or upload S-record object code through the GPIB port

Command Format: `GPS [D{<base>} | U {<count>} {<memory address>} {<ImAddrr>}]`

Action: If D/U is D, downloads S-record image to GPIB-VXI/C at:

```
Download Address = S-record address + <ImAddr> - <base> + <memory address>
```

The download terminates with EOI. The GPS command will accept Motorola S0, S2, S3, and S9 records.

If D/U is U, uploads `<count>` bytes of data from memory starting at `<memory address>`. The S-record address is:

```
S-record address = GPIB-VXI/C memory address - <memory address> + <ImAddr>
```

Example: Download S-record to S-record address.

```
GPS D 0 0 0
```

pROBE Output Routing Control Commands

The pROBE output routing commands are described below.

- OMD
- OME

The pROBE output routing control commands give you the capability to make a log of pROBE output in the GPIB-VXI/C's memory. You can use the OME command to initiate recording, and the OMD command to terminate it. The log could then be uploaded to an external host using the GP command.

OMD

Purpose: Disable copying of pROBE output to memory.

Command Format: OMD

Action: No subsequent pROBE output characters are copied to memory. The last address where data was stored will be displayed on the console.

Example:
```
OMD
```

OME

Purpose: Enable copying of pROBE output to memory.

Command Format:
```
OME {address}
```

Action: All subsequent characters that are printed to the console, including characters typed at the keyboard and the pROBE> prompt are copied to memory starting at <address>.

Example:
```
OME
```

Macro and Script Commands

Descriptions of macro and script commands are given on the following pages.

- MA
- MD
- MI
- MS

A *macro* is a command that is expanded by pROBE according to the contents of a section of memory. When a macro is called, pROBE looks up its address in a macro table and executes the character sequence located at the macro address until it encounters the macro termination character (FFh).

You can call a macro by entering the macro name, followed by up to ten parameters. Insert blank spaces to delimit the parameters from the macro name and from each other. Parameters containing spaces must be enclosed in double quotes (for example, "This is a parameter containing spaces.").

Macro parameters are indicated in the macro text with an *@N* character sequence, where *N* is a number in the range 0 to 9. @0 corresponds to the first macro parameter, and @9 is the tenth macro parameter. As the interpreter encounters each parameter sequence in the macro text, it substitutes the macro parameter for the *@N* character sequence. You can nest macros or call them recursively, with each call having its own independent set of up to ten parameters. In addition, you can download macros or enter them as keystroke sequences from the console.

The MA command designates a macro name for the macro table and records the macro's content. The MI command removes a macro from the macro table. MD displays the current macro names.

The MS command initiates the execution of a *script*. While scripts are similar to macros, one primary difference is that scripts are referenced by their start address rather than by a name. Scripts differ also in that parameters are not allowed. A pROBE command string can be downloaded from an external host to GPIB-VXI/C memory and executed as a script.

You can abort commands by typing <CTRL>-<C>, and you can abort macro and script execution by typing <CTRL>-<A>. You can pause command responses to the serial port by using the XON/XOFF protocol (<CTRL>-<Q>, <CTRL>-<S>).

MA

Purpose: Create a macro.

Command Format: `MA {{M {address} | B} {macro name}}`

Action: If the first parameter is `M` (a memory address macro), `<macro name>` is added to the macro table. The first character of the macro is at location `<address>`.

If the first parameter is `B` (a buffer macro) `<macro name>`, you will be prompted to enter the character string, which is recorded until you enter <CTRL>-<D>. `<macro name>` is transparently assigned a start address and added to the macro table.

Example: Type in macro `startup`.

```
MA B startup
```

MD

Purpose: Display macros.

Command Format: `MD`

Action: Each macro text is displayed, with a pause between macros for you to enter <CR> or `Q`. <CR> causes the next macro to be displayed, while `Q` quits.

When all macros have been displayed, the available macro memory is displayed.

Example: `MD`

MI

Purpose: Remove macro or initialize macro table.

Command Format: `MI {macro name | -A}`

Action: If the parameter is a `<macro name>`, the macro `<macro name>` is removed from the macro table.

If the parameter is `-A`, the macro table is reinitialized (all macro names are removed).

Example: Reinitialize macro table.

```
MI -A
```

MS

Purpose: Initiate memory script.

Command Format: `MS {address}`

Action: Execute memory script previously downloaded to `<address>`. pROBE will execute the character sequence beginning at `<address>` until it encounters the script termination character (FFh).

Example: Execute a memory script that has been loaded into memory at GPIB-VXI/C local address 0x30000.

```
MS 30000
```

Hardware Access Commands

The hardware access commands are described on the following pages.

- MO
- P
- VR

You can use the MO command to manipulate the Slot 0 MODID hardware. The GPIB-VXI/C MODID register can be read and written to. You can also use the MO command to assert a particular MODID line on the VXI backplane.

The P command displays and changes the contents of RAM and registers accessible through the local address space. It is similar to the built-in pROBE command pm but is of a more interactive nature.

The VR command accesses the contents of any device's VXI-defined and device-dependent registers, which can then be displayed or altered manually. VR recognizes a complete set of register name mnemonics for Register-Based and Message-Based devices, which simplifies its usage.

MO

Purpose: Access local Slot 0 MODID hardware. GPIB-VXI/C must be configured for and in Slot 0.

Command Format: `MO {D | R | S {num} | <value>}`

Action: Command `MO D` disables the MODID line drivers by writing 0000h to the MODID register.

Command `MO R` reads and displays the contents of the MODID register.

Command `MO S <num>` enables the MODID line drivers, and asserts the line for Slot `<num>`.

Command `MO <value>` writes `<value>` ORed with 2000h to the MODID register. This causes `<value>` to be driven on the MODID lines, because 2000h sets the MODID enable bit.

`<num>` and `<value>` are hexadecimal values.

Example: Assert MODID line for Slot 5.

```
MO S 5
```

P

Purpose: Patch memory. Reads and writes memory beginning at location `<address>`.

Command Format: `P [B | W | L] [R] {address}`

where `<address>` is a hexadecimal value.

The first parameter sets the read/write access mode. `B` sets the mode to 8-bit (byte). `W` sets the mode to 16-bit (word). `L` sets the mode to 32-bit (longword). The default read/write mode is 8-bit (byte).

The `R` parameter causes pROBE to preread and display the next memory location automatically (without the second <CR>).

Action: `<address>` is displayed, followed by a colon prompt. You can perform various functions at this point.

- To display the hexadecimal value in memory at `<address>`, enter a <CR> at the colon prompt. The data is displayed, followed by a colon prompt.
- To write a new value, type a hexadecimal value at the colon prompt, followed by a carriage return. Each time a new byte value is entered, the same address is repeated until no new data is entered.
- To skip the displayed address, type `n` at the colon prompt. This is useful, for example, if you want to read the contents of a location without altering them. Remember that if the `R` parameter is used, pROBE will preread the location, so in this case entering `n` is no different than entering a <CR>.
- To cycle the read/write access mode through word, longword, and then back to byte, type `W` at the colon prompt.
- To enter a new patch address, type a period at the colon prompt.
- To return to pROBE, type `Q` at the colon prompt.

Example: Patch memory at location 5607h in longword mode and with preread activated.

```
P L R 5607
```

VR

Purpose: Access VXI registers.

Command Format: `VR {logical address} {R | W {value}} {reg name | offset}`

Action: Read or write VXI registers for device at `<logical address>`.

`<logical address>`, `<value>`, and `<offset>` are hexadecimal values.

The `R` parameter causes the register contents to be read and displayed. The `W` parameter causes the value `<value>` to be written to the register.

`<reg name>` is a two-letter mnemonic for the register name, and `<offset>` is the address offset of the register relative to the base address of the `<logical address>`, as follows:

Register	<reg name>	<offset>	Access
ID	ID	00h	R
Device type	DT	02h	R
Status	ST	04h	R
Control	CO	04h	W
Offset	OF	06h	R/W
Attribute - memory	AT	08h	R
MODID - Register-Based	MO	08h	R/W
Protocol	PR	08h	R
Signal	SI	08h	W
Response	RE	0Ah	R
Read Data Low	DL	0Eh	W
Write Data Low	DL	0Eh	R
Subclass	SU	1Eh	R

To display the register name mnemonics, enter the `VR` command without the register name, followed by a `?` in response to the register name prompt. The command interpreter only associates the register name with the mnemonic; it does not attempt to intercept illegal accesses (for example, an attempt to read the Control register will read the Status register). The registers at 10h through 1Ch, and 20h through 3Eh are device-dependent, and therefore can only be accessed through their offset value. Offsets must be even values, and only word accesses are permitted.

Examples: Write 0000h to logical address 24h Read Data Low register.

```
VR 24 W 0 DL
```

Read logical address 5 device-dependent register at offset 20h.

```
VR 5 R -20
```

VXI Device Information Commands

Descriptions of the VXI device information commands are given on the following pages.

- `FI`
- `FV`

The `FI` and `FV` commands display VXI device information for a particular device or for all devices in the VXI system, respectively. These commands display information similar to that returned by the local command `RmEntry?`, but they also display the state of the VXI-defined Status, Response, and Protocol register bits.

FI

Purpose: Display VXI device information for a logical address.

Command Format: `FI {logical address}`

Action: Displays device type, logical address, slot number, manufacturer ID number and name, model code and model, and status register contents for `<logical address>`. If the device is Message-Based, the Response and Protocol register contents are also displayed.

The state of the Status and Response register bits are also indicated by a printout of their names. Upper case letters indicate a logical TRUE state of a flag, while lower case letters indicate a logical FALSE state. Notice that active high (1) bits are TRUE when their value is 1 (as in DOR). Active low (0) bits are TRUE when their value is 0 (as in Err*). The printout of the bit states are based upon logical values, and are independent of the active level of the bit.

Example: Get information about logical address 0.

```
FI 0
```

FV

Purpose: Display VXI device information for all logical addresses.

Command Format: FV

Action: Displays device type, logical address, slot number, manufacturer ID number and name, model code and model, and status register contents for all known logical addresses. If the device is Message-Based, the Response and Protocol register contents are also displayed.

The state of the Status and Response register bits are also indicated by a printout of their names. Upper case letters indicate a logical TRUE state of a flag, while lower case letters indicate a logical FALSE state. Notice that active high (1) bits are TRUE when their value is 1 (as in DOR). Active low (0) bits are TRUE when their value is 0 (as in Err*). The printout of the bit states are based upon logical values, and are independent of the active level of the bit.

Example: FV

Appendix A
Code Instruments Source Code

This appendix contains a summary of the contents of the C and assembly source code files that you will need in order to implement Code Instruments (CIs) on the GPIB-VXI/C. These files can be found in machine-readable form on the GPIB-VXI/C Distribution Disk Code Instrument Examples/Shell Files, source (part number 420400-44), that you received from National Instruments as part of this kit.

Note: This summary contains the actual filenames. MS-DOS has truncated some of the filenames that appear here.

```
================================================================
```

National Instruments GPIB-VXI Sample Code Instrument
Copyright 1991 National Instruments Corporation
All rights reserved.

The following is a short file-by-file description of the uses of each file used by a Code Instrument.

The function-specific code for the CDS 852 Code Instrument are in the files worker.c and cds852.c. The function-specific code for the DMAmove Code Instrument is in the file worker.c only.

```
================================================================
```

******* C files *******

asynch.c

Contains most of the C code that controls the Asynch process. This file contains the process entry point routine as well as its main functions. The Asynch process is responsible for handling all asynchronous events including Word Serial commands/responses and VXI Response/Event signals. This code should be nearly the same for any Code Instrument.

cds852.c

Contains all of the code specific to a CDS 852 Adapter Code Instrument. This code consists of the init, reset, read, write, and local parameter changing functions.

init.c

Contains all of the code for the Pre-init sequence. This is used to link the Code Instrument with the GPIB-VXI runtime system.

SampleWSsl.c

Contains a sample code shell for use under the Asynch process when taking over the physical Word Serial registers of the GPIB-VXI.

UserLibCallsInit.c

Contains all of the code necessary to access a user-defined function library contained in another Code Instrument. This is a sample and is nonfunctional.

UserLibTable.c

Contains all of the code and data structures necessary to implement a user-defined function library. This is a sample and is nonfunctional.

worker.c

Contains the main loop structure for the Worker process, which will be nearly the same for all Code Instruments. The Worker process is responsible for handling all of the synchronous operations (read, write, and so on). It simply needs to make calls to the appropriate read, write, and local command set routines.

******* Assembly files *******

DCIheader.asm

Contains the code necessary to reserve space for the Code Instrument header. It also contains a pointer to the Pre-init routine and the default setup parameters for EPROMed Code Instruments.

DciLibCalls.asm

Contains all of the code needed to interface the National Instruments Library functions (pSOS, pREP/C, and National Instruments-specific) contained in the National Instruments-supplied firmware.

UserLibCalls.asm

Contains all of the code and definitions needed in order to link with a user-defined function library contained in another Code Instrument. This is a sample and is nonfunctional.

******* Include (*.h) files *******

ctype.h and stdio.h

These files are copies provided by Software Components Group and are needed to supply C with expected function names and variables.

DataSizes.h

Contains the basic type definitions for standard variables in C (for example, uint8 = unsigned char). All of the code is supplied using these type definitions instead of the standard definitions, in order to ease porting of code and to avoid conflicting data sizes (for example, int = 16 or 32 bits) among many C compilers.

DciStruct.h

Contains all of the defines and structure definitions needed in order to access/report information about the Code Instrument. This file includes the C structure definition of the header.

InclWS.h

Contains all of the defines and structure definitions needed in order to perform both Servant and Commander Word Serial operations. These operations include buffer reads and writes, as well as command sending and query responding.

CommonBuf.h

Contains all of the defines and structure definitions in order to access a 'CommonBuf' buffer. 'CommonBuf' buffers are the mode used to transfer all data between resources on the GPIB-VXI. These resources include GPIB I/O, Word Serial Servant I/O, Word Serial Commander I/O, GPIB-VXI Local Parser I/O, as well as Code Instrument I/O. 'CommonBuf' buffers will be used for any future enhancements.

ResManIncl.h

Contains all of the defines and structure definitions needed in order to access Resource Manager entries for a particular device. These structure definitions allow a Code Instrument direct access to the tables contained on the GPIB-VXI. No function calls are required.

signals.h

Contains the definitions of all of the VXI Response and Event signals.

DciFuncs.h

An include file that gives all of the type definitions for the library functions supplied with the Code Instruments.

S_EVENT.h

Contains the definitions for the different pSOS signals. Do not confuse pSOS signals with VXI signals; they are completely different. pSOS signals are used for process synchronicity.

DCI_DEBUG.h

An include file used to turn on and off certain debug modes built into the sample Code Instrument source code.

UserDMA.h

An include file used to call the function DMAmove. This file contains all of the necessary mode and return code defines.

dips.h

An include file used to access the onboard dip switches.

trig.h

An include file which contains all of the parameter and return code defines for use with the National Instruments Trigger Interface Chip (TIC) trigger functions for the GPIB-VXI/C.

******* Misc. files *******

dci.bat

A simple batch file which calls the "nmk" utility to use the file dcimake as a source makefile for a Downloadable Code Instrument.

dcimake

A sample MicroSoft makefile that will work with the Code Instrument structure supplied here and with the MicroTek MCC68K C Compiler. It allows you to automatically 'make' a Downloaded Code Instrument (DCI).

eci.bat

A simple batch file which calls the "nmk" utility to use the file ecimake as a source makefile for a EPROMed Code Instrument.

ecimake

A sample MicroSoft makefile that will work with the Code Instrument structure supplied here and with the MicroTek MCC68K C Compiler. It allows you to automatically 'make' an EPROMed Code Instrument (ECI) image.

Appendix B
GPIB-VXI/C VXI Trigger Support

This appendix contains an overview of the VXI triggering capabilities of the GPIB-VXI/C and GPIB-VXI/CP.

Both of these interfaces contain a custom ASIC designed by National Instruments called the Trigger Interface Chip (TIC). The TIC gives direct access/control to the VXI trigger lines as well as some unique hardware features such as a 16-bit counter/timer, a dual 5-bit tick timer, and a 10 by 10 crosspoint switch matrix. It also includes some minimal signal conditioning such as signal inversion, variable pulse stretching, and synchronization with a clock source. The TIC supports all VXI-defined trigger protocols on the eight VXI TTL trigger lines and the two P2 connector ECL trigger lines.

In addition, the TIC has 10 external connections referred to as General Purpose Input/Output (GPIO) connections. Figure B-1 shows the configuration of the GPIOs on the GPIB-VXI/C and the GPIB-VXI/CP. You can route a GPIO to any or all of the VXI trigger lines or the 5-bit tick timer. By using the built-in 10 MHz clock or GPIO 16 kHz, 4.9152 MHz, or TRIG IN connections, you can generate a square wave on any or all of the backplane trigger lines. By using the dual 5-bit tick timers, you can divide any of these frequencies down to a lower frequency. You can also disconnect a GPIO from its external connection and use it as a crosspoint switch location. In this mode, you can use any single trigger line as the internal source for the GPIO and any or all of the remaining trigger lines as the destination.

Refer to the *GPIB-VXI/C Trigger Functions* section of Chapter 6, *C Function Calls*, for information on programming the TIC.

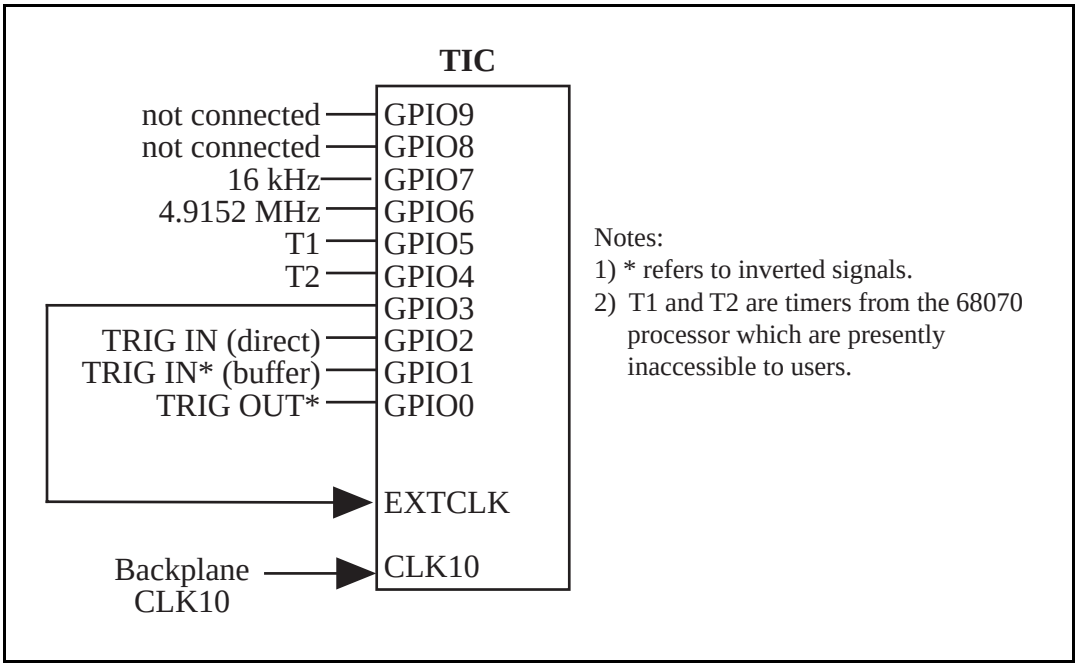


Figure B-1. GPIB-VXI/C GPIO Connections

Appendix C
Customer Communication

For your convenience, this appendix contains a form to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Corporate Headquarters
(512) 795-8248
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	(03) 879 9422	(03) 879 9179
Austria	(0662) 435986	(0662) 437010-19
Belgium	02/757.00.20	02/757.03.11
Denmark	45 76 26 00	45 76 71 11
Finland	(90) 527 2321	(90) 502 2930
France	(1) 48 14 24 00	(1) 48 14 24 14
Germany	089/741 31 30	089/714 60 35
Italy	02/48301892	02/48301915
Japan	(03) 3788-1921	(03) 3788-1923
Netherlands	03480-33466	03480-30673
Norway	32-848400	32-848600
Spain	(91) 640 0085	(91) 640 0533
Sweden	08-730 49 70	08-730 43 70
Switzerland	056/20 51 51	056/20 51 55
U.K.	0635 523545	0635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___)_____ Phone (___)_____

The problem is _____

_____

_____

_____

_____

List any error messages _____

_____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

_____

_____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **GPIB-VXI/CP Software Reference Manual**

Edition Date: **December 1993**

Part Number: **320405-01**

Please comment on the completeness, clarity, and organization of the manual.

If you find errors in the manual, please record the page numbers and describe the errors.

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (_____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678

Glossary

Asynch process	One of two required pSOS processes for a CI. The Asynch process handles all asynchronous events.
backplane	An assembly, typically a printed circuit board, with 96 pin connectors and signal paths that bus the connector pins. VXIbus systems have either two sets of bused connectors, designated J1 and J2 backplanes, or three sets of bused connectors, designated J1, J2, and J3 backplanes.
CI	See *Code Instrument*.
code instrument	CI; a proprietary National Instruments software structure that uses software to emulate the capabilibites of a VXI Message-Based device.
command	Causes the GPIB-VXI/C to take some action.
Commander	A Message-Based device that is also a bus master and can control one or more Servants.
commonBuf	A National Instruments-defined data structure used internally on the GPIB-VXI/C to transfer data from any I/O source to any other I/O souce.
console response	Returned in the form of readable sentences, which is better suited for interactive command entry.
DCI	See *Downloaded CI*.
diagnostics mode	Mode in which you can perform extensive offline diagnostic tests of the GPIB-VXI/C.
Downloaded CI	DCI; a form of CI that is downloaded into the GPIB-VXI/C's RAM memory.
dynamic configuration device	DC device; a device that initially has a logical address of 255. The RM subsequently assigns it a different, unique logical address.
ECI	See *EPROMed CI.*
EEPROM	Electrically Eraseable Programmable Read Only Memory.
EPROM	Eraseable Programmable Read Only Memory.
EPROMed CI	ECI; a form of CI that is user-installed into EPROMs.
global CI RAM area	An area of GPIB-VXI/C RAM reserved for the data area of Static CIs.

GPIB	General Purpose Interface Bus
GPIB-VXI/C local command set	Consists of commands and queries.
GPIO	General Purpose Input/Output
hex	hexadecimal
Hz	hertz
IEEE	Institute of Electrical and Electronic Engineers
in.	inches
K	kilobytes of memory (1,024 bytes)
logical address	An 8-bit number that uniquely identifies each VXIbus device in a system. It defines a device's A16 register address and indicates Commander/Servant relationships.
M	megabytes of memory (1,048,576 bytes)
memory map	An association between resources and CPU addresses. A hardware memory map maps physical resources such as RAM and EPROM to particular CPU addresses. A software memory map maps particular code segments to particular CPU addresses.
Message-Based device	An intelligent device that implements the defined VXIbus registers and communication protocols.
MODID lines	VXI backplane signals used by the Resource Manager (through the use of the Slot 0 device) in order to perform slot associations for logical addresses. There are 13 MODID lines, one for each slot in a full-size mainframe.
module	Typically consists of a board assembly and its associated mechanical parts, front panel, optional shields, and so on. A module contains everything required to occupy a slot in a mainframe. A module may occupy one or more slots.
nonvolatile configuration mode	Mode in which you can edit the contents of the nonvolatile EEPROM memory.
parse	The act of interpreting an ASCII character string as a command to perform a device-specific action.
peek	To read the contents.
PI	Position independent.
poke	To write a value.

position independent code	PI code; code that can be executed anywhere in a computer's memory map.
position independent CI	PI CI; a specific object code format for a CI. A PI CI uses PC-relative code, PC-relative initialized data, and A5-relative variable data to make it position independent.
pre-init routine	A name given to a CI's Bootstrap function. The Pre-init routine emulates the VXI CONFIG state and provides information to the GPIB-VXI/C runtime system in order to link the CI into the system.
pREP/C	An extended C library interface for use when writing C code for the pSOS operating system kernel. It is commercially available from Software Components Group, Inc.
pROBE	A low-level interactive debugger for use with the pSOS operating system. It is commercially available from Software Components Group, Inc. VXI pROBE is an enhanced version of pROBE supplied on developmental versions of the GPIB-VXI/C.
program mode response	Has a terse data-only format that is intended for a control program to read and parse.
pSOS	A small, multitasking operating system kernel used on the GPIB-VXI/C. It is commercially available from Software Components Group, Inc.
query	Like *command*, causes the GPIB-VXI/C to take some action, but it always returns a response containing data or other information.
RCI	See *Resident CI*.
Register-Based device	A Servant only device that supports VXIbus configuration registers. Register-Based devices are typically controlled by Message-Based devices via device-dependent register reads and writes.
Resident CI	RCI; a CI that is supplied by National Instruments and resides in the firmware.
Resource Manager	A Message-Based Commander located at Logical Address 0 that provides configuration management services such as address map configuration, Commander/Servant mappings, self-test and diagnostic management.
RM	See *Resource Manager*.
S-Record image	A commonly used ASCII format for binary object files.
Servant	A device that is controlled by a Commander. Any device can be a Servant.

static CI	A specific object code format for a CI. A static CI has a *hard coded* address for code and data. It can reside at only one specific address in the GPIB-VXI/C's memory map.
static configuration device	SC device; a device that has its logical address set by static means, such as by a DIP switch.
system configuration table	During the execution of the RM and general configuration operations, the GPIB-VXI/C builds up a table of system configuration information. Each device has an entry in the table containing the device's logical address, its Commander's logical address, its secondary address, slot number, device class, manufacturer ID number, model code, memory space requirement, memory base address, and memory size. This table remains after the RM and general configuration operations are complete. It is accessible through the GPIB-VXI/C local command set.
TIC	Trigger Interface Chip
VME	Versa Module Eurocard or IEEE 1014.
VXIbus	VMEbus Extensions for Instrumentation.
VXI pROBE mode	Mode in which you can use the enhanced pROBE debugger. This mode is available only with the GPIB-VXI/C development firmware option.
VXI system mode	The startup mode for normal operation in a VXI system.
Word Serial communication	The simplest form of communication required by Message-Based devices. It utilizes the A16 communication registers to transfer data using a simple polling handshake method.
Word Serial Protocol	The rules and regulations involved in performing Word Serial communication.
worker process	One of two required pSOS processes for a CI. The Worker process receives and processes messages from its Commander and Servants in a device-specific manner.

Index

Numbers

68881 math coprocessor
  Disable68881 function, 6-18
  Enable68881 function, 6-20
  purpose and use, 3-10 to 3-11
852 adapter CI. *See also* asynch.c file.
  definition of, 1-7, 3-3
  precaution against modifying distribution files, 3-4
  source code supplied for, 1-7
  using as a shell, 3-3 to 3-4

A

acceptor trigger functions, 6-22
AcknowledgeTrig function, 6-23
assembler requirements for developing CIs, 1-8
assembly files, A-2
Asynch process, 2-10 to 2-11, 3-5
asynch.c file, 3-5 to 3-11
  68881 math coprocessor, 3-10 to 3-11
  accessing nonvolatile configuration information, 3-10
  Control register writes, 3-8 to 3-9
  definition of, A-1
  DOR (Data Out Ready) requests, 3-8
  event/response signals/interrupts, 3-7 to 3-8
  physical Word Serial commands and queries, 3-7
  purpose and use, 3-5
  VXI startup sequence, 3-9
  Word Serial commands and queries, 3-5 to 3-7
  Word Serial Data In notices, 3-8
  Word Serial messages, 3-10
  worker.c file, 3-9
Asynchronous Mode Control command, 3-6

B

Begin Normal Operation command, 3-9
BOOT command, VXI pROBE, 8-7
branch word instruction, 4-3

C

C files, A-1 to A-2
C function calls
  GPIB-VXI/C resource access functions
    ClearParser, 6-16
    DCIParserDeq, 6-16
    DCIParserEnq, 6-17
    DCIParserQuery, 6-18
    Disable68881, 6-18
    DMAmove, 6-19
    Enable68881, 6-20
    NVconf, 6-20
    overview, 6-14 to 6-15
    ReenableHandler, 6-21
    ReserveHandler, 6-21
  GPIB-VXI/C trigger functions
    acceptor trigger functions, 6-22
    AcknowledgeTrig, 6-23
    DisableTrigSense, 6-24
    EnableTrigSense, 6-25
    GetTrigHandler, 6-26
    map trigger functions, 6-22
    MapTrigToTrig, 6-27 to 6-28
    overview, 6-22 to 6-23
    SetTrigHandler, 6-29
    source trigger functions, 6-22
    SrcTrig, 6-30 to 6-31
    TrigAssertConfig, 6-32 to 6-33
    TrigCntrConfig, 6-34
    TrigExtConfig, 6-35 to 6-36
    trigger configuration functions, 6-23
    TrigTickConfig, 6-37 to 6-38
    UnMapTrigToTrig, 6-39
    WaitForTrig, 6-40
  interface with National Instruments-supplied functions, 6-2
  interface with user-defined functions/libraries, 6-2 to 6-3
  purposes, 6-1
  VXI register access functions
    overview, 6-10
    SafeReadByte, 6-11
    SafeReadWord, 6-11
    SafeWriteByte, 6-12
    SafeWriteWord, 6-12
    VXIregBase, 6-13

Word Serial Protocol functions
CIAbortNormalOperation, 6-5
CIEndNormalOperation, 6-5
CIReleaseDevice, 6-6
ControlOut, 6-6
CreateBuf, 6-7
DCIwsClear, 6-7
DuplicateCommonBuf, 6-8
HWwsClear, 6-8
overview, 6-4 to 6-5
SendDORrequest, 6-9
SignalOut, 6-9
CDS 73A-852 adapter. *See* 852 adapter CI.
cds852.c file, A-1
CI structure. *See* Code Instrument structure.
CIAbortNormalOperation function, 6-5
CIEndNormalOperation function, 6-5
CIReleaseDevice function, 6-6
CIs. *See* Code Instruments (CIs).
Clear command, handling of, 3-6, 7-2 to 7-3
ClearParser function, 6-16
ClearServants function, 7-3
Code Instrument development
compiler, assembler and linker requirements, 1-8
development system configuration, 1-7
position independent code versus static code, 1-8 to 1-9
Code Instrument structure
Asynch process, 2-10 to 2-11
hardware interrupts, 2-11
header structure, 2-9
initialization routine and process structures, 2-9
memory structure
overview, 2-1
Position Independent CI memory structure, 2-6 to 2-7
Position Independent DCI memory structure, 2-7 to 2-8
Position Independent ECI memory structure, 2-8 to 2-9
Static CI memory structure, 2-1 to 2-3
Static DCI memory structure, 2-3 to 2-4
Static ECI memory structure, 2-5
Pre-init routine, 2-9 to 2-10
Worker process, 2-11
Code Instruments (CIs). *See also* designing Code Instruments (CIs); downloaded CIs; EPROMed CIs.
advantages, 1-5 to 1-6
capabilities, 3-2 to 3-3
characteristics, 1-4 to 1-5
communication protocol implementation, 1-6
communication with local command parser, 1-5
compared with physical Message-Based Commander, 1-4, 1-6
definition of, 1-3
forms, 1-4
operation and capabilities, 1-3 to 1-4
Resident CIs, 1-7
source code
assembly files, A-2
C files, A-1 to A-2
include files, A-3
miscellaneous files, A-4 to A-5
summary, 1-5 to 1-6
typical applications, 1-4
VXIbus system's view of Code Instruments (CIs), 1-6
Commander conflicts, 1-1
Commander/Servant relationships
operation without CIs, 1-1 to 1-3
support for one Commander/Servant connection, 1-3
CommonBuf structure, 3-8, 6-4
CommonBuf.h file, 7-2, A-3
compiler requirements, 1-8
CONF command, VXI pROBE, 8-7
Control Event command, 3-6
Control In event message, 3-8 to 3-9
Control register writes, 3-8 to 3-9
Control Response command, 3-6
ControlOut function, 6-6
CreateBuf function, 6-7
creating CIs. *See* designing Code Instruments (CIs); Downloaded CIs; EPROMed CIs.
ctype.h file, A-3
customer communication, *xiii*, C-1

D

Data Out Ready (DOR) requests, 3-8. *See also* SendDORrequest function.
DataSizes.h file, A-3
dci.bat file, A-4
DCI_DEBUG.h file, A-4
DCIDownLdPI command, 4-4
DCIDownLoad command, 4-3, 4-4
DciFuncs.h file, A-4
DCIheader.asm file, 2-9, A-2

DciLibCalls.asm file, 3-11, 6-2, A-2
dcimake file, A-4
DCIParserDeq function, 6-16
DCIParserEnq function, 6-17
DCIParserQuery function, 6-18
DCIs. *See* downloaded CIs.
DCISetup? command, 4-3, 5-1
DCISetupPI? command, 2-7, 4-3, 4-4, 5-1
DciStruct.h file, 2-9, 3-4, A-3
DCIwsClear function, 6-7
d_ctrl(DEV_WS,..) call, 7-7
d_ctrl(DEV_WSsl,..) call, 7-11
debugging. *See* VXI pROBE.
designing Code Instruments (CIs)
    852 adapter CI used as a shell, 3-3 to 3-4
    asynch.c file, 3-5 to 3-11
        68881 math coprocessor, 3-10 to 3-11
        accessing nonvolatile configuration information, 3-10
        Control register writes, 3-8 to 3-9
        DOR (Data Out Ready) requests, 3-8
        event/response signals/interrupts, 3-7 to 3-8
        physical Word Serial commands and queries, 3-7
        VXI startup sequence, 3-9
        Word Serial commands and queries, 3-5 to 3-7
        Word Serial Data In notices, 3-8
        Word Serial messages, 3-10
        worker.c file, 3-9
    Code Instrument capabilities, 3-2 to 3-3
    design process, 3-1 to 3-2
    implementing function libraries, 3-11
    init.c file, 3-4 to 3-5
    precaution against modifying CI distribution files, 3-4
device-dependent commands/queries, 3-7
device information commands. *See* VXI pROBE commands.
DIAG command, VXI pROBE, 8-7
dips.h file, A-4
Disable68881 function, 6-18
DisableTrigSense function, 6-24
DMAmove function, 6-19
documentation
    conventions, *xii*
    organization of manual, *xi-xii*
    related documentation, *xiii*
    related manual, *xii*
DOR (Data Out Ready) requests, 3-8. *See also* SendDORrequest function.
Downloaded CIs. *See also* Position Independent DCIs; Static DCIs.
    creating
        creation sequence, 4-3
        DCI and ECI compatibility issues, 4-1 to 4-3, 5-1
        DCI segment ordering and content, 4-1 to 4-2
        ECI segment ordering and content, 4-2
        initialization, 4-2 to 4-3
        memory image format, 4-3 to 4-4
        memory map creation, 4-3 to 4-4
        PI DCI creation sequence, 4-4
    definition of, 1-7
    memory requirements, 2-3
d_read(DEV_WS,..) call, 7-8
d_read(DEV_WSsl,..) call, 7-12
DuplicateCommonBuf function, 6-8
d_write(DEV_WS,..) call, 7-9
d_write(DEV_WSsl,..) call, 7-13

E

ECI command, VXI pROBE, 5-2, 8-8
eci.bat file, A-5
ecimake file, A-5
ECIs. *See* EPROMed CIs.
Enable68881 function, 6-20
EnableTrigSense function, 6-25
EPROMed CIs. *See also* Position Independent ECIs; Static ECIs.
    creating
        converting DCIs to ECIs, 5-1
        differences between DCIs and ECIs, 5-1
        ECI EPROM image versus PI ECI EPROM image, 5-1
        memory image format, 5-2
        Position Independent ECI memory map, creating, 5-2
        Static ECI memory map, creating, 5-1
    definition of, 1-7
    memory requirements, 2-3
event/response signals/interrupts, 3-7 to 3-8

F

FI command, VXI pROBE, 8-20
function calls. *See* C function calls.
function libraries, implementing, 3-11
FV command, VXI pROBE, 8-21

G

GD command, VXI pROBE, 8-10
GDO command, VXI pROBE, 8-10
GE command, VXI pROBE, 8-11
General Purpose Input Output (GPIO) connections, B-1
GEO command, VXI pROBE, 8-11
GetTrigHandler function, 6-26
global CI RAM area
  definition of, 2-1
  designating in RAM, 2-3
  Static DCI memory, 2-4
GP command, VXI pROBE, 8-11
GPIB buffer transfers, 7-2
GPIB port control commands. *See* VXI pROBE commands.
GPIB-VXI/C
  kit contents, 1-1
  operation without CIs, 1-1 to 1-3
GPIB-VXI/C resource access functions
  ClearParser, 6-16
  DCIParserDeq, 6-16
  DCIParserEnq, 6-17
  DCIParserQuery, 6-18
  Disable68881, 6-18
  DMAmove, 6-19
  Enable68881, 6-20
  NVconf, 6-20
  overview, 6-14 to 6-15
  ReenableHandler, 6-21
  ReserveHandler, 6-21
GPIB-VXI/C trigger functions
  acceptor trigger functions, 6-22
  AcknowledgeTrig, 6-23
  DisableTrigSense, 6-24
  EnableTrigSense, 6-25
  GetTrigHandler, 6-26
  map trigger functions, 6-22
  MapTrigToTrig, 6-27 to 6-28
  overview, 6-22 to 6-23
  SetTrigHandler, 6-29
  source trigger functions, 6-22
  SrcTrig, 6-30 to 6-31
  TrigAssertConfig, 6-32 to 6-33
  TrigCntrConfig, 6-34
  TrigExtConfig, 6-35 to 6-36
  trigger configuration functions, 6-23
  TrigTickConfig, 6-37 to 6-38
  UnMapTrigToTrig, 6-39
  WaitForTrig, 6-40
GPIO connections, B-1
GPS command, VXI pROBE, 8-12

H

HandleSignal function, 3-7
HandleWordSerialCommand function, 3-5 to 3-6
hardware access commands. *See* VXI pROBE commands.
hardware interrupts, 2-11
header structure, 2-9
HELP command, VXI pROBE, 8-5
HWwsClear function, 6-8

I

IEEE-488 VXI translation, 1-3
IN command, VXI pROBE, 8-8
include files, A-3
InclWS.h file, 7-1, A-3
init.c file, 3-4 to 3-5, A-1
initialization routine
  Downloaded CIs, 4-2 to 4-3
  init.c file, 3-4 to 3-5
  Pre-init routine, 2-9 to 2-10
interrupts
  Asynch process, 2-10 to 2-11, 3-5
  event/response signals/interrupts, 3-7 to 3-8
  hardware interrupts, 2-11
  RORA and ROAK interrupters, 3-8
  Unrecognized Command event, 7-3

L

LaSaddr local command, 3-7
linker requirements, 1-8
local command parser, 1-5

M

MA command, VXI pROBE, 8-15
macro and script commands. *See* VXI pROBE commands.
map trigger functions, 6-22
MapTrigToTrig function, 6-27 to 6-28
math coprocessor
  Disable68881 function, 6-18
  Enable68881 function, 6-20
  purpose and use, 3-10 to 3-11
MD command, VXI pROBE, 8-15
ME command, VXI pROBE, 8-8
memory image format
  Downloaded CIs, 4-4
  EPROMed CIs, 5-2
memory map, creating
  Downloaded CIs, 4-3 to 4-4
  EPROMed CIs, 5-2
  Position Independent ECIs, 5-2
memory structure
  converting DCIs to ECIs, 4-1 to 4-2
  overview, 2-1
  Position Independent CIs, 2-6 to 2-7
  Position Independent DCIs, 2-7 to 2-8
  Position Independent ECIs, 2-8 to 2-9
  Static CIs, 2-1 to 2-3
  Static DCIs, 2-3 to 2-4
  Static ECIs, 2-5
MI command, VXI pROBE, 8-16
Microtec MCC68K Compiler/Assembler/Librarian/Lin package, 1-8, 1-9
MO command, VXI pROBE, 8-17
mode control commands. *See* VXI pROBE commands.
MS command, VXI pROBE, 8-16

N

National Instruments-supplied functions, 6-2
non-VXI devices, examples, 1-5
nonvolatile configuration information, accessing, 3-10
NVconf function, 3-10, 6-20

O

OASYS-Green Hills C compiler/Assembler/Librarian/Lin package, 1-8, 1-9
OMD command, VXI pROBE, 8-13
OME command, VXI pROBE, 8-13

P

P command, VXI pROBE, 8-18
physical Message-Based Commander, 1-4, 1-6
physical Word Serial commands and queries, 3-7
PI CIs. *See* Position Independent CIs.
PI DCIs. *See* Position Independent DCIs.
PI ECIs. *See* Position Independent ECIs.
polled I/O operations, avoiding, 3-9
Position Independent CIs, 2-6 to 2-7
Position Independent DCIs
  creation sequence, 4-4
  DCI and ECI compatibility issues, 4-1 to 4-2
  memory structure, 2-7 to 2-8
Position Independent ECIs
  DCI and ECI compatibility issues, 4-1 to 4-2
  memory structure, 2-8 to 2-9
  Static ECI EPROM image versus PI ECI EPROM image, 5-1
Position independent (PI) code, compared with static code, 1-8 to 1-9
Pre-init routine
  init.c file, 3-4 to 3-5
  precaution for modifying, 3-4
  purpose and use, 2-9 to 2-10
pROBE commands. *See* VXI pROBE commands.
processes
  Asynch process, 2-10 to 2-11, 3-5
  hardware interrupts, 2-11
  Worker process, 2-11

R

RBO command, VXI pROBE, 8-8
RCIs. *See* Resident CIs.
Read Protocols command, 3-6
Read Status Byte command, 3-6
Read Status Byte query, 3-8
ReenableHandler function, 6-21
Release Device command, 3-6
REQT event handler, 3-8
ReserveHandler function, 3-8, 6-21
Resident CIs, 1-7

ResManIncl.h file, A-3
resource access functions. *See* GPIB-VXI/C resource access functions.
RS-232 port, 1-3

S

SafeReadByte function, 6-11
SafeReadWord function, 6-11
SafeWriteByte function, 6-12
SafeWriteWord function, 6-12
SampleWSsl.c file, A-2
SendDORrequest function, 6-9
SetTrigHandler function, 2-11, 6-29
setup parameters, 4-3
S_EVENT.h file, A-4
SignalOut function, 6-9
signals/interrupts, handling, 3-7 to 3-8
signals.h file, A-3
source code files
  assembly files, A-2
  C files, A-1 to A-2
  include files, A-3
  miscellaneous files, A-4 to A-5
source trigger functions, 6-22
SrcTrig function, 6-30 to 6-31
Static CI memory structure, 2-1 to 2-3
static code, compared with position independent (PI) code, 1-8 to 1-9
Static DCIs
  creation sequence, 4-3 to 4-4
  DCI and ECI compatibility issues, 4-1 to 4-2
  memory map creation, 4-3 to 4-4
  memory structure, 2-3 to 2-4
Static ECIs
  DCI and ECI compatibility issues, 4-1 to 4-2
  memory structure, 2-5
  Static ECI EPROM image versus PI ECI EPROM image, 5-1
stdio.h file, A-3
structures. *See* Code Instrument structure.

T

technical support, C-1
TIC capabilities, B-1
TrigAssertConfig function, 6-32 to 6-33
TrigCntrConfig function, 6-34
TrigExtConfig function, 6-35 to 6-36
Trigger command, 3-6
trigger functions. *See* GPIB-VXI/C trigger functions.
Trigger Interface Chip (TIC), B-1
trigger support, B-1
trig.h file, A-4
TrigTickConfig function, 6-37 to 6-38

U

UnMapTrigToTrig function, 6-39
Unrecognized Command event, handling, 7`3
user-defined functions/libraries, 6-2 to 6-3
UserDMA.h file, A-4
UserLibCalls.asm file, 3-11, 6-2 to 6-3, A-2
UserLibCallsInit.c file, 3-11, 6-2 to 6-3, A-2
UserLibTable.c file, 3-11, 6-2 to 6-3, A-2

V

VME A16 and A24 space, 6-10
VR command, VXI pROBE, 8-19
VXI pROBE
  entering and exiting, 8-1 to 8-2
  menu
    Do a GPIB download or upload, 8-2
    Do a hex dump of memory, 8-2
    Initialize pROBE, 8-2
    Jump to execution address, 8-2
    Patch memory, 8-2
    Return to pROBE, 8-3
    Trap to pROBE, 8-3
  overview, 8-1
VXI pROBE commands
  device information commands
    FI, 8-20
    FV, 8-21
  GPIB port control commands
    GD, 8-10
    GDO, 8-10
    GE, 8-11
    GEO, 8-11
    GP, 8-11
    GPS, 8-12
    overview, 8-10
  hardware access commands
    MO, 8-17
    overview, 8-17
    P, 8-18
    VR, 8-18

HELP, 8-5
macro and script commands
MA, 8-15
MD, 8-15
MI, 8-16
MS, 8-16
overview, 8-14
mode control commands
BOOT, 8-7
CONF, 8-7
DIAG, 8-7
ECI, 5-2, 8-8
IN, 8-9
ME, 8-9
overview, 8-6
RBO, 8-9
notation and syntax, 8-4
overview, 8-4
pROBE output routing control commands
OMD, 8-13
OME, 8-13
VXI register access functions
overview, 6-10
SafeReadByte, 6-11
SafeReadWord, 6-11
SafeWriteByte, 6-12
SafeWriteWord, 6-12
VXIregBase, 6-13
VXI startup sequence, 3-9
VXIbus
communication protocols, 1-6
interaction with Code Instruments (CIs), 1-6
VXIregBase function, 6-13

W

WaitForTrig function, 6-40
Word Serial Command device driver calls
d_ctrl(DEV_WS,..), 7-7
d_read(DEV_WS,..), 7-8
d_write(DEV_WS,..), 7-9
functions performed by, 7-1
overview, 7-6
Word Serial commands and queries
Asynchronous Mode Control, 3-6
Clear, 3-6
Control Event, 3-6
Control Response, 3-6
device-dependent commands/queries, 3-7
handled by Asynch process, 3-5 to 3-6
physical Word Serial commands and queries, 3-7
Read Protocols, 3-6
Read Status Byte, 3-6
Release Device, 3-6
Trigger, 3-6
Word Serial communication
communication protocol implementation, 1-6
examples, 7-3 to 7-5
receiving buffers, 7-4
sending buffers, 7-5
sending commands, 7-3
sending responses, 7-4
Word Serial Data In Asynch event message, 3-8
Word Serial drivers
buffer-based communication, 7-1 to 7-2
clear command, handling, 7-2 to 7-3
GPIB buffer transfers, 7-2
implementing Word Serial operations, 6-10
overview, 7-1
Unrecognized Command event, handling, 7-3
Word Serial Command device driver calls
d_ctrl(DEV_WS,..), 7-7
d_read(DEV_WS,..), 7-8
d_write(DEV_WS,..), 7-9
Word Serial communication examples, 7-3 to 7-5
receiving buffers, 7-4
sending buffers, 7-5
sending commands, 7-3
sending responses, 7-4
Word Serial Slave driver calls
d_ctrl(DEV_WSsl,..), 7-11
d_read(DEV_WSsl,..), 7-12
d_write(DEV_WSsl,..), 7-13
overview, 7-10
Word Serial messages, 3-10
Word Serial Protocol functions
CIAbortNormalOperation, 6-5
CIEndNormalOperation, 6-5
CIReleaseDevice, 6-6
ControlOut, 6-6
CreateBuf, 6-7
DCIwsClear, 6-7
DuplicateCommonBuf, 6-8
HWwsClear, 6-8
overview, 6-4 to 6-5
SendDORrequest, 6-9

SignalOut, 6-9
Word Serial Servant, 1-3
Word Serial Slave driver calls
d_ctrl(DEV_WSsl,..), 7-11
d_read(DEV_WSsl,..), 7-12
d_write(DEV_WSsl,..), 7-13
overview, 7-10
WordSerEna command, 3-7
Worker process, 2-11
worker.c file, 3-9, A-2
WSbuf structure, 7-1
WScom structure, 7-1
WSresp structure, 7-1

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP