HP EPC-7 486

Embedded Computer, 100 MHz Processor

EPConnect/VXI for DOS Manual Set

VOL 3 of 3

RadiSys® Corporation

15025 S.W. Koll Parkway

Beaverton, OR 97006

Phone: (503) 646-1800

FAX: (503) 646-1850

http://www.radisys.com

RadiSys®
CORPORATION

07-0230-02

December 1994

EPC and RadiSys are registered trademarks of RadiSys Corporation.

This manual part number is 07-0230-02.
It contains manuals 07-0157-02 and 07-0139-02.

July 1994

Bus Management for DOS Programmer's Reference Guide

RadiSys® Corporation

15025 S.W. Koll Parkway

Beaverton, OR 97006

Phone: (503) 646-1800

FAX: (503) 646-1850

07-0157-02 December 1994

August 1990

Software License and Warranty

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THE DISKETTE OR DISK UNIT PACKAGE. BY OPENING THE PACKAGE, YOU INDICATE THAT YOU ACCEPT THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, YOU SHOULD PROMPTLY RETURN THE UNOPENED PACKAGE, AND YOU WILL BE REFUNDED.

LICENSE

You may:

1. Use the product on a single computer;
2. Copy the product into any machine-readable or printed form for backup or modification purposes in support of your use of the product on a single computer;
3. Modify the product or merge it into another program for your use on the single computer—any portion of this product merged into another program will continue to be subject to the terms and conditions of this agreement;
4. Transfer the product and license to another party if the other party agrees to accept the terms and conditions of this agreement—if you transfer the product, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copy not transferred, including all modified versions and portions of the product contained in or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification, or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PRODUCT OR ANY COPY, MODIFICATION, OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION, OR MERGED PORTION OF THE PRODUCT TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM

The license is effective until terminated. You may terminate it at any time by destroying the product and all copies, modifications, and merged portions in any form. The license will also terminate upon conditions set forth elsewhere in this agreement or if you fail to comply with any of the terms or conditions of this agreement. You agree upon such termination to destroy the product and all copies, modifications, and merged portions in any form.

LIMITED WARRANTY

RadiSys Corporation ("RadiSys") warrants that the product will perform in substantial compliance with the documentation provided. However, RadiSys does not warrant that the functions contained in the product will meet your requirements or that the operation of the product will be uninterrupted or error-free.

RadiSys warrants the diskette(s) on which the product is furnished to be free of defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment to you.

LIMITATIONS OF REMEDIES

RadiSys' entire liability shall be the replacement of any diskette that does not meet RadiSys' limited warranty (above) and that is returned to RadiSys.

IN NO EVENT WILL RADISYS BE LIABLE FOR ANY DAMAGES, INCLUDING LOST PROFITS OR SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE PRODUCT EVEN IF RADISYS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

GENERAL

You may not sublicense the product or assign or transfer the license, except as expressly provided for in this agreement. Any attempt to otherwise sublicense, assign, or transfer any of the rights, duties, or obligations hereunder is void.

This agreement will be governed by the laws of the state of Oregon.

If you have any questions regarding this agreement, please contact RadiSys by writing to RadiSys Corporation, 15025 SW Koll Parkway, Beaverton, Oregon 97006.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATION BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

NOTES

Table of Contents

1

1. Introducing Bus Management for DOS

This manual is intended for programmers using the Bus Management for DOS programming interface to develop programs that control VXI I/O modules via the VXI expansion interface on an EPC.

The Bus Management library is one of the application programming interfaces (APIs) that are part of EPConnect. You are expected to have read the *EPConnect/VXI for DOS & Windows User's Guide* for an understanding of what is in EPConnect, to learn the terms and conventions used in this manual set, and how to install and configure the Bus Management for DOS API for use on your system. You are not expected to have in-depth knowledge of DOS.

The Bus Management for DOS API provides a powerful interface for interacting with the VXIbus. RadiSys offers considerable flexibility by supplying interfaces for several high-level languages. By observing the MS Pascal binding conventions, you can use EPConnect with these languages. See Chapter 4, *Advanced Topics,* for more information on programming.

Chapter 1 introduces you to the RadiSys Bus Management for DOS environment. In it you will find the following:

- What is in this manual and how to use it
- What is Bus Management for DOS?
- Programming, Compiling and Linking
- What to do next

1.1 How This Manual is Organized

This manual has five chapters:

Chapter 1, *Introduction*, introduces Bus Management for DOS and this manual.

Chapter 2, *Function Descriptions*, describes the major categories of functions and gives complete descriptions of each function. Function descriptions are alphabetic by function name.

Chapter 3, *Advanced Topics*, provides information for developing advanced applications.

Chapter 4, *Error Messages*, contains an alphabetic listing of error messages generated by EPConnect device drivers.

Chapter 5, *Support and Service*, describes how to contact RadiSys Technical Support for support and service.

1.2 What is Bus Management for DOS?

Bus Management for DOS consists of those portions of the EPConnect software package that are required by "C/C++" and Basic programmers developing VXI applications that run under DOS on a RadiSys Embedded Personal Computer (EPC). Figure 1-1 is a diagram of the Bus Management for DOS software architecture that shows how the architecture relates to the VXIbus.

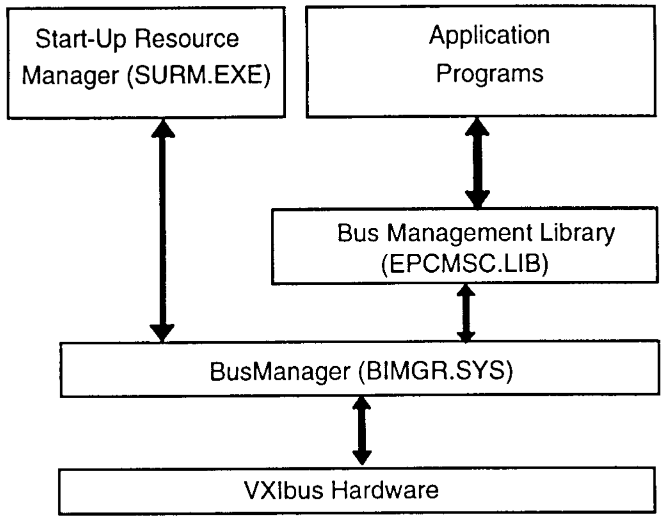

Figure 1-1. Bus Management for DOS Architecture

1.2.1 Bus Management Library and BusManager Device Driver

Bus Management for DOS consists of an application interface library (**EPCMSC.LIB**) and a device driver (**BIMGR.SYS**). User-written DOS applications access the VXIbus hardware by calling the functions supported by the interface library, which in turn call the BusManager device driver. These functions allow DOS applications to do the following:

- Handle VME interrupts and system errors.
- Transfer blocks of data to and from VXIbus devices, with BERR detection.
- Control VXIbus word serial registers.
- Control EPC slave memory
- Query EPC driver, firmware, and hardware version or type.

The Bus Management library supports MS Basic compilers and ANSI-standard "C" compilers, such as Microsoft C/C++ and Borland C/C++.

The Bus Management Library is fully reentrant.

1.2.2 SURM

The Start-Up Resource Manager (SURM) is a DOS application that determines the physical content of the system and configures the devices. It is typically the first program to run after DOS boots. The SURM is the EPConnect implementation of resource manager defined in the VXIbus specification. However, SURM extends the specification definition to include non-VXIbus devices, such as VME devices and GPIB instruments. The SURM uses the **DEVICES** file to obtain device information not directly available from the devices. SURM accesses VXIbus devices in the system directly.

1.3 Programming, Compiling and Linking

This section contains information about programming with Bus Management for DOS. Included is a list of the header files provided, the programming interfaces, and compiling and linking hints.

1.3.1 Header Files

Bus Management for DOS provides the following header files:

BMBLIB.BI An MS BASIC header file containing constant and function declarations required for using EPConnect with MS BASIC.

BUSMGR.H A "C" header file containing the constant definitions, macro definitions, and function prototypes required to compile applications using any Microsoft or Borland "C" or C++ compiler.

BUSMGR.INC A copy of **BUSMGR.H** that's been converted so that it is suitable for inclusion into an assembly language source file.

EPC_OBM.H A "C" header file containing the constant definitions, macro definitions, structure definitions, and function prototypes required to compile EPConnect applications for DOS.

EPC_OBM.H should never be included in a source file directly. **BUSMGR.H** includes **EPC_OBM.H**.

EPCSTD.H A "C" header file containing macro definitions to standardize non-ANSI, compiler-dependent keywords. By using the macros defined here, an application can compile successfully using any revision of Microsoft or Borland "C" or C++ compiler without modifying the source file.

EPCSTD.H should never be included in a source file directly. **BUSMGR.H** includes **EPC_OBM.H**.

VMEREGS.H A "C" header file containing constant and macro definitions for accessing the EPC VMEbus control registers.

VMEREGS.INC A copy of **VMEREGS.H** that has been converted so that it is suitable for inclusion into an assembly language source file.

All Bus Management for DOS header files contain an **#if/#endif** pair surrounding the contents of the header file so that the file can be included multiple times without causing compiler errors.

All "C" header files also contain **extern "C"{}** bracketing for C++ compilers. Because **extern "C"** is strictly a C++ keyword, it is also bracketed and only visible when compiling under C++ and not standard "C."

1.3.2 Programming Interface

Bus Management for DOS functions are accessible through interfaces for assembly language, "C", and BASIC languages. The following table shows the interface libraries and definition files for each of the language interfaces.

Language	*Library files*	*Definition files*
MS "C"	**EPCMSC.LIB**	**BUSMGR.H**
Borland "C"	**EPCMSC.LIB**	**BUSMGR.H**
MS BASIC	**EPCMSC.LIB**	**BMBLIB.BI**
Assembly	**EPCMSC.LIB**	**BUSMGR.INC**

The use of these files is discussed in the following sections.

Calling Bus Management for DOS From MS "C" and QuickC

The "C" language interface is designed to work with Version 5.1 and later versions of the Microsoft "C" compiler and libraries. The libraries are created for the large memory model (far code and far data). This is sufficient for linking programs of any model size, due to the prototyping of all library functions in the include files. The include files provide strong type checking and convert near code and data to far code and data for programs using the small (near code and near data), compact (near code and far data), or medium (far code and near data) memory models.

Calling EPConnect From Borland Turbo C

Bus Management for DOS was designed to work with the Microsoft "C" compilers and can be used with the Borland "C" compilers as well.

Calling EPConnect from MS BASIC

The BASIC language interface is designed to work with Version 7.0 and later versions of the Microsoft BASIC compiler and libraries. The libraries are created for the large memory model (far code and far data). This is sufficient for linking programs of any model size, due to the prototyping of all library functions in the include files. The include files provide strong type checking and convert near code and data to far code and data for programs using the small (near code and near data), compact (near code and far data), or medium (far code and near data) memory models.

Calling Bus Management for DOS From Assembly Language

Assembly language programs can use Bus Management for DOS functions through the **BMINT** interrupt (interrupt 66h). Include the file **BUSMGR.INC**, which contains a set of data definitions needed to call Bus Management for DOS functions, in your assembly language program.

1.3.3 Compiling and Linking Applications

NOTE: For specific compiler and/or linker options, refer to your compiler's documentation.

The following examples assume that EPConnect software has been installed in the **C:\EPCONNEC** directory.

Compiling and Linking C/C++ Applications

When compiling Bus Management for DOS applications, ensure that the Bus Management for DOS header files are in the compiler search path by doing one of the following:

1. Specify the entire header file pathname when including the header file in the source file.

2. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path at compiler invocation time.

3. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path environment variable.

Also, ensure that Bus Management for DOS libraries are in the linker search path by doing one of the following:

1. Specify the entire library pathname when linking object files.

2. Specify **C:\EPCONNEC\LIB** as part of the linker library search path.

Compiling and Linking MS BASIC Applications

When compiling Bus Management for DOS BASIC applications, ensure that the **BMBLIB.BI** header file is in the compiler search path by doing one of the following:

1. Specify the entire header file pathname when including the header file in the source file.

2. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path at compiler invocation time.

3. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path environment variable INCLUDE.

Also, ensure that Bus Management for DOS libraries are in the linker search path by doing one of the following:

1. Specify the entire library pathname when linking object files.

2. Specify **C:\EPCONNEC\LIB** as part of the linker library search path.

1.4 What to do Next

1. If Bus Management for DOS software is not pre-installed on your system, install and configure your system using the procedures in Chapter 2 of the *EPConnect/VXI for DOS & Windows User's Guide*.

2. Refer to the error messages in Chapter 5 of this manual for corrective action information about device driver installation errors.

3. Refer to the function descriptions in Chapter 2 of this manual for details about a function and/or its parameters to develop applications.

4. Refer to the sample programs included with EPConnect software under the **C:\EPCONNEC\SAMPLES\BUSMGR.DOS** directory.

2. Function Descriptions

2

2.1 Introduction

This chapter lists the Bus Management for DOS functions by category and by name. It is for the programmer who needs a particular fact, such as what function performs a specific task or what a function's arguments are.

The first section lists the functions categorically by the task each performs. It also gives you a brief description of what each function does. The second section lists the functions alphabetically and describes each function in detail.

2.2 Functions by Category

The categorical listing provides an overview of the operations performed by the EPConnect functions. Included with each category is a description of the operations performed, a listing of the functions in the category, and a brief description of each function.

The categories of the Bus Management for DOS library functions include:

- Bus Access
- Byte-Swapping
- Block Copy
- Interrupt and Error Handling
- Bus Control
- Commander Functionality
- Servant Functionality
- Event/Response Functions
- Other Functions

2

2.2.1 Bus Access Functions

Bus Access functions allow Bus Management applications to access VXIbus registers and VMEbus memory. Bus Access functions include the following:

EpcGetAccMode	Queries the current bus access mode.
EpcGetAmMap	Queries the current access mode and bus window base address.
EpcMapBus	Maps the bus window onto the VMEbus.
EpcRestState	Restores an access mode and a bus window base that were previously saved by a call to **EpcSaveState**.
EpcSaveState	Preserves the current access mode and bus window in a caller-supplied area.
EpcSetAccMode	Defines the current bus access mode.
EpcSetAmMap	Defines the bus access mode and bus window base.

2.2.2 Byte-Swapping Functions

Byte-swapping functions convert data from Intel (80x86) format to Motorola (68xxx) format and vice versa. Byte-swapping functions include the following:

EpcMemSwapL	Byte-swaps an array of 32-bit values.
EpcMemSwapW	Byte-swaps an array of 16-bit values.
EpcSwapL	Byte-swaps a single 32-bit value.
EpcSwapW	Byte-swaps a single 16-bit value.

2.2.3 Block Copy Functions

The block copy functions efficiently copy blocks of memory between EPC memory and VMEbus memory.

Block Copy functions include the following:

EpcFromVme	Copies consecutive VMEbus locations to consecutive EPC locations using the current access mode.
EpcFromVmeAm	Copies consecutive VMEbus locations to consecutive EPC locations using the specified access mode.
EpcToVme	Copies consecutive EPC locations to consecutive VMEbus locations using the current access mode.
EpcToVmeAm	Copies consecutive EPC locations to consecutive VMEbus locations using the specified access mode.

2.2.4 Interrupt and Error Handling Functions

A handler is a subroutine that is called when an interrupt or error occurs. This comparatively low-level passing of control requires that the handler obey some rather strict rules, but it allows quick response to other devices. Refer to Chapter 4, *Advanced Topics*, for more information about interrupt and error handling.

Interrupt and error handling functions include the following:

EpcCkIntr	Queries the VMEbus interrupt being asserted by this EPC.
EpcDisErr	Disables a specified error without affecting handler assignment.
EpcDisIntr	Disables a specified interrupt without affecting handler assignment.
EpcEnErr	Enables a specified error without affecting handler assignment.
EpcEnIntr	Enables a specified interrupt without affecting handler assignment.
EpcGetError	Queries a specified error's current handler function and stack.
EpcGetIntr	Queries an interrupt's current handler function and stack.
EpcSetError	Defines a specified error's handler function and stack.
EpcSetIntr	Defines a specified interrupt's handler function and stack.
EpcSigIntr	Signals (asserts or deasserts) a VMEbus interrupt.
EpcWaitIntr	Waits for an interrupt to occur.

2.2.5 Bus Control Functions

Bus control functions give applications access to EPC and VXIbus control and configuration parameters. Bus Control functions include the following:

EpcGetSlaveAddr	Queries the current address space and base address of the EPC's slave memory.
EpcGetSlaveBase	Queries the current base address of the EPC's slave memory.
EpcGetUla	Queries the unique logical address (ULA) of the EPC.
EpcSetSlaveAddr	Defines the current address space and base address of the EPC's slave memory.
EpcSetSlaveBase	Defines the current base address of the EPC's slave memory.
EpcSetUla	Defines the unique logical address (ULA) of the EPC.
EpcVmeCtrl	Queries or defines VMEbus interface control bits.
EpcVxiCtrl	Queries or defines VXIbus interface control bits.

2.2.6 Commander Functionality

2

Commander functions control the EPC's message registers. When two devices on the system communicate directly, one device is the *commander* and the other device is the *servant*. A device may be the commander to any number of servants, but each device may be a servant to only one commander. At the root of this tree there is one device that has no commander, only zero or more servants. This device is called the *top-level commander*.

Commander functions include the following:

EpcElwsCmd	Sends an extended longword serial command.
EpcLwsCmd	Sends a longword serial command.
EpcWsCmd	Sends a word serial command.
EpcWsRcvStr	Receives a series of bytes.
EpcWsSndStr	Sends a series of bytes, setting the END bit on the last byte.
EpcWsSndStrNe	Sends a series of bytes without setting the END bit on the last byte.
EpcWsStat	Returns the word-serial status of a device.

2.2.7 Event/Response Functions

VXIbus *events* and *responses* (collectively called *E/Rs*) get special handling. They arrive either in the signal register or as the Status/ID returned in response to an interrupt acknowledge for a VMEbus interrupt. All E/Rs are queued, to preserve the sequence of responses and events.

When a value is placed in the signal register, the signal FIFO is emptied into the BusManager-maintained E/R queue. The BusManager uses the hardware signal interrupt internally to maintain this queue. VMEbus interrupts may be designated as sources of events and responses so that the Status/IDs returned in response to interrupt acknowledges are recognized as E/Rs and placed in the E/R queue as well.

Event and Response functions include the following:

EpcErGet	Dequeues and returns the oldest event/response.
EpcErQue	Queues the supplied value as the newest element in the event/response queue.
EpcErRedir	Assigns a VMEbus interrupt as a VXIbus event/response interrupt.
EpcErUnredir	De-assigns a VMEbus interrupt as a VXIbus event/response interrupt.

2.2.8 Servant Functionality

EPConnect provides support for using an EPC as a message-based servant device in a VXIbus system. This functionality is specific to the VMEbus extension for instrumentation (VXI) and is not supported by most VMEbus modules.

Servant functions include the following:

EpcWsServArm	Arms the EPC so that it can receive a command.
EpcWsServPeek	Waits for a command to arrive without removing the incoming command.
EpcWsServRcv	Waits for a command to arrive and receives the incoming command.
EpcWsServSend	Sends a response to the EPC's commander.
EpcErServIntr	Sends an event/response to a commander using a VMEbus interrupt.
EpcErServSig	Sends an event/response to a commander using a VXIbus signal.

2.2.9 Other Functions

This section describes functions that allow you to get information about the version of the BusManager software, the EPC hardware, and the BIOS. A function that indicates whether the BusManager device driver is currently loaded in the system and a function to obtain descriptive error strings are also provided.

"Other" functions include the following:

EpcBiosVer	Queries the BIOS version number.
EpcBmVer	Queries the BusManager software version number.
EpcCkBm	Determines whether the BusManager software is currently loaded.
EpcErrStr	Returns a string describing the specified BusManager error:
EpcHwVer	Queries the EPC's hardware version number.

2.3 Functions By Name

2

This section contains an alphabetical listing of the BusManager library functions. Each listing describes the function, gives its invocation sequence and arguments, discusses its operation, and lists its returned values.

Each Bus Management program should call **EpcCkBm** once, and test for **EPC_SUCCESS** to verify that the BusManager is operational.

EpcBiosVer

Description Queries the BIOS version number.

2

C Synopsis

short FAR PASCAL
EpcBiosVer(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcBiosVer%**
biosversion% = **EpcBiosVer%**

Remarks This function returns the version number of the EPC BIOS. The BIOS version number consists of the major and minor version numbers of the BIOS that is installed in the EPC. The BIOS version number is returned with the major version number in the high-order byte and the minor version number in the low-order byte.

See Also **EpcBmVer, EpcCkBm, EpcHwVer.**

EpcBmVer

Description Queries the Bus Manager for DOS software version number.

C Synopsis

short FAR PASCAL
EpcBmVer(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcBmVer%**
bmversion% = **EpcBmVer%**

Remarks The function returns the version number of the Bus Manager for DOS software. The Bus Manager for DOS version number consists of a major version and minor version number assigned to the Bus Manager software running on the EPC. The Bus Manager version number is returned with the major version number in the high-order byte and the minor version number in the low-order byte.

See Also **EpcBiosVer, EpcCkBm, EpcHwVer .**

EpcCkBm

Description Determines whether the Bus Manager for DOS software is currently loaded.

C Synopsis

short FAR PASCAL
EpcCkBm(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcCkBm%**
ok% = **EpcCkBm%**

Remarks The function determines whether the BusManager driver is installed in the system, is in operation, and is able to communicate with the calling application.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	The library was unable to access the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcBiosVer, EpcBmVer, EpcHwVer.**

EpcCkIntr

2

Description Queries the VMEbus interrupt being asserted by this EPC.

C Synopsis

short FAR PASCAL
EpcCkIntr(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcCkIntr%**
interrupt% = **EpcCkIntr%**

Remarks This function returns the number of the VMEbus interrupt being asserted by this EPC. If no interrupt is being asserted (that is, if the last interrupt has been acknowledged) then zero is returned. Interrupt acknowledgment is simply a hardware handshake and not an indication that the remote interrupt handling code has been executed.

Return Value The following return values are supported:

Constant	Description
0	No VMEbus interrupts are asserted.
BM_VME_INTR1	The EPC is currently asserting VMEbus interrupt 1.
...	
BM_VME_INTR7	The EPC is currently asserting VMEbus interrupt 7.

See Also **EpcSigIntr.**

2

EpcDisErr

Description Disables a specified error without affecting handler assignment.

C Synopsis

short FAR PASCAL
EpcDisErr(short *error***);**

error	Error number

MS BASIC Synopsis

DECLARE FUNCTION **EpcDisErr%**(BYVAL *error%*)
ok% = **EpcDisErr%**(*error%*)

Remarks The function disables the specified *error* without affecting the handler assignment. If the specified *error* condition occurs, the associated handler is not called. Use **EpcEnErr** to enable a disabled *error*.

The parameter *error* specifies the error condition to disable. The following constants define valid values for *error*:

Constant	Description
BM_SYSFAIL_ERR	SYSFAIL assertion.
BM_BERR_ERR	VMEbus BERR.
BM_ACFAIL_ERR	ACFAIL assertion.
BM_WATCHDOG_ERR	Watchdog timer expiration.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	The library was unable to access the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcEnErr, EpcGetError, EpcSetError.**

EpcDisIntr

Description Disables a specified interrupt without affecting handler assignment.

C Synopsis

short FAR PASCAL
EpcDisIntr(short *interrupt*);

interrupt	Interrupt number.

MS BASIC Synopsis

DECLARE FUNCTION **EpcDisIntr**%(BYVAL *interrupt%*)
ok% = **EpcDisIntr**%(*interrupt%*)

Remarks The parameter *interrupt* specifies the interrupt condition to disable. The following constants define valid values for *interrupt*:

Constant	Description
BM_MSG_INTR	Message interrupt.
BM_VME_INTR1	VMEbus interrupt 1.
...	
BM_VME_INTR7	VMEbus interrupt 7.
BM_ER_INTR	Event/Response interrupt.
BM_TTLTRG0_INTR	TTL trigger interrupt 0 (EPC-7 only).
...	
BM_TTLTRG7_INTR	TTL trigger interrupt 7 (EPC-7 only).

The function is used to temporarily mask off an interrupt. Use **EpcEnIntr** to enable a disabled interrupt.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	The library was unable to access the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcEnIntr, EpcGetIntr, EpcSetIntr, EpcWaitIntr.**

EpcEnErr

Description Enables a specified error without affecting handler assignment.

C Synopsis

short FAR PASCAL
EpcEnErr(short *error*);

error	Error number.

MS BASIC Synopsis

DECLARE FUNCTION **EpcEnErr**%(BYVAL *error*%)
ok% = **EpcEnErr**%(*error*%)

Remarks The parameter *error* specifies the error condition to enable. The following constants define valid values for error:

Constant	Description
BM_SYSFAIL_ERR	SYSFAIL assertion.
BM_BERR_ERR	Bus error (BERR).
BM_ACFAIL_ERR	ACFAIL assertion.
BM_WATCHDOG_ERR	Watchdog timer expiration.

The function enables reception of an error condition. **EpcEnErr** should only be used to reverse the effect of a previous **EpcDisErr**, because no check is made to make sure a handler is assigned to the specified error. If no handler is assigned for the specified error, the error is associated with a default handler. This default handler disables the error when it occurs.

EpcEnErr enables the specified error unconditionally -- there is no nesting of **EpcDisErr/EpcEnErr** pairs.

Calling **EpcSetError** to assign a handler to an error immediately enables the specified error, and a call to **EpcEnErr** is unnecessary.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcDisErr, EpcGetError, EpcSetError.**

2

EpcEnIntr

Description Enables a specified interrupt without affecting handler assignment.

C Synopsis

short FAR PASCAL
EpcEnIntr(short *interrupt*);

interrupt	Interrupt number.

MS BASIC Synopsis

DECLARE FUNCTION **EpcEnIntr**%(BYVAL *interrupt%*)

ok% = **EpcEnIntr**%(*interrupt%*)

Remarks The parameter *interrupt* specifies the interrupt condition to enable. The following constants define valid values for *interrupt*:

Constant	Description
BM_MSG_INTR	Message interrupt.
BM_VME_INTR1	VMEbus interrupt 1.
...	
BM_VME_INTR7	VMEbus interrupt 7.
BM_ER_INTR	Event/Response interrupt.
BM_TTLTRG0_INTR	TTL trigger interrupt 0 (EPC-7 only).
...	
BM_TTLTRG7_INTR	TTL trigger interrupt 7 (EPC-7 only).

The function enables reception of an interrupt condition. **EpcEnIntr** function should only be used in conjunction with **EpcDisIntr**, because no check is made to make sure a handler is assigned to the specified interrupt.

EpcEnIntr enables the specified interrupt unconditionally - there is no "nesting" of **EpcDisIntr/EpcEnIntr** pairs.

Calling **EpcSetIntr** to assign a handler to a bus interrupt immediately enables the specified interrupt; a call to **EpcEnIntr** is unnecessary.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcDisIntr, EpcGetIntr, EpcSetIntr, EpcWaitIntr.**

2

EpcErGet

Description Dequeues and returns the oldest event/response.

C Synopsis

short FAR PASCAL
EpcErGet(unsigned short FAR * *er_pointer*);

er_pointer	Location where the dequeued event/response will be placed..

MS BASIC Synopsis

NONE

Remarks This function dequeues and returns the oldest event/response. If the returned value is the last entry in the queue, the E/R interrupt is deasserted.

Return Value This function returns **TRUE** if the queue is non-empty.

See Also **EpcErRedir, EpcErQue, EpcErUnredir.**

EpcErQue

2

Description Queues the supplied value as the newest element in the event/response queue.

C Synopsis

short FAR PASCAL
EpcErQue(unsigned short *er*);

er	Event/response value to be queued.

MS BASIC Synopsis

NONE

Remarks This function queues *er* as the newest element in the event/response queue. The E/R interrupt is asserted (since the queue is now non-empty). If the handler is installed for the E/R interrupt and the E/R interrupt is enabled, the installed handler will be called before this function returns.

Return Value This function returns **FALSE** if the queue is full.

See Also **EpcGetError.**

EpcErRedir

2

Description Assigns a VMEbus interrupt as a VXIbus interrupt.

C Synopsis

short FAR PASCAL
EpcErRedir(short *interrupt*);

interrupt VMEbus interrupt from which to redirect E/Rs

MS BASIC Synopsis

DECLARE FUNCTION **EpcErRedir%**(BYVAL *interrupt%*)

ok% = **EpcErRedir%**(*interrupt%*)

Remarks This function allows a commander to redirect the designated *interrupt* as a source for receipt of events and responses from servants.

The following constants define valid values for *interrupt*:

Constant	Description
BM_VME_INTR1	VMEbus interrupt 1.
....	
BM_VME_INTR7	VMEbus interrupt 7.

When an interrupt is redirected, the interrupt is enabled.

At system restart no interrupts are redirected. Any number of VMEbus interrupts may be redirected.

There must be a redirected interrupt any time there is a slave-only VXIbus interrupter device, because slave-only devices cannot write to the signal register and must then communicate using interrupts.

An interrupt may not both be redirected and have a handler assigned to it; if it does, **ERR_FAIL** is returned.

After a redirected interrupt is asserted and acknowledged, the low 16 bits of the returned Status/ID are placed in the E/R queue. An E/R interrupt is then asserted (because the queue is no longer empty).

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcErGet, EpcErUnredir.**

2

EpcErServIntr

Description Sends an event/response to a commander using a VMEbus interrupt.

C Synopsis

short FAR PASCAL
EpcErServIntr(short *interrupt*, **unsigned short** *er*);

interrupt	VMEbus interrupt to assert to send the event/response.
er	Event/response value to send.

MS BASIC Synopsis

DECLARE FUNCTION **EpcErServIntr**%(BYVAL *interrupt%*, BYVAL *er%*)

ok% = **EpcErServIntr**%(*interrupt%*, *er%*)

Remarks Sends an event/response to a commander device using a VMEbus interrupt. This function is used to implement a VXIbus servant interface on the EPC.

The following constants define valid values for *interrupt*:

Constant	Description
BM_VME_INTR1	VMEbus interrupt 1(EPC-2 and EPC-7 only).
....	
BM_VME_INTR7	VMEbus interrupt 7 (EPC-2 and EPC-7 only).

If a word serial command from the commander is present in the EPC's message register, that command is saved before the register is used. If the register contains outgoing data, this function waits until the commander has read the data before signaling the interrupt.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also EpcErServSig.

EpcErServSig

Description Sends an event/response to a commander using a VXIbus signal.

C Synopsis

short FAR PASCAL
EpcErServSig(unsigned short *ula,* **unsigned short** *er*);

ula ULA of the commander to which the signal is sent.

er Event/response value to send.

MS BASIC Synopsis

DECLARE FUNCTION **EpcErServSig%**(BYVAL *ula%*, BYVAL *er%*)

ok% = **EpcErServSig%**(*ula%*, *er%*)

Remarks Signals the EPC's commander by placing a value in the commander's signal register. This function is used in implementing a VXIbus servant interface on the EPC.

Return Value The following return values are supported:

Constant	Description
ERR_BERR	Commander has no signal register, or its signal queue is full.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcErServIntr.**

EpcErUnredir

Description Deassigns a VMEbus interrupt as a VXIbus Event/Response interrupt.

C Synopsis

short FAR PASCAL
EpcErUnredir(short *interrupt*);

interrupt	VMEbus interrupt from which to stop redirecting ERs

MS BASIC Synopsis

DECLARE FUNCTION **EpcErUnredir**%(BYVAL *interrupt%*)

ok% = **EpcErUnredir**%(*interrupt%*)

Remarks This function deassigns *interrupt* as a VXIbus Event/Response interrupt and makes it available as a regular VMEbus interrupt.

The following constants define valid values for *interrupt*:

Constant	Description
BM_VME_INTR1	VMEbus interrupt 1.
....	
BM_VME_INTR7	VMEbus interrupt 7.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while attempting to unredirect an interrupt that is not redirected.
EPC_SUCCESS	Successful function completion.

See Also **EpcErGet, EpcErRedir.**

2

EpcErrStr

Description Queries a string describing a specified BusManager error.

C Synopsis

char FAR * FAR PASCAL
EpcErrStr(int *retcode***);**

retcode BusManager return value.

MS BASIC Synopsis

NONE

Remarks The function returns a pointer to a string describing the BusManager return value *retcode*:

```
short retcode;
if ((retcode = EpcCkBm() !=EPC_SUCCESS)  {
   printf("Error: %\n", EpcErrStr(retcode));
   exit(1);
}
```

Return Value NONE

See Also **EpcCkBm.**

EpcElwsCmd

Description Sends an extended longword serial command.

C Synopsis

short FAR PASCAL
EpcElwsCmd(unsigned short *ula*, **unsigned short FAR*** *command*, **unsigned short** *wait*);

ula	Servant's unique logical address.
command	Command to send.
wait	Timeout, in milliseconds.

MS BASIC Synopsis

DECLARE FUNCTION **EpcElwsCmd%**(BYVAL *ula%*, SEG *cmd%*, BYVAL *wait%*)

DIM *cmd%*[3]

ok% = **EpcElwsCmd%**(*ula%*, *cmd%*, *wait%*)

Remarks Send one extended longword serial command. A command will be sent only when the servant device's WRDY bit is set.

Note: Extended longword serial commands do not generate a reply.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_BERR	A bus error occurred sending a word serial command.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_RBERR	A bus error occurred receiving a word serial command response.
ERR_RTIMEOUT	A timeout occurred receiving a word serial command response.
ERR_TIMEOUT	A timeout occurred sending a word serial command.
ERR_WS	A word serial protocol error occurred.

See Also **EpcLwsCmd, EpcWsCmd.**

EpcFromVme

2

Description Copies data from consecutive VMEbus locations to consecutive EPC locations using the current access mode.

C Synopsis

unsigned short FAR PASCAL
EpcFromVme(short *width,* **unsigned long** *source,* **char FAR** **dest,* **unsigned short** *count);*

width	Number of data bits to copy per bus access.
source	Source address on the VMEbus.
dest	Destination address in EPC memory.
count	Number of bytes to transfer.

MS BASIC Synopsis

DECLARE FUNCTION **EpcFromVme**%(BYVAL *width%*,
BYVAL *source&*, SEG *dest%*, BYVAL count%)

DIM *source%*[...]

ok% = **EpcFromVme**%(*width%*, *source&*, *dest%*, *count%*)

Remarks This function copies data from consecutive VMEbus locations to consecutive EPC locations using the current access mode. The current access mode is the address modifier and byte order set by the most recent **EpcRestState** or **EpcSetAmMap** call. The bus window is saved, altered as necessary during the copy, and restored upon completion of the copy. This function is intended for use in transferring large amounts of data to consecutive locations.

The *count* parameter should always express the number of bytes to be transferred regardless of the copy width specified. Setting *count* to zero specifies a transfer of zero bytes and nothing is transferred.

2

The *width* parameter specifies whether data is to be moved in 8-bit, 16-bit, or 32-bit chunks. Transfers are always aligned on natural boundary; 16-bit quantities are written to the VMEbus only at even addresses, and 32-bit quantities are written to the VMEbus only at addresses evenly divisible by 4.

Valid values for the *width* parameter are as follows:

Constant	Description
BM_W8	8-bit copy width
BM_W8O	8-bit copy width, odd-only copy
BM_W16	16-bit copy width
BM_W32	32-bit copy width
BM_FASTCOPY	Don't check for intermediate bus errors. This constant can be OR'd with one of the previous constants to increase copy speed.

Transfers to non-aligned locations are done in a read-modify-write fashion – a chunk is read from the destination, the bytes to be transferred are copied to the corresponding bytes in the chunk, and the chunk is replaced. For example, a copy of 32-bit chunks to a non-aligned address would occur in the following manner. The leading 32-bit word would be read from the destination, modified, and written back. Next, all whole (aligned) 32-bit values would be transferred. Finally, the trailing 32-bit word would be read from the destination, modified, and replaced.

Notes:

- This "read-modify-write" sequence is done in software, and is *not* a RMW bus cycle.
- If an unmodified byte in the leading or trailing word of a non-aligned transfer contains a semaphore that is signaled while the copy is taking place, the signal may be lost.

When you specify 8-bit, odd-only transfers (**BM_W8O**), the VMEbus address "spins" twice as fast as the EPC address. That is, for `i = 0` to `(count - 1), dest + i` receives `src + (i × 2) + 1`.

By default, BERR is checked after every transfer. If there is an error, the copy is aborted but the BERR error handler is not called. This eliminates the requirement that the calling program coordinate with the BERR handler. Errors are reflected by a non-zero return value.

If you OR the *width* parameter with **BM_FASTCOPY** before calling the copy function, BERR is checked only after transfers to nonaligned locations. Fast copying uses "Move String" instructions to quickly copy blocks of data. By taking advantage of pipelining in the processor and the VMEbus interface hardware, fast copy transfers are five times faster than transfers without **BM_FASTCOPY**. There are risks, however: a BERR may go undetected, or the BERR error handler may be called erroneously (if a transfer – still in the pipeline when the function returns – causes a BERR). Generally you should select the fast copy option.

BM_FASTCOPY is ignored when you specify 8-bit, odd-only transfers (**BM_W8O**).

2

Return Value The following return values are supported:

Constant	Description
ERR_BERR	The function returns the number of bytes not transferred.
EPC_SUCCESS	Successful function completion.

See Also **EpcFromVmeAm, EpcRestState, EpcSetAmMap, EpcToVme, EpcToVmeAm.**

EpcFromVmeAm

2

Description Copies consecutive VMEbus locations to consecutive EPC locations using the specified access mode.

C Synopsis

unsigned short FAR PASCAL
EpcFromVmeAm(short *mode,* **short** *width,* **unsigned long** *source,*
char FAR **dest,* **unsigned short** *count)***;**

mode	Access mode.
width	Number of data bits to copy per bus access.
source	Source address on the VMEbus.
dest	Destination address in EPC memory.
count	Number of bytes to transfer.

MS BASIC Synopsis

DECLARE FUNCTION **EpcFromVmeAm**%(BYVAL *mode%*,
BYVAL *width%*, BYVAL *source&*, SEG *dest%*,
BYVAL *count%*)

DIM src%[...]

ok% = **EpcFromVmeAm**%(*mode%*, *width%*, *source&*, *dest%*,
count%)

Remarks This function copies data from consecutive VMEbus locations to consecutive EPC locations using the specified access mode. The current access mode and bus window are saved, altered as specified during the copy, and restored upon completion of the copy.

The parameter *mode* is an OR'd combination of a byte order constant and an address modifier constant.

The returned access mode is an OR'd combination of a byte order constant and an address modifier constant:

0	0	0	0	0	0	0	*byte order*	0	0	*addrmod*

The following constants are valid byte order constants:

Constant	Description
BM_IBO	Little-endian (Intel 386-style) byte order
BM_MBO	Big-endian (Motorola 68000-style) byte order

The following constants define valid address modifier constants:

Constant	Description
A16N	A16 non privileged address modifier
A16S	A16 supervisor address modifier
A24ND	A24 non privileged data address modifier
A24NP	A24 non privileged program address modifier
A24SD	A24 supervisor data address modifier
A24SP	A24 supervisor program address modifier
A32ND	A32 non privileged data address modifier
A32NP	A32 non privileged program address modifier
A32SD	A32 supervisor data address modifier
A32SP	A32 supervisor program address modifier

The *width* parameter specifies whether data is to be moved in 8-bit, 16-bit, or 32-bit chunks. VMEbus transfers are always aligned on natural boundary; 16-bit quantities are written to the VMEbus only at even addresses, and 32-bit quantities are written to the VMEbus only at addresses evenly divisible by 4.

Valid values for the *width* parameter are defined as follows:

Constant	Description
BM_W8	8-bit copy width
BM_W8O	8-bit copy width, odd-only copy
BM_W16	16-bit copy width
BM_W32	32-bit copy width
BM_FASTCOPY	Don't check for intermediate bus errors. This constant can be OR'd with one of the previous constants to increase copy speed.

Transfers to non-aligned locations are done in a read-modify-write fashion – a chunk is read from the destination, the bytes to be transferred are copied to the corresponding bytes in the chunk, and the chunk is replaced. For example, a copy of 32-bit chunks to a non-aligned address would occur in the following manner. The leading 32-bit word would be read from the destination, modified, and written back. Next, all whole (aligned) 32-bit values would be transferred. Finally, the trailing 32-bit word would be read from the destination, modified, and replaced.

Notes:

- This "read-modify-write" sequence is done in software, and is *not* a RMW bus cycle.
- If an unmodified byte in the leading or trailing word of a non-aligned transfer contains a semaphore that is signaled while the copy is taking place, the signal may be lost.

When you specify 8-bit, odd-only transfers (**BM_W8O**), the VMEbus address "spins" twice as fast as the EPC address. That is, for `i = 0 to (count - 1), dest + i` receives `src + (i × 2) + 1`.

By default, BERR is checked after every transfer. If there is an error, the copy is aborted but the BERR error handler is not called. This eliminates the requirement that the calling program coordinate with the BERR handler. Errors are reflected by a non-zero return value.

If you OR the *width* with **BM_FASTCOPY** before calling the copy function, BERR is checked only after transfers to nonaligned locations. Fast copying uses "Move String" instructions to move "blocks" of data. By taking advantage of pipelining in the processor and the VMEbus interface hardware, fast copy transfers are five times faster than transfers without **BM_FASTCOPY**. There are risks, however: a BERR may go undetected, or the BERR error handler may be called erroneously (if a transfer – still in the pipeline when the function returns – causes a BERR). Generally, however, you should select the fast copy option.

The Fast Copy flag (**BM_FASTCOPY**) is ignored when you specify 8-bit, odd-only transfers (**BM_W8O**).

Return Value

The function returns **EPC_SUCCESS** on successful completion. Otherwise, the function returns the number of bytes ***not*** transferred. This indicates there was a VMEbus error (BERR).

See Also

EpcFromVme, EpcToVme, EpcToVmeAm.

EpcGetAccMode

Description Queries the current bus access mode.

C Synopsis

short FAR PASCAL
EpcGetAccMode(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcGetAccMode%**
oldmode% = **EpcGetAccMode%**

Remarks The function returns the EPC's current access mode.

The returned access mode is an OR'd combination of a byte order constant and an address modifier constant:

0	0	0	0	0	0	0	*byte order*	0	0	*addrmod*

The following constants are valid byte order constants:

Constant	Description
BM_IBO	Little-endian (Intel 386-style) byte order
BM_MBO	Big-endian (Motorola 68000-style) byte order

2

The following constants define valid address modifier constants:

Constant	Description
A16N	A16 non privileged address modifier
A16S	A16 supervisor address modifier
A24ND	A24 non privileged data address modifier
A24NP	A24 non privileged program address modifier
A24SD	A24 supervisor data address modifier
A24SP	A24 supervisor program address modifier
A32ND	A32 non privileged data address modifier
A32NP	A32 non privileged program address modifier
A32SD	A32 supervisor data address modifier
A32SP	A32 supervisor program address modifier

Although still supported, **EpcGetAccMode** functionality has been superseded by **EpcGetAmMap**.

Return Value

If successful, the function returns the bus' current access mode. Otherwise, the function returns **ERR_FAIL**.

See Also

EpcGetAmMap, EpcRestState, EpcSaveState, EpcSetAccMode, EpcSetAmMap.

2

EpcGetAmMap

Description Queries the current access mode and bus window base address.

C Synopsis

short FAR PASCAL
EpcGetAmMap(unsigned short FAR **accessmode,* **unsigned long FAR** **busaddress);*

accessmode	Location where the current access mode will be placed.
busaddress	Location where the current bus window address will be placed.

MS BASIC Synopsis

DECLARE FUNCTION **EpcGetAmMap**%(SEG *accessmode%*, SEG *busaddress&*)

returncode% = **EpcGetAmMap**%(*accessmode%*, *busaddress&*)

Remarks The returned access mode is an OR'd combination of a byte order constant and an address modifier constant, as follows:

0	0	0	0	0	0	0	*byte order*	0	0	*addrmod*

The following constants are valid byte order constants:

Constant	Description
BM_IBO	Little-endian (Intel 386-style) byte order
BM_MBO	Big-endian (Motorola 68000-style) byte order

The following constants define valid address modifier constants:

Constant	Description
A16N	A16 non privileged address modifier
A16S	A16 supervisor address modifier
A24ND	A24 non privileged data address modifier
A24NP	A24 non privileged program address modifier
A24SD	A24 supervisor data address modifier
A24SP	A24 supervisor program address modifier
A32ND	A32 non privileged data address modifier
A32NP	A32 non privileged program address modifier
A32SD	A32 supervisor data address modifier
A32SP	A32 supervisor program address modifier

Return Value

The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.

See Also

EpcGetAccMode, EpcMapBus, EpcRestState, EpcSaveState, EpcSetAccMode, EpcSetAmMap.

EpcGetError

Description Queries a specified error's current handler function and stack.

2

C Synopsis

void (FAR CDECL * FAR PASCAL
EpcGetError(short *error,* **char FAR * FAR *** *stack*)**(unsigned**
long *error)*;

error	Error number.
stack	Location where a pointer to the current stack will be placed.

MS BASIC Synopsis

NONE

Remarks The function returns the addresses of the specified *error*'s current handler function and stack.

The following constants define valid values for *error*:

Constant	Description
BM_SYSFAIL_ERR	SYSFAIL assertion.
BM_BERR_ERR	VMEbus BERR.
BM_ACFAIL_ERR	ACFAIL assertion.
BM_WATCHDOG_ERR	Watchdog timer expiration.

An error handler function has the following calling semantics:

void FAR CDECL
error_handler (**unsigned long** *error*);

If *stack* is NULL, the current stack pointer is not returned.

Return Value If successful, the function returns the address of the current error handler. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcDisErr, EpcEnErr, EpcSetError.**

EpcGetIntr

Description Queries a specified interrupt's current handler function and stack.

C Synopsis

void (FAR CDECL * FAR PASCAL EpcGetIntr(short *interrupt,* **char FAR * FAR** *stack))(***unsigned long** *data);*

interrupt	Interrupt number.
stack	Location where a pointer to the current stack will be placed.

MS BASIC Synopsis

NONE

Remarks The function returns the addresses of the specified *interrupt*'s current handler function and stack.

The following constants define valid values for *interrupt*:

Constant	Description
BM_MSG_INTR	Message interrupt.
BM_VME_INTR1	VMEbus interrupt 1.
...	
BM_VME_INTR7	VMEbus interrupt 7.
BM_ER_INTR	Event/Response interrupt.
BM_TTLTRG0_INTR	TTL trigger interrupt 0 (EPC-7 only).
...	
BM_TTLTRG7_INTR	TTL trigger interrupt 7 (EPC-7 only).

An interrupt handler function has the following calling semantics:

void FAR CDECL
interrupt_handler (**unsigned long** *data*);

If *stack* is NULL, the current stack pointer is not returned.

Return Value If successful, the function returns the address of the current interrupt handler. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcDisIntr, EpcEnIntr, EpcSetIntr, EpcWaitIntr.**

2

EpcGetSlaveAddr

Description Queries the current address space and base address of the EPC's slave memory.

C Synopsis

short FAR PASCAL
EpcGetSlaveAddr(unsigned short FAR **addrspace*, **unsigned long FAR** **slavebase*);

addrspace	Pointer to a location where the current address space will be placed.
slavebase	Pointer to a location where the current base address will be placed.

MS BASIC Synopsis

DECLARE FUNCTION **EpcGetSlaveAddr**%(SEG *addrspaceptr%*, SEG *slavebaseptr&*)

returncode% = **EpcGetSlaveAddr**%(*addrspace%*, *slavebase&*)

Remarks The slave memory base address defines where the EPC's slave memory appears on the VMEbus (if it is enabled). Return values for the variables **slavebase* and **addrspace* are as follows:

EPC type	*slavebase	*addr space
EPC-2	0x18000000, 0x19000000, ... , 0x1F000000	BM_A32
	EPC_SLAVE_MEMORY_DISABLED	N/A
EPC-7	0x000000, 0x400000, ..., 0xC00000	BM_A24
	0x00000000, 0x01000000, ..., 0xFF000000	BM_A32
	EPC_SLAVE_MEMORY_DISABLED	N/A
EPC-8	EPC_SLAVE_MEMORY_DISABLED	N/A

A24 base addresses are aligned on a 4 MByte boundary, and only the first 4 MBytes of the EPC's slave memory is mapped to the bus. A32 base addresses are aligned on a 16 MByte boundary, and only the first 16 MBytes of the EPC's slave memory is mapped to the bus.

If the EPC's slave memory is disabled, a slave memory base address of **EPC_SLAVE_MEMORY_DISABLED** is returned.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcGetSlaveBase, EpcSetSlaveAddr, EpcSetSlaveBase.**

EpcGetSlaveBase

Description Queries the current base address of the EPC's slave memory.

C Synopsis

unsigned long FAR PASCAL
EpcGetSlaveBase(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcGetSlaveBase&**

slavebase& = **EpcGetSlaveBase&**

Remarks The slave base address for each EPC type and address space supported is one of the following:

EPC type	Slave Base	Address Space
EPC-2	0x18000000, 0x19000000, ... , 0x1F000000	**BM_A32**
	EPC_SLAVE_MEMORY_DISABLED	**N/A**
EPC-7	0x000000, 0x400000, ..., 0xC00000	**BM_A24**
	0x00000000, 0x01000000, ..., 0xFF000000	**BM_A32**
	EPC_SLAVE_MEMORY_DISABLED	**N/A**
EPC-8	**EPC_SLAVE_MEMORY_DISABLED**	**N/A**

A24 base addresses are aligned on a 4 MByte boundary, and only the first 4 MBytes of the EPC's slave memory is mapped to the bus. A32 base addresses are aligned on a 16 MByte boundary, and only the first 16 MBytes of the EPC's slave memory is mapped to the bus.

If the EPC's slave memory is disabled, a slave memory base address of **EPC_SLAVE_MEMORY_DISABLED** is returned.

Return Value This function returns the current base address where the EPC memory appear on the VMEbus. The address space is not returned by this function. If not successful, the function returns **ERR_FAIL**.

See Also **EpcGetSlaveAddr, EpcSetSlaveAddr, EpcSetSlaveBase.**

EpcGetUla

Description Queries the unique logical address (ULA) of the EPC.

C Synopsis

short FAR PASCAL
EpcGetUla(void)

MS BASIC Synopsis

DECLARE FUNCTION **EpcGetUla%**

ula% = **EpcGetUla%**

Remarks The ULA is used to determine the base address of the VMEbus registers in A16 space, as follows:

```
A16_Address = (ULA<<6)+0xC000;
```

Return Value If successful, the function returns the EPC's current ULA. Otherwise, the function returns **ERR_FAIL.**

See Also **EpcSetUla.**

EpcHwVer

Description Queries the EPC hardware version number.

C Synopsis

short FAR PASCAL
EpcHwVer(void);

MS BASIC Synopsis

DECLARE FUNCTION **EpcHwVer%**
hwversion% = **EpcHwVer%**

Remarks The function returns the version number of the EPC hardware.

Return Value If successful, the function returns the version number of the EPC hardware. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcBiosVer, EpcBmVer, EpcCkBm.**

EpcLwsCmd

2

Description Sends a longword serial command.

C Synopsis

short FAR PASCAL EpcLwsCmd(unsigned short *ula*, **unsigned long** *command*, **unsigned long FAR** * *result_ptr*, **unsigned short** *wait*);

ula	Servant's unique logical address.
command	Command to send.
result_ptr	Address of result.
wait	Timeout, in milliseconds.

MS BASIC Synopsis

DECLARE FUNCTION **EpcLwsCmd%**(BYVAL *ula%*, BYVAL *cmd&*, SEG *result&*, BYVAL *wait%*)

ok% = **EpcLwsCmd%**(*ula%*, *cmd&*, *result&*, *wait%*)

DECLARE FUNCTION **EpcLwsCmdNr%**(BYVAL *ula%*, BYVAL *cmd&*, BYVAL *wait%*)

ok% = **EpcLwsCmdNr%**(*ula%*, *cmd&*, *wait%*)

Remarks Sends one longword serial command. A command will be sent only when the servant device's WRDY bit is set.

In the C interface, if *result_ptr* is non-NULL, the function waits for a result and returns it in the location pointed to by *result_ptr*.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

2

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_BERR	A bus error occurred sending a word serial command.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_RBERR	A bus error occurred receiving a word serial command response.
ERR_RTIMEOUT	A timeout occurred receiving a word serial command response.
ERR_TIMEOUT	A timeout occurred sending a word serial command.
ERR_WS	A word serial protocol error occurred.

See Also **EpcElwsCmd, EpcWsCmd.**

EpcMapBus

2

Description Maps the bus window onto the VMEbus.

C Synopsis

char FAR * FAR PASCAL
EpcMapBus(unsigned long *busaddr)***;**

busaddr Desired bus address.

MS BASIC Synopsis

DECLARE FUNCTION **EpcMapBus&**(BYVAL *busaddr&*)
Vmeptr& = **EpcMapBus&**(*busaddr&*)

DECLARE SUB **EpcMapBusB**(BYVAL *busaddr&*, SEG *busseg%*, SEG *busoff%*)
CALL **EpcMapBusB**(*busaddr&*, *busseg%*, *busoff%*)

Remarks This function is provided for compatibility with existing applications. **EpcSetAmMap** is the preferred method of mapping the bus.

Given a bus address, **EpcMapBus** sets the VMEbus mapping registers and returns a pointer to the bus window. Within the context of the current access mode, you can use this pointer to get to the bus. You must remap the bus, however, when an address range extends beyond the 64 KB-aligned bus window.

Because the bus window is 64 KB in size and aligned on a 64 KB boundary, the BusManager uses only the high-order 16 bits of the address to set the mapping. The low-order 16 bits are passed back to the caller unchanged. The segment portion of the return value is set to the physical location of the VMEbus window. It is not guaranteed that this implementation will be retained in future versions of the bus mapping hardware.

Return Value If successful, the function returns a pointer to the specified bus address. Otherwise, it returns a null pointer.

See Also **EpcGetAmMap, EpcRestState, EpcSaveState, EpcSetAmMap.**

EpcMemSwapL

Description Byte-swaps an array of 32-bit values.

C Synopsis

void FAR PASCAL
EpcMemSwapL(unsigned long FAR **buffer*, **unsigned short** *entrycount*);

buffer	Array of 32-bit elements to be swapped.
entrycount	Number of 32-bit elements in *buffer*.

MS BASIC Synopsis

DECLARE SUB **EpcMemSwapL**(SEG *buffer&*, BYVAL *entrycount%*)

CALL **EpcMemSwapL**(*buffer&*, *entrycount%*)

Remarks This function swaps the bytes in each 32-bit element in the buffer such that 32-bit values stored in Intel byte order are transformed to the Motorola byte order and vice versa.

For example, given:

```
    unsigned long value[] =
  {0x11223344L, 0x55667788L};
```

the following call:

```
    EpcMemSwapL(buffer, 2);
```

results in this output:

```
    value[0] = 0x44332211L
    value[1] = 0x88776655L
```

See Also **EpcMemSwapW, EpcSwapL, EpcSwapW.**

EpcMemSwapW

Description Byte-swaps an array of 16-bit values.

C Synopsis

void FAR PASCAL
EpcMemSwapW(unsigned short FAR ****buffer,* **unsigned short
*entrycount***);**

buffer	Array of 16-bit elements to be swapped.
entrycount	Number of 16-bit elements in *buffer.*

MS BASIC Synopsis

DECLARE SUB **EpcMemSwapW**(SEG *buffer%*, BYVAL *entrycount%*)

CALL **EpcMemSwapW**(*buffer%*, *entrycount%*)

Remarks This function swaps the bytes in each 16-bit element in the buffer.

For example, given the following:

```
unsigned short buffer[] =
{ 0x1122, 0x3344, 0x5566, 0x7788 };
```

this call:

```
EpcMemSwapW(buffer, 4);
```

returns the following:

```
buffer[0] = 0x2211
buffer[1] = 0x4433
buffer[2] = 0x6655
buffer[3] = 0x8877
```

See Also **EpcMemSwapL, EpcSwapL, EpcSwapW.**

EpcRestState

Description Restores an access mode and a bus window base that were previously saved by a call to **EpcSaveState**.

C Synopsis

short FAR PASCAL
EpcRestState(unsigned long FAR* *state_stash*);

state_stash Pointer to a 4-byte area in which the mapping state will be saved.

MS BASIC Synopsis

DECLARE FUNCTION **EpcRestState**(SEG *state_stash&*)

Ok% = **EpcRestState**(*state_stash&*)

Remarks This function does not check the validity of the internal format.

Return Value If successful, the function restores the specified access mode and bus window. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcGetAccMode, EpcGetAmMap, EpcMapBus, EpcSaveState, EpcSetAccMode, EpcSetAmMap.**

EpcSaveState

Description Preserves the current access mode and bus window base in a caller-supplied area.

C Synopsis

void FAR PASCAL
EpcSaveState(unsigned long FAR* *state_stash*);

state_stash	Pointer to a 4-byte area in which the mapping state has been saved.

MS BASIC Synopsis

DECLARE SUB **EpcSaveState**(SEG *state_stash&*)

CALL **EpcSaveState**(*state_stash&*)

Remarks This function preserves the current access mode and bus window base in a caller-supplied area. This function does not check the validity of the internal format.

Return Value NONE

See Also **EpcGetAccMode, EpcGetAmMap, EpcMapBus, EpcRestState, EpcSetAccMode, EpcSetAmMap.**

EpcSetAccMode

Description Defines the current bus access mode.

C Synopsis

short FAR PASCAL
EpcSetAccMode(short *mode*);

mode	Desired access mode.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSetAccMode%**(BYVAL *mode%*)

ok% = **EpcSetAccMode%**(*mode%*)

Remarks The function defines the EPC's current access mode.

The returned access mode is an OR'd combination of a byte order constant and an address modifier constant.

0	0	0	0	0	0	0	*byte order*	0	0	*addrmod*

Valid byte order constants are the following:

Constant	Description
BM_IBO	Intel (80x86-style) byte ordering
BM_MBO	Motorola (68000-style) byte ordering

Valid address modifier constants are the following:

Constant	Description
A16N	A16 non-privileged address modifier
A16S	A16 supervisor
A24ND	A24 non-privileged data address modifier
A24NP	A24 non-privileged program address modifier
A24SD	A24 supervisor data address modifier
A24SP	A24 supervisor program address modifier
A32ND	A32 non-privileged data address modifier
A32NP	A32 non-privileged program address modifier
A32SD	A32 supervisor data address modifier
A32SP	A32 supervisor program address modifier

Note that **EpcSetAmMap** is the preferred method of setting the bus access parameters.

Return Value

The following return values are supported:

Constant	Description
EPC_SUCCESS	The function completed successfully.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_UNSUPPORTED_FNCT	The function requires unsupported functionality (most likely, Motorola 68000 [big-endian] byte swapping).

See Also

EpcGetAccMode, EpcGetAmMap, EpcRestState, EpcSaveState, EpcSetAmMap.

2

EpcSetAmMap

Description Defines the current bus access mode and bus window base.

C Synopsis

short FAR PASCAL
EpcSetAmMap(unsigned short *accessmode,* **unsigned long** *busaddress,* **void FAR * FAR *** *mapped_ptr*);

accessmode	Desired access mode.
busaddress	Desired bus address.
mapped_ptr	Returned pointer to desired address space.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSetAmMap**%(BYVAL *accessmode%*, BYVAL *busaddress&*, SEG *mapped_ptr&*)

returncode% = **EpcSetAmMap**%(*accessmode%*, *busaddress&*, *mapped_ptr&*)

DECLARE FUNCTION **EpcSetAmMapB**%(BYVAL *accessmode%*, BYVAL *busaddress&*, SEG *busseg%*, SEG *busoff%*)

returncode% = **EpcSetAmMapB**%(*accessmode%*, *busaddress&*, *busseg%*, *busoff%*)

Remarks The function defines the EPC's current bus access mode and bus window base address.

The returned access mode is an OR'd combination of a byte order constant and an address modifier constant.

0	0	0	0	0	0	0	*byte order*	0	0	*addrmod*

2

Valid byte order constants are the following:

Constant	Description
BM_IBO	Intel (80x86-style) byte ordering
BM_MBO	Motorola (68000-style) byte ordering

Valid address modifier constants are the following:

Constant	Description
A16N	A16 non-privileged address modifier
A16S	A16 supervisor
A24ND	A24 non-privileged data address modifier
A24NP	A24 non-privileged program address modifier
A24SD	A24 supervisor data address modifier
A24SP	A24 supervisor program address modifier
A32ND	A32 non-privileged data address modifier
A32NP	A32 non-privileged program address modifier
A32SD	A32 supervisor data address modifier
A32SP	A32 supervisor program address modifier

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	The function completed successfully.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_UNSUPPORTED_FNCT	The function requires unsupported functionality, most likely Motorola 68000 (big-endian) byte swapping.

See Also **EpcMapBus, EpcSetAccMode, EpcSaveState.**

EpcSetError

2

Description Defines a specified error's handler function and stack.

C Synopsis

void (FAR CDECL * FAR PASCAL
EpcSetError(short *error,*
void (FAR CDECL * *new_handler*)**(unsigned long** *error*)
char FAR * *new_stack*, **char FAR * FAR ***
prev_stack))**(unsigned long** *error*);

error	Error number.
new_handler	Address of new error handler.
new_stack	Base address of new stack.
prev_stack	Location where the base address of the current stack will be placed.

MS BASIC Synopsis

NONE

Remarks The function defines the handler and stack addresses for an *error* and returns the current handler and stack addresses.

The following constants define valid values for *error*:

Constant	Description
BM_SYSFAIL_ERR	SYSFAIL assertion.
BM_BERR_ERR	VMEbus BERR.
BM_ACFAIL_ERR	ACFAIL assertion.
BM_WATCHDOG_ERR	Watchdog timer expiration.

An error handler function has the following calling semantics:

void FAR CDECL
new_handler **(unsigned long** *error*);

Error handling works similarly to interrupt handling, with two exceptions:

1) Where an interrupt handler is passed the Status/ID of the VMEbus interrupter, an error handler is passed the error number.

2) The BusManager clears all error conditions before calling the handler.

If *prev_stack* is null, the previous stack pointer is not returned.

To remove an assigned handler, call this function with *new_handler* set to null. The BusManager will assign the "do-nothing" function and disable the interrupt.

This function returns the address of the handler previously assigned to the specified interrupt. If no handler has been assigned (or if the interrupt ,was last connected to the "do-nothing" function), this function returns the address of the "do-nothing" function.

Calling **EpcSetError** to assign a handler to a VMEbus error immediately enables the specified interrupt.

Return Value If successful, the function returns the address of the current error handler. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcDisErr, EpcEnErr, EpcGetError.**

EpcSetIntr

Description Defines a specified interrupt's handler function and stack.

C Synopsis

void (FAR CDECL * FAR PASCAL
EpcSetIntr(short *interrupt,*
void (FAR CDECL * *new_handler*)(**unsigned long** *data*),
char FAR * *new_stack,*
char FAR * FAR * *prev_stack*))(**unsigned long** *data*);

MS BASIC Synopsis

NONE

Remarks The function defines the handler and stack addresses for an *interrupt* and returns the current handler and stack addresses.

The parameter *interrupt* specifies the interrupt condition to disable. The following constants define valid values for *interrupt*:

Constant	Description
BM_MSG_INTR	Message interrupt.
BM_VME_INTR1	VMEbus interrupt 1.
...	
BM_VME_INTR7	VMEbus interrupt 7.
BM_ER_INTR	Event/Response interrupt.
BM_TTLTRG0_INTR	TTL trigger interrupt 0 (EPC-7 only).
...	
BM_TTLTRG7_INTR	TTL trigger interrupt 7 (EPC-7 only).

An interrupt handler function has the following calling semantics:

void FAR CDECL
new_handler (**unsigned long** *data*)

The following actions are taken when the specified interrupt occurs:

1) Disable processor interrupt.

2) Acknowledge the programmable interrupt controllers (PICs).

3) If this is a VMEbus interrupt, acknowledge it. If it is a message interrupt, disabled it. (Message interrupts are enabled by the message-passing functions, described elsewhere in this chapter.)

4) Push the bus state (access mode and bus window) onto the stack.

5) Switch to the handler's stack.

6) If this is a VMEbus interrupt, zero-extend the 16-bit Status/ID value from the interrupt acknowledgment to a long (32-bit) value. Note that a 16-bit Status/ID is always requested — it is up to the handler to know the actual size (8 or 16 bits) of the Status/ID that the device returns.

7) The interrupt handler is invoked by means of a FAR call, and is passed a 32-bit parameter. It returns with a RET instruction to the BusManager.

8) The BusManager switches to its own stack, restores the saved bus state, and enables processor interrupts.

If the BusManager detects an interrupt that has no handler assigned, the BusManager invokes a "do-nothing" function.

To remove an assigned handler, call this function with *new_handler* set to null. The BusManager will assign the "do-nothing" function and disable the interrupt.

This function returns the address of the handler previously assigned to the specified interrupt. If no handler has been assigned (or if the interrupt was last connected to the "do-nothing" function), this function returns the address of the "do-nothing" function.

If *prev_stack* is null, then it is not set to the previous stack pointer by this function. If *prev_stack* is not null, then the value at the location to which it points is set to null by this function.

Calling **EpcSetIntr** to assign a handler to a bus interrupt immediately enables the specified interrupt. A call to **EpcEnIntr** is unnecessary.

Return Value If successful, the function returns the address of the current interrupt handler. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcDisIntr, EpcEnIntr, EpcGetIntr.**

2

EpcSetSlaveAddr

Description Defines the address space and base address of the EPC's slave memory.

C Synopsis

short FAR PASCAL
EpcSetSlaveAddr(unsigned short *addrspace,* **unsigned long** *slavebase*);

addrspace	New address space.
slavebase	New slave base address.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSetSlaveAddr%**(BYVAL *addrspace%*, BYVAL *slavebase&*)

returncode% = **EpcSetSlaveAddr%**(*addrspace%*, *slavebase&*)

Remarks The function defines the address space and base address of the EPC's slave memory. Valid values for *addrspace* and *slavebase* are the following:

EPC type	**slavebase*	**addr space*
EPC-2	0x18000000, 0x19000000, ... , 0x1F000000	BM_A32
	EPC_SLAVE_MEMORY_DISABLED	N/A
EPC-7	0x000000, 0x400000, ..., 0xC00000	BM_A24
	0x00000000, 0x01000000, ..., 0xFF000000	BM_A32
	EPC_SLAVE_MEMORY_DISABLED	N/A
EPC-8	EPC_SLAVE_MEMORY_DISABLED	N/A

A24 base addresses are aligned on a 4 MByte boundary, and only the first 4 MBytes of the EPC's slave memory is mapped to the bus. A32 base addresses are aligned on a 16 MByte boundary, and only the first 16 MBytes of the EPC's slave memory is mapped to the bus.

To disable slave memory, call this function with a slave base address of **EPC_SLAVE_MEMORY_DISABLED**.

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	The slave base address is not supported on this EPC.
EPC_SUCCESS	Successful function completion.

See Also **EpcGetSlaveAddr, EpcGetSlaveBase, EpcSetSlaveBase.**

2

EpcSetSlaveBase

Description Defines the current base address of the EPC's slave memory.

C Synopsis

short FAR PASCAL
EpcSetSlaveBase(unsigned long *slavebase*);

slavebase New slave base address.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSetSlaveBase%**(BYVAL *slavebase&*)
returncode% = **EpcSetSlaveBase%**(*slavebase&*)

Remarks The function defines the base address of the EPC's slave memory. Valid values for *slavebase* and the implied address space are the following:

EPC type	Slave Base Address	Implied Slave Address Space
EPC-2	0x18000000, 0x19000000, ... , 0x1F000000	BM_A32
	EPC_SLAVE_MEMORY_DISABLED	N/A
EPC-7	0x000000, 0x400000, ..., 0xC00000	BM_A24
	0x00000000, 0x01000000, ..., 0xFF000000	BM_A32
	EPC_SLAVE_MEMORY_DISABLED	N/A
EPC-8	EPC_SLAVE_MEMORY_DISABLED	N/A

A24 base addresses are aligned on a 4 Mbyte boundary, and only the first 4 Mbytes of the EPC's slave memory is mapped to the bus. A32 base addresses are aligned on a 16 MByte boundary, and only the first 16 MBytes of the EPC's slave memory is mapped to the bus.

To disable slave memory, call this function with a slave base address of **BM_SLAVE_MEMORY_DISABLED**.

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_FAIL	The slave base address is not supported on this EPC.

See Also **EpcGetSlaveAddr, EpcGetSlaveBase, EpcSetSlaveAddr.**

EpcSetUla

Description Defines the EPC's unique logical address (ULA).

C Synopsis

short FAR PASCAL
EpcSetUla(unsigned short *ula***);**

ula	New unique logical address.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSetUla%**(BYVAL *ula%*)
returncode% = **EpcSetUla%**(*ula%*)

Remarks The ULA is used to determine the base address of the EPC configuration registers in A16 space, as follows:

```
A16_Address =(ULA<<6)+0xC000;
```

Return Value The following return values are supported:

Constant	Description
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
EPC_SUCCESS	Successful function completion.

See Also **EpcGetUla.**

EpcSigIntr

Description Signals (asserts or deasserts) a VMEbus interrupt.

C Synopsis

short FAR PASCAL
EpcSigIntr(short *interrupt*);

interrupt Interrupt number.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSigIntr**%(BYVAL *interrupt%*)
ok% = **EpcSigIntr**%(*interrupt%*)

Remarks The function asserts or deasserts a VMEbus interrupt.

The parameter *interrupt* specifies the VMEbus to assert or deassert. The following values are valid:

Value	Description
0	Deassert the currently asserted VMEbus interrupt.
BM_VME_INTR1	Assert VMEbus interrupt 1.
...	
BM_VME_INTR7	Assert VMEbus interrupt 7.

If *interrupt* is non-zero and the EPC is not asserting an interrupt, then the appropriate VMEbus interrupt (1 through 7) is asserted. If the *interrupt* is non-zero and the EPC is asserting an interrupt, then the function fails. If *interrupt* is zero and the EPC is already asserting an interrupt, then the bus interrupt is deasserted and the function succeeds. It is not an error to deassert an interrupt when no interrupt is asserted - this function always succeeds if *interrupt* is set to zero.

2

Return Value The following return value is supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.

See Also **EpcDisIntr, EpcEnIntr, EpcGetIntr, EpcSetIntr.**

EpcSwapL

Description Byte-swaps a single 32-bit value.

C Synopsis

unsigned long FAR PASCAL
EpcSwapL(unsigned long *value*);

value 32-bit value to be swapped.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSwapL**&(BYVAL *value*&)
newvalue& = **EpcSwapL**&(*value*&)

Remarks This function swaps the bytes in the supplied 32-bit *value* and returns the result.

For example, the following call:

```
EpcSwapL(0x11223344);
```

returns the value `0x44332211`.

See Also **EpcMemSwapL, EpcMemSwapW, EpcSwapW.**

EpcSwapW

Description Byte-swaps a single 16-bit value.

C Synopsis

unsigned short FAR PASCAL
EpcSwapW(unsigned short *value*);

value 16-bit value to be swapped.

MS BASIC Synopsis

DECLARE FUNCTION **EpcSwapW%**(BYVAL *value%*)

newvalue% = **EpcSwapW%**(*value%*)

Remarks This function swaps the bytes in the supplied 16-bit *value* and returns the result.

For example, the following call:

```
EpcSwapW(0x1122);
```

returns the value `0x2211`.

See Also **EpcMemSwapL, EpcMemSwapW, EpcSwapL.**

EpcToVme

Description Copy consecutive EPC locations to consecutive VMEbus locations using the current access mode.

C Synopsis

unsigned short FAR PASCAL
EpcToVme(short *width*, **char FAR** **source*, **unsigned long** *dest*, **unsigned short** *count*);

width	Number of data bits to copy per bus access.
source	Source address in EPC memory.
dest	Destination VMEbus address.
count	Number of bytes to transfer.

MS BASIC Synopsis

DECLARE FUNCTION **EpcToVme%**(BYVAL *width%*, SEG *source%*, BYVAL *dest&*, BYVAL *count%*)

DIM src%[...]

ok% = **EpcToVme%**(*width%*, *source%*, *dest&*, *count%*)

Remarks This function copies data from consecutive EPC locations to consecutive VMEbus locations using the current access mode. The current access mode set by the most recent **EpcRestState** or **EpcSetAmMap** is saved, the bus window is altered as necessary during the copy, and the access mode is restored.

This function is intended for transferring large amounts of data to consecutive locations.

The *count* parameter always specifies the number of bytes to transfer, regardless of the specified *width*. Setting *count* to zero specifies a transfer of zero bytes.

2

The *width* parameter specifies whether data is to be moved in 8-bit, 16-bit, or 32-bit chunks. Transfers are always aligned on natural boundary; 16-bit quantities are written to the VMEbus only at even addresses, and 32-bit quantities are written to the VMEbus only at addresses evenly divisible by 4.

Valid values for the *width* parameter are the following:

Constant	Description
BM_W8	8-bit copy width
BM_W8O	8-bit copy width, odd-only copy
BM_W16	16-bit copy width
BM_W32	32-bit copy width
BM_FASTCOPY	Don't check for intermediate bus errors. This constant can be OR'd with one of the previous constants to increase copy speed.

Transfers to non-aligned locations are done in a read-modify-write fashion – a chunk is read from the destination, the bytes to be transferred are copied to the corresponding bytes in the chunk, and the chunk is replaced. For example, a copy of 32-bit chunks to a non-aligned address would occur in the following manner. The leading 32-bit word would be read from the destination, modified, and written back. Next, all whole (aligned) 32-bit values would be transferred. Finally, the trailing 32-bit word would be read from the destination, modified, and replaced.

Notes:

- This "read-modify-write" sequence is done in software, and is *not* an RMW bus cycle.
- If an unmodified byte in the leading or trailing word of a non-aligned transfer contains a semaphore that is signaled while the copy is taking place, the signal may be lost.

When you specify 8-bit, odd-only transfers (**BM_W8O**), the VMEbus address "spins" twice as fast as the EPC address. That is, for `i = 0` to `(count - 1)`, `dest + (i x 2) + 1` receives `src + i`.

By default, BERR is checked after every transfer. If there is an error, the copy is aborted but the BERR error handler is not called. This eliminates the requirement that the calling program coordinate with the BERR handler. Errors are reflected by a non-zero return value.

If you OR the *width* parameter with **BM_FASTCOPY** before calling the copy function, BERR is checked only after transfers to nonaligned locations. Fast copying uses "Move String" instructions to copy "blocks" of data. By taking advantage of pipelining in the processor and the VMEbus interface hardware, fast copy transfers are five times faster than transfers without **BM_FASTCOPY**. There are risks, however: a BERR may go undetected, or the BERR error handler may be called erroneously (if a transfer – still in the pipeline when the function returns – causes a BERR). In general, you should select the fast copy option.

The fast copy flag (**BM_FASTCOPY**) is ignored when you specify 8-bit, odd-only transfers (**BM_W8O**).

Return Value

The function returns **EPC_SUCCESS** on successful completion. Otherwise, the function returns the number of bytes ***not*** transferred. This indicates there was a VMEbus error (BERR).

See Also

EpcFromVme, EpcFromVmeAm, EpcRestState, EpcSetAmMap, EpcToVmeAm.

2

EpcToVmeAm

Description Copies consecutive EPC locations to consecutive VMEbus locations using a specified access mode.

C Synopsis

unsigned short FAR PASCAL
EpcToVmeAm(short *mode*, **short** *width*, **char** **source*, **unsigned long** *dest*, **unsigned short** *count)*;

mode	Access mode.
width	Number of data bits to copy per bus access.
source	Source address in EPC memory.
dest	Destination VMEbus address.
count	Number of bytes to transfer.

MS BASIC Synopsis

DECLARE FUNCTION **EpcToVmeAm**%(BYVAL *mode%*, BYVAL *width%*, SEG *source%*, BYVAL *dest&*, BYVAL *count%*)

DIM *source%*[...]

ok% = **EpcToVmeAm**%(*mode%*, *width%*, *source%*, *dest&*, *count%*)

Remarks This function copies data from consecutive EPC locations to consecutive bus locations using the specified access mode. The current access mode and bus window are saved, altered as specified during the copy, and restored upon completion of the copy.

The parameter *mode* is an OR'd combination of a byte order constant and an address modifier constant:

The returned access mode is an OR'd combination of a byte order constant and an address modifier constant:

0	0	0	0	0	0	0	*byte order*	0	0	*addrmod*

The following constants define valid byte order constants:

Constant	Description
BM_IBO	Little-endian (Intel 386-style) byte order
BM_MBO	Big-endian (Motorola 68000-style) byte order

The following constants define valid address modifier constants:

Constant	Description
A16N	A16 non privileged address modifier
A16S	A16 supervisor address modifier
A24ND	A24 non privileged data address modifier
A24NP	A24 non privileged program address modifier
A24SD	A24 supervisor data address modifier
A24SP	A24 supervisor program address modifier
A32ND	A32 non privileged data address modifier
A32NP	A32 non privileged program address modifier
A32SD	A32 supervisor data address modifier
A32SP	A32 supervisor program address modifier

The *width* parameter specifies whether data is to be moved in 8-bit, 16-bit, or 32-bit chunks. VMEbus transfers are always aligned on natural boundary; 16-bit quantities are written to the VMEbus only at even addresses, and 32-bit quantities are written to the VMEbus only at addresses evenly divisible by 4.

Valid values for the *width* parameter are the following:

Constant	Description
BM_W8	8-bit copy width
BM_W80	8-bit copy width, odd-only copy
BM_W16	16-bit copy width
BM_W32	32-bit copy width
BM_FASTCOPY	Don't check for intermediate bus errors. This constant can be OR'd with one of the previous constants to increase copy speed.

Transfers to non-aligned locations are done in a read-modify-write fashion – a chunk is read from the destination, the bytes to be transferred are copied to the corresponding bytes in the chunk, and the chunk is replaced. For example, a copy of 32-bit chunks to a non-aligned address would occur in the following manner. The leading 32-bit word would be read from the destination, modified, and written back. Next, all whole (aligned) 32-bit values would be transferred. Finally, the trailing 32-bit word would be read from the destination, modified, and replaced.

Notes:

- This "read-modify-write" sequence is done in software, and is *not* a RMW bus cycle.
- If an unmodified byte in the leading or trailing word of a non-aligned transfer contains a semaphore that is signaled while the copy is taking place, the signal may be lost.

When you specify 8-bit, odd-only transfers (**BM_W8O**), the VMEbus address "spins" twice as fast as the EPC address. That is, for `i = 0` to `(count - 1)`, `dest + (i x 2) + 1` receives `src + i`.

By default, BERR is checked after every transfer. If there is an error, the copy is aborted but the BERR error handler is not called. This eliminates the requirement that the calling program coordinate with the BERR handler. Errors are reflected by a non-zero return value.

If you OR the *width* with **BM_FASTCOPY** before calling the copy function, BERR is checked only after transfers to nonaligned locations. Fast copying uses "Move String" instructions to copy "blocks" of data. By taking advantage of pipelining in the processor and the VMEbus interface hardware, fast copy transfers are five times faster than transfers without **BM_FASTCOPY**. There are risks, however: a BERR may go undetected, or the BERR error handler may be called erroneously (if a transfer – still in the pipeline when the function returns – causes a BERR). In general, you should select the fast copy option.

The Fast Copy flag (**BM_FASTCOPY**) is ignored when you specify 8-bit, odd-only transfers (**BM_W80**).

Return Value The function returns **EPC_SUCCESS** on successful completion. Otherwise, the function returns the number of bytes ***not*** transferred, indicating a bus error (BERR).

See Also **EpcFromVme, EpcFromVmeAm, EpcToVme.**

2

EpcVmeCtrl

Description Queries or defines VMEbus interface control bits.

C Synopsis

short FAR PASCAL
EpcVmeCtrl(unsigned short *opcode,* **unsigned short** *flag*);

opcode	Read, assert or deassert flag.
flag	Possible flags are described below.

MS BASIC Synopsis

DECLARE FUNCTION **EpcVmeCtrl%**(BYVAL *code%*, BYVAL *flag%*)

value% = **EpcVmeCtrl%**(*code%*, *flag%*)

Remarks The function reads, asserts or deasserts VMEbus interface control bits. The parameter *flag* defines the desired control bit and *opcode* defines whether to read, assert, or deassert the bit.

Valid values for *opcode* are the following:

Code	Description
CTRL_READ	read flag
CTRL_ASSERT	assert flag
CTRL_DEASSERT	deassert flag

Valid values for *flag* are the following:

Flag	Description
VME_SYSFAIL_EN	SYSFAIL out enable
VME_SYSRESET_EN	SYSRESET in enable
VME_SYSRESET	SYSRESET out
VME_PASSTEST	self test pass
VME_EXTTEST	in extended self test
VME_WATCHDOG	watchdog timer expired (read only)
VME_ACFAIL_IN	ACFAIL asserted (read only)

Code	Description
VME_BERR_IN	BERR asserted (destructive read)
VME_SYSFAIL_IN	SYSFAIL asserted (read only)
VME_A24_SLAVE	A24 slave (always zero on EPC-8)
VME_ACCESS	VME access
VME_WRITE	VME write
VME_PIPELINE_BUSY	VME pipeline busy
VME_STICKY_BERR	sticky BERR
VME_SIGNAL	SIGNAL register available
VME_SLAVE_EN	VME slave enable (always zero on EPC-8)

Return Value

When *opcode* is **CTRL_READ**, the function returns zero if the control bit specified by flag is deasserted and if it is asserted. Note that the function hides whether the logic of the control bit is negative-TRUE or positive-TRUE.

For *opcode* values of **CTRL_ASSERT** and **CTRL_DEASSERT**, the following values are returned:

ERR_FAIL	The specified *opcode* or *flag* value is invalid.
EPC_SUCCESS	Successful function completion.

EpcVxiCtrl

Description Queries or defines VXIbus interface control bits.

C Synopsis

short FAR PASCAL
EpcVxiCtrl(unsigned short *code*, **unsigned short** *flag*);

code	Read, assert, or deassert flag
flag	Possible flags are described below.

MS BASIC Synopsis

DECLARE FUNCTION **EpcVxiCtrl**%(BYVAL *code*%, BYVAL *flag*%)

value% = **EpcVxiCtrl**%(code%, flag%)

Remarks The function reads, asserts or deasserts VXIbus interface control bits. The parameter *flag* defines the desired control bit and *opcode* defines whether to read, assert, or deassert the bit.

Valid values for *opcode* are the following:

Code	Description
CTRL_READ	read flag
CTRL_READ_STATE	read trigger
CTRL_ASSERT	assert flag
CTRL_DEASSERT	deassert flag

Valid values for *flag* are the following:

Flag	Description
0	Data Input Ready (DIR)
1	Data Output Ready (DOR)
2	ERR Flag
OLRM_TTLTRG0	TTL Trigger Line 0 (EPC-2 and EPC-7 only)
OLRM_TTLTRG1	TTL Trigger Line 1 (EPC-2 and EPC-7 only)

OLRM_TTLTRG2	TTL Trigger Line 2 (EPC-2 and EPC-7 only)
OLRM_TTLTRG3	TTL Trigger Line 3 (EPC-2 and EPC-7 only)
OLRM_TTLTRG4	TTL Trigger Line 4 (EPC-2 and EPC-7 only)
OLRM_TTLTRG5	TTL Trigger Line 5 (EPC-2 and EPC-7 only)
OLRM_TTLTRG6	TTL Trigger Line 6 (EPC-2 and EPC-7 only)
OLRM_TTLTRG7	TTL Trigger Line 7 (EPC-2 and EPC-7 only)
OLRM_ECLTRG1	ECL Trigger Line 1 (EPC-2 and EPC-7 only)
OLRM_ECLTRG2	ECL Trigger Line 2 (EPC-2 and EPC-7 only)

Return Value

When *opcode* is **CTRL_READ** or **CTRL_READ_STATE**, the function returns zero if the control bit specified by flag is deasserted and if it is asserted. Note that the function hides whether the logic of the control bit is negative-TRUE or positive-TRUE.

For *opcode* values of **CTRL_ASSERT** and **CTRL_DEASSERT**, the following values are returned:

ERR_FAIL	The specified *opcode* or *flag* value is invalid.
EPC_SUCCESS	Successful function completion.

EpcWaitIntr

Description Waits for an interrupt to occur.

C Synopsis

short FAR PASCAL
EpcWaitIntr(unsigned short *mask,* **unsigned long FAR *** *status,*
unsigned long *waittime*);

short FAR PASCAL
EpcWaitIntr2(unsigned short *mask,* **unsigned long FAR *** *status,*
unsigned long FAR* *memwaittime*);

mask	Mask of interrupts to await.
memwaittime	Address location containing the number of milliseconds to wait before returning.
waittime	Number of milliseconds to wait before returning.
status	Returned Status/ID.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWaitIntr%**(BYVAL *mask%*, SEG *status&*, BYVAL *waittime&*)

ok% = **EpcWaitIntr%**(*mask%*, *status&*, *waittime&*)

DECLARE FUNCTION **EpcWaitIntr2%**(BYVAL *mask%*, SEG *status&*, SEG *memwaittime&*)

ok% = **EpcWaitIntr2%**(*mask%*, *status&*, *memwaittime&*)

Remarks These functions wait up to *waittime* (or **memwaittime*) milliseconds for one of the interrupts specified by *mask* to occur.

The parameter *mask* specifies the interrupt(s) to await. It is an OR'd combination of the following:

Value	Description
1<<BM_MSG_INTR	Message interrupt.
1<<BM_VME_INTR1	VMEbus interrupt 1.
...	
1<<BM_VME_INTR7	VMEbus interrupt 7.
1<<BM_ER_INTR	Event/Response interrupt.

Both **EpcWaitIntr** and **EpcWaitIntr2** return the mask of the highest priority interrupt that occurs, zero if the timer expires before any of the awaited interrupts occur, and **ERR_FAIL** if some other error occurs. Functions **EpcWaitIntr** and **EpcWaitIntr2** differ in that **EpcWaitIntr** takes milliseconds as a parameter, while **EpcWaitIntr2** takes a pointer to milliseconds as a parameter and modifies the contents of that location to reflect the number of milliseconds remaining when an interrupt occurs.

The timer value is expressed in milliseconds. If *waittime* (or the value stored at the location specified by *memwaittime*) is zero, only one check will be made before returning. If no interrupt handler exists for this interrupt, **EpcWaitIntr** sends the appropriate interrupt acknowledgment before returning to the caller. The bus state is not saved or restored.

Upon function completion, *status* contains the status/ID of the interrupt. A 16-bit interrupt acknowledge (IACK) cycle is performed when a VMEbus interrupt arrives. It is up to the calling program to know whether the device generating the interrupt returns an 8-bit or 16-bit Status/ID. For compatibility with future products, this value is zero-extended to 32 bits.

If an interrupt also has a handler assigned to it, then that handler is executed before this call returns (see **EpcSetIntr**).

Whenever an interrupt occurs, that fact is remembered and will be returned by **EpcWaitIntr**. This behavior eliminates the race condition that would otherwise exist between the device generating the interrupt and the program waiting for the interrupt. However, it can cause the BusManager to remember "stale" interrupts. To avoid this problem, repeatedly call **EpcWaitIntr** with a timeout of zero milliseconds before using a device, until no interrupts are returned. This clears out any stale interrupts for that device.

Notes:

- To use the DOS clock for tracking elapsed time, this function enables processor interrupts for the duration of its execution.
- Only the highest-priority interrupt is handled within a given call, where VMEbus interrupt 7 is highest and the message interrupt is lowest. Other interrupts are left pending.

Return Value If successful, the function returns a non-negative value. Otherwise, the function returns **ERR_FAIL**.

See Also **EpcSetIntr, EpcEnIntr.**

EpcWsCmd

Description Sends a word serial command.

C Synopsis

short FAR PASCAL
EpcWsCmd(unsigned short *ula,* **unsigned short** *command,* **unsigned short FAR** * *result_ptr,* **unsigned short** *wait*);

ula	Servant's unique logical address.
command	Command to send
result_ptr	Address of result
wait	Timeout, in milliseconds.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsCmd%**(BYVAL *ula%*, BYVAL *cmd%*, SEG *result%*, BYVAL *wait%*)

ok% = **EpcWsCmd%**(*ula%*, *cmd%*, *result%*, *wait%*)

DECLARE FUNCTION **EpcWsCmd%**(BYVAL *ula%*, BYVAL *cmd%*, BYVAL *wait%*)

ok% = **EpcWsCmd%**(*ula%*, *cmd%*, *wait%*)

Remarks Sends a word serial command. A command will be sent only when the servant device's WRDY bit is set.

In the C interface, if *result_ptr* is non-NULL, waits for a result and returns it in the location pointed to by *result_ptr*.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

2

2

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_BERR	A bus error occurred sending a word serial command.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_RBERR	A bus error occurred receiving a word serial command response.
ERR_RTIMEOUT	A timeout occurred receiving a word serial command response.
ERR_TIMEOUT	A timeout occurred sending a word serial command.
ERR_WS	A word serial protocol error occurred.

See Also **EpcLwsCmd.**

2

EpcWsRcvStr

Description Receives a series of bytes.

C Synopsis

short FAR PASCAL
EpcWsRcvStr(unsigned short *ula*, **char FAR *** *msg_ptr*, **short** *len*, **short FAR *** *bytecnt_ptr*, **unsigned short** *wait*);

ula	Servant's unique logical address
msg_ptr	Message buffer
len	Message buffer length
bytecnt_ptr	Number of bytes received
wait	Timeout, in milliseconds

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsRcvStr**%(*ula*%, *msg*$, *bytecnt*%, *wait*%)

ok% = **EpcWsRcvStr**%(*ula*%, *msg*$, *bytecnt*%, *wait*%)

Remarks Receives a series of bytes via the word serial BYTE REQUEST command. BYTE REQUEST commands are sent only when the device's DOR (Data Output Ready) and WRDY (Write Ready) bits are set.

If *bytecnt_ptr* is non-NULL, the C interface returns the number of bytes received in the location pointed to by *bytecnt_ptr*.

The MS BASIC interface uses a fixed internal buffer of 512 bytes to construct strings, and received messages are limited to that size.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

The MS BASIC interface doesn't require a length parameter—it passes the length of the message as part of the string descriptor.

This function terminates successfully when a byte with the END bit set is received. It will also terminate when the buffer is full, when a timeout occurs, when a VXIbus error occurs, or when a Word Serial Protocol error is detected.

If the buffer fills before the set END bit is detected, this function returns **ERR_BUFFER_FULL.** Subsequent calls retrieve more data; so you can use a series of calls to **EpcWsRcvStr** to receive long strings.

Return Value

The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_BERR	A bus error occurred sending a word serial command.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_BUFFER_FULL	The specified buffer is full.
ERR_RBERR	A bus error occurred receiving a word serial command response.
ERR_RTIMEOUT	A timeout occurred receiving a word serial command response.
ERR_TIMEOUT	A timeout occurred sending a word serial command.
ERR_WS	A word serial protocol error occurred.

See Also

EpcWsSndStr, EpcWsSndStrNe.

EpcWsServArm

Description Arms the EPC so that it can receive a command.

C Synopsis

short FAR PASCAL
EpcWsServArm(short *code*);

code Arming code.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsServArm**%(BYVAL *code%*)
ok% = **EpcWsServArm**%(*code%*)

Remarks Valid code values are the following:

Constant	Description
BM_WSRCV_DISARM	Disarm commander reception.
BM_WSRCV_ARM	Arm command reception.
BM_WSRCV_ARMandENABLE	Arm command reception and enable the message interrupt.
BM_WSRCV_FDISARM	Forcefully disarm command reception.
BM_WSRCV_FARM	Forcefully arm command reception.
BM_WSRCV_FARMandENABLE	Forcefully arm command reception and enable the message interrupt.

Arming for command receipt sets the VMEbus-readable bit WRDY (write ready), indicating that a command can be accepted. In addition, the message interrupt may be enabled to inform the program when the command arrives. You must call this function before trying to receive a command.

Arming codes **BM_WSRCV_DISARM**, **BM_WSRCV_ARM**, and **BM_WSRCV_ARMandENABLE** obey the EPC locking protocol, allowing multiple controllers to communicate with the same device. This protocol requires that the VMEbus response register not be touched by a controller unless they are going to send a command. In environments where this rule may not be obeyed, use the "force" versions of these sub functions (**BM_WSRCV_FDISARM**, **BM_WSRCV_FARM**, and **BM_WSRCV_FARMandENABLE**).

Return Value

The function returns the following return values:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.

See Also

EpcWsServPeek, EpcWsServRcv, EpcWsServSend.

EpcWsServPeek

Description Waits for a command to arrive without removing the incoming command.

C Synopsis

short FAR PASCAL
EpcWsServPeek(unsigned long FAR * *command,* **unsigned long**
waittime);

short FAR PASCAL
EpcWsServPeek2(unsigned long FAR * *command,* **unsigned**
long FAR * *memwaittime*);

command	Word serial command received.
waittime	Number of milliseconds to wait before returning.
memwaittime	Address of the number of milliseconds to wait before returning.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsServPeek%**(SEG *command&*,
BYVAL *waittime&*)
ok% = **EpcWsServPeek%**(*command&*, *waittime&*)

DECLARE FUNCTION **EpcWsServPeek2%**(SEG *command&*,
SEG *memwaittime&*)
ok% = **EpcWsServPeek2%**(*command&*, *memwaittime&*)

2

Remarks

Both **EpcWsServPeek** and **EpcWsServPeek2** wait for a command to arrive and return it to the caller. The command stays available for subsequent **EpcWsServPeek** and **EpcWsServRcv** calls. **EpcWsServPeek** and **EpcWsServPeek2** differ in that **EpcWsServPeek** takes a timeout parameter while **EpcWsServPeek2** takes a pointer to a timeout parameter and modifies the value to reflect the number of milliseconds remaining when a command arrives.

You must call **EpcWsServArm** before calling this function. Otherwise, **EpcWsServPeek** returns invalid data.

The command size may be 2 or 4 bytes on an EPC-2 or EPC-7 or 2 bytes on an EPC-6. When a 2-byte command is received, the two unused high-order bytes are undefined.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

Return Value

The function returns the size of the command (in bytes) if a command arrives. If no command arrives with the specified time, the function returns zero. Otherwise, the function returns **ERR_FAIL**.

See Also

EpcWsServArm, EpcWsServRcv.

2

EpcWsServRcv

Description Waits for a command to arrive and receive the incoming command.

C Synopsis

short FAR PASCAL
EpcWsServRcv(short *code,* **unsigned long FAR *** *command,*
unsigned long *waittime*);

short FAR PASCAL
EpcWsServRcv2(short *code,* **unsigned long FAR *** *command,*
unsigned long FAR * *memwaittime*);

code	Arming code.
command	Word serial command received.
waittime	Number of milliseconds to wait before returning.
memwaittime	Address of the number of milliseconds to wait before returning.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsServRcv%**(BYVAL *code%*, SEG *command&*, BYVAL *waittime&*)
ok% = **EpcWsServRcv%**(*code%*, *command&*, *waittime&*)

DECLARE FUNCTION **EpcWsServRcv2%**(BYVAL *code%*, SEG *command&*, SEG *memwaittime&*)
ok% = **EpcWsServRcv2%**(*code%*, *command&*, *memwaittime&*)

2

Remarks

EpcWsServRev and **EpcWsServRev2** wait for a command to arrive and returns the command to the caller. **EpcWsServRcv** and **EpcWsServRcv2** differ in that **EpcWsServRcv** takes a timeout as a parameter, while **EpcWsServRcv2** takes a pointer to a timeout parameter and modifies the timeout to reflect the number of milliseconds remaining when a command is received.

The parameter *code* specifies the arming option to perform after receiving the command. Valid values for *code* are the following:

Constant	Description
BM_WSRCV_DISARM	Disarm command reception.
BM_WSRCV_ARM	Arm command reception.
BM_WSRCV_ARMandENABLE	Arm command reception and enable the message.

If a command is received, the action specified in *code* is performed after the receipt and before **EpcWsServRcv** returns. That action is an integral part of the receipt, so race conditions are avoided.

You must call **EpcWsServArm** before calling this function. Otherwise, **EpcWsServRcv** returns invalid data.

The command size may be 2 or 4 bytes on an EPC-2 or EPC-7, or 2 bytes on an EPC-6. When a 2-byte command is received, the two unused high-order bytes are undefined.

To use the DOS clock for tracking elapsed time, this function enables processor interrupts while it operates.

Return Value

The function returns the size of the command (in bytes) if a command is received. If no command is received within the specified time, the function returns zero. Otherwise, the function returns **ERR_FAIL.**

See Also

EpcWsServArm, EpcWsServSend, EpcWsServPeek.

EpcWsServSend

Description Sends a response to the EPC's commander.

C Synopsis

short FAR PASCAL
EpcWsServSend(short *code,* **void FAR *** *command,* **unsigned long** *waittime*);

short FAR PASCAL
EpcWsServSend2(short *code,* **void FAR *** *command,* **unsigned long FAR *** *memwaittime*);

code	Send operation code.
command	Word serial response to send.
waittime	Number of milliseconds to wait before returning.
memwaittime	Address of the number of milliseconds to wait before returning.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsServSend%**(BYVAL *code%*, SEG *command&*, BYVAL *waittime&*)
ok% = **EpcWsServSend%**(*code%*, *command&*, *waittime&*)

DECLARE FUNCTION **EpcWsServSend2%**(BYVAL *code%*, SEG *command&*, SEG *memwaittime&*)
ok% = **EpcWsServSend2%**(*code%*, *command&*, *memwaittime&*)

2

Remarks

EpcWsServSend and **EpcWsServSend2** send a word serial command response to this EPC's commander. **EpcWsServSend** and **EpcWsServSend2** differ in that **EpcWsServSend** takes a timeout parameter, while **EpcWsServSend2** takes a pointer to a time-out parameter and modifies the timeout to reflect the number of milliseconds remaining when the response was received by the EPC's commander. Before the command is sent, however, the VMEbus data register must be cleared (that is, RRDY and WRDY must both be false). The register is cleared only when it is read by the commander, and the *waittime* (or *memwaittime*) parameter lets you prevent the function from waiting indefinitely.

The parameter *code* specifies the send operation. Valid values are the following:

Value	Description
0	Send no command response -- wait for the previous command response to be received.
1	Send no command response -- wait for the previous command response to be received and enable the message interrupt.
2	Send a 16-bit command response.
3	Send a 16-bit command response and enable the message interrupt.
4	Send a 32-bit command response. (EPC-2 and EPC-7 only)
5	Send a 32-bit command response and enable the message interrupt. (EPC-2 and EPC-7 only)

To use the DOS clock for tracking elapsed time, this function enables processor interrupts while it operates.

Return Value

The function supports the following return values:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.

See Also EpcWsServArm, EpcWsServPeek, EpcWsServRcv.

EpcWsSndStr

Description Sends a series of bytes setting the END bit on the last byte.

C Synopsis

short FAR PASCAL
EpcWsSndStr(unsigned short *ula*, **char FAR *** *msg_ptr*, **short** *len*, **short FAR *** *bytecnt_ptr*, **unsigned short** *wait*);

ula	Servant's unique logical address.
msg_ptr	Address of string to send.
len	Message length.
bytecnt_ptr	Number of bytes sent.
wait	Timeout, in milliseconds.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsSndStr%**(BYVAL *ula%*, *msg$*, SEG *bytecnt%*, BYVAL *wait%*)

ok% = **EpcWsSndStr%**(*ula%*, *msg$*, *bytecnt%*, *wait%*)

Remarks Sends a series of bytes via the word serial BYTE AVAILABLE command. BYTE AVAILABLE commands are sent only when the device's DIR (Data Input Ready) and WRDY bits are set. This function sets the END bit in the last command of the series.

Using the C interface, if *bytecnt_ptr* is non-NULL, this function returns the number of bytes sent in the location pointed to by *bytecnt_ptr*.

The BASIC interface doesn't require a length parameter—it passes the length of the message as part of the string descriptor.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_BERR	A bus error occurred sending a word serial command.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_TIMEOUT	A timeout occurred sending a word serial command.
ERR_WS	A word serial protocol error occurred.

See Also EpcWsStat, EpcWsSndStrNe.

EpcWsSndStrNe

Description Sends a series of bytes without setting the END bit on the last byte.

C Synopsis

short FAR PASCAL
EpcWsSndStr(unsigned short *ula*, **char FAR** * *msg_ptr*, **short** *len*, **short FAR** * *bytecnt_ptr*, **unsigned short** *wait*);

ula	Servant's unique logical address.
msg_ptr	Address of string to send.
len	Message length.
bytecnt_ptr	Number of bytes sent.
wait	Timeout, in milliseconds.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsSndStrNe%**(BYVAL *ula%*, *msg$*, SEG *bytecnt%*, BYVAL *wait%*)

ok% = **EpcWsSndStrNe%**(*ula%*, *msg$*, *bytecnt%*, *wait%*)

Remarks This function works the same as **EpcWsSndStr**, except that it does not set the END bit in the last command of the series.

To use the DOS clock for tracking elapsed time, the function enables processor interrupts for the duration of its execution.

Return Value The following return values are supported:

Constant	Description
EPC_SUCCESS	Successful function completion.
ERR_BERR	A bus error occurred sending a word serial command.
ERR_FAIL	A failure occurred while the library was communicating with the BusManager driver.
ERR_TIMEOUT	A timeout occurred sending a word serial command.
ERR_WS	A word serial protocol error occurred.

See Also EpcWsStat, EpcWsSndStrNe.

2

EpcWsStat

Description Returns the word serial status of the designated device.

C Synopsis

short FAR PASCAL
EpcWsStat(unsigned short *ula*);

ula Servant's unique logical address.

MS BASIC Synopsis

DECLARE FUNCTION **EpcWsStat%**(BYVAL *ula%*)

status% = **EpcWsStat%**(*ula%*)

Remarks Returns the status of the designated device or a negative number (indicating failure). Bits 14–8 of the returned status are set to bits 14–8 of the servant's Response Register. These bits are: a reserved bit (14), DOR, DIR, ERR*, Read Ready, Write Ready, and FHS*.

Return Value NONE

See Also **EpcWsCmd.**

NOTES

3. OLRM Functions

The On-Line Resource Manager (OLRM) gives application programs a high-level language interface to the devices on the VXIbus, and manages serially reusable resources such as interrupt and trigger lines. The OLRM allows non-VXIbus devices to be viewed in the same way as VXIbus devices.

The OLRM is attribute oriented, and allows devices to be addressed by either symbolic device name or logical address. It consists of the following functions:

OLRMAllocate	Allocates trigger and interrupt line resources.
OLRMDeallocate	Places the specified resources in the deallocated state.
OLRMGetBoolAttr	Returns boolean information about a specified device.
OLRMGetList	Returns a list of information and the number of elements in the list.
OLRMGetNumAttr	Returns numeric information about the specified device.
OLRMGetStringAttr	Returns ASCII information about a specified device.
OLRMRename	Changes the symbolic name of a device.

For all but **OLRMAllocate** and **OLRMDeallocate**, the first two parameters are an ASCII device name and a numeric logical address. One or the other is used to refer to the device. In the C interface, the ASCII device name is used if the parameter is non-null—the second parameter is ignored. If the ASCII device name is null, the second parameter is used. In the Basic interface, an empty string indicates that the second parameter is to be used.

Unless otherwise noted, these functions return meaningless results when called with inappropriate parameters (such as asking for the memory speed of a register-based VXIbus device).

3.1 Calling the OLRM From MS C and QuickC

3

The C language interface is designed to work with Microsoft C compilers (versions 5.1 and later).

Your C application can be compiled in any of the memory models. To make OLRM independent of the memory models, all calls to OLRM are of type **far Pascal**.

The following examples show how the MS C functions are used:

Example 1
```
    if (OLRMGetBoolAttr("scanner1",0,OLRM_SIGREG)) ...
```
Tests device `scanner1` for a signal register.

Example 2
```
    i = OLRMGetNumAttr("Wavegen",0,OLRM_SLOT);
```
Gets the slot number of device `Wavegen`.

Example 3
```
    i = OLRMGetNumAttr("globalmem",0,OLRM_ADDRESS_BASE);
```
Gets the memory base address of device `globalmem`.

Example 4
```
  manufname = OLRMGetStringAttr("Wavegen" ,0, OLRM_MANUFACTURER,
    manufname);
```
Gets the symbolic manufacturer's name of device `Wavegen`.

Example 5
```
    OLRMGetList(NULL,0,OLRM_DEVICES,256,lalist);
```
Gets a list of logical addresses of all devices.

Example 6
```
    OLRMRename(NULL,25,"Mil1553");
```
Renames the device with logical address 25 as `Mil1553`.

Example 7
```
i = OLRMAllocate(OLRM_TTLTRGANY2);
```
Allocates any two adjacent TTL trigger lines.

3.2 Calling the OLRM From MS BASIC and QuickBASIC

The BASIC interface is designed to work with Microsoft QuickBASIC and Compiled BASIC. The following examples show how the MS BASIC functions are used:

Example 1
```
IF OLRMGetBoolAttr%("scanner1",0,OLRM_SIGREG) <> 0 ...
```
Tests device `scanner1` for a signal register.

Example 2
```
i% = OLRMGetNumAttr%("Wavegen",0,OLRM_SLOT)
```
Gets the slot number of device `Wavegen`.

Example 3
```
i% = OLRMGetNumAttr%("globalmem",0,OLRM_ADDRESS_BASE)
```
Gets the memory base address of device `globalmem`.

Example 4
```
CALL OLRMGetStringAttr("Wavegen" ,0, OLRM_MANUFACTURER, manufname$)
```
Gets the symbolic manufacturer's name of device `Wavegen`.

Example 5
```
retval% = OLRMGetList%("",0,OLRM_DEVICES,256,lalist$)
```
Gets a list of logical addresses of all devices.

Example 6
```
triggers% = OLRMAllocate%(OLRM_TTLTRGANY2)
```
Allocates any two adjacent TTL trigger lines.

3.4 Functions by Name

This section contains an alphabetical listing of the SICL library functions. Each listing describes the function, gives its invocation sequence and arguments, discusses its operation, and lists its returned values. Where usage of the function may not be clear, an example with comments is given. Each function description begins on a new page.

OLRMAllocate

Description Allocates trigger and interrupt line resources.

C Synopsis

unsigned short FAR PASCAL
OLRMAllocate(unsigned short *resource*);

resource Trigger and interrupt line to be allocated.

MS BASIC Synopsis

DECLARE FUNCTION **OLRMAllocate**%(BYVAL *resource%*)

ok% = **OLRMAllocate**%(*resource%*)

Remarks Allocates trigger and interrupt line resources. Resources can be allocated specifically ("give me TTL trigger line 4") and generically ("give me two TTL trigger lines").

The *resource* parameter may be one of the following:

OLRM_TTLTRG0	**OLRM_TTLTRG0123**	**OLRM_ECLTRG23**
OLRM_TTLTRG1	**OLRM_TTLTRG4567**	**OLRM_ECLTRG45OL**
OLRM_TTLTRG2	**OLRM_TTLTRGANY**	**OLRM_ECLTRGANY**
OLRM_TTLTRG3	**OLRM_TTLTRGANY2**	**OLRM_ECLTRGANY2**
OLRM_TTLTRG4	**OLRM_TTLTRGANY4**	**OLRM_IRQ1**
OLRM_TTLTRG5	**OLRM_ECLTRG0**	**OLRM_IRQ2**
OLRM_TTLTRG6	**OLRM_ECLTRG1**	**OLRM_IRQ3**
OLRM_TTLTRG7	**OLRM_ECLTRG2**	**OLRM_IRQ4**
OLRM_TTLTRG01	**OLRM_ECLTRG3**	**OLRM_IRQ5**
OLRM_TTLTRG23	**OLRM_ECLTRG4**	**OLRM_IRQ6**
OLRM_TTLTRG45	**OLRM_ECLTRG5**	**OLRM_IRQ7**
OLRM_TTLTRG67	**OLRM_ECLTRG01**	**OLRM_IRQANY**

You can request the allocation of specific resources, groups of resources (such as TTL triggers 0 and 1), and "any" resources. To accommodate D-size systems, the available resources include the extra four ECL triggers (lines 2-5) on the P3 connector.

To permit computation with these resource values, the encodings are numerically equivalent to the lowest-numbered resource of a class. For example, **OLRM_TTLTRG1** is equal to **OLRM_TTLTRG0** + 1, and **OLRM_IRQ3** is equal to **OLRM_IRQ1** + 2.

Notes:

- Since the **OLRM_ECLTRGANY** and **OLRM_ECLTRGANY2** parameters could allocate ECL triggers 2-5 (nonexistent in a C-size system), one should avoid using these in a C-size system.
- All resources are not necessarily available for allocation when the system is initialized. Specifically, the SURM allocates interrupt lines as described through the Configurator.

Return Value If the resource was allocated, the resource number is returned. In the case of multiple allocations (**OLRMAllocate(OLRM_TTLTRGANY2**), for example), the value returned is' that of the lowest-numbered of the resources allocated. The returned value is 0 if the function fails (that is, if the resource is already allocated, insufficient resources are available, or the resource is unknown).

See Also **OLRMDeallocate.**

OLRMDeallocate

Description Places the specified resources in the deallocated state.

C Synopsis

void FAR PASCAL
OLRMDeallocate(unsigned short *resource*);

resource Trigger or interrupt to be deallocated.

MS BASIC Synopsis

DECLARE SUB **OLRMDeallocate**(BYVAL *resource%*)

CALL **OLRMDeallocate**(*resource%*)

Remarks Places the specified resource(s) in the deallocated state, making them available for allocation. The resource parameters can be any of those specified under **OLRMAllocate** (except for the ***ANY** values).

Return Value None.

See Also **OLRMAllocate.**

3

OLRMGetBoolAttr

Description Returns boolean information about the specified device.

C Synopsis

unsigned short FAR PASCAL
OLRMGetBoolAttr(char FAR **devname*, **unsigned short** *ula*,
unsigned short *attr*);

devname	Device name.
ula	Unique logical address
attr	Attribute

MS BASIC Synopsis

DECLARE FUNCTION **OLRMGetBoolAttr**%(*devname*$,
BYVAL *ula*%, BYVAL *attr*%)

value% = **OLRMGetBoolAttr**%(*devname*$, *ula*%, *attr*%)

Remarks Returns requested information about specified device. The device can be addressed by its symbolic name or logical address.

Attr may be one of the following. The VXIbus source "devtab" is the internal device table maintained by the SURM and OLRM.

Attr	**Source**
OLRM_REGISTER_DEVICE	ID register
OLRM_MEMORY_DEVICE	ID register
OLRM_EXTENDED_DEVICE	ID register
OLRM_MESSAGE_DEVICE	devtab
OLRM_A16_ONLY	ID register
OLRM_A16_A24	ID register
OLRM_A16_A32	ID register
OLRM_A24A32_ENABLED	status register
OLRM_MODID	status register
OLRM_EXTENDED_TEST	status register
OLRM_PASSED	status register
OLRM_SUPVSR_ONLY	memory attribute register
OLRM_BT	memory attribute register
OLRM_N_P	memory attribute register
OLRM_D32	memory attribute register
OLRM_CMDR	message protocol register
OLRM_SIGREG	message protocol register
OLRM_MASTER	message protocol register
OLRM_INTERRUPTER	message protocol register
OLRM_FHS	message protocol register
OLRM_SHMEM	message protocol register
OLRM_DOR	message response register
OLRM_DIR	message response register
OLRM_ERR	message response register
OLRM_RRDY	message response register
OLRM_WRDY	message response register
OLRM_FHS_ACTIVE	message response register
OLRM_LOCKED	message response register
OLRM_FAILED	devtab
OLRM_NOTVXI	devtab
OLRM_MEM_ALLOCATED	devtab
OLRM_EXISTS	devtab
OLRM_HAS_SERVANTS	devtab

If the device is a VXIbus device, most of these attributes cause a VXIbus access.

Return Value The boolean value returned is always of positive logic, regardless of the polarity of the actual VXIbus-defined bit. For instance, the attribute **OLRM_MODID** returns **TRUE** if the device's MODID bit is 0; **OLRM_N_P** returns **TRUE** if a RAM device is nonvolatile or a ROM device electrically programmable.

Most of the attributes are named the same way as in the VXIbus specification. The **OLRM_FAILED** attribute denotes whether the SURM reported the device as failed and placed the device in the safe state. The **OLRM_NOTVXI** attribute denotes whether the device is not a VXIbus device. The **OLRM_MEM_ALLOCATED** attribute denotes whether address space for the device was reserved or allocated in the A24 or A32 address space. The **OLRM_EXISTS** attribute denotes whether the device (specified by symbolic name or logical address) is a known device. The **OLRM_HAS_SERVANTS** attribute denotes whether the device has been assigned any servants by the SURM.

In the event of an error, such as specifying a nonexistent device or calling this function with a VXIbus attribute for a VMEbus device, this function returns 0.

See Also **OLRMGetNumAttr, OLRMGetList, OLRMGetStringAttr.**

OLRMGetList

Description Returns a list of information and the number of elements in the list.

C Synopsis

unsigned short FAR PASCAL
OLRMGetList(char FAR **devname*, **unsigned short** *ula*, **unsigned short** *attr*, **unsigned short** *size*, **char FAR *** *list*);

devname	Device name.
ula	Unique logical address.
attr	Attribute.
size	Maximum list size, in bytes.
list	Pointer to a buffer where the attribute list will be placed.

3

MS BASIC Synopsis

DECLARE FUNCTION **OLRMGetList%**(*devname*$, BYVAL *ula%*, BYVAL *attr%*, *value*$)

retval% = **OLRMGetList%**(*devname*$, *ula%*, *attr%*, *value*$)

Returns a list of information (as bytes in a character array) and the number of elements in *list*. The *size* parameter specifies the maximum number of bytes returned in *list* (the return value is not influenced by *size* and thus may be greater than *size*).

Attr may be either of the following. The source devtab is the internal device table maintained by the SURM and OLRM.

Attr	Source
OLRM_DEVICES	devtab
OLRM_SERVANTS	devtab

If the attribute is **OLRM_DEVICES**, the *devname* and *ula* arguments are ignored. The logical addresses of all VXIbus and pseudo-VXIbus devices in the system are returned in the list.

If the attribute is **OLRM_SERVANTS,** the logical addresses of the specified device's servants are returned in the list. The device can be addressed by symbolic name (*devname*) or logical address.

Return Value The function returns the number of byte elements in the attribute list. If an error occurs, this function returns 0.

See Also **OLRMGetBoolAttr, OLRMGetNumAttr, OLRMGetStringAttr.**

OLRMGetNumAttr

Description Returns requested numeric information about the specified device.

C Synopsis

unsigned short FAR PASCAL
OLRMGetNumAttr(char FAR **devname,* **unsigned short** *ula;*
unsigned short *attr***);**

devname	Device name.
ula	Unique logical address
attr	Attribute

MS BASIC Synopsis

DECLARE FUNCTION **OLRMGetNumAttr**%(*devname*$,
BYVAL *ula%*, BYVAL *attr%*)

value% = **OLRMGetNumAttr**%(*devname*$, *ula%*, *attr%*)

Remarks Returns requested numeric information about the specified device. The device can be addressed by its symbolic name or logical address.

Attr may be one of the following. The source "devtab" is the internal device table maintained by the SURM and OLRM.

Attr	Source
OLRM_CLASS	VXIbus ID register
OLRM_ADDRESS_MODE	VXIbus ID register
OLRM_MANUFACTURER	VXIbus ID register
OLRM_REQ_MEMORY	VXIbus device-type register
OLRM_MODEL	VXIbus device-type register
OLRM_ADDRESS_BASE	VXIbus offset register
OLRM_MEMORY_TYPE	VXIbus memory attribute register
OLRM_SPEED	VXIbus memory attribute register
OLRM_LOG_ADDR	devtab
OLRM_SLOT	devtab
OLRM_CMDR	devtab
OLRM_BID	VXIbus ID register
OLRM_BDT	VXIbus device-type register
OLRM_BSC	VXIbus status register
OLRM_BSO	VXIbus offset register
OLRM_BAT	VXIbus memory attribute register
OLRM_BPR	VXIbus message protocol register
OLRM_BRE	VXIbus message response register
OLRM_BMH	VXIbus message data-high register
OLRM_BML	VXIbus message data-low register

If the device is a VXIbus device, most of these attributes cause a VXIbus access.

The available attributes cover both fields as well as entire registers. The encoding is the same as defined in the VXIbus specification (for example, **OLRM_CLASS** returns a value in the range 0-3).

The **OLRM_LOG_ADDR** attribute denotes the logical address of the device. The **OLRM_SLOT** attribute denotes the slot in which the device resides. The **OLRM_CMDR** attribute denotes the logical address of the device's commander. Every device has a commander. The commander of the top level commander is itself. The **OLRM_BID, OLRM_BDT, OLRM_BSC, OLRM_BSO, OLRM_BAT, OLRM_BPR, OLRM_BRE, OLRM_BMH,** and **OLRM_BML** attributes denote the value of the entire VXIbus register.

Return Value

In the event of an error, such as calling this function with a VXIbus attribute for a VMEbus device, this function returns 0xFFFF.

See Also

OLRMGetBoolAttr, OLRMGetList, OLRMGetStringAttr.

3

OLRMGetStringAttr

Description Returns ASCII information about the specified device.

C Synopsis

char FAR * FAR PASCAL
OLRMGetStringAttr(char FAR **devname,* **unsigned short** *ula,*
unsigned short *attr,* **char FAR** *string***);**

devname	Device name.
ula	Unique logical address
attr	Attribute
string	String

MS BASIC Synopsis

DECLARE SUB **OLRMGetStringAttr**%(*devname*$, BYVAL
ula%, BYVAL *attr*%, *value*$) .

CALL **OLRMGetStringAttr**%(*devname*$, *ula*%, *attr*%, *value*$)

Remarks Returns requested ASCII information about a specific device. The device can be addressed by symbolic name or logical address.

Attr may be one of the following. The source "devtab" is the internal device table maintained by the SURM and OLRM.

Attr	Source
OLRM_DEVICE_NAME	devtab
OLRM_MANUFACTURER	devtab
OLRM_MODEL	devtab

These attributes are the symbolic values as reported by the SURM. The caller is responsible for allocating at least 13 bytes for the fourth parameter (the output string). The value of the attribute is placed in this string and the address of this string is returned.

Return Value If an error occurs, this function returns a null pointer.

See Also OLRMGetBoolAttr, OLRMGetList, OLRMGetNumAttr.

3

OLRMRename

Description Changes the symbolic name of a device.

C Synopsis

char FAR * FAR PASCAL
OLRMRename(char FAR * *devname*, **unsigned short** *ula*, **char *
FAR** *newname*);

MS BASIC Synopsis

NONE

Remarks Changes the symbolic name of a device. The device can be addressed by its symbolic name or logical address. If the device is found, its name is changed to that of *newname* (or the first 12 characters of *newname*) and the returned value is identical to the *newname* parameter. If the device cannot be found, or if any other error occurs, the function returns NULL.

The name change is lost when the machine is shut down or rebooted.

Return Value If the device cannot be found, or if any other error occurs, the function returns NULL.

4. Advanced Topics

This chapter discusses topics of interest to advanced application programmers. Topics include:

- Byte Ordering and Data Representation
- Handler Operations
- Programming Interface
- Writing Device Drivers
- "C" Optimization

4

4.1 Byte Ordering and Data Representation

Byte ordering adds complexity to the VMEbus interface. Many VMEbus devices use the data formats of Motorola microprocessors. Others, including RadiSys EPC controllers, use the data format of Intel microprocessors. Although the Motorola and Intel microprocessors use the same data types, the hardware representations of these data types differ.

Figure 4-1 shows how the same sequence of bytes in memory is interpreted by Intel and Motorola microprocessors. Memory value 11 is at the lowest address and memory value 88 is at the highest address. The data widths shown correspond to the data operand sizes found on both microprocessors.

4

Memory Value	Intel Order	Data Width	Motorola Order
11	11	8 bits	11
22	2211	16 bits	1122
33			
44	44332211	32 bits	11223344
55			
66			
77			
88	8877665544332211	64 bits	1122334455667788

Figure 4-1. Byte Order Example

5.1.1 Byte Swapping Functions

The **EpcMemSwapW** and **EpcSwapW** functions convert 16-bit data between Intel and Motorola byte orders. The **EpcSwapL** and **EpcMemSwapL** functions convert 32-bit data between Intel and Motorola byte orders. Note that 8-bit data does not require conversion.

The block transfer functions (**EpcFromVme, EpcFromVmeAm, EpcToVme,** and **EpcToVmeAm**) conditionally perform byte-swapping.

4.1.2 Correcting Data Structure Byte Ordering

Even if byte-swapping occurs during a block transfer function, byte ordering problems occur when data is copied between Motorola and Intel memory using a different data width than the width of the operand itself. This situation occurs when a data structure containing mixed-type fields is copied in a single operation.

The following code fragment illustrates how to use the **EpcMemSwapL** or the **EpcMemSwapW** functions to correct the byte order in the local copy of the data structure:

```
struct DataStructure
{
        char        field8;
        short       field16;
        long        field32;
} data;

/* Copy the data structure to local memory from the VMEbus. */

EpcFromVme(BM_W8, address, (char FAR*) &data, sizeof(struct
DataStructure));

/* Byte-swap the individual structure fields (data.field8 is an
8-bit field, so it is already correct).
*/

EpcMemSwapW(&data.field16,1);
EpcMemSwapL(&data.field32,1);
```

In the above example, the data structure was copied from VMEbus memory one byte at a time. To copy data from EPC memory to Motorola-ordered memory, byte-swap the fields of the structure in local memory (using the above byte swapping functions) and copy the data using the **EpcToVme** or **EpcToVmeAm** function.

It is sometimes more efficient to copy blocks of data using a data transfer width greater than the expected data width. If you use a greater data transfer width to copy data structures containing mixed-type fields to/from Motorola-order memory, do not use the byte-swapping feature. Swap the data structure fields individually.

4.2 EPConnect Handler Execution Under DOS

Installed interrupt and error handler functions execute as part of a separate thread under DOS. This feature implies that an EPConnect handler function can only call fully reentrant "C" library and EPConnect library functions. Also, an EPConnect handler can only invoke fully reentrant DOS functionality.

These conditions must be true before an application's handlers can execute:

- The application must use **EpcSetError** or **EpcSetIntr** to install a handler function.
- An error or interrupt must occur.

EPConnect discards all interrupts and errors that occur before the application installs a handler and enables interrupt or error reception.

When an application installs a handler and enables interrupt or error reception, the handler processes the interrupt or error as soon as they are received. Under DOS, the installed handler executes as part of an interrupt thread, with processor interrupts disabled, and using the installed handler's stack.

4.3 Writing Device Drivers

This chapter describes how you use the EPConnect programming interface in drivers for VXIbus devices connected to EPCs. You are assumed to have some experience writing DOS device drivers and to have read the BusManager documentation.

4.3.1 General Information

VMEbus device drivers fall into one of two categories:

- **Program-specific drivers**. These are drivers that are a part of a program. Typically, a program-specific driver consists of a set of functions. Most device drivers fall into this category.
- **Resident drivers**. These are drivers that are loaded at boot time. A resident driver is usually built as a DOS driver and loaded in the **CONFIG.SYS** file. A resident driver can also be built as a *terminate-and-stay-resident program* (TSR) and loaded in the **AUTOEXEC.BAT** file.

Program-specific drivers have a totally flexible applications interface – calls may be added easily. Such a driver is relatively easy to implement, but controls the device only while the program is running.

Resident drivers can make a device accessible to all programs by designating it as a DOS device or by defining a service accessible through a DOS interrupt. Resident drivers are much more difficult to write: they are typically written in assembly language and often require the creation of an interface library to give higher-level languages access to device services. The BusManager is an example of a resident driver. It must be loaded before any other resident driver that uses BusManager functions.

4.3.2 Using the VMEbus Window

Access to a device is gained primarily through its control and status registers. These registers are addressable locations, usually in the VMEbus A16 address space, accessible through the EPC VMEbus window. The VMEbus window is a 64KB region of memory which can be mapped to any section of the A16, A24, or A32 address spaces that starts on a 64KB boundary. The bus window is only a VMEbus *master* – it has no slave address and cannot be the destination of an access by other boards. This means, for instance, that a VMEbus device cannot do a direct memory access into the bus window.

The mapping of the bus window onto the VMEbus address space is controlled by the BusManager device driver (**BUSMGR.SYS**). The BusManager provides all the services necessary to use the bus window. BusManager functions that pertain to the bus window include:

- **EpcSetAmMap**. Sets the mapping of the bus window into VMEbus space and sets the address modifier (A16, A24, or A32) and the byte order (either Intel-style or Motorola-style).
- **EpcSaveState**. Stores the bus window mapping, address modifier, and byte order (collectively know as the *state*) in a caller-specified location.
- **EpcRestState** Restores a previously saved state, using the internal representation created by a **EpcSaveState** call.

Several drivers may simultaneously use the bus window, each mapping it to a different location, so take care to save and restore the state used by each driver.

4.3.3 Interrupts

It is often desirable to use the seven VMEbus interrupts generated by a device to control its operation. Several devices may trigger the same interrupt, but all the drivers responding to a given interrupt must run on the same processor and coordinate among themselves. Put another way, each VMEbus interrupt must be handled by exactly one processor.

When the BusManager detects an interrupt for which it is enabled, it issues a 16-bit interrupt acknowledge (IACK) cycle on the VMEbus and gets back an 8-bit or 16-bit Status/ID response from the interrupting device. This Status/ID information is made available to the driver, but the BusManager cannot detect the actual size of the response – it is up to the driver to know whether the response contains 8 or 16 significant bits.

BusManager functions for dealing with interrupts include:

- **EpcWaitInt**. Causes the caller to wait until one of a specified set of interrupts occurs or until a timer expires.
- **EpcSetIntr**. Declares the routine that is called when the specified interrupt occurs.
- **EpcDisIntr**. Tells the BusManager to ignore the specified interrupt.
- **EpcEnIntr**. Tells the BusManager to react to the specified interrupt.

Waiting for Interrupts

The easiest way to deal with device interrupts is to use the **EpcWaitIntr** function. No interrupt handler needs to be set up and no stack needs to be established. This function waits for one of a set of interrupts to occur (or for a specified amount of time to elapse). You *poll* an interrupt by calling the **EpcWaitIntr** function with a timer duration of 0.

If the "awaited" interrupt is enabled and has an assigned handler, that handler is invoked before control returns from the **EpcWaitIntr** call.

By keeping track of interrupts that have occurred before the call to **EpcWaitIntr**, the BusManager assures that no race condition arises. A side effect of "remembering" an interrupt is that old interrupts may still be recorded long after they are significant. As a consequence, drivers that use this function should include in their initialization phase a call to **EpcWaitIntr** with a timer duration of zero (0) to remove any remembered interrupts.

Interrupt Handlers

Polling interrupts is easy for single devices and gives reasonable response time. In a multi-tasking environment, however, it may be more appropriate to install interrupt handlers.

Interrupt handlers are described in more detail in the language-specific sections of this chapter.

4.3.4 Building Resident Drivers

There is much more to know about writing resident device drivers than can be covered in this guide. *The Microsoft MS-DOS Operating System Programmer's Guide* has an excellent section on building resident drivers.

4.3.5 Writing Device Drivers In MS C and QuickC

The Microsoft "C" and QuickC EPConnect interfaces provides access to all BusManager functions. This section is designed for use by readers experienced in writing drivers and interrupt code and familiar with the Microsoft C (version 5.1 or later) compiler, linker, and (where necessary) assembler, and with Microsoft QuickC.

Note: If you are using version 6.0 of the Microsoft "C" compiler, please read the section *C Optimization*.

Using the MS C EPConnect Interface

To use EPConnect functions in a driver, include the appropriate header files in the modules in which the functions are used, and link your driver object files with the library files. The header files contain function prototypes, structure definitions, and constants associated with the EPConnect BusManager functions. (See the section *Programming Interface* for a description of the EPConnect definition files.)

Note: By default, the Microsoft linker allows a program to have 128 segments. The MS "C" library has over 100 segments. If the linker reports "too many segments" you should instruct the linker to allocate more space for segment information. To do so, include the option `/SE:`*nn* on the linker command line, where *nn* is some value greater than 128. (The greater the value you specify, the more space the linker allocates and the slower the linking phase becomes.) Start by specifying 150 for *nn*, then adjust the value to suit your time and space requirements.

If you request more space than the linker can allocate, it will report "requested segment limit too high." Specify a smaller value for *nn* in the /SE command line option.

Using the MS QuickC EPConnect Interface

The Microsoft QuickC EPConnect interface is the same as that for Microsoft "C".

You may link your applications in the QuickC programming environment with the "C" libraries by specifying them in the Program List for the applications through the QuickC Program List facility.

Example 1: Using the VMEbus Window

Access to a device is gained primarily through its control and status registers. These registers are addressable locations, usually in the VMEbus A16 address space, accessible through the EPC VMEbus window. The VMEbus window is a 64KB region of memory which can be mapped to any section of the A16, A24, or A32 address spaces that starts on a 64KB boundary. The bus window is only a VMEbus *master* – it has no slave address and cannot be the destination of an access by other boards. This means, for instance, that a VMEbus device cannot do a direct memory access into the bus window.

The mapping of the bus window onto the VMEbus address space is controlled by the BusManager device driver (**BUSMGR.SYS**). The BusManager provides all the services necessary to use the bus window. BusManager functions that pertain to the bus window include:

- **EpcSetAmMap**. Sets the mapping of the bus window into VMEbus space and sets the address modifier (A16, A24, or A32) and the byte order (either Intel-style or Motorola-style).
- **EpcSaveState**. Stores the bus window mapping, address modifier, and byte order (collectively know as the *state*) in a caller-specified location.
- **EpcRestState**. Restores a previously saved state, using the internal representation created by a **EpcSaveState** call.

Several drivers may simultaneously use the bus window, each mapping it to a different location, so take care to save and restore the state used by each driver. The following code fragment demonstrates how this is done.

```
# include "\epconnec\include\busmgr.h"
    ...
```

```
long MyState;  /* my bus window state */

/*
 * Device Registers
 */

struct my_device {
    unsigned short status;  /* status register */
    unsigned short data[4]; /* data I/O */
};

/* point to device registers */

struct my_device FAR *MyDev;
    ...
/*
 * InitMyDriver -- Initialization entry point for my driver
 */

InitMyDriver()
{
    long old_state;

    /* save state on entry */
    EpcSaveState(&old_state);

    /* set to big endian and A24 space, and map the bus */
    EpcSetAmMap(BM_MBO | A24N, 0x400340L, &MyDev);

    /* speed later access */
    EpcSaveState(&MyState);
        ...

    /* restore entry state */
    EpcRestState(&old_state);
}
/*
 * MyDoOp -- Do an operation on My device
 */

short MyDoOp(op, arg)
short op;
short arg;
{
    long old_state;
        ...

    /* save entry state */
    EpcSaveState(&old_state);

    /*restore device state */
    EpcRestState(&MyState);

    [manipulate device registers pointed to by MyDev]

    /* restore entry state */
```

```
    EpcRestState(&old_state);
}
```

Note how the **EpcSaveState** and **EpcRestState** operations are used to speed the setup of the bus window.

Example 2: Waiting for Interrupts

The easiest way to deal with device interrupts is to use the **EpcWaitIntr** function. No interrupt handler needs to be set up and no stack needs to be established. This function waits for one of a set of interrupts to occur (or for a specified amount of time to elapse). You *poll* an interrupt by calling the **EpcWaitIntr** function.

The following code fragment shows an example of waiting for an interrupt.

```
long status;  /* returned Status/ID */
    ...

EpcEnIntr(MY_INTR)
EpcSaveState(&old_state);
EpcRestState(&MyState);
MyDev->data[0] = DATA1;  /* load up data ports */
MyDev->data[1] = DATA2;
MyDev->status |= DEV_GO;  /* turn on go bit */
if (EpcWaitIntr((1<<MY_INTR), &status, 0) != (1<<MY_INTR)) {
    /*
     * No interrupt!
     */
        ...
    EpcRestState(&old_state);
    return (FAILURE);
}

/*
 * Process interrupt
 */
    ...
EpcRestState(&old_state);
return (SUCCESS);
```

Hint: To increase parallelism, consider designing your application so that, instead of issuing a command to the VMEbus device and waiting for it to finish, you wait for the previous device command to complete and *then* issue the new command.

If the "awaited" interrupt is enabled and has an assigned handler, that handler is invoked before control returns from the **EpcWaitIntr** call.

By keeping track of interrupts that have occurred before the call to **EpcWaitIntr**, the BusManager assures that no race condition arises. A side effect of "remembering" an interrupt is that old interrupts may be recorded long after they are significant. As a consequence, drivers that use this function should include in their initialization phase a call to **EpcWaitIntr** with a timer duration of zero (0) to remove any remembered interrupts.

Example 3: Implementing Interrupt Handlers

Polling interrupts is easy for single devices and gives reasonable response time. In a multi-tasking environment, however, it may be more appropriate to install interrupt handlers.

The BusManager handles only those VMEbus interrupts to which handlers are assigned. Interrupts that have no assigned handlers are ignored by the BusManager when they occur, on the assumption that some other processor on the VMEbus system will handle those interrupts.

When an interrupt that has a handler assigned to it is detected, the BusManager performs the following operations:

1) Disables processor interrupts
2) Acknowledges the processor interrupt (to eliminate race conditions)
3) Determines which VMEbus interrupt was detected
4) Performs the IACK cycle to get the Status/ID and clear the interrupt
5) Saves the current bus state on the BusManager's stack
6) Switches to the handler's stack
7) Performs an ordinary **FAR** call to the handler, passing it the Status/ID
8) Switches back to the BusManager's stack
9) Restores the saved bus state
10) Scans for another interrupt; (if found, continues at step 3)
11) Returns to the interrupted DOS routine and enables processor interrupts.

Each interrupt handler has its own stack, which should have been allocated previously. This stack must have sufficient capacity to store the actual parameters and local variables within the interrupt handler as well as those of subsequent functions which it may call. A stack size of 256 bytes is suitable in most applications. This stack is not where the C compiler expects it to be, so the interrupt handler must be compiled using the following flags:

`/Gs` Turn off stack checking. Without this option, the handler will immediately report a stack overflow.

/Au*xx* Tell the compiler that `SS != DS`, and to reload `DS` upon entry. The *xx* signifies the desired memory model, as described in the following table.

Model	**Flag**	**Address** *size*
Small	/Ausn	Near data, near code
Medium	/Auln	Near data, far code
Compact	/Ausf	Far data, near code
Large	/Aulf	Far data, far code

Using the /Au*xx* flag means that only a far pointer can take the address of a location or variable on the stack.

If the array for the stack is a **near** array (compiled with the small or medium model, or explicitly declared as such), the /Au*xx* flag is unnecessary, because the BusManager sets `DS` equal to `SS`. In other words, if the array used for the stack has the same segment value as your **near** data, then the BusManager will correctly set the data segment register when entering the handler.

In any case, the handler function itself must be declared **far**, so that the function entry/exit properly matches the way it is called.

Because Microsoft does not supply libraries that match custom memory models, Microsoft "C" library functions cannot be called from the handler. Moreover, DOS is not reentrant so no DOS operations can be used within the handler.

The handler *must* return to the BusManager – that is, **setjmp()** and **longjmp()** constructs are not allowed. However, any BusManager function may be called by the handler. At the very least, most handlers will use **EpcRestState** to reset their device registers.

The following example shows how to set up an interrupt handler:

```
# include "\epconnec\include\busmgr.h"

# ifndef NULL
#  define NULL ((char far *)0)
# endif

# define STKSIZE 256  /* size of intr handler stack */
char MyStack[STKSIZE];  /* interrupt stack */
extern void far MyIntr();  /* interrupt handler */
    ...
/*
 * Set Up Interrupt Handler
 *   Don't worry about previous handler for now
 */
    (void)EpcSetIntr(MY_INTR, MyIntr, &MyStack[STKSIZE], NULL);
        ...
```


The handler for interrupt number **MY_INTR** has been set to the function **MyIntr()** and will be called using **MyStack**. Note that **MyStack** is statically allocated (*not* put on the stack), and that the value passed for the initial stack pointer is the location just beyond the end of the array. The first push will fill the last element of the array, and so on.

For this example, information about the previous handler is not saved – the return value of **EpcSetIntr()** is discarded. The null pointer is specified as the address in which to return the previous stack so it, too, is discarded.

The interrupt handler is compiled separately with the following command:

```
cl /c /Gs /G2 /Ausn myintr.c
```

The interrupt handler code follows:

```
# include "\epconnec\include\busmgr.h"
    ...
extern long MyState;  /* window setting for driver */
extern struct my_device far *MyDev;  /* point to dev regs */
    ...
void far MyIntr(sid)
long sid;
{
    short stat;

    EpcRestState(&MyState);  /* restore window */
    stat = MyDev->status;
        ...
}
```

Note that since the BusManager saves and restores the state in the process of calling and returning from the interrupt handler, there is no need for the handler to save and restore the state.

Resident drivers remain installed for as long as DOS is running; however, program-specific drivers leave memory when the program terminates, so they *must* deassign their interrupt handlers. Your device driver applications *must* deassign their interrupt handlers before they terminate. Otherwise, the memory pointed to by those interrupt handlers will be unassigned or overwritten after the program terminates and the corresponding interrupt will cause the computer to crash.

4

The following code segment shows how to deassign the handler for a program-specific driver:

```
(void) EpcSetIntr(MY_INTR, (void (CDECL FAR *)())NULL,
    (char FAR *)NULL, NULL);
```

Setting a null interrupt handler causes an internal do-nothing handler to be set and the interrupt to be disabled. This is preferable to a simple **EpcDisIntr** because it sets the handler address to a "safe" value.

4.3.6 Writing Device Drivers In Turbo C

The Borland Turbo "C" EPConnect interface provides access to all BusManager functions. This section is designed for use by readers experienced in writing drivers and interrupt code and familiar with the Turbo "C" (version 1.5 or 2.0) compiler, linker, and (where necessary) assembler.

Using the Turbo "C" EPConnect Interface

To use EPConnect functions in a driver, include the appropriate header files in the modules in which the functions are used, and link your driver object files with the library files. The header files contain function prototypes, structure definitions, and constants associated with the EPConnect BusManager functions. (See the section *Programming Interface* for a description of the EPConnect definition files.)

Turbo "C" programs *must not* be compiled with the "-A" option, which forces strict ANSI compatibility – the EPConnect interface library uses Pascal calling conventions, which are disabled by this flag.

Each interrupt handler has its own stack, which should have been allocated previously. This stack must have sufficient capacity to store the actual parameters and local variables within the interrupt handler as well as those of subsequent functions which it may call. A stack size of 256 bytes is suitable in most applications. This stack is not where the "C" compiler expects it to be, so you must take the following steps:

- Compile your program with the `-ml` flag, specifying the large memory model. This tells the compiler that `SS != DS` and specifies a **far** entry point. (For speed, individual arrays may be typed **near**.)
- Let the following two lines be the first executable statements in your interrupt handler:

  ```
  asm mov ax,DGROUP
  asm mov ds,ax
  ```

  These lines reload the data segment register with the environment in which the program was linked, allowing access to string constants and global variables.

  Note: Initialization of automatic variables (as in `int a = j+1;`) constitutes executable statements, and cannot precede the `asm` statements.

Most Turbo "C" library routines are not reentrant, and reentrancy bugs are difficult to track down, so you are advised not to call library functions from your handler. Moreover, DOS is not reentrant, so no DOS operations can be used within the handler.

The handler *must* return to the BusManager – that is, **setjmp()** and **longjmp()** constructs are not allowed. However, any BusManager function may be called by the handler. At the very least, most handlers will use **EpcRestState** to reset their device registers.

The following example shows how to set up an interrupt handler:

```
# include "\epconnec\include\busmgr.h"

# ifndef NULL
#  define NULL ((char far *)0)
# endif

# define STKSIZE 256  /* size of intr handler stack */
char MyStack[STKSIZE];  /* interrupt stack */
extern void far MyIntr();  /* interrupt handler */
    ...
/*
 * Set Up Interrupt Handler
 *   Don't worry about previous handler for now
 */
    (void)EpcSetIntr(MY_INTR, MyIntr, &MyStack[STKSIZE], NULL);
        ...
```

The handler for interrupt number **MY_INTR** has been set to the function **MyIntr()** and will be called using **MyStack**. Note that **MyStack** is statically allocated (*not* put on the stack), and that the value passed for the initial stack pointer is the location just beyond the end of the array. The first push will fill the last element of the array, and so on.

For this example, information about the previous handler is not saved – the return value of **EpcSetIntr()** is discarded. The null pointer is specified as the address in which to return the previous stack so it, too, is discarded.

The interrupt handler code follows:


```
# include "\epconnec\include\busmgr.h"
    ...
extern long MyState;  /* window setting for driver */
extern struct my_device far *MyDev;  /* point to dev regs */
    ...
void far MyIntr(sid)
long sid;
{
    short stat;

    asm mov ax,DGROUP
    asm mov ds,ax

    EpcRestState(&MyState);   /* restore window */
    stat = MyDev->status;
        ...
}
```

Note that since the BusManager saves and restores the state in the process of calling and returning from the interrupt handler, there is no need for the handler to save and restore the state.

Resident drivers remain installed for as long as DOS is running; however, program-specific drivers leave memory when the program terminates, so they *must* deassign their interrupt handlers. Your device driver applications *must* deassign their interrupt handlers before they terminate. Otherwise, the memory pointed to by those interrupt handlers will be unassigned or overwritten after the program terminates and the corresponding interrupt will cause the computer to crash.

The following code segment shows how to deassign the handler for a program-specific driver:

```
(void) EpcSetIntr(MY_INTR, (void (far *)())NULL,
    (char far *)NULL, NULL);
```

Setting a null interrupt handler causes an internal do-nothing handler to be set and the interrupt to be disabled. This is preferable to a simple **EpcDisIntr** because it sets the handler address to a "safe" value.

4.3.7 C Optimization

Under certain circumstances, your "C" compiler may introduce an error into your application. In the following example, variable *vmeptr* points to a 16-bit value that is ANDed with 8000h:

```
int far * vmeptr;
  ...
   EpcSetAmMap( A32SD | BM_MBO, vmeaddress, &vmeptr );
   if ( *vmeptr & 0x8000 ) ...
     ...
```


Some compilers eliminate the and of 00 with the low-order byte of the value pointed to by *vmeptr* (because 0 and any value returns 0). Such compilers generate the following assembly language for the second statement:

```
  ...
les   bx,dword ptr [vmeptr]    ; load es:bx with address of vmeptr
test  byte ptr es:[bx+1],80    ; look only at high byte of vmeptr
  ...
```

This seemingly reasonable optimization has serious implications for hardware that requires full-word accesses to invoke needed side effects.

The EPC hardware allows word and double-word references to VMEbus memory to specify byte order as "big-endian" (Motorola style) or "little-endian" (Intel style). For big-endian references, the hardware swaps the bytes so the application receives them in the right order. In the example just shown, however, the compiler eliminates the comparison of the low-order byte. As a result, no full-word access is made, the byte swapping does not occur, and the wrong byte of **vmeptr* is compared to 0x80. (This optimization also causes an obvious problem for hardware that responds only to full-word access.)

According to the ANSI specification of the "C" language, declaring a variable as *volatile* should prevent the compiler from optimizing memory references; that is, references to memory for *volatile* variables must be made exactly as they are written in the source code. This solution does not always have the desired effect, however. The MS "C" compiler 6.0, for example, generates the assembly language shown for the second statement, even when executed with the /Od flag to disable optimization.

You can avoid these problems altogether by making a temporary version of the value pointed to by *vmeptr* and using the temporary version for the AND and the comparison. Modified in this way, the example code becomes

```
int wordcache;
int far * vmeptr;
  ...

   EpcSetAmMap( A32SD | BM_MBO, vmeaddress, &vmeptr );
   if ( (wordcache = *vmeptr) & 0x8000 ) ...
     ...
```

This solution has been tested successfully for versions 5.1 and 6.0 of the Microsoft "C" compiler.

5. Error Messages

This chapter contains an alphabetic listing of error messages that may be generated by the Bus Manager Device Driver (**BIMGR.SYS**).

The error messages listed in this chapter are system-level errors, not application errors returned by EPConnect function calls. Errors that may be returned by a function call are listed in the description of that function in Chapter 2, *Function Descriptions*.

All error messages appear only on the console.

Accompanying each error message is the probable cause of the error, a suggested action to take to correct the error, and the source of the error.

5

Bad parameter */parameter* -- Missing "=" or ":"

Cause	*Parameter* specified on the BIMGR.SYS installation line of the **CONFIG.SYS** file is incorrectly formatted. **BIMGR.SYS** was not installed.
Corrective Action	Correct *parameter* format (refer to *EPConnect/VXI for DOS and Windows User's Guide* for a list of valid options) and reboot.
Source	**BIMGR.SYS**

Bad value for parameter */parameter* -- should be *valid_value*

Cause	The value of *parameter* on the **BIMGR.SYS** installation line in the **CONFIG.SYS** file is not valid. **BIMGR.SYS** was not installed.
Corrective Action	Change value of *parameter* to *valid_value* and reboot.
Source	**BIMGR.SYS**

5

*** EPConnect BusManager NOT INSTALLED due to configuration errors ***

Cause	One or more parameters on the **BIMGR.SYS** installation line of the **CONFIG.SYS** file is not valid.
Corrective Action	Correct invalid parameter (refer to *EPConnect/VXI for DOS and Windows User's Guide* for a list of valid options) and reboot.
Source	**BIMGR.SYS**

ERROR: Unknown EPC Hardware!

Cause	**BIMGR.SYS** does not recognize the EPC hardware. **BIMGR.SYS** was not installed.
Corrective Action	Verify that **BIMGR.SYS** version supports EPC model number. Install correct **BIMGR.SYS** version, update **CONFIG.SYS** installation line, and reboot.
Source	**BIMGR.SYS**

ERROR: VXI hardware not responding!

Cause	**CONFIG.SYS** tried to load **BIMGR.SYS** on a non-EPC computer, or there is a problem with the VXIbus interface registers on the EPC. **BIMGR.SYS** was not installed.
Corrective Action	Verify the state of the hardware by rebooting the system and checking the EPC power-on self-test (POST) results.
Source	**BIMGR.SYS**

Interrupt Stack Overflow Detected in BusManager *** --Hit CTRL-ALT-DEL to reboot

Cause	**BIMGR.SYS** detected an overflow in the **BIMGR.SYS** stack.
Corrective Action	Correct nesting error in **BIMGR.SYS** calls by user-installed VXIbus interrupt handlers.
Source	**BIMGR.SYS**

Unrecognized flag: */flag_value*

Cause	*Flag_value* specifies an unrecognized **BIMGR.SYS** installation parameter in the **CONFIG.SYS** file. **BIMGR.SYS** was not installed.
Corrective Action	Correct or delete *flag_value* (refer to *EPConnect/VXI for DOS Programmer's Reference* for a list of valid options) and reboot.
Source	**BIMGR.SYS**

6. Support and Service

6.1 In North America

6.1.1 Technical Support

RadiSys maintains a technical support phone line at (503) 646-1800 that is staffed weekdays (except holidays) between 8 AM and 5 PM Pacific time. If you have a problem outside these hours, you can leave a message on voice-mail using the same phone number. You can also request help via electronic mail or by FAX addressed to RadiSys Technical Support. The RadiSys FAX number is (503) 646-1850. The RadiSys E-mail address on the Internet is *support@radisys.com*. If you are sending E-mail or a FAX, please include information on both the hardware and software being used and a detailed description of the problem, specifically how the problem can be reproduced. We will respond by E-mail, phone or FAX by the next business day.

Technical Support Services are designed for customers who have purchased their products from RadiSys or a sales representative. If your RadiSys product is part of a piece of OEM equipment, or was integrated by someone else as part of a system, support will be better provided by the OEM or system vendor that did the integration and understands the final product and environment.

6.1.2 Bulletin Board

RadiSys operates an electronic bulletin board (BBS) 24 hours per day to provide access to the latest drivers, software updates and other information. The bulletin board is not monitored regularly, so if you need a fast response please use the telephone or FAX numbers listed above.

The BBS operates at up to 14400 baud. Connect using standard settings of eight data bits, no parity, and one stop bit (8, N, 1). The telephone number is (503) 646-8290.

6.2 Other Countries

Contact the sales organization from which you purchased your RadiSys product for service and support.

Index

I

I

SICL for DOS

Programmer's

Reference Guide

RadiSys® Corporation

15025 S.W. Koll Parkway

Beaverton, OR 97006

Phone: (503) 646-1800

FAX: (503) 646-1850

07-0139-02 December 1994

October 1992

Software License and Warranty

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THE DISKETTE OR DISK UNIT PACKAGE. BY OPENING THE PACKAGE, YOU INDICATE THAT YOU ACCEPT THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, YOU SHOULD PROMPTLY RETURN THE UNOPENED PACKAGE, AND YOU WILL BE REFUNDED.

LICENSE

You may:

1. Use the product on a single computer;
2. Copy the product into any machine-readable or printed form for backup or modification purposes in support of your use of the product on a single computer;
3. Modify the product or merge it into another program for your use on the single computer—any portion of this product merged into another program will continue to be subject to the terms and conditions of this agreement;
4. Transfer the product and license to another party if the other party agrees to accept the terms and conditions of this agreement—if you transfer the product, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copy not transferred, including all modified versions and portions of the product contained in or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification, or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PRODUCT OR ANY COPY, MODIFICATION, OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION, OR MERGED PORTION OF THE PRODUCT TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM

The license is effective until terminated. You may terminate it at any time by destroying the product and all copies, modifications, and merged portions in any form. The license will also terminate upon conditions set forth elsewhere in this agreement or if you fail to comply with any of the terms or conditions of this agreement. You agree upon such termination to destroy the product and all copies, modifications, and merged portions in any form.

LIMITED WARRANTY

RadiSys Corporation ("RadiSys") warrants that the product will perform in substantial compliance with the documentation provided. However, RadiSys does not warrant that the functions contained in the product will meet your requirements or that the operation of the product will be uninterrupted or error-free.

RadiSys warrants the diskette(s) on which the product is furnished to be free of defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment to you.

LIMITATIONS OF REMEDIES

RadiSys' entire liability shall be the replacement of any diskette that does not meet RadiSys' limited warranty (above) and that is returned to RadiSys.

IN NO EVENT WILL RADISYS BE LIABLE FOR ANY DAMAGES, INCLUDING LOST PROFITS OR SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE PRODUCT EVEN IF RADISYS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

GENERAL

You may not sublicense the product or assign or transfer the license, except as expressly provided for in this agreement. Any attempt to otherwise sublicense, assign, or transfer any of the rights, duties, or obligations hereunder is void.

This agreement will be governed by the laws of the state of Oregon.

If you have any questions regarding this agreement, please contact RadiSys by writing to RadiSys Corporation, 15025 SW Koll Parkway, Beaverton, Oregon 97006.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATION BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

NOTES

Table of Contents

1

1. Introducing SICL for DOS

This manual is intended for programmers using the SICL for DOS programming interface to develop applications that control I/O modules via the VXI expansion interface on an EPC. You are expected to have read the *EPConnect/VXI for DOS & Windows User's Guide* for an understanding of what is in EPConnect/VXI, how to configure it with DOS, and how to use the Start-Up Resource Manager (SURM). You are not expected to have in-depth knowledge of DOS.

This chapter introduces you to the RadiSys® Standard Instrument Control Library (SICL) for DOS. In it you will find the following:

- What is in this manual and how to use it
- What is SICL for DOS?
- Programming, Compiling and Linking
- What to do next

1.2 How This Manual is Organized

This manual has five chapters:

Chapter 1, *Introduction*, introduces SICL for DOS and this manual.

Chapter 2, *Function Descriptions*, describes the major categories of SICL function calls and gives complete descriptions of each SICL library function call. The function call descriptions also contain a supporting example or a reference to an example that demonstrates use of the function call. Function call descriptions are alphabetic by function names.

Chapter 3, *Advanced Topics*, provides information for the advanced application developer.

Chapter 4, *Error Messages,* contains an alphabetic listing of error messages generated by SICL.

Chapter 5, *Support and Service*, describes how to contact RadiSys Technical Support.

1.2 What is SICL For DOS?

SICL for DOS is the RadiSys implementation of the SICL standard as defined by Hewlett Packard. It is a runtime library for use by C programmers that are developing portable instrument control applications that run on a RadiSys VXIbus Embedded Personal Computer (EPC®). SICL for DOS (referred to as SICL in this manual) is written for use with and supports only ANSI standard C/C++ compilers (for example, Microsoft C/C++ and Borland C/C++).

The library contains functions that allow DOS-based applications running on a VXIbus embedded controller to control VXIbus instruments or General Purpose Interface Bus (GPIB) instruments. An instrument control connection is called a session. Sessions can be to a single instrument (device) or to all instruments (interface) and must be on one bus, VXIbus or GPIB. The maximum number of open sessions is 512, 256 for VXIbus and 256 for GPIB.

SICL functions allow C/C++ programmers to take full advantage of the connected instrument capabilities, including:

- Sending and receiving messages.
- Requesting a status byte from a device.
- Receiving asynchronous service requests (SRQ) from devices.
- Clearing a device or interface.
- Locking and unlocking devices and interfaces.
- Controlling time-outs.
- Controlling interrupt, service request (SRQ), and error handling.
- Using symbolic names for devices and interfaces.
- Formatted and unformatted I/O.
- Bus mapping and copy functions
- Register based command messages

1.2.1 Conformance to the SICL Standard

The RadiSys implementation of SICL for DOS conforms to revision 3.5 of the Hewlett Packard SICL standard. This implementation supports level 2F: device and interface sessions for both non-formatted and formatted I/O. This implementation of SICL does not support communications with commanders.

1.2.2 Portability

Applications written using SICL easily port to other environments with little or no change, as long as the new environment supports an equivalent level of the SICL standard.

1.2.3 Transparency

SICL defines one consistent interface for communicating with both VXIbus and GPIB devices. In addition, SICL supports symbolic naming of devices and interfaces. These features allow applications that communicate with one instrument on one interface (VXI or GPIB) to communicate with an equivalent instrument on the other interface without program modification or recompilation.

1.2.4 SICL for DOS Architecture

Figure 1-1 is a diagram of the SICL for DOS software architecture that shows how the architecture relates to the VXI hardware and where SICL fits in the architecture. User-written DOS and Windows™ applications can access the VXI hardware using the Bus Management Library or by using a user-written driver.

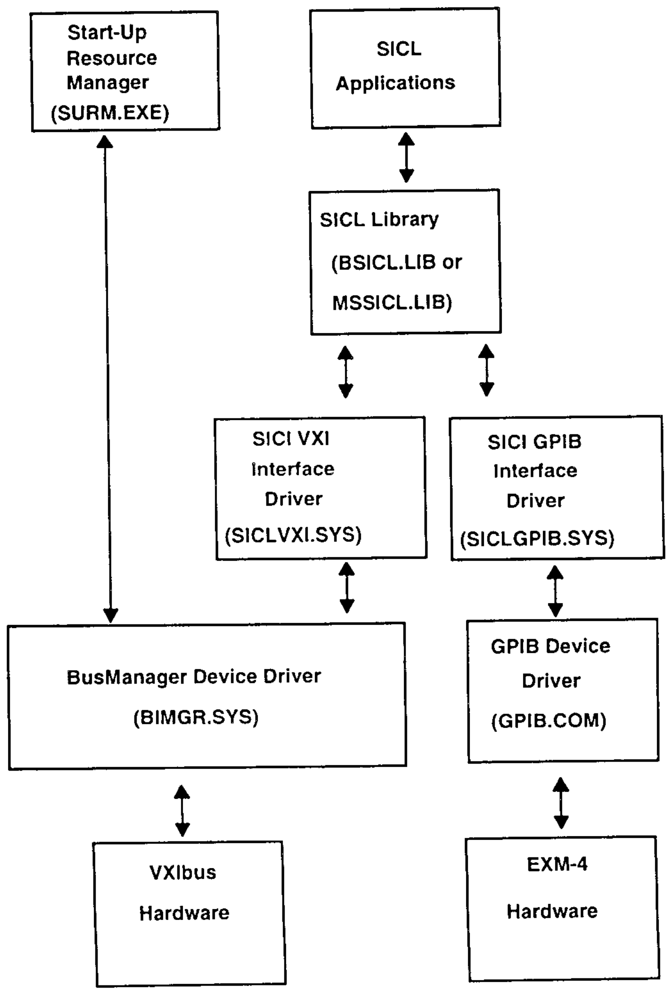

Figure 1-1. SICL for DOS Software Architecture

1.2.5 SICL VXI Interface Driver and BusManager Device Driver

The SICL VXI interface driver and the BusManager device driver provide VXI-interface specific and hardware-specific support to SICL.

1.2.6 SICL GPIB Interface Driver and GPIB Device Driver

The SICL GPIB interface driver and the GPIB device driver provide GPIB-interface specific and hardware-specific support to SICL.

1.2.7 SICL

The SICL interface is independent of the operating system, the hardware platform, and the communication interface. Programs that use SICL port easily to another controller platform as long as the new platform also uses a compatible SICL library. Portability is both at the source code level and at the interface level. Programs written to communicate with an instrument on a given interface can be used to communicate with an equivalent instrument on another interface without modification.

1.2.8 SURM

The Start-Up Resource Manager (SURM) determines the physical content of the system and configures the devices. It is typically the first program to run after DOS boots. The SURM is the EPConnect implementation of the resource manager defined in the VXIbus specification. However, SURM extends the specification definition to include non-VXIbus devices, such as GPIB instruments. The SURM uses the **DEVICES** file to obtain device information not directly available from the devices. SURM accesses VXIbus devices in the system directly.

1.3 Programming, Compiling and Linking

This section contains information about programming with SICL for DOS. Included is a list of the header files provided, the programming interfaces, and compiling and linking hints.

1.3.1 Header File

The **SICL.H** header file contains constants, type definitions, macros, and function prototypes for all SICL functions. It also contains an include directive for the EPConnect header file **EPCSTD.H**.

Figure 1-2 shows the structure of **SICL.H**. It contains two sections: one defining standard constants, structures, and functions and another defining non-standard constants, structures, and functions.

```
#ifndef SICL_H
#define SICL_H
...body of the standard header file...

#ifndef STD_SICL

...body of non-standard header file...
#endif  /* STD_SICL */

#endif  /* SICL_H */
```

Figure 1-2. Default SICL.H File

An **#if/#endif** pair surrounds the contents of the **SICL.H** header file so that you can include the file multiple times without causing compiler errors.

The include file also contains **extern "C"{}** bracketing for the C++ compiler. Because **extern "C"** is strictly a C++ keyword, it is also bracketed and only visible when compiling under C++ and not standard C. If your compiler does not define the __CPLUSPLUS manifest constant or Borland's __TCPLUSPLUS or BCPLUSPLUS manifest constants, you are required to bracket the **SICL.H** and **EPCSTD.H** files with extern "C" when compiling C++ SICL programs.

1.3.2 Compiling and Linking SICL for DOS Applications

NOTE: For specific compiler and/or linker options, refer to your vendor's documentation.

The following examples assume that EPConnect software has been installed in the **C:\EPCONNEC** directory.

When compiling SICL applications, ensure that **SICL.H** and **EPCSTD.H** are in the compiler search path by doing one of the following:

1. Specify the entire file pathname when including the header file in the source file.

2. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path at compiler invocation time.

3. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path environment variable.

When linking a SICL for DOS application, the link must include the appropriate SICL library files. For Microsoft C/C++ compilers, the SICL library is **MSSICL.LIB** and for Borland C/C++ compilers, the SICL library is **BSICL.LIB**. In addition, you must also specify the low-level EPConnect library (i.e., **EPCMSC.LIB**).

Ensure that either **MSSICL.LIB** or **BSICL.LIB** and **EPCMSC.LIB** are in the linker search path by doing one of the following:

1. Specify the entire file pathname on the linker command line.

2. Specify **C:\EPCONNEC\LIB** as part of the linker library search path.

1.4 What to do Next

Follow these instructions to begin creating SICL for DOS applications:

1. If SICL is not pre-installed on your system, install and configure the SICL library using the procedures in Chapter 2 of the *EPConnect/VXI for DOS & Windows User's Guide*.

2. If necessary, refer to the error messages in Chapter 4 of this manual for corrective action information about device driver installation errors.

3. Use the function descriptions in Chapter 2 of this manual for details about a function and/or its parameters to develop applications. Most functions have accompanying examples that demonstrate the function's use.

2. Function Descriptions

2

This chapter lists the SICL functions by category and by name. It is for the programmer who needs a particular fact, such as what function performs a specific task or what a function's arguments are.

The first section lists the functions categorically by the task each performs. It also gives you a brief description of what each function does. The second section lists the functions alphabetically and describes each function in detail.

2.1 Functions by Category

The categorical listing provides an overview of the operations performed by the SICL functions. Included with each category is a description of the operations performed, a listing of the functions in the category, and a brief description of each function.

The categories of the library routines include:

- Session Handling
- Formatted I/O
- Unformatted I/O
- Asynchronous Event Control
- Memory Mapping
- Memory Mapped I/O
- Error Handling
- Locking
- Device and Interface Control
- VXI Interface Control
- GPIB Interface Control

2.1.1 Session Handling

Session handling category functions open sessions, get information about sessions, and close sessions. The category includes these functions:

iclose	Closes a session.
igetaddr	Gets a pointer to the session's address string.
igetdata	Gets a pointer to a session's application data structure.
igetdevaddr	Gets a device address.
igetintftype	Gets a session's interface type.
igetlu	Gets a session's logical unit.
igetsesstype	Gets a session's type
igettimeout	Gets a session's current timeout value.
iopen	Opens a session.
isetdata	Stores a pointer to the session data structure.
itimeout	Set a session's timeout value.

2.1.2 Formatted I/O

Formatted I/O eliminates the need to convert internal C types to types understood by the device or interface. Format strings in the **iprintf**, **ipromptf**, and **iscanf** functions direct formatting and conversion. These format strings are similar to format strings found in standard C **printf** and **scanf** functions. All formatting and conversion operations are compatible with IEEE 488.2 style character and number formats. Formatted I/O operations also use buffers to queue characters into large blocks to improve performance.

Do not mix the formatted I/O functions with unformatted I/O calls within a session.

The **iprintf** function and the write portion of the **ipromptf** function use the write buffer. When the write buffer is full or when it receives an END-bit character it is flushed (its contents is sent to the device). It also flushes immediately after the write portion of an **ipromptf** call.

The **iscanf** function and the read portion of the **ipromptf** function use the read buffer. The read buffer flushes (discards its contents) automatically before the write portion of an **ipromptf** call.

The functions **iflush** and **isetbuf** control read/write buffer operations.

The formatted I/O category functions include:

iflush	Flushes the read and/or write formatted I/O buffers.
iprintf	Formats and writes data to a device or interface.
ipromptf	Sends formatted data to and reads formatted data from a device or interface.
iscanf	Reads and formats data from a device or interface.
isetbuf	Sets the size of the formatted I/O read and/or write buffers.

2.1.3 Unformatted I/O

Unformatted I/O provides a method to send and receive arbitrary blocks of data to and from a device. No formatting or conversion is performed. Using unformatted I/O provides the greatest control when accessing a system device. Do not mix the unformatted I/O functions with formatted I/O calls within a session. The unformatted I/O category functions include:

igettermchr	Gets a session's current termination character.
inbread	Reads data from a device or interface without blocking.
inbwrite	Writes data to a device or interface without blocking.
iread	Reads data from a device or interface.
itermchr	Specifies a session's termination character.
iwrite	Writes data to a device or interface.

2.1.4 Asynchronous Event Control

An asynchronous event is an event that can occur anytime during the execution of a program. In SICL, an asynchronous event occurs when a SRQ occurs or an enabled interrupt occurs. The executing handler identifies the event's source. The asynchronous event control category functions include:

igetonintr	Queries the session's current interrupt handler.
igetonsrq	Queries the session's current service request (SRQ) handler.
iintroff	Disables SRQ and interrupt event processing.
iintron	Enables processing of SRQ and interrupt events.
ionintr	Installs a session's interrupt handler.
ionsrq	Installs a service request (SRQ) handler.
isetintr	Enables and disables interrupt reception.
iwaithdlr	Waits for a SRQ or interrupt handler function to execute.

2.1.5 Memory Mapping

The memory mapping functions map a subset of memory space into the user's address space, free user memory when the space is no longer needed, and get memory space mapping information. Memory mapping category functions include:

imap	Maps a portion of a VXIbus address space into user memory space.
imapinfo	Queries address space mapping capabilities for the specified interface.
iunmap	Deletes an address space mapping.

2.1.6 Memory Mapped I/O

The memory mapped I/O functions copy bytes, words, and longwords from one location to another. The locations can be either a sequence of memory locations or a FIFO register. The memory mapped I/O functions include:

ibblockcopy	Copies bytes from one set of sequential memory locations to another.
ibpeek	Reads a byte stored at a mapped address.
ibpoke	Writes a byte to a mapped address.
ibpopfifo	Copies bytes from a single memory location (FIFO register) to sequential memory locations.
ibpushfifo	Copies bytes from sequential memory locations to a single memory location (FIFO register).
ilblockcopy	Copies a block of 32-bit words from one set of sequential memory locations to another.
ilpeek	Reads a 32-bit word stored at a mapped address.
ilpoke	Writes a 32-bit word to a mapped address.
ilpopfifo	Copies 32-bit words from a single memory location (FIFO register) to sequential memory locations.
ilpushfifo	Copies 32-bits words from sequential memory locations to a single memory location (FIFO register).

iwblockcopy	Copies blocks of 16-bit words from one set of sequential memory locations to another.
iwpeek	Reads a 16-bit word stored at an address.
iwpoke	Writes a 16-bit word to an address.
iwpopfifo	Copies 16-bit words from a single memory location (FIFO register) to sequential memory locations.
iwpushfifo	Copies 16-bits words from sequential memory locations to a single memory location (FIFO register).

2.1.7 Error Handling

Many of the SICL functions can generate errors. Errors usually return a special value (a null pointer or a non-zero error code) to indicate the error. In addition, the application program can designate a procedure to execute when an error occurs. The error handling category functions include these functions:

icauseerr	Set a process' most recent error number.
igeterrno	Gets an error number.
igeterrstr	Gets an error string.
igetonerror	Queries the current error handler.
ionerror	Installs an error handler.

2.1.8 Locking

A device or interface can be locked by a process to prevent access by another process. Locking is useful when multiple processes attempt simultaneous device or interface access. A locked device or interface can cause the accessing process to suspend or generate an error. The locking category functions include:

igetlockwait	Gets a session's current lock-wait flag.
ilock	Locks a device or interface.
isetlockwait	Determines whether accessing a locked device or interface suspends the calling process or generates an error.
iunlock	Unlocks a device or interface.

2.1.9 Device and Interface Control

The device and interface control category contains functions that direct operations common to devices and interfaces. It also contains functions that set local and remote operation of devices. The device and interface control category functions include:

iclear	Clears a device or an interface.
ihint	Defines the type of communication a device driver should use.
ilocal	Puts a device in local mode.
ireadstb	Reads the status byte from a device.
iremote	Puts a device in remote mode.
isetstb	Sets this controller's status byte.
itrigger	Sends a trigger to a device or interface.
ixtrig	Asserts and deasserts one or more triggers to an interface.

2.1.10 VXI Interface

The VXI functions control a VXI interface and includes these functions:

ivxibusstatus	Gets the VXI bus status.
ivxigettrigroute	Gets the current trigger routing.
ivxirminfo	Gets VXI device information.
ivxiservants	Gets a list of VXI servants.
ivxitrigoff	Deasserts VXIbus trigger lines.
ivxitrigon	Asserts VXIbus trigger lines.
ivxitrigroute	Routes VXIbus trigger lines.
ivxiwaitnormop	Waits for a normal operation of a VXI interface.
ivxiws	Sends a word-serial command to a VXI device.

2.1.11 GPIB Interface

The GPIB interface functions control a GPIB interface and includes these functions:

igpibatnctl	Controls the state of the ATN line during GPIB writes.
igpibbusstatus	Gets GPIB status.
igpibllo	Puts all GPIB devices into local-lockout mode.
igpibpassctl	Passes active controller status to another GPIB interface.
igpibppoll	Executes a parallel poll.
igpibppollconfig	Configures a GPIB device's response to a parallel poll.
igpibrenctl	Controls the state of the GPIB REN line.
igpibsendcmd	Writes command bytes to a GPIB interface.

2.2 Functions by Name

This section contains an alphabetical listing of the SICL library functions. Each listing describes the function, gives its invocation sequence and arguments, discusses its operation, and lists its returned values. Where usage of the function may not be clear, an example with comments is given. Each function description begins on a new page.

2

ibblockcopy

Description

Copies bytes from one set of sequential memory locations to another.

int PASCAL
ibblockcopy(INST *id*, **unsigned char** **src*, **unsigned char** **dest*,
unsigned long *count*);

id	Pointer to a session structure.
src	Source address.
dest	Destination address.
count	Number of bytes to copy.

Remarks

This function copies bytes from successive memory locations beginning at *src* into successive memory locations beginning at *dest*. *Count* specifies the number of data bytes to transfer and has a maximum value of 0x10000. *Id* identifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Src* and/or *dest* is null.

See Also ibpeek, ibpoke, ibpopfifo, ibpushfifo, ilblockcopy, imap, iwblockcopy

Example


```
/*
//      This example uses ibblockcopy function to read a VXI
//      register of the device configured as ULA 0.  The bit
//      encodings of this register are defined by the VXI
//      specification.  For this particular example, the
//      program is using the Device class bits.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define     VXIREGISTEROFFSET       0xc000

void main(void)
{   INST instance;
    char *vxiregisters;
    int returncode, errornumber;
    char deviceclass;
    char *dclass[] = { "Memory",
                  "Extended",
                  "Message Based",
                  "Register Based" };
    char *sessionname = "vxi";

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /* Map in A16 space */
    vxiregisters = imap(instance,I_MAP_A16,0,0,NULL);
    if (vxiregisters == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
```

2

```
returncode = ibblockcopy(instance,
                        (unsigned char *)
                        ((unsigned long) vxiregisters +
                        (unsigned long) VXIREGISTEROFFSET),
                        (unsigned char *) &deviceclass,
                        1L);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to copy ID register, error = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(3);
    }
    fprintf(stdout,
            "Class of device at ULA 0 is %s.",
             dclass[(deviceclass >> 6) & 0x3]);
    exit(0);
}
```

ibpeek

Description

Reads a byte stored at a mapped address.

volatile unsigned char PASCAL ibpeek(volatile unsigned char **addr*)**;**

addr Address of byte.

Remarks

The *addr* pointer should be a mapped pointer returned by a previous **imap** call.

Return Value

The function returns the 8-bit value stored at *addr*.

See Also

ibpoke, ilpeek, imap, iwpeek

Example

```
/*
//      This example uses ibpeek to read a VXI
//      register of the device configured as ULA 0.  The bit
//      encodings of this register are defined by the VXI
//      specification.  For this particular example, the
//      program is using the Address space bits.
*/

#include <stdlib.h>
#include <stdio.h>
#include "sicl.h"


void main(void)
{   INST instance;
    int errornumber;
    char *vxiregisters;
    unsigned char addressspace;
    char *deviceclass[] = { "A16/A24",
                         "A16/A32",
                         "RESERVED",
                         "A16 Only" };
    char *sessionname = "vxi";
```

2

```
    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /* Map in A16 space */
    vxiregisters = imap(instance,I_MAP_A16,0,0,NULL);
    if (vxiregisters == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    addressspace = (unsigned char) ((ibpeek((unsigned char *)
        ((unsigned long) vxiregisters + 0xC000L))
        & 0x30) >> 4);
    fprintf(stdout,
        "Address space of device at ULA 0 is %s.",
        deviceclass[addressspace & 0x03]);
    exit(0);
}
```

ibpoke

Description Writes a byte to a mapped address.

void PASCAL
ibpoke(volatile unsigned char ***dest*****, unsigned char** *value***);**

dest	Destination address.
value	Byte to write.

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call.

Return Value The function returns no value.

See Also **ibpeek, ilpoke, imap, iwpoke**

Example

```
/*
//      This example uses ibpoke to write to the VXI
//      register of the device configured as ULA 0.  For this
//      particular example, the program assumes the device
//      is an EPC.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    char *vxiregisters;
    int errornumber;
    char *sessionname = "vxi";
    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /* Map in A16 space */
    vxiregisters = imap(instance,I_MAP_A16,0,0,NULL);
    if (vxiregisters == NULL) {
        errornumber = igeterrno();
```

2

```
        fprintf(stderr,
            "\tUnable to map in A16 space, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    vxiregisters += 0xc005;
    /*
    //  Clearing the high bit of the VXI Status/control register
    //  causes the EPC-7 to ignore A32 accesses.
    */
    ibpoke(vxiregisters,(unsigned char) (ibpeek(vxiregisters) &
        ~0x80));
    exit(0);
}
```

ibpopfifo

Description

Copies bytes from a single memory location (FIFO register) to sequential memory locations.

int PASCAL
ibpopfifo(INST *id*, **unsigned char** **fifo*, **unsigned char** **dest*,
unsigned long *count*);

id	Pointer to a session structure.
fifo	FIFO pointer.
dest	Destination address.
count	Number of bytes to copy.

Remarks

This function copies *count* bytes from *fifo* into successive memory locations beginning at *dest*. *Count* specifies the number of data bytes to transfer and has a maximum value of 0x10000. *Id* identifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Fifo* and/or *dest* is null.

See Also ibpushfifo, ilpopfifo, imap, iwpopfifo

2

Example

```
/*
//      This example uses ibpopfifo to read from a
//      hypothetical VXI fifo at offset 0.
*/

#include <stdlib.h>
#include <stdio.h>
#include "sicl.h"

void main(void)
{   INST instance;
    unsigned char *vxi;
    int returncode, errornumber;
    unsigned char datafifo[5];
    char *sessionname = "vxi";

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxi = (unsigned char *) imap(instance,I_MAP_A16,0,0,NULL);
    if (vxi == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = ");
        fprintf(stderr,
            "%s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    /*
    // Read the Fifo 5 times, storing the values into datafifo[]
    */
    returncode = ibpopfifo(instance,vxi,datafifo,
        (long) sizeof(datafifo));
```

```
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to read the fifo at address ");
        fprintf(stderr,
            "%p\n\r\tError = %s (%d) \n\r",
            vxi,
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    exit(0);
}
```

ibpushfifo

Description

Copies bytes from sequential memory locations to a single memory location (FIFO register).

int PASCAL
ibpushfifo(INST ***id*****, unsigned char ****src*****, unsigned char ****fifo*****,**
unsigned long ***count*****);**

id	Pointer to a session structure.
src	Source address.
fifo	FIFO pointer.
count	Number of bytes to copy.

Remarks

This function copies *count* bytes from the sequential memory locations beginning at *src* into the FIFO at *fifo*. *Count* specifies the number of data bytes to transfer and has a maximum value of 0x10000. *Id* specifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Src* and/or *fifo* is null.

See Also **ibpopfifo, ilpushfifo, imap, iwpushfifo**

Example

```
/*
//      This example uses ibpushfifo to write values
//      to a hypothetical VXI fifo at offset 0.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define     VXIREGISTEROFFSET       0xc000

void main(void)
{   INST instance;
    char *vxi;
    int returncode, errornumber;
    unsigned char datafifo[] = { 0xf1, 0xf2, 0xf3, 0xf4, 0xf5 };
    char *sessionname = "vxi";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxi = imap(instance,I_MAP_A16,0,0,NULL);  /* Map in A16 space */
    if (vxi == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = ");
        fprintf(stderr,
            "%s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    /*
```

2

2

```
    // Write to the fifo 5 times, storing 0xf1, 0xf2, 0xf3,
    // 0xf4 and 0xf5.
    */
    returncode = ibpushfifo(instance,
                      (unsigned char *) vxi,
                      datafifo,
                      (unsigned long) sizeof(datafifo));
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to write to the fifo at address ");
        fprintf(stderr,
            "%p\n\r\tError = %s (%d) \n\r",
            vxi,
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    exit(0);
}
```

icauseerr

Description

Set a process' most recent error number.

void PASCAL
icauseerr(INST *instance*, **int** *error*, **int** *callhandler*);

instance	A pointer to a session structure.
error	An error number.
callhandler	A flag indicating whether or not to call the process' currently installed error handler.

Remarks

The function sets the process' most recent error number to *error* for creating user defined errors. If *error* is not **I_ERR_NOERROR** and *callhandler* is non-zero and the process has an error handler installed, the function also calls the installed error handler. A process' most recent error number can be queried using **igeterrno**. A process' error handler can be set using **ionerror** and queried using **igetonerror**.

Return Value

The function does not return a value.

See Also

igeterrno, **igeterrstr**, **igetonerror**, **ionerror**

Example

See **igetonerr**.

2

iclear

Description

Clears a device or an interface.

int PASCAL
iclear(INST *id*);

id Pointer to a session structure.

Remarks

For VXI device sessions, the function issues a DEVICE CLEAR word-serial command to the device. Only message based VXI devices are supported. Other VXI devices cause an error.

For VXI interface sessions, the function issues a SYSRESET signal (SYSRESET is pulsed).

For GPIB device sessions, the function issues a device clear command to the device.

For GPIB interface sessions, the function issues an interface clear signal (IFC is pulsed).

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies an interface or commander session or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iclose, iopen, itimeout**

Example

```
/*
//      Call iclear() to assert IFC (GPIB).
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "gpib";

    /*
    // Open a GPIB interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
    /* pulse IFC for GPIB interface sessions */
    returncode = iclear(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIclear call failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(4);
    }
    exit(0);
}
```

2

iclose

Description Closes a session.

int PASCAL
iclose(INST *id*);

id	Pointer to a session structure.

Remarks This function invalidates the **INST** handle pointed to by *id*.

An implicit **iclose** occurs for all currently open sessions when an application terminates.

Closing a session releases all resources associated with the session, including locks (if the closing function set the locks), I/O buffers, and address space mappings.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.

See Also **iopen**

Example

```
/*
//     This example uses explicit calls to iclose to
//     release the session's resources.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int *vxiregisters;
    int errornumber;
    char *sessionname = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }

    vxiregisters = (int *) imap(instance,I_MAP_VXIDEV,0,0,NULL);
    if (vxiregisters == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,"\tUnable to map in VXI registers\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    (void) iclose(instance);
    /*
    //    ...
    //
    //    Instance handle no longer valid.  Memory references
    //    via vxiregisters may be undefined.
    */
    exit(0);
}
```

2

iflush

Description

Flushes the read and/or write formatted I/O buffers.

int PASCAL
iflush(INST *id*, **int** *buffermask*);

id	Pointer to a session structure.
buffermask	Selects the buffer(s) to clear.

Remarks

This function clears the read buffer or writes the contents of the **iprintf** and **ipromptf** write buffer. *Buffermask* must be an OR'd combination of the these constants:

Constant	Description
I_BUF_READ	Clears the session read buffer then reads from the device or interface session pointed to by *id* until an **END** indicator is read. Clearing the read buffer ensures that the next call to **iscanf** reads data directly from the device rather than reading data that was previously buffered.
I_BUF_WRITE	Writes all data in the write buffer to the device or interface session pointed to by *id*.

If a specified buffer is empty or has already been flushed, this call has no effect.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iprintf, ipromptf, iscanf, isetbuf, itimeout**

Example

```
/*
//      Use iflush() to explicitly flush the write buffer.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vdev1";

#if !defined(I_SICL_FMTIO)
    fprintf(stderr,
           "\tFormatted I/O is not supported on this
          implementation");
    exit(0);
#endif
```

2

```
    /*
    //  Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = isetbuf(instance,I_BUF_WRITE,100);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to create a 100 byte buffer\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    /*
    //      Write bcc\n to the buffer.   Use -t to prevent an
    //      implicit buffer flush.
    */
    (void) iprintf(instance,"bcc%-t\n");
    returncode = iflush(instance,I_BUF_WRITE);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to flush buffer\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(3);
    }
    exit(0);
}
```

igetaddr

Description

Gets a pointer to the session's address string.

int PASCAL
igetaddr(INST *id*, **char** ***address*);

id	Pointer to a session structure.
address	Pointer to a location where the function stores the session's address string.

Remarks

This function returns a pointer to the session address string of the session pointed to by *id*. The returned address is the address of the session address string passed to **iopen** when it opened the session.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Address* is null.

See Also

iopen

2

2

Example

```
/*
//      Use igetaddr() to get the session name.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionaddress;
    char *sessionname = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igetaddr(instance,&sessionaddress);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to get session's string address\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,"Session address = <%s>",sessionaddress);
    exit(0);
}
```

igetdata

Description

Gets a pointer to a session's application data structure.

int PASCAL
igetdata(INST *id*, **void** ***data*);

id	Pointer to a session structure.
data	Pointer to a location where the function stores the data structure.

Remarks

This function places an application specific data structure to the data structure of the session pointed to by *id* in the address pointed to by *data*. The **isetdata** function establishes the session data structure.

The session data structure is a 4-byte memory block. Its contents are application specific. Typically, it contains a pointer to an application's data structure.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Data* is null.

See Also

isetdata

Example

```
/*
//      Use isetdata()/igetdata() to cache a user pointer
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"
```

2

```
void main(void)
{  INST instance = 0, previousinstance = 0, nextinstance = 0;
    int primary, secondary, returncode, lu, session = 0;
    register int devtype, devnumber;
    char *devtypes[] = { "vdevx", "gdevx" };

    /*
    // Open all device session with names gdev[0-9] and vdev[0-9]
    */
    for (devtype = 0; devtype < 2; devtype++) {
        for (devnumber = 0; devnumber < 10; devnumber++) {
            *(devtypes[devtype] + 4) = (char)
            (((char) devnumber) + (char) '0');
            instance = iopen(devtypes[devtype]);
            /*
            //         Link the sessions together by placing
            //         the instance address into the data
            //         structure address
            */
            if (instance != NULL) {
                if (nextinstance == 0)
                    nextinstance = instance;
                if (previousinstance != 0) {
                    returncode =
                        isetdata(previousinstance,instance);
                    if (returncode != I_ERR_NOERROR) {
                        fprintf(stderr,
                        "\tUnable to set structure address\n\r");
                        fprintf(stderr,
                                "\tError = %s (%d) \n\r",
                                igeterrstr(returncode),
                                 returncode);
                        exit(1);
                    }
                }
            returncode = isetdata(instance,0);
            if (returncode != I_ERR_NOERROR) {
                fprintf(stderr,
                        "\tUnable to set structure address\n\r");
                fprintf(stderr,
                        "\tError = %s+ (%d) \n\r",
                         igeterrstr(returncode),
                         returncode);
                exit(2);
            }
            previousinstance = instance;
            }
        }
    }
```

```
    /*
    //      traverse the session chain
    */
    instance = nextinstance;
    while (instance) {
        returncode = igetdata(instance,&nextinstance);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable to get structure address\n\r");
            fprintf(stderr,
                "\tError = %s (%d) \n\r",
                igeterrstr(returncode),
                returncode);
            exit(3);
            }
        returncode = igetlu(instance,&lu);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable to get logical unit id\n\r");
            fprintf(stderr,
                "\tError = %s (%d) \n\r",
                igeterrstr(returncode),
                returncode);
            exit(4);
        }
        (void) igetdevaddr(instance,&primary,&secondary);
        instance = nextinstance;
        fprintf(stdout,
            "Session %d \tlogical unit = %d ",session++,lu);
        fprintf(stdout,
            "\tprimary address = %d\n\r",
            primary);
    }
    exit(0);
}
```

2

igetdevaddr

Description

Gets a device address.

int PASCAL
igetdevaddr(INST *id*, int **primary*, int **secondary*);

id	Pointer to a device session structure.
primary	Pointer to a location where the function stores the session's primary address.
secondary	Pointer to a location where the function stores the session's secondary address.

Remarks

The function returns the primary address and the secondary address of the session pointed to by *id* in the locations pointed to by *primary* and *secondary*, respectively.

The function is valid only for device sessions.

For VXI devices, *primary* is the device's ULA.

If a secondary address does not exist or the session is for a VXI device, *secondary* is set to −1.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* is an interface or commander session or *primary* and/or *secondary* is null.

See Also

iopen

Example

```c
/*
//      Call igetdevaddr() to obtain the primary and
//      secondary addresses.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, primary, secondary, errornumber;
    char *sessionname = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igetdevaddr(instance,&primary,&secondary);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIgetdevaddr failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,
            "Session <%s> primary address = %d",
            sessionname,
            primary);
    fprintf(stdout,
            ", secondary address = %d\n\r",
            secondary);
    exit(0);
}
```

igeterrno

Description

Gets an error number.

int PASCAL
igeterrno(void);

Return Value

This function returns the error number of the most recently executed SICL function.

See Also

igeterrstr

Example

See **ibblockcopy**.

igeterrstr

Description Gets an error string.

char *PASCAL
igeterrstr(int *error***);**

error	Error number.

Remarks This function returns a pointer to an ASCII string corresponding to the error number specified by *error*.

If passed an invalid error code, the function returns a null string pointer.

See Also **igeterrno**

Example See **ibblockcopy**.

igetintftype

Description Gets a session's interface type.

int PASCAL
igetintftype(INST *id*, **int** **intftype*);

id	Pointer to a interface session structure.
intftype	Pointer to a location where the function stores the interface type.

Remarks This function places the interface type of the session pointed to by *id* in the location pointed to by *intftype*. The following are valid interface type constants:

Constant	Description
I_INTF_GPIB	GPIB interface
I_INTF_VXI	VXI interface

The function is valid only for interface sessions.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Intftype* is null.

See Also **iopen**

Example

```
/*
//      Call igetintftype() to obtain the device session's
//      interface type
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define DIM(x)              (sizeof(x)/sizeof(char *))

char *names[] = { "1", "2", "vdev1", "gdev1"} ;
char *interfacetypes[] = { "I_INTF_GPIB", "I_INTF_VXI" };


void main(void)
{   INST instance;
    int returncode, facetype;
    register short dinductor;

    for (dinductor = 0; dinductor < DIM(names); dinductor++) {
        instance = iopen(names[dinductor]);
        if (instance == NULL) continue;
        returncode = igetintftype(instance,&facetype);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIgetdevaddr call failed\n\r");
            fprintf(stderr,
                "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
            exit(1);
        }
        fprintf(stdout,
           "Session <%s> interface type = \t%s\n\r",
           names[dinductor],
           interfacetypes[facetype]);
    }
    exit(0);
}
```

2

igetlockwait

Description

Gets a session's current lock-wait flag.

int PASCAL
igetlockwait(INST *id*, **int** **waitflag*);

id	Pointer to a session structure.
waitflag	Pointer to the location where the function stores the lock-wait flag.

Remarks

This function places the current state of the lock-wait flag of the session pointed to by *id* in the location pointed to by *waitflag*. The **isetlockwait** function sets the session's lock-wait flag state.

Under DOS, a session's lock-wait flag has no effect. Locking conflicts always generate an **I_ERR_LOCKED** error because DOS does not support process preemption.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	W*aitflag* pointer is null.

See Also

ilock, isetlockwait, iunlock

2

Example

```
/*
//      Call igetlockwait() to obtain the session's
//      wait flag.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{    INST instance;
     int returncode, errornumber, waitflag;
     char *sessionname = "vdev1";

     /*
     // Open a device session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     returncode = igetlockwait(instance,&waitflag);
     if (returncode != I_ERR_NOERROR) {
         fprintf(stderr,
             "\tIgetlockwait call failed\n\r");
         fprintf(stderr,
             "\tError = %s (%d) \n\r",
             igeterrstr(returncode),returncode);
         exit(2);
     }
     fprintf(stdout,"Lockwait flag = %d",waitflag);
     exit(0);
}
```

2

igetlu

Description Gets a session's logical unit.

int PASCAL
igetlu(INST *id*, int **lu*);

id	Pointer to a session structure.
lu	Pointer to the location where the function stores the logical unit.

Remarks This function places the logical unit of the session pointed to by *id* in the location pointed to by *lu*.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Lu* is null.

See Also **iopen**

Example See **igetdata**.

igetonerror

Description Queries the current error handler.

int PASCAL
igetonerror(void (CDECL***errorhandler***)(INST** ***id*****, int** ***error*****));**

errorhandler	Pointer to a location where the function stores the current error handler.

Remarks This function queries the current error handler. The **ionerror** function defines the error handler.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*ErrorHandler* is null.

See Also **ionerror**

Example

```
/*
//      This example uses igetonerror and ionerror
//      to manipulate the error handler.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

volatile short          errordetected   = 0;

#define MYERROR         255
```

2

```
void
console(char *astring)
{   char    achar;

    while (*astring) {
        achar = *astring++;
        ASM
            mov     ah,0eh
            mov     al,achar
            mov     bx,3
            int     010h
        ENDASM
    }
}


void CDECL
myhandler(INST instance,int error)
{   char *sessionaddress;
    char errorstring[9] = {0};

    (void) igetaddr(instance,&sessionaddress);
    /*
    //      we can't use DOS to write in interrupt handlers
    */
    console("Error ");
    itoa(error,errorstring,10);
    console(errorstring);
    console(" detected for ");
    console(sessionaddress);
    console("\n\r");
    errordetected = 1;
}

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vxi";
    void (CDECL *previoushandle)(INST instance,int error);

    /*
    //  Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
                "\tUnable to open <%s>, error = %s (%d)\n\r",
                sessionname,
                igeterrstr(errornumber),errornumber);
        exit(1);
    }
```

```
    /*
    //      Get the previously installed error handler.  (Should be
    //      NULL until set by ionerror).
    */
    returncode = igetonerror(&previoushandle);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
                "\tIgetonerror call failed\n\r");
        fprintf(stderr,
                "\terror = %s (%d) \n\r",
                igeterrstr(errornumber),errornumber);
        exit(2);
    }
    returncode = ionerror(myhandler);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
                "\tIonerror call failed\n\r");
        fprintf(stderr,
                "\terror = %s (%d) \n\r",
                igeterrstr(errornumber),errornumber);
        exit(3);
    }
    /*
    //      The following function should fail.  Only device
    //      sessions can use I_MAP_VXIDEV, this session is an
    //      interface session
    */
    (void) imap(instance,I_MAP_VXIDEV,0,0,NULL);
    if (errordetected != 0)
        fprintf(stdout,
                "Error handler execution successful\n\r");
    else
        fprintf(stdout,
                "Error handler execution unsuccessful\n\r");
    /*
    //      Force a user defined error
    */
    icauseerr(instance,MYERROR,1);
    /*
    //      Deinstall our error handler by restoring the original
    //      handler.  The handler can also be disabled by installing
    //      a NULL handler.
    */
    (void)  ionerror(previoushandle);
    exit(0);
}
```

2

igetonintr

Description Queries the session's current interrupt handler.

int PASCAL
igetonintr(INST *id*, **void (CDECL****intrhandler*)(**INST** *id*, **long**
data1,**long** *data2*);

id	Pointer to a session structure.
intrhandler	Pointer to a location where the function stores the current interrupt handler.

Remarks This function queries the current interrupt handler in use by the device or interface session pointed to by *id*. The **ionintr** function defines a device's interrupt handler.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Intrhandler* is null.

See Also **ionintr**

Example

```
/*
//      This example sets, generates and processes interrupts
//      using igetonintr, ionintr, isetintr and iintron/introff.
*/

#include <stdio.h>
#include <stdlib.h>
#include "busmgr.h"
#include "sicl.h"

/* removes compiler warning message (compiler specific) */
#define REMOVEWARNING(x)   x = x

#define INTERRUPTENABLE    1
#define INTERRUPTDISABLE   0
#define INTERRUPTS         7        /* interrupts 1-7 */
#define WAITTIME           (1000L*30L*1)
#define TIMERINT           8
```

2

```
volatile unsigned long Vmeinterruptcount = 0;
void (INTERRUPT *timerfunction)();
volatile unsigned long Tick = 0;

void
console(char *astring)
{   char    achar;

    while (*astring) {
        achar = *astring++;
        ASM
            mov     ah,0eh
            mov     al,achar
            mov     bx,3
            int     010h
        ENDASM
    }
}

static void
reverse(char s[])       /* K & R -- page 59  */
{   register int i, j;
    int slen;
    char c;

    slen = 0;
    while(s[slen++]);
    for (i = 0, j = slen-2; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

static void
myitoa(long n,char s[])  /* K & R -- page 60   */
{   long i, sign;

    if ((sign = n) < 0) n = -n;
    i = 0;
    do {
        s[(int) (i++)] = (char) ((char) (n % 10) + '0');
    } while ((n /= 10) > 0);
    if (sign < 0) s[(int) (i++)] = (char) '-';
    s[(int) i] = (char) '\0';
    reverse(s);
}

void CDECL
vmehandler(INST instance, long interruptsource, long junk)
{   char abuffer[10];
    char *sessionaddress;

    Vmeinterruptcount++;
    /*
    // Can't use stdio from interrupt handlers.
```

2

```
    */
    console("handler : vmehandler,  Interrupt source <");
    myitoa(interruptsource,abuffer);
    console(abuffer);
    console(">\n\r");
    console("Interrupt <");
    myitoa(Vmeinterruptcount,abuffer);
    console(abuffer);
    console(">\n\r");
    if (igetaddr(instance,&sessionaddress) == I_ERR_NOERROR) {
       console("Session address = <");
       console(sessionaddress);
       console(">\n\r");
    }
    REMOVEWARNING(junk);
}

#if !defined(__TURBOC__)

void INTERRUPT
mytimer()
{    Tick--;
     if (Tick == 0) {
        EpcSigIntr(3);
     }
     Vmeinterruptcount = 1;
     _chain_intr(timerfunction);
}

void
installtimer(void (INTERRUPT *newfunction)(),unsigned short timeout)
{    _disable();
     Tick = 18 * timeout;
     timerfunction = _dos_getvect(TIMERINT);
     _dos_setvect(TIMERINT,newfunction);
     _enable();
}

void
deinstalltimer()
{    _disable();
     _dos_setvect(TIMERINT,timerfunction);
     _enable();
}

#endif

void main(void)
{    INST instance;
     int returncode, errornumber;
     char *sessionname = "vxi";
     register short iinductor;
     void (CDECL *oldhandler)(INST instance,
                        long interruptsource,
                        long junk);

     /*
```

```
// Open a device session
*/
instance = iopen(sessionname);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionname,
        igeterrstr(errornumber),errornumber);
    exit(1);
}
returncode = ionintr(instance,vmehandler);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,
        "\tUnable to set interrupt handler\n\r");
    fprintf(stderr,
        "\terror = %s (%d)\n\r",
        igeterrstr(returncode),returncode);
    exit(2);
}
returncode = isetintr(instance,I_INTR_VXI_VME,INTERRUPTENABLE);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,
        "\tUnable to enable interrupt reception\n\r");
    fprintf(stderr,
        "\terror = %s (%d)\n\r",
        igeterrstr(returncode),returncode);
    exit(3);
}
```

2

```
    /*
    // Cycle through the VME interrupts
    */
    for (iinductor = 0; iinductor <= INTERRUPTS; iinductor++) {
        if (EpcSigIntr(iinductor) != EPC_SUCCESS) {
            fprintf(stderr,
            "\tUnable to generate a VME interrupt\n\r");
            exit(4);
        }
    }
    if (Vmeinterruptcount != INTERRUPTS) {
        fprintf(stderr,
            "\tExpected interrupt processing not detected\n\r");
        exit(5);
    }
#if !defined(__TURBOC__)
    /*
    // Create a new thread to assert a VME interrupt.
    */
    Vmeinterruptcount = 0;
    installtimer(mytimer,15);
    /*
    //     Wait for the completion of one more interrupt handler
    //     invocation
    */
    returncode = iwaithdlr(WAITTIME);
    deinstalltimer();
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIwaithdlr failed\n\r");
        fprintf(stderr,
            "\terror = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(6);
    }
    if (Vmeinterruptcount == 0) {
        fprintf(stderr,
            "\tExpected interrupt processing not detected\n\r");
        exit(7);
    }
#endif
    /*
    //     Keep interrupt processing off while the interrupt
    //     handler is being written
    */
    returncode = iintroff();
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIintroff failed\n\r");
        fprintf(stderr,
            "\terror = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(8);
    }
```

```
    /*
    // Restore the previous interrupt
    */
    returncode = igetonintr(instance,&oldhandler);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable execute igetonintr successfully\n\r");
        fprintf(stderr,
            "\terror = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(9);
    }
    fprintf(stdout,"Interrupt testing successful\n\r");
    exit(0);
}
```

2

igetonsrq

Description

Queries the session's current service request (SRQ) handler.

int PASCAL
igetonsrq(INST *id*, **void (CDECL*********srqhandler*)(**INST** *id*));

id	Pointer to a device session structure.
srqhandler	Pointer to a location where the function stores the current SRQ handler.

Remarks

This function queries the current SRQ handler of the session pointed to by *id*. The function **ionsrq** defines the session's SRQ handler.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies an interface or commander session or *srqhandler* is null.

See Also

ionsrq

Example

```
/*
//      This example sets, generates and processes SRQs.
*/

#define EPC2        1
#include <conio.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include "busmgr.h"
#include "olrm.h"
#include "sicl.h"
#include "vmeregs.h"
```

```
/* remove's compiler warning message (compiler specific) */
#define REMOVEWARNING(x)   x = x


void
console(char *astring)
{   char    achar;

    while (*astring) {
        achar = *astring++;
        ASM
            mov     ah,0eh
            mov     al,achar
            mov     bx,3
            int     010h
        ENDASM
    }
}

void CDECL
srqhandler(INST instance)
{   char *sessionaddress;

    /*
    //  Can't use stdio from srq handlers.
    */
    console("handler : srqhandler\n\r");
    if (igetaddr(instance,&sessionaddress) == I_ERR_NOERROR) {
        console("Session address = <");
        console(sessionaddress);
        console(">\n\r");
    }
}

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vdev1";
    void (CDECL *oldhandler)(INST instance);
    unsigned short ula;

    /*
    //  Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
```

2

2

```
        returncode = ionsrq(instance,srqhandler);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable to set srq handler\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(2);
        }
        /*
        // Queue a REQUEST TRUE event from a servant device.
        */
        ula = OLRMGetNumAttr(sessionname, 0, OLRM_LOG_ADDR);
        if (ula == 0xFFFF ||
            EpcErQue((short) (ula | 0xFD00)) == (short) FALSE) {
            fprintf(stderr,
                "Unable to generate an SRQ_EVENT interrupt\n\r");
            exit(3);
        }
        /*
        //      Keep srq processing off while the handler
        //      is being written
        */
        returncode = iintroff();
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIintroff failed\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(4);
        }
        /*
        // Restore the previous srq handler
        */
        returncode = igetonsrq(instance,&oldhandler);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable execute igetonsrq successfully\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(7);
        }
        fprintf(stdout,"SRQ testing successful\n\r");
        exit(0);
    }
```

igetsesstype

Description

Gets a session's type.

int PASCAL
igetsesstype(INST *id*, **int** **sessiontype*);

id	Pointer to a session structure.
sessiontype	Pointer to the location where the functions stores the session's type.

Remarks

This function places the session type of the session pointed to by *id* in the location pointed to by *sessiontype*. The following are valid *sessiontype* constants:

Constant	Description
I_SESS_DEV	Device session
I_SESS_INTR	Interface session

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Sessiontype* is null.

See Also

iopen

2

Example

```
/*
//      Call igetsesstype() to retrieve the session type
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, sessiontype, errornumber;
    char *sessionname1 = "gdev1";
    char *sessionname2 = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname1);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname1,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igetsesstype(instance,&sessiontype);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIgetsesstype call failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,"Session <%s> type is ",sessionname1);
    if (sessiontype == I_SESS_DEV)
        fprintf(stdout,"<Device session>\n\r");
    else
        fprintf(stdout,"<Interface session>\n\r");
    (void) iclose(instance);
    instance = iopen(sessionname2);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname2,
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
```

```
        returncode = igetsesstype(instance,&sessiontype);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIgetsesstype call failed\n\r");
            fprintf(stderr,
                "\tError = %s (%d) \n\r",
                igeterrstr(returncode),returncode);
            exit(4);
        }
        fprintf(stdout,"Session <%s> type is ",sessionname2);
        if (sessiontype == I_SESS_DEV)
            fprintf(stdout,"<Device session>\n\r");
        else
            fprintf(stdout,"<Interface session>\n\r");
        exit(0);
    }
```

igettermchr

2

Description

Gets a session's current termination character.

int PASCAL
igettermchr(INST *id*, **int** **termchr*);

id	Pointer to a session structure.
termchr	Pointer to a location where the functions stores the current termination character.

Remarks

This function places the current termination character of the session pointed to by *id* in the location pointed to by *termchr*.

The default termination character for a session is −1 (no termination character set). Use **itermchr** to set a termination character.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Termchr* is null.

See Also

inbread, **iread**, **itermchr**

Example

```
/*
//      Call igettermchr() to retrieve the session's
//      termination character.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, termchar, errornumber;
    char *sessionname = "vdevl";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igettermchr(instance,&termchar);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIgettermchr call failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    /*
    //      Default is -1
    */
    if (termchar == -1) {
        returncode = itermchr(instance,(int) '\n');
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tItermchr call failed\n\r");
            fprintf(stderr,
                "\tError = %s (%d) \n\r",
                igeterrstr(returncode),returncode);
            exit(3);
        }
    }
    exit(0);
}
```

igettimeout

Description Gets a session's current timeout value.

int PASCAL
igettimeout(INST *id*, **long** **timeout*);

id	Pointer to a session structure.
timeout	Pointer to a location where the function stores the timeout value.

Remarks This function places the current timeout value of the session pointed to by *id* in the location pointed to by *timeout*. Timeout values are specified in milliseconds.

The default timeout value for a session is 0 (no timeout set). A *timeout* value less than zero also indicates that no timeout is set. Use **itimeout** to set a session timeout value.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Timeout* is null.

See Also **itimeout**

Example

```
/*
//      Call igettimeout() to retrieve the session's
//      timeout character value.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"
```

```
void main(void)
{   INST instance;
    int returncode, errornumber;
    long timeout;
    char *sessionname = "vdevl";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /*
    //  ...
    //
    */
    returncode = igettimeout(instance,&timeout);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIgettimeout call failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    /*
    //     Default value is 0
    */
    if (timeout == 0) {
        /*
        // Set the timeout to 1/2 second
        */
        returncode = itimeout(instance,500L);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tItimeout call failed\n\r");
            fprintf(stderr,
                "\tError = %s (%d) \n\r",
                igeterrstr(returncode),returncode);
            exit(3);
        }
    }
    exit(0);
}
```

2

igpibatnctl

2

Description Controls the state of the ATN line during GPIB writes.

int PASCAL
igpibatnctl(INST *id*, **int** *atnstate*);

id	Pointer to a GPIB interface session structure.
atnstate	ATN line state.

Remarks This function defines the state of the ATN line during future write operations using the GPIB interface session pointed to by *id*. A write operation can occur either directly or indirectly from calls to **iflush**, **inbwrite**, **iprintf**, **ipromptf**, **isetbuf**, and **iwrite**.

This function is valid only for GPIB interface sessions.

Setting *atnstate* equal to zero deasserts the ATN line during subsequent writes. Setting *atnstate* to a non-zero value asserts the ATN line during subsequent writes.

Bytes sent over the GPIB interface when ATN is asserted cause the interface to interpret the bytes as commands. Bytes sent when ATN is deasserted are interpreted as data.

The state of the ATN line is undefined following all other SICL calls.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.

I_ERR_PARAM *Id* specifies a device or commander session.

See Also **iflush, inbwrite, iprintf, ipromptf, isetbuf, iwrite**

Example

```
/*
//     This example uses igpibatnctl to configure the ATL
//     line for commands or data.
*/

#define ATNDATA     0
#define ATNCOMMAND -1


#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionnames = "gpib";

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igpibatnctl(instance,ATNDATA);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibatnctl\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    (void) iprintf(instance,"DATA TEST\n");
    exit(0);
}
```

2

igpibbusstatus

Description Gets GPIB status.

int PASCAL
igpibbusstatus(INST *id*, **int** *request*, **int** **result*);

id	Pointer to a GPIB interface session structure.
request	Status request.
result	Pointer to the location where the stores the GPIB interface status.

Remarks This function places the GPIB interface status requested by *request* in the location pointed to by *result*. The following are valid constants for *request*:

Constant	Description
I_GPIB_BUS_REM	Get the interface remote state (1 = remote, 0 = not remote).
I_GPIB_BUS_SRQ	Get the SRQ state (1 = SRQ asserted, 0 = SRQ not asserted). On an EPC-2 or on an EPC-7 with EXM-4 modules installed, the SRQ line state can be accurately monitored only when the interface is in the active controller state.
I_GPIB_BUS_SYSCTLR	Get the interface system controller state (1 = system controller, 0 = not system controller).
I_GPIB_BUS_ACTCTLR	Get the interface active controller state (1 = active controller, 0 = not active controller).

I_GPIB_BUS_TALKER	Get interface addressed-to-talk state (1 = addressed-to-talk, 0 = not addressed-to-talk).
I_GPIB_BUS_LISTENER	Get interface addressed-to-listen state (1 = addressed-to-listen, 0 = not addressed-to-listen).
I_GPIB_BUS_ADDR	Get the interface primary bus address.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot determine GPIB status.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_NOTSUPP	The hardware/software platform does not support the specified *request*.
I_ERR_PARAM	*Id* specifies a device or commander session, *Request* is invalid, or *result* is null.

See Also **iopen**

Example

```
/*
//      This example calls igpibbusstatus to display
//      the GPIB bus status information.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"
```

2

```
#define DIM(x)              (sizeof(x)/sizeof(int))

int requests[] = { I_GPIB_BUS_REM,
                   I_GPIB_BUS_SRQ,
                   I_GPIB_BUS_SYSCTLR,
                   I_GPIB_BUS_ACTCTLR,
                   I_GPIB_BUS_TALKER,
                   I_GPIB_BUS_LISTENER,
                   I_GPIB_BUS_ADDR };


char *requeststrings[] = {
                  "I_GPIB_BUS_REM",
                  "I_GPIB_BUS_SRQ",
                  "I_GPIB_BUS_SYSCTLR",
                  "I_GPIB_BUS_ACTCTLR",
                  "I_GPIB_BUS_TALKER",
                  "I_GPIB_BUS_LISTENER",
                  "I_GPIB_BUS_ADDR" };


void main(void)
{   INST instance;
    int returncode, errornumber, result;
    char *sessionname = "GPIB";
    register short vinductor;

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
```

```
    for (vinductor = 0; vinductor < DIM(requests); vinductor++) {
        returncode = igpibbusstatus(instance,
                            requests[vinductor],
                            &result);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable to execute igpibbusstatus\n\r");
            fprintf(stderr,
                "\tRequest = %s",
                requeststrings[vinductor]);
            fprintf(stderr,
                "\tError = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(2);
        }
        fprintf(stdout,"%s = \t%d\n\r",
            requeststrings[vinductor],
            result);
    }
    exit(0);
}
```

igpibllo

Description

Puts all GPIB devices into local-lockout mode.

int PASCAL
igpibllo(INST *id*);

id	Pointer to a GPIB interface session structure.

Remarks

This function sends the GPIB LLO (local lockout) command to all devices on the GPIB interface of the session pointed to by *id*.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot execute LLO on the interface.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_PARAM	*Id* specifies a device or commander session.
I_ERR_TIMEOUT	A timeout occurred.

See Also

iopen, itimeout

Example

```
/*
//      This example uses igpibllo to put all GPIB devices
//      into local-lockout mode.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionnames = "gpib";

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /*
    //    None there is no way to automatically verify that the LLO
command
    //    was received.
    */
    returncode = igpibllo(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibllo\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    exit(0);
}
```

2

igpibpassctl

Description Passes active controller status to another GPIB interface.

int PASCAL
igpibpassctl(INST *id*, **int** *busaddress*);

id	Pointer to a GPIB interface session structure.
busaddress	GPIB address of new active controller interface.

Remarks This function passes active controller state from the GPIB interface of the session pointed to by *id* to the GPIB interface whose address is *busaddress*.

Busaddress must be between zero and 30, inclusive.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot pass active controller states to the specified device.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_PARAM	*Id* specifies a device or commander session, or *busaddress* is invalid.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iopen, itimeout**

Example

```
/*
//      This example uses igpibpassctl to pass active control
//      to another GPIB interface.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{    INST instance, itfinstance;
     int returncode, errornumber, primary, secondary;
     char *sessionnames[] = { "gpib", "gdev1" };

     /*
     //  Open an interface session
     */
     itfinstance = iopen(sessionnames[0]);
     if (itfinstance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionnames[0],
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     /*
     //  Open a device session
     */
     instance = iopen(sessionnames[1]);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionnames[1],
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     returncode = igetdevaddr(instance, &primary, &secondary);
     if (returncode != I_ERR_NOERROR) {
         fprintf(stderr,
             "\tUnable to execute igetdevaddr\n\r");
         fprintf(stderr,
             "\tError = %s (%d)\n\r",
             igeterrstr(returncode),returncode);
         exit(2);
     }
```

```
    returncode = igpibpassctl(itfinstance,primary);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibpassctl\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(3);
    }
    exit(0);
}
```

igpibppoll

Description

Executes a parallel poll.

int PASCAL
igpibppoll(INST *id*, **int** **polldata*);

id	Pointer to a GPIB interface session structure.
polldata	Pointer to the location where the function stores the parallel poll result.

Remarks

This function executes a parallel poll of the GPIB interface of the session pointed to by *id*. The parallel poll results are placed in the lower 8-bits of the location pointed to by *polldata*.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot execute a parallel poll.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_PARAM	*Id* specifies a device or commander session, or *polldata* is null.
I_ERR_TIMEOUT	A timeout occurred.

See Also

iopen, igpibppollconfig, itimeout

2

Example

```
/*
//      This example calls igpibpollconfig configure a device's
//      response to a parallel poll.  Additionally, igpibppoll
//      is called to verify correct execution of the poll
//      configuration call.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

/* GPIB response line 7, no service req */
#define POLLCONFIG          0x47

void main(void)
{   INST instance;
    int returncode, errornumber, polldata;
    char *sessionnames[] = { "gdev1", "gpib" };

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames[0]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[0],
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igpibppollconfig(instance,POLLCONFIG);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibppoll\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    (void) iclose(instance);
    instance = iopen(sessionnames[1]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[1],
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
```

```
    returncode = igpibppoll(instance,&polldata);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibppoll\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(4);
    }
    if (polldata != 0x80) {
        fprintf(stderr,
            "\tIgpibpoll received %x, expected %x\n\r",
            polldata,
            1 << (POLLCONFIG & 0x0f));
        exit(5);
    }
    fprintf(stdout,"Poll data = <%d>",polldata);
    exit(0);
}
```

igpibppollconfig

Description

Configures a GPIB device's response to a parallel poll.

int PASCAL
igpibppollconfig(INST *id*, **int** *configparam*);

id	Pointer to a GPIB device session structure.
configparam	Device configuration.

Remarks

This function configures the parallel poll response of the GPIB device session pointed to by *id*. *Configparam* specifies the GPIB device's response to future parallel polls.

Specifying *configparam* equal to −1 disables the device from responding to parallel polling. Specifying *configparam* greater that or equal to zero enables the device's response to a parallel poll. The lower four bits of *configparam* configure the parallel poll response. Bits 0, 1, and 2 specify the GPIB response lines. Bit 3 specifies the meaning of a parallel poll response (1 = service request, 0 = no service request).

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot define the specified device's PPOLL configuration.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_PARAM	*Id* specifies an interface or commander session.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iopen; itimeout**

Example See **igpibppoll**.

2

igpibrenctl

Description

Controls the state of the GPIB REN line.

int PASCAL
igpibrenctl(INST *id*, **int** *renstate*);

id	Pointer to a GPIB interface session structure.
renstate	REN line state.

Remarks

This function defines the REN line state of the GPIB interface of the session pointed to by *id*.

Specifying a *renstate* equal to zero deasserts the REN line. Specifying *renstate* as non-zero asserts the REN line.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot set REN line state on the interface.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_PARAM	*Id* specifies a device or commander session.
I_ERR_TIMEOUT	A timeout occurred.

See Also

iopen, itimeout

2

Example

```
/*
//      This example uses igpibrenctl to configure the GPIB
//      REN line.
*/

#define RENASSERT  -1
#define RENDEASSERT       0


#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionnames = "gpib";

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igpibrenctl(instance,RENASSERT);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibrenctl\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    exit(0);
}
```

2

igpibsendcmd

Description

Writes command bytes to a GPIB interface.

int PASCAL
igpibsendcmd(INST *id*, **char** **buffer*, **int** *buffersize*);

id	Pointer to a GPIB interface session structure.
buffer	Pointer to a data source buffer.
buffersize	Data buffer size, in bytes.

Remarks

This function writes data from the buffer pointed to by *buffer* to the GPIB interface of the session pointed to by *id* with the ATN line asserted. *Buffersize* specifies the number of data bytes in the buffer.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function cannot send the command data.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-GPIB interface type.
I_ERR_PARAM	*Id* specifies a device or commander session, or *buffer* is null.
I_ERR_TIMEOUT	A timeout occurred.

See Also

iopen, itimeout

2

Example

```
/*
//      This example uses igpibsendcmd to send commands
//      to the GPIB interface.
*/

#define RENASSERT  -1
#define RENDEASSERT      0


#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance, itfinstance;
    int returncode, errornumber, commandlength, itfprimary,
        primary, secondary;
    char *sessionnames[] = { "gpib", "gdev1" };
    char commandlist[5] = { 0 };

    /*
    // Open an interface session
    */
    itfinstance = iopen(sessionnames[0]);
    if (itfinstance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[0],
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = igpibbusstatus(itfinstance,
                    I_GPIB_BUS_ADDR,
                    &itfprimary);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibbusstatus\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    instance = iopen(sessionnames[1]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[1],
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
```

2

```
    returncode = igetdevaddr(instance, &primary, &secondary);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igetdevaddr\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(4);
    }
    commandlist[0] = 0x3F;                                  /* UNL */
    commandlist[1] = (char) (itfprimary + 0x40);            /* MTA */
    commandlist[2] = (char) (primary + 0x20);               /* LAG */
    if (secondary == -1) commandlength = 3;
    else {
        commandlist[3] = (char) (secondary + 0x60);         /* SCG */
        commandlength = 4;
    }
    returncode = igpibsendcmd(itfinstance,
        commandlist,commandlength);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute igpibsendcmd\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(5);
    }
    exit(0);
}
```

ihint

Description

Defines the type of communication a device driver should use.

int PASCAL
ihint(INST *id*, int *hint*);

id	Pointer to a session structure.
hint	Communications type.

Remarks

For SICL, this function checks for errors and returns. *Hint* is ignored. Valid *hint* constants are:

Constant	Description
I_HINT_DONTCARE	No communications preference.
I_HINT_USEDMA	Use DMA, if possible.
I_HINT_USEINTR	Use interrupts, if possible.
I_HINT_USEPOLL	Use polling, if possible.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Hint* is invalid.

2

iintroff

Description Disables SRQ and interrupt event processing.

int PASCAL
iintroff(void);

Remarks This function disables processing of SRQ and interrupt events for the calling process.

When event processing is disabled, SRQ and interrupt events are queued. The *eventqueuesize* variable in the SICLIF file sets the number of SRQ and interrupt events that can be queued while event processing is disabled. If an attempt to queue an event causes the queue to overflow, the event is discarded and the error message "SICL event queue overflow -- event lost!" is sent to the console.

By default, SRQ and interrupt event processing are enabled.

Use **iintron** to re-enable SRQ and interrupt event processing.

SRQ and interrupt event disabling can be nested. Each call to **iintroff** should be paired with one, and only one, call to **iintron**.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_NOERROR	Successful function completion.

See Also **iintron**

Example See **igetonintr**.

iintron

Description Enables processing of SRQ and interrupt events.

int PASCAL
iintron(void);

Remarks This function enables processing of SRQ and interrupt events by the calling process.

By default, SRQ and interrupt event processing is enabled.

Use **iintroff** to disable SRQ and interrupt event processing.

Attempting to enable SRQ and interrupt event processing when it is already enabled results in an **I_ERR_OS** error.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_NOERROR	Successful function completion.
I_ERR_OS	Asynchronous event handling is already enabled.

See Also **iintroff, ionintr, ionsrq, isetintr**

Example See **igetonintr**.

2

ilblockcopy

Description

Copies a block of 32-bit words from one set of sequential memory locations to another.

int PASCAL
ilblockcopy(INST *id*, **unsigned long** **src*, **unsigned long** **dest*,
unsigned long *count*, **int** *swap*);

id	Pointer to a session structure.
src	Source pointer.
dest	Destination pointer.
count	Number of 32-bit words to copy.
swap	Byte swap flag.

Remarks

Copies 32-bit words from successive memory locations beginning at *src* into successive memory locations beginning at *dest*. *Count* specifies the number of 32-bit words to transfer and has a maximum value of 0x4000. *Id* specifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

When *swap* is non-zero and a VXIbus access is made, the function byte-swaps the 32-bit words to or from Motorola byte ordering as necessary. When *swap* is zero, no byte swapping occurs. The following lists the possible scenarios when accessing EPC and VXIbus memory:

src	dest	swap	Result
EPC	EPC	0	No byte-swapping
EPC	EPC	Non-zero	No byte-swapping
EPC	VXI	0	No byte-swapping
EPC	VXI	Non-zero	One byte-swap
VXI	EPC	0	No byte-swapping
VXI	EPC	Non-zero	One byte-swap
VXI	VXI	0	No byte-swapping
VXI	VXI	Non-zero	Two byte-swaps (equivalent to no byte-swap)

For byte-swapping to work properly, all VXIbus access must be aligned on a 32-bit boundary.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Src* and/or *dest* is null.

See Also

ibblockcopy, **ilpeek**, **ilpoke**, **ilpopfifo**, **ilpushfifo**, **imap**, **iwblockcopy**

Example

See **iwblockcopy**.

2

ilocal

Description Puts a device in local mode.

int PASCAL
ilocal(INST *id*);

id	Pointer to a device session structure.

Remarks With VXI device sessions, this function supports only message-based VXI devices.

For VXI device sessions, the function issues a CLEAR LOCK word-serial command to the device. Only message-based VXI devices are supported. Use with other VXI devices cause an error.

For GPIB device sessions, the function addresses the device to listen, then sends the GTL (go to local) command.

This function supports only device sessions. Specifying an interface session is an error.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.

I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies an interface or commander session or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iremote, itimeout**

Example

```
/*
//     This example uses ilocal to put the specified
//     GPIB device into local mode.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "gdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ilocal(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIlocal call failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    exit(0);
}
```

2

ilock

Description

Locks a device or interface.

int PASCAL
ilock(INST *id*);

id Pointer to a session structure.

Remarks

This function locks the device or interface session pointed to by *id* to prevent access by other processes.

Locking an interface session locks the entire interface. Only the calling process can access devices on the interface.

Locking a device session prevents all other processes from locking or accessing the device. It also prevents other processes from locking the interface. It does not prevent other processes from locking or accessing other devices on the interface.

Locking conflict resolution is set by **isetlockwait**. However, under DOS, a locking conflict always results in an **I_ERR_LOCKED** error because DOS does not support process preemption.

Locks can be nested. Each **ilock** call must be paired with a corresponding **iunlock** call.

Locking affects these SICL functions:

imap	**inbread**	**isetstb**
iclear	**inbwrite**	**itrigger**
iflush	**iopen**	**ivxigettrigroute**
igpibatnctl	**igpibllo**	**ivxitrigoff**
igpibpassctl	**iprintf**	**ivxitrigon**
igpibppoll	**ipromptf**	**ivxitrigroute**
igpibppollconfig	**iread**	**ivxiwaitnormop**
igpibrenctl	**ireadstb**	**ivxiws**
igpibsendcmd	**iremote**	**iwrite**
ilocal	**iscanf**	**ixtrig**
ilock	**isetbuf**	

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.

See Also

igetlockwait, isetlockwait, itimeout, iunlock

Example

```
/*
//      This example uses ilock/iunlock to lock the device access
//      from other processes.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"
```

2

```
void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ilock(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to lock <%s>\n\r",
            sessionname,
            igeterrstr(returncode),returncode);
        exit(2);
    }
    /*
    //      Processing of the critical section goes here
    //      ...
    */
    returncode = iunlock(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to unlock <%s>\n\r",
            sessionname,
            igeterrstr(returncode),returncode);
        exit(3);
    }
    exit(0);
}
```

ilpeek

Description Reads a 32-bit word stored at a mapped address.

volatile unsigned long PASCAL
ilpeek(volatile unsigned long ***addr*);**

addr Address of a 32-bit word.

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

Return Value The function returns the 32-bit word contained at *addr*.

See Also **ibpoke, ibpeek, imap, iwpeek**

Example

```
/*
//      This example uses ilpeek to read our own slave
//      memory thru the VXIbus.
*/

#include <stdlib.h>
#include <stdio.h>
#include "busmgr.h"
#include "sicl.h"

void main(void)
{   INST instance;
    int errornumber, returncode, result;
    char *lowpage;
    unsigned long lowmemory;
    char *sessionnames[] = { "vxi", "vdevl" };
    unsigned long *baseoffset = (unsigned long *) 0x400L;

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames[0]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[0],
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
```

2

```
/*
//      Find where our memory begins
*/
returncode = ivxibusstatus(instance,
                    I_VXI_BUS_SHM_PAGE,
                    &result);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,
        "\tUnable to execute ivxibusstatus\n\r");
    fprintf(stderr,
        "\tError = %s (%d) \n\r",
        igeterrstr(returncode),returncode);
    exit(2);
}
(void) iclose(instance);
/*
// Open a device session
*/
instance = iopen(sessionnames[1]);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionnames[1],
        igeterrstr(errornumber),errornumber);
    exit(3);
}
/* Map in A24 space */
lowpage = imap(instance,I_MAP_A24,result >> 8,1,NULL);
if (lowpage == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to map in A24 space, error = %s (%d) \n\r",
        igeterrstr(errornumber),errornumber);
    exit(4);
}
```

```
    /*
    //    Reading the 400th long word of VME memory at our base
    //    address should return the same value as reading 0:400
    //    through PC memory
    */
    lowmemory = ilpeek((unsigned long *)
                ((unsigned long) lowpage+
                (unsigned long) baseoffset));
    EpcMemSwapL(&lowmemory,1);
    if (lowmemory != *baseoffset) {
        fprintf(stderr,
            "\tVME memory at page %x longword offset %lx ",
            result >> 8,baseoffset);
        fprintf(stderr,"= %08.8lx\n\r",lowmemory);
        fprintf(stderr,"\tExpected %08.8lx\n\r",*baseoffset);
        exit(5);
    }
    fprintf(stdout,"VME memory at page %x longword offset %lx = ",
            result >> 8,baseoffset);
    fprintf(stdout,"%08.8lx\n\r",lowmemory);
    exit(0);
}
```

2

ilpoke

Description Writes a 32-bit word to a mapped address.

void PASCAL
ibpoke(volatile unsigned long **dest*, **unsigned long** *value*);

dest	Destination address.
value	32-bit word to write.

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

Return Value The function returns no value.

See Also **ibpeek, ibpoke, imap, iwpoke**

Example

```
/*
//      This example uses ilpoke to write into
//      DOS's communication area via VME memory.
*/

#include <stdlib.h>
#include <stdio.h>
#include "sicl.h"
#include "busmgr.h"

#define FOOTPRINT         0x12345678L
```

```
void main(void)
{   INST instance;
    int errornumber, returncode, result;
    char *lowpage;
    long *doscom = (long *) 0x4f0L;
    char *sessionnames[] = { "vxi", "vdev1" };

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames[0]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[0],
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /*
    //      Find where our memory begins
    */
    returncode = ivxibusstatus(instance,
                      I_VXI_BUS_SHM_PAGE,
                      &result);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute ivxibusstatus\n\r");
        fprintf(stderr,
            "\tError = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    (void) iclose(instance);
    /*
    // Open a device session
    */
    instance = iopen(sessionnames[1]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[1],
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
    /* Map in A24 space */
    lowpage = imap(instance,I_MAP_A24,result >> 8,1,NULL);
    if (lowpage == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A24 space, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(4);
    }
```

2

```
    /*
    //      Write into DOS's communication area at PC address
    //      4f0:0
    */
    ilpoke((unsigned long *)
           ((unsigned long) lowpage+(unsigned long) doscom),
           FOOTPRINT);
    EpcMemSwapL((unsigned long *) doscom,1);
    if (*doscom != FOOTPRINT) {
        fprintf(stderr,
            "\tVME memory at page %x longword offset %lx ",
            result >> 8,doscom);
        fprintf(stderr,"= %08.8lx\n\r",*doscom);
        fprintf(stderr,"\tExpected %08.8lx\n\r",FOOTPRINT);
        exit(5);
    }
    fprintf(stdout,"VME memory at page %x longword offset %lx = ",
            result >> 8,doscom);
    fprintf(stdout,"%08.8lx\n\r",*doscom);
    exit(0);
}
```

ilpopfifo

Description

Copies 32-bit words from a single memory location (FIFO register) to sequential memory locations.

int PASCAL
ibpopfifo(INST *id*, **unsigned long** **fifo*, **unsigned long** **dest*, **unsigned long** *count*, **int** *swap*);

id	Pointer to a session structure.
fifo	FIFO pointer.
dest	Destination address.
count	Number of 32-bit words to copy.
swap	Byte swap flag.

Remarks

This function copies *count* 32-bit words from *fifo* into sequential memory locations beginning at *dest*. *Count* specifies the number of 32-bit words to transfer and has a maximum value of 0x4000. *Id* specifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap-around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

When *swap* is non-zero and a VXIbus access is made, the function byte-swaps the 32-bit words to or from Motorola byte ordering as necessary. When *swap* is zero, no byte swapping occurs. The following table lists the possible scenarios when accessing EPC and VXIbus memory:

2

src	*dest*	*swap*	Result
EPC	EPC	0	No byte-swapping
EPC	EPC	Non-zero	No byte-swapping
EPC	VXI	0	No byte-swapping
EPC	VXI	Non-zero	One byte-swap
VXI	EPC	0	No byte-swapping
VXI	EPC	Non-zero	One byte-swap
VXI	VXI	0	No byte-swapping
VXI	VXI	Non-zero	Two byte-swaps (equivalent to no byte-swap)

For byte-swapping to work properly, all VXIbus access must be aligned on a 32-bit boundary.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Fifo* and/or *dest* is null.

See Also **ibpopfifo, ilpushfifo, imap, iwpopfifo**

Example

```
/*
//      This example uses ilpopfifo to read from a
//      hypothetical VXI fifo at offset 0.
*/

#include <stdlib.h>
#include <stdio.h>
#include "sicl.h"

#define NOSWAP          0       /* 0 indicates no byte swapping */
```

```
void main(void)
{   INST instance;
    unsigned long *vxi;
    int returncode, errornumber;
    unsigned long datafifo[5];
    char *sessionname = "vxi";

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxi = (unsigned long *) imap(instance,I_MAP_A16,0,0,NULL);
    if (vxi == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = ");
        fprintf(stderr,
            "%s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    /*
    // Read the Fifo 5 times, storing the values into datafifo[]
    */
    returncode = ilpopfifo(instance,
                      vxi,
                      datafifo,
                      (long) (sizeof(datafifo)/sizeof(long)),
                      NOSWAP);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to read the fifo at address ");
        fprintf(stderr,
            "%p\n\r\tError = %s (%d) \n\r",
            vxi,
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    exit(0);
}
```

2

ilpushfifo

Description

Copies 32-bits words from sequential memory locations to a single memory location (FIFO register).

int PASCAL
ilpushfifo(INST *id*, **unsigned long** **src*, **unsigned long** **fifo*,
unsigned long *count*, **int** *swap*);

id	Pointer to a session structure.
src	Source address.
fifo	FIFO pointer.
count	Number of 32-bit words to copy.
swap	Byte swap flag.

Remarks

Copies *count* 32-bit words from the sequential memory locations beginning at *src* into the FIFO at *fifo*. *Count* specifies the number of 32-bit words to transfer and has a maximum value of 0x4000. *Id* specifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap-around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

When *swap* is non-zero and a VXIbus access is made, the function byte-swaps the 32-bit words to or from Motorola byte ordering as necessary. When *swap* is zero, no byte swapping occurs. The following lists the possible scenarios when accessing EPC and VXIbus memory:

src	dest	swap	Result
EPC	EPC	0	No byte-swapping
EPC	EPC	Non-zero	No byte-swapping
EPC	VXI	0	No byte-swapping
EPC	VXI	Non-zero	One byte-swap
VXI	EPC	0	No byte-swapping
VXI	EPC	Non-zero	One byte-swap
VXI	VXI	0	No byte-swapping
VXI	VXI	Non-zero	Two byte-swaps (equivalent to no byte-swap)

For byte-swapping to work properly, all VXIbus access must be aligned on a 32-bit boundary.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Src* and/or *fifo* is null.

See Also **ibpopfifo, ibpushfifo, imap, iwpushfifo**

Example

```
/*
//      This example uses ilpushfifo to write values
//      to a hypothetical VXI fifo at offset 0.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define NOSWAP          0       /* 0 indicates no byte swapping */
```

2

```
void main(void)
{   INST instance;
    char *vxi;
    int returncode, errornumber;
    unsigned long datafifo[] = { 0x1L, 0x2L, 0x3L, 0x4L, 0x5L };
    char *sessionname = "vxi";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxi = imap(instance,I_MAP_A16,0,0,NULL);  /* Map in A16 space */
    if (vxi == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = ");
        fprintf(stderr,
            "%s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    /*
    // Write to the fifo 5 times, storing 0x00000001L, 0x00000002L,
    // 0x00000003L, 0x00000004L, 0x00000005L
    */
    returncode = ilpushfifo(instance,
                    (unsigned long *) vxi,
                    datafifo,
                    (unsigned long) sizeof(datafifo)/sizeof(long),
                    NOSWAP);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to write to the fifo at address ");
        fprintf(stderr,
            "%p\n\r\tError = %s (%d) \n\r",
            vxi,
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    exit(0);
}
```

imap

Description Maps a portion of a VXIbus address space into user memory space.

char * PASCAL
imap(INST ***id*****, int** *mapspace*, **unsigned int** *pagestart*, **unsigned int** *pagecnt*, **char** **suggestedaddress*);

id	Pointer to a session structure.
mapspace	Address space to map.
pagestart	Starting page number.
pagecnt	Number of pages to map.
suggestedaddress	User suggested pointer to the mapped memory location.

Remarks Although **imap** returns a pointer to the designated portion of VXIbus, the pointer cannot be used directly because the byte order is not defined. Byte order is defined when the returned pointer is used in a mapped memory I/O function.

The address space to be mapped depends on *id* and *mapspace*. The following are valid constants for *mapspace*:

Constant	Description
I_MAP_A16	Map the A16 address space. Valid for VXI device and interface sessions.
I_MAP_A24	Map the A24 address space (page size 64K bytes). Valid for VXI device and interface sessions.
I_MAP_A32	Map the A32 address space (page size 64K bytes). Valid for VXI device and interface sessions.
I_MAP_VXIDEV	Map a VXI device's configuration registers. Valid only for VXI device sessions.

2

I_MAP_EXTEND	Map the A24/A32 address space that corresponds to this EPC. Valid only for VXI device sessions (EPC-2 and EPC-7 only).

When *mapspace* is **I_MAP_EXTEND**, the A16 registers for the device determine the location of the address space. *Pagestart* is the offset, in 64K pages, into the extended memory of the device. *Pagecnt* is the amount of memory, in 64K pages, to map.

The *suggestedaddress* variable is NULL.

Use **imapinfo** to obtain a valid page size parameter for a given address space.

The DOS real mode implementation limits mapping to A16 space or one A24 or A32 space page at a time.

When *mapspace* is either **I_MAP_A16** or **I_MAP_VXIDEV**, the *pagestart* and *pagecnt* variables are ignored.

Unmap the current space before attempting to map another address space. Unmap the address space when it is no longer needed to free hardware resources for other processes.

For DOS applications, the **imap** function cannot suspend execution of the calling process; therefore, when sufficient resources are not available to satisfy the request, the **imap** function returns an **I_ERR_NORSRC** error.

Return Value

If successful, the function returns a pointer to the mapped address. Otherwise, a null pointer is returned. Possible errors include:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The system cannot execute the specified mapping.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.

I_ERR_NOERROR Successful function completion.

I_ERR_NORSRC The system contains insufficient resources to satisfy the specified map request.

I_ERR_NOTSUPP *Id* specifies an interface type that does not support memory mapping (e.g., GPIB).

I_ERR_PARAM *Id* specifies a session whose type is inconsistent with the given *mapspace*, *pagestart/pagecnt* are inconsistent with the capabilities of the hardware/software platform and/or the given *mapspace*, or *mapspace* is invalid.

See Also **imapinfo, iopen, iunmap**

Example

```
/*
//      This example uses imap to map the VXI registers
//      into the application's memory space.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int *vxiregisters;
    int returncode, errornumber;
    int vxiid;
    char *sessionname = "vdevl";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxiregisters = (int *) imap(instance,I_MAP_VXIDEV,0,0,NULL);
    if (vxiregisters == NULL) {
        errornumber = igeterrno();
```

2

```
        fprintf(stderr,
            "\tUnable to map in VXI registers");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    returncode = iwblockcopy(instance,
                        (unsigned short *) vxiregisters,
                        (unsigned short *) &vxiid,
                        1L,
                        -1);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to copy ID register, ");
        fprintf(stderr,
            "error = %s (%d)\n\r",
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    fprintf(stdout,"Manufacturer ID of device <%s> is %d",
            sessionname,
            vxiid & 0xfff);
    exit(0);
}
```

imapinfo

Description

Queries address space mapping capabilities for the specified interface.

int PASCAL
imapinfo(INST *id*, **int** *mapspace*, **int** **numwindows*, **int** **windowsize*);

id	Pointer to a session structure.
mapspace	Address space.
numwindows	Pointer to a location where the function stores the total number of windows.
windowsize	Pointer to a location where the function stores the window size, in pages.

Remarks

This function queries *mapspace* on the interface of the session pointed to by *id* and obtains the number of mapping windows available and the size of each window. It does not identify which window is in use by another process.

When there is more than one window size available, *windowsize* points to a location containing the smallest window size.

The following constants define valid values for *mapspace*:

Constant	Description
I_MAP_A16	Map the A16 address space
I_MAP_A24	Map the A24 address space (page size 64K bytes)
I_MAP_A32	Map the A32 address space (page size 64K bytes)

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Mapspace* is invalid or *numwindows* and/or *windowsize* is null.

See Also **imap, iopen**

Example

```
/*
//      This example calls imapinfo to determine the window(s)
//      count and size.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{    INST instance;
     int returncode, windowcount, windowsize, errornumber;
     char *sessionname = "vdevl";

     /*
     // Open a device session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
```

```
    returncode = imapinfo(instance,
                    I_MAP_A32,
                    &windowcount,
                    &windowsize);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tImapinfo call failed, error = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,
            "The VXI interface contains %d window(s) of %d page(s)",
            windowcount,
            windowsize);
    exit(0);
}
```

2

inbread

Description

Reads data from a device or interface without blocking.

int PASCAL
inbread(INST *id*, **char** **buf*, **unsigned long** *bufsize*, **int** **reason*,
unsigned long **actualcnt*);

id	Pointer to a session structure.
buf	Pointer to the data buffer.
bufsize	Number of data bytes to read.
reason	Pointer to a location where the function stores the read termination bit mask.
actualcnt	Pointer to a location where the function stores the actual number of bytes read from the device.

Remarks

This function reads *bufsize* bytes from the device or interface of the session pointed to by *id* and stores them in the buffer specified by *buf*. *Bufsize* has a maximum value of 0x10000. It performs no formatting or data conversion.

Reading ends when *bufsize* bytes are read, an END indicator is received, a termination character is received, or the device or interface does not send data. Unlike the **iread** function, this function does not block if the device or interface does not send data.

When *id* specifies a device session, data is read using interface independent communications methods. When *id* specifies an interface session, data is read in raw mode using interface specific methods.

For VXI device sessions, the function issues BYTE REQUEST word-serial commands. Only message based VXI devices are supported, other VXI devices cause an error.

For VXI interface sessions, the function generates an **I_ERR_PARAM** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, the function issues the interface's listen address, followed by the device's talk address. Finally, the function reads the data bytes.

For GPIB interface sessions, the function reads data from the GPIB interface without performing any addressing.

If *reason* is not null, the function stores a bit mask describing why the read terminated in the referenced memory location. The following constants define valid bits in the mask pointed to by *reason*:

Constant	Description
I_TERM_CHR	Termination character received (see **itermchr**)
I_TERM_END	END indicator received
I_TERM_MAXCNT	*Bufsize* bytes read
I_TERM_NON_BLOCKED	The device or interface was not ready to send more data

When *reason* is **I_TERM_NON_BLOCKED**, no other termination reasons are possible. Conversely, **I_TERM_NON_BLOCKED** is not possible when any of the other three termination conditions exist.

If *actualcnt* is not null, the function stores the number of bytes read in the referenced memory location.

2

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred during the read operation.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred during the read operation.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface session or a VXI device that is not message-based, or *buf* is null.

See Also

igettermchr, inbwrite, iread, itermchr, iwrite

Example

```
/*
//      This example calls inbread to read
//      an instrument's response without waiting
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, reason = 0, errornumber, position = 0;
    unsigned long readcount;
    char buffer[50] = {0};
    char *sessionname = "vdevl";
```

2

```
/*
// Open a device session
*/
instance = iopen(sessionname);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionname,
        igeterrstr(errornumber),errornumber);
    exit(1);
}
(void) iprintf(instance,"rmx\n");
do {
    returncode = inbread(instance,
                    &buffer[position],
                    sizeof(buffer),
                    &reason,
                    &readcount);
    position += (int) readcount;
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tinbread failed, error = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
} while (reason != I_TERM_END);
buffer[(short) position] = (char) '\0';
fprintf(stdout,"The data read from %s is %s\n\r",
        sessionname,
        buffer);
fprintf(stdout,"Read termination reason(s):\n\n\r");
if (reason & I_TERM_CHR) fprintf(stdout,"\tI_TERM_CHR\n\r");
if (reason & I_TERM_END) fprintf(stdout,"\tI_TERM_END\n\r");
if (reason & I_TERM_MAXCNT)
    fprintf(stdout,"\tI_TERM_MAXCNT\n\r");
exit(0);
}
```

2

inbwrite

Description

Writes data to a device or interface without blocking.

int PASCAL
inbwrite(INST *id*, **char** **buf*, **unsigned long** *bufsize*, **int** *end*,
unsigned long **actualcnt*, **int** **done*);

id	Pointer to a session structure.
buf	Pointer to the data buffer.
bufsize	Length, in bytes, of data buffer.
end	END indicator flag.
actualcnt	Pointer to a location where the functions stores the actual number of bytes written.
done	Pointer to a location where the functions store a flag indicating write completion status.

Remarks

This function writes the *bufsize* bytes at *buf* to the device or interface of the session pointed to by *id*. *Bufsize* has a maximum value of 0x10000. It performs no formatting or data conversion.

Writing ends when *bufsize* bytes are written or the device or interface is not ready to receive data. Unlike the **iwrite** function, this function does not block if the device is not ready to receive data.

When *id* specifies a device session, the function writes data using interface dependent communication methods. When *id* specifies an interface session, the function writes data in raw mode using interface specific methods.

If *end* is non-zero, the function writes an END indicator with the last data byte. If *end* is zero, the function does not write an END indicator with the last data byte.

If *actualcnt* is not null, the function stores the number of data bytes written in the referenced memory location.

The function writes a one into the location referenced by *done* after it writes all the specified data bytes. Until all data bytes are written, the function writes a zero into the location referenced by *done*. *Done* cannot be null.

For VXI device sessions, the function issues BYTE AVAILABLE word-serial commands and supports only message based VXI devices. Other VXI devices cause an error.

For VXI interface sessions, the function generates an **I_ERR_PARAM** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's talk address, followed by the device's listen address. Finally, the function writes the data.

For GPIB interface sessions, the function writes bytes directly to the interface without performing any addressing. The ATN line state determines if the bytes are interpreted as command bytes.

2

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred during the write operation.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred during the write operation.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based, or *buf* and/or *done* is null.

See Also **inbread, inbwrite, iread, iwrite**

Example

```
/*
//      This example calls inbwrite to write to an instrument
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define EOI         -1              /* set the end indicator */

void main(void)
{   INST instance;
    int returncode, errornumber, done = 0, count = 4, position = 0;
    char *sessionname = "vdev1";
    unsigned long actualcount;
    char *writestring = "rmx\n";
```

```
        /*
        // Open a device session
        */
        instance = iopen(sessionname);
        if (instance == NULL) {
            errornumber = igeterrno();
            fprintf(stderr,
                "\tUnable to open <%s>, error = %s (%d)\n\r",
                sessionname,
                igeterrstr(errornumber),errornumber);
            exit(1);
        }
        do {
            returncode = inbwrite(instance,
                                  &writestring[position],
                                  count,
                                  EOI,
                                  &actualcount,
                                  &done);
            count -= (int) actualcount;
            position += (int) actualcount;
            if (returncode != I_ERR_NOERROR) {
                fprintf(stderr,
                    "\tInbwrite failed, error = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
                exit(2);
            }
        } while (!done);
        fprintf(stdout,"%d bytes written to <%s>",position,sessionname);
        exit(0);
    }
```

2

ionerror

Description Installs an error handler.

int PASCAL
ionerror(void (CDECL * *errorhandler*)**(INST** *id*, **int** *error*));

errorhandler Pointer to an error handler function.

Remarks This function installs the function pointed to by *errorhandler* as the function to call when an error occurs.

The SICL library assumes error handler functions have the following interface:

void CDECL
errorhandler(**INST** *id*, **int** *error*);

where *id* identifies the device or interface session generating the error and *error* is an error constant defining the error.

SICL defines two default error handlers:

Constant	Description
I_ERROR_EXIT	Writes an error message to STDERR and terminates the process.
I_ERROR_NO_EXIT	Writes an error message to STDERR and allows process to continue.

For DOS, the default error handlers send descriptive information to the console without terminating the process. The functionality required to write to STDERR and terminate a process is non-reentrant, and cannot be used in an error handler. (See Chapter 4, *Advanced Topics*).

Installing a null error handler removes the current error handler.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_NOERROR	Successful function completion.

See Also **igetonerror**

Example See **igetonerror**.

2

ionintr

Description

Installs a session's interrupt handler.

int PASCAL
ionintr(INST *id*, **void (CDECL** **intrhandler*)(**INST** *id*, **long**
data1, **long** *data2*));

id	Pointer to a session structure.
intrhandler	Pointer to an interrupt handler function.

Remarks

This function installs the function pointed to by *intrhandler* as the function to call when the device or interface session pointed to by *id* processes an interrupt event.

The SICL library assumes that interrupt handler functions have the following interface:

void CDECL
intrhandler(**INST** *id*, **long** *data1*, **long** *data2*);

where *id* identifies the device or interface session receiving the interrupt, *data1* identifies the interrupt (**I_INTR_TRIG**, etc.).

Data2 has meaning on an EPC-7 only for **I_INTR_TRIG** interrupts to a VXI interface session when it identifies the trigger(s) causing the interrupt. *Data2* has these constants:

Constant	Description
I_TRIG_STD	Standard trigger.
I_TRIG_EXT0	EXT trigger 0, if it is mapped as an input trigger (see **ivxitrigroute**).
I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.

I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

The trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxirigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Proper VXI trigger interrupt operation on an EPC-7 requires direct program manipulation of EPC-7 hardware, refer to Chapter 4, *Advanced Topics*, for additional information.

This function does not enable interrupt reception or processing. See **isetintr** to enable interrupt reception and **iintroff** and **iintron** to disable and enable interrupt processing, respectively.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a commander session.

See Also **igetonintr, iintroff, iintron, isetintr**

Example

```
/*
//      This example sets, generates and processes interrupts
//      using igetonintr, ionintr, isetintr and iintron/introff.
*/

#include <stdio.h>
#include <stdlib.h>
#include "busmgr.h"
#include "sicl.h"
```

2

```
/* remove's compiler warning message (compiler specific) */
#define REMOVEWARNING(x)   x = x

#define INTERRUPTENABLE    1
#define INTERRUPTDISABLE   0
#define INTERRUPTS         7       /* interrupts 1-7 */
#define WAITTIME           (1000L*30L*1)
#define TIMERINT           8

volatile unsigned long Vmeinterruptcount = 0;
void (INTERRUPT *timerfunction)();
volatile unsigned long Tick = 0;

void
console(char *astring)
{   char    achar;

    while (*astring) {
        achar = *astring++;
        ASM
            mov     ah,0eh
            mov     al,achar
            mov     bx,3
            int     010h
        ENDASM
    }
}

static void
reverse(char s[])       /* K & R -- page 59   */
{   register int i, j;
    int slen;
    char c;

    slen = 0;
    while(s[slen++]);
    for (i = 0, j = slen-2; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

static void
myitoa(long n,char s[])  /* K & R -- page 60   */
{   long i, sign;

    if ((sign = n) < 0) n = -n;
    i = 0;
    do {
        s[(int) (i++)] = (char) ((char) (n % 10) + '0');
    } while ((n /= 10) > 0);
    if (sign < 0) s[(int) (i++)] = (char) '-';
    s[(int) i] = (char) '\0';
    reverse(s);
}
```

```
void CDECL
vmehandler(INST instance, long interruptsource, long junk)
{   char abuffer[10];
    char *sessionaddress;

    Vmeinterruptcount++;
    /*
    // Can't use stdio from interrupt handlers.
    */
    console("handler : vmehandler,  Interrupt source <");
    myitoa(interruptsource,abuffer);
    console(abuffer);
    console(">\n\r");
    console("Interrupt <");
    myitoa(Vmeinterruptcount,abuffer);
    console(abuffer);
    console(">\n\r");
    if (igetaddr(instance,&sessionaddress) == I_ERR_NOERROR) {
       console("Session address = <");
       console(sessionaddress);
       console(">\n\r");
    }
    REMOVEWARNING(junk);
}

#if !defined(__TURBOC__)

void INTERRUPT
mytimer()
{    Tick--;
     if (Tick == 0) {
        EpcSigIntr(3);
     }
     Vmeinterruptcount = 1;
     _chain_intr(timerfunction);
}

void
installtimer(void (INTERRUPT *newfunction)(),unsigned short timeout)
{   _disable();
    Tick = 18 * timeout;
    timerfunction = _dos_getvect(TIMERINT);
    _dos_setvect(TIMERINT,newfunction);
    _enable();
}

void
deinstalltimer()
{   _disable();
    _dos_setvect(TIMERINT,timerfunction);
    _enable();
}

#endif

void main(void)
{   INST instance;
```

```
    int returncode, errornumber;
    char *sessionname = "vxi";
    register short iinductor;
    void (CDECL *oldhandler)(INST instance,
                        long interruptsource,
                        long junk);

    /*
    //  Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ionintr(instance,vmehandler);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to set interrupt handler\n\r");
        fprintf(stderr,
            "\terror = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    returncode = isetintr(instance,I_INTR_VXI_VME,INTERRUPTENABLE);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to enable interrupt reception\n\r");
        fprintf(stderr,
            "\terror = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(3);
    }
    /*
    //  Cycle through the VME interrupts
    */
    for (iinductor = 0; iinductor <= INTERRUPTS; iinductor++) {
        if (EpcSigIntr(iinductor) != EPC_SUCCESS) {
            fprintf(stderr,"\tUnable to generate a VME
interrupt\n\r");
            exit(4);
        }
    }
    if (Vmeinterruptcount != INTERRUPTS) {
        fprintf(stderr,
            "\tExpected interrupt processing not detected\n\r");
        exit(5);
    }
#if !defined(__TURBOC__)
```

```
        /*
        // Create a new thread to assert a VME interrupt.
        */
        Vmeinterruptcount = 0;
        installtimer(mytimer,15);
        /*
        //      Wait for the completion of one more interrupt handler
        //      invocation
        */
        returncode = iwaithdlr(WAITTIME);
        deinstalltimer();
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIwaithdlr failed\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(6);
        }
        if (Vmeinterruptcount == 0) {
            fprintf(stderr,
                "\tExpected interrupt processing not detected\n\r");
            exit(7);
        }
    #endif
        /*
        //      Keep interrupt processing off while the interrupt
        //      handler is being written
        */
        returncode = iintroff();
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIintroff failed\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(8);
        }
        /*
        // Restore the previous interrupt
        */
        returncode = igetonintr(instance,&oldhandler);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable execute igetonintr successfully\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(9);
        }
        fprintf(stdout,"Interrupt testing successful\n\r");
        exit(0);
    }
```

2

ionsrq

Description

Installs a service request (SRQ) handler.

int PASCAL
ionsrq(INST *id*, **void (CDECL****srqhandler*)(**INST** *id*));

id	Pointer to a device session structure.
srqhandler	Pointer to a SRQ handler function.

Remarks

This function installs the function pointed to by *srqhandler* as the function to call when the device session pointed to by *id* processes a service request event.

The SICL library assumes that SRQ handler functions have the following interface:

void CDECL
srqhandler(**INST** *id*);

where *id* identifies the device requesting service.

SRQ reception is always enabled.

This function does not enable or disable SRQ processing. Use **iintroff** to disable SRQ processing and **iintron** to enable SRQ processing. By default, SRQ processing is enabled.

If an interface device driver receives a SRQ and cannot determine the SRQ source, it passes the SRQ to all device sessions on the interface. Therefore, a SRQ handler cannot assume that its corresponding device generated the SRQ. Use the **ireadstb** function to determine whether the corresponding device generated the SRQ.

If a process has two or more sessions that refer to the same device and a SRQ request occurs, the SRQ handlers for each of the two different device sessions are called.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies an interface or commander session.

See Also

igetonsrq, **ireadstb**

Example

See **igetonsrq**

2

iopen

Description Opens a session.

int PASCAL
INST iopen(char ***addr*****);**

addr	Device or interface address string

Remarks This function opens a session for communicating with the device or interface specified by the address string *addr. Addr* cannot be null.

An address string for interfaces has this form :

logical unit | *symbolic name*

where *logical unit* is an integer greater than zero and less than 32767 and *symbolic name* is any sequence of letters, digits, underscores, and dashes that begins with a letter. The following are valid interface addresses:

7	An interface at *logical unit* 7
vxi	A *symbolic name* for the VXIbus interface

An address string for devices has this form :

(i/f address, primary address [, *secondary address*])|
symbolic name

where *i/f address* is *logical unit* | *symbolic name* (the same as the address string for interfaces), *primary address* is interface specific (normally a positive integer, but can be a string or sequence of bytes), *secondary address* is also interface specific, and *symbolic name* is the same as the address string for interfaces . The following are valid device addresses:

7,23	*I/f address* is *logical unit* 7 and *primary address* of the device is 23.
vxi,128	*I/f address* is *symbolic name* vxi and *primary address* is *ula* 128.
meter	The device has *symbolic name* meter

Logical units, symbolic names, and the corresponding device driver names are defined in the SICLIF file in the ...\EPCONNEC directory. By default, the SICLIF file defines the following interfaces:

Logical Unit	Symbolic Name	Device Name
2	vxi	vxi$1
2	VXI	vxi$1
2	mxi	vxi$1
2	MXI	vxi$1
1	gpib	gpib$1
1	GPIB	gpib$1
1	hpib	gpib$1
1	HPIB	gpib$1

Symbolic device names are defined in the DEVICES file in the ...\EPCONNEC directory. If no configured name matches the a VXI device, the VXI device gets a symbolic name generated by the SURM. The SURM assigned names may change if the system configuration is changed. The VXI Configurator defines symbolic devices and their attributes.

If an interface and a device have the same name, the session opens as an interface session because interface names are searched first.

Address strings that begin with ASCII digits "0" through "9" are considered logical unit numbers.

2

Return Value If successful, the function returns a pointer to the new session. Otherwise, a null pointer is returned. Possible errors are:

Constant	Description
I_ERR_BADADDR	*Addr* specifies an invalid primary or secondary address, or references an invalid device.
I_ERR_LOCKED	*Addr* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	The device driver corresponding to *addr* is not installed.
I_ERR_NORSRC	The system contains insufficient resources to open the session.
I_ERR_NOTSUPP	The implementation does not support commander sessions.
I_ERR_SYMNAME	*Addr* specifies an invalid symbolic interface or device name.
I_ERR_SYNTAX	*Addr* specifies a syntactically incorrect address.

See Also **iclose**

Example

```
/*
//      Use iopen to establish some sessions
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"
```

2

```
void main(void)
{   INST instances[6] = {0};
    int errornumber, icount = 0, i;
    char *interfaces[] = { "1", "2" };
    char *sessions[] = { "vdev1", "gdev1" };

    for (i = 0; i < 2; i++) {
        /*
        //  Open the interfaces
        */
        instances[icount] = iopen(interfaces[i]);
        if (instances[icount] == NULL) {
            errornumber = igeterrno();
            fprintf(stderr,
                "\tUnable to open <%s>, error = %s (%d)\n\r",
                 interfaces[i],
                igeterrstr(errornumber),errornumber);
            exit(1);
        }
        icount++;
    }
    for (i = 0; i < 2; i++) {
        /*
        //  Open the device sessions
        */
        instances[icount] = iopen(sessions[i]);
        if (instances[icount] == NULL) {
            errornumber = igeterrno();
            fprintf(stderr,
                "\tUnable to open <%s>, error = %s (%d)\n\r",
                 sessions[i],
                igeterrstr(errornumber),errornumber);
            exit(2);
        }
        icount++;
    }
    /*
    // Open some devices with a hardcoded interface
    */
    instances[icount] = iopen("2,1");
    if (instances[icount] == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <2,1>, error = %s (%d)\n\r",
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
    icount++;
```

2

```
    /*
    //  Open some devices with a hardcoded interface
    */
    instances[icount] = iopen("vxi,1");
    if (instances[icount] == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <vxi,1>, error = %s (%d)\n\r",
            igeterrstr(errornumber),errornumber);
        exit(4);
    }
    /*
    //      ....
    */
    exit(0);
}
```

iprintf

Description

Formats and writes data to a device or interface.

int CDECL
iprintf(INST ***id*****, char** ****format*** **[,** ***argument*****]...);**

id	Pointer to a session structure.
format	Pointer to a format control string.
argument	Optional arguments.

Remarks

This function writes characters and values to the device or interface of the session pointed to by *id*. *Format* is a string of ordinary characters, special formatting character sequences, and format specifications that control how to format and convert each *argument*. Ordinary characters and special formatting character sequences are written as they are encountered. The following defines valid special formatting sequences:

Sequence	Description
\n	Write the ASCII line-feed character. The END indicator is also automatically sent, but can be disabled using the **–t** type character.
\r	Write the ASCII carriage return character.
****	Write the backslash (\\) character.
\t	Write the ASCII tab character.
\###	Write the ASCII character specified by the three digit octal value ###.
\\"	Write the ASCII double-quote (") character.

Format specifications always begin with the percent sign (%) and are processed left to right. The first format specification causes the first *argument* value to be converted and written. The second format specification causes conversion and writing of the second *argument*, and so forth. To eliminate unpredictable results, there must be an *argument* for each format specification. If there are more *arguments* than format specifications, the excess *arguments* are ignored.

Floating point format types use non-reentrant C library calls; therefore, do not use **iprintf** function calls with floating point types within interrupt, SRQ, and error handlers.

To eliminate unpredictable results, do not mix **inbwrite** with **iprintf** and **iwrite** calls within a session.

Format Specification Fields

There are six format specification fields. Each field is a character, a series of characters, or a number that specifies how to convert and write the associated *argument*. A format specification has these fields:

%[*flags*] [*width*] [*.precision*] [*distance*] [*size*] *type*

Field	Description
type	Required character that determines how to interpret the associated *argument* (character, string, number, or pointer.)
flags	Optional characters that control the justification of characters and the printing of signs, blanks, decimal points. It also controls the printing of binary, octal and hexadecimal prefixes. More than one *flag* can appear in a format specification.
width	Optional character that specifies the minimum number of characters to write.

precision	Optional character that specifies the number of characters to write after the decimal point for numeric formats. For string formats, *precision* specifies the maximum number of characters to write.
distance	Optional character prefix that refers to the near or far object.
size	Optional character that specifies an argument size modifier.

The simplest format contains only the % sign and a *type* field character. The optional fields, that appear before the *type* field character control other formatting aspects. Any character that follows the % sign that is not a valid format field is interpreted as data.

Type Field Character

The *type* field character is the only required format specification field and determines whether the associated argument is interpreted as a character, string, number, or pointer. It also controls writing of the END indicator when a linefeed character is written. The following lists the valid *type* field characters and describes how the associated *argument* is interpreted:

Character	Type	Description
d	**int**	Signed decimal integer.
i	**int**	Signed decimal integer.
u	**int**	Unsigned decimal integer.
o	**int**	Unsigned octal integer.
x	**int**	Unsigned hexadecimal integer, using lower case letters.
X	**int**	Unsigned hexadecimal integer, using upper case letters.

2

f	**double**	Signed value having the form [–]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point depends on the *precision* field value.
e	**double**	Signed value having the form [-]*d.dddd*e[*sign*]*ddd*, where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or –.
E	**double**	Same as **e**, but the *argument* uses "E" instead of "e".
g	**double**	Signed value in the **f** or **e** format, whichever is more compact for the given value and *precision*. The **e** format is used only when the exponent of the value is less than –4 or greater than or equal to the *precision* value. Trailing zeros and decimal point are written only if necessary.
c	**int**	Single character.
C	**int**	Single character with the END indicator appended.
s	Pointer	Pointer to a null-terminated string. The null character or the *precision* value determines the length of the formatted string.

S	Pointer	Pointer to a null-terminates string that is written as an IEEE 488.2 STRING RESPONSE DATA block. The string is enclosed in double quotes ("). Double quotes within the string are double quoted ("").
n	Pointer to integer	Pointer to the number of characters converted and written to the buffer. This value is stored in the integer whose address is given as the argument.
p	Far pointer to **void**	Prints the address pointed to by the argument in the form *xxxx:yyyy*, where *xxxx* is the segment and *yyyy* is the offset, and the digits *x* and *y* are uppercase hexadecimal digits; **%hp** indicates a near pointer and prints only the offset of the address.
b	Pointer to data block	Pointer to a block of data that is written as an IEEE 488.2 DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA block. *Flags* must contain a long specifying the maximum the number of elements (specified by the *size* **w**, **i**, **z**, or **Z** or default) in the data block or an asterisk. An asterisk specifies that the next two arguments contain the number of bytes to write and a pointer to the data block, respectively. The number of bytes to write is an unsigned long type and has a maximum value of 0xFFFF. *Width* and *precision* are not allowed.

B	Pointer to data block	Same as **b**, except that the data block is written as an IEEE 488.2 INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA. This format writes the END indicator.
–t	N/A	Turns off sending of the END indicator when an ASCII line feed character is written from within the format string. The flag does not affect transmission of the END indicator for conversion with *types* **s**, **S**, **c**, and **C**.
+t	N/A	Turns on sending of the END indicator when an ASCII line feed character is written from within the format string. The flag does not affect transmission of the END indicator for conversion with *types* **s**, **S**, **c**, and **C**.

Flags Field Characters

The *flags* field character is optional and controls the justification of characters and the writing of signs, blanks, and decimal points. It also controls the writing of binary, octal, and hexadecimal prefixes, and modifies the meaning of the *type* field character. More than one *flags* character can be used in a format specification. The following describes the *flags* field characters and the defaults when that *flags* is not specified:

Flags	Definition	Default
–	Left-justify the result within the given field width.	Right justify.
+	Prefix data with a sign (+ or –) if the data is of a signed type. Can be used with *flags* @**1**, @**2**, or @**3**. Not valid with *flags* @**H**, @**Q**, or @**B**.	Only negative values are prefixed.
blank	Prefix with a blank if the value is signed and positive; the blank is ignored if both the "blank" and "+" flags appear. Can be used with *flags* @**1**, @**2**, or @**3**, but not valid with *flags* @**H**, @**Q**, or @**B**	No blank appears.
0	If *width* is prefixed with **0**, pad with zeros until the minimum width is reached. If "**0**" and "–" are specified, the **0** is ignored. If **0** is specified with an integer format (**i**, **u**, **x**, **X**, **o**, **d**), the **0** is ignored.	No padding
#	When used with *types* **o**, **x**, or **X**, prefixes any non-zero output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with *types* **e**, **E**, or **f**, always forces the output value to contain a decimal point.	Decimal point appears only if digits follow it.
	When used with *types* **g** or **G**, forces the output value to always contain a decimal point and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.

2

	Ignored when used with *types* **c**, **d**, **i**, **u**, or s.	
@1	Converts the *type* to an integer with no decimal point (NR1 compatible). Valid only with types **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@2	Converts the *type* to a number with at least one digit to the right of the decimal point (NR2 compatible). Valid only with the **d**, **f**, **e**, E, **g**, and **G** types.	Format data based on *type* only.
@3	Converts the *type* to a floating point number with exponential notations (NR3 compatible). Valid only with types **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@H	Create an IEEE 488.2 HEXADECIMAL NUMERIC RESPONSE DATA number (e.g. #H4A81). Valid only with *types* **d**, **f**, **e**, E, **g**, and **G**.	Format data based on *type* only.
@Q	Create an IEEE 488.2 OCTAL NUMERIC RESPONSE DATA number (e.g. #Q17774). Valid only with *types* **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@B	Create an IEEE 488.2 BINARY NUMERIC RESPONSE DATA number (e.g. #B11011000). Valid only with *types* **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.

Width Field Character

The *width* field character is optional and contains a non-negative decimal integer that specifies the minimum number of characters written. If the number of characters to write is less than the specified *width*, blanks are added to the left or right of the value, depending on whether the – flag is specified, until the minimum width is reached. If *width* is prefixed with the "0" flag, zeros are added until the minimum with is reached.

The *width* field character never causes the value to be truncated. If the number of characters to write is greater than the specified width or *width* is not given, all characters of the value are written (subject to *precision*).

If *width* is an asterisk (*), the next *argument* from the argument list is treated as an **int** and supplies the width value. The value to format immediately follows the *precision* value in the argument list. A nonexistent or small field does not cause truncation. If the result of the conversion is wider than the field width, the field expands to contain the conversion result.

Precision Field Character

The *precision* field is an option and contains a non-negative decimal integer, preceded by a period, that specifies the number of characters to write. Unlike the *width* field, *precision* can cause truncation of the output value, or rounding in the case of a floating point number.

If *precision* is an asterisk (*), the next *argument* from the argument list is treated as an **int** and supplies the precision value. The value to format immediately follows the *precision* value in the argument list. The following describes how *precision* values affect the various *types* (defaults are actions when *precision* is omitted with the *type*.)

Type	Meaning	Default
d, i, u, o, x, X	Specifies the minimum number of digits to write. If the number of digits in the argument is less than *precision*, the output is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default is 1.
e, E	Specifies the number of digits to write after the decimal point. The last written digit is rounded.	Default is 6. If *precision* is 0 or the period appears without a number following it, no decimal point is written.
f	Specifies the number of digits to write after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default is 6. If *precision* is 0 or the period appears without a number following it, no decimal point is written.
g, G	Specifies the maximum number of significant digits to write.	Six significant digits are written with any trailing zeros truncated.
c, C	No effect	Character is written.
s, S	Specifies the maximum number of character to write. Characters in excess of *precision* are not written	Characters are written until a null character is encountered.

If the *argument* corresponding to a floating-point specifier is infinite, indefinite, or not a number (NAN), the **iprintf** function returns the following:

Value	Returned Value
+ infinity	**1.#inf** *random-digits*
–infinity	**–1.#inf** *random-digits*
Indefinite	*digit*.**#IND** *random-digits*
NAN	*digit*.**#NAN** *random-digit*

Distance Field Character

The optional *distance* prefix refers to the distance to the object being printed (**F**ar or **N**ear).

F and **N** are not part of the ANSI or SICL definition and should not be used if ANSI or SICL portability is required.

The following demonstrates the use of **F, N, h,** and **l**.

Sample Code	Action
iprintf("%Ns");	Write **near string**
iprintf("%Fs");	Write **far string**
iprintf("%Nn");	Write **char** count in **near int**
iprintf("%Fn");	Write **char** count in **far int**
iprintf("%hp");	Write a 16-bit pointer (*xxxx*)
iprintf("%lp");	Write a 32-bit pointer (*xxxx:xxxx*)
iprintf("%Nhn");	Write **char** count in **near short int**
iprintf("%Nln");	Write **char** count in **near long int**
iprintf("%Fhn");	Write **char** count in **far short int**
iprintf("%Fln");	Write **char** count in **far int**

The specifications **%hs** and **%ls** have no meaning.

Size Field Character

The *size* field character is optional and is an *argument* modifier. The following defines the valid *size* entries:

Character	Description
h	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a **short int** or with *type* **u** to specify a **short unsigned int**. If used with type **p** , it indicates a 16-bit pointer (offset only).
l	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a long int. Use with the *type* **u** to specify a **long unsigned int**. Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **double** rather than a **float**. If used with *type* **p** , it indicates a 32-bit pointer.
	Use with *types* **b**.and **B** to specify that the argument is a pointer to an array of **long unsigned ints** (32-bits). The data block is sent as an array of 32-bit words. The longwords are byte swapped and padded as necessary so that they conform to IEEE 488.2.
L	Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **long double**.
w	Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **unsigned shorts** (16-bits). The data block is sent as an array of 16-bit words. *Flags* must be a long and specifies the number of words in the data block. The words are byte swapped and padded as necessary so that they conform to IEEE 488.2.

2

z	Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **floats**. The data block is sent as an array of 32-bit IEEE-754 floating point numbers. If the internal floating point representation of the computer is not IEEE-754 compliant, the numbers are converted before being written.
Z	Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **doubles**. The data block is sent as an array of 64-bit IEEE-754 floating point numbers. If the internal floating point representation of the computer is not IEEE-754 compliant, the numbers are converted before being written.

Return Value The function returns an integer indicating the actual number of format conversions performed. Conversions that require multiple arguments are counted as one conversion for the return value. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred during the write operation.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred during the write operation.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

2

See Also **iflush, ipromptf, iscanf, isetbuf, iwrite**

Example

```
/*
//      This program illustrates output formatting with iprintf
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void
check(int returncode);

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *startstring = "BEGIN";
    short blockresponsedata[4] = { 1, 2, 3, 4 };
    char end = ';';
    int index = 1;
    double seed = 3825.1e+15;
    char *sessionname = "EPC2";

#if defined(I_SICL_FMTIO)
    fprintf(stderr,
        "\tFormatted I/O is not supported on this implementation");
    exit(0);
#endif
    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = iprintf(instance,"%s\n",startstring);
    check(returncode);
    returncode = iprintf(instance,"%@Hd\n",index);
    check(returncode);
    returncode = iprintf(instance,"%le\n",seed);
    check(returncode);
    returncode = iprintf(instance,"%@Bg\n",seed);
    check(returncode);
    returncode = iprintf(instance,"%4wB\n",blockresponsedata);
    check(returncode);
    returncode = iprintf(instance,"%C",end);
    check(returncode);
}

void
```

```
check(int returncode)
{   int errornumber;

    /*
    //  Iprintf returns the number of format conversion.
    */
    errornumber = igeterrno();
    if (returncode != 1 || errornumber != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIprintf failed, error = %s (%d)\n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    exit(0);
}
```

2

iprompttf

Description

Sends formatted data to and reads formatted data from a device or interface.

int CDECL
ipromptf(INST *id*, **char** **writeformat*, **char** **readformat*
[,*argument*]...);

id	Pointer to a session structure.
writeformat	Pointer to write format.
readformat	Pointer to read format.
argument	Optional input arguments and (or pointer(s)) to the location(s) where the function stores the formatted data.

Remarks

This function performs both an **iprintf** function and an **iscanf** function in a single call. First data is written, then it is read.

Writeformat points to a format specification string that writes data to the device or interface of the session pointed to by *id*. It uses the number of *arguments* necessary to satisfy the format specification. The write format specification is identical to the **iprintf** format specification.

Readformat points to a read data format specification string that reads data from the device or interface of the session pointed to by *id*. *Readformat* uses the remaining arguments to satisfy the read format specification. The read format specification is identical to the **iscanf** format specification.

Interrupts that occur while a read is being executed are not processed until the read completes.

Return Value The function returns an integer indicating the total number of format conversions performed by both format specifications. Conversions that require multiple arguments are counted as one conversion for the return value. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iprintf, iscanf**

Example

```
/*
//      This example calls iprompt to program and
//      read an instrument.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char buffer[50] = {0};
    char *sessionname = "vdev1";

#if defined(I_SICL_FMTIO)
    fprintf(stderr,
          "\tFormatted I/O is not supported on this
implementation");
    exit(0);
#endif
```

2

```
    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ipromptf(instance,"rmx\n","%s",buffer);
    if (returncode != 1) {
        fprintf(stderr,
            "\tUnexpected number of Ipromptf conversions\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,
            "The data read from <%s> is %s\n\r",\
            sessionname,
            buffer);
    exit(0);
}
```

iread

Description

Reads data from a device or interface.

int PASCAL
iread(INST *id*, **char** **buf*, **unsigned long** *bufsize*, **int** **reason*,
unsigned long **actualcnt*);

id	Pointer to a session structure.
buf	Pointer to the data buffer.
bufsize	Number of data bytes to read.
reason	Pointer to the location where the functions stores the cause of read termination bit mask.
actualcnt	Pointer to a location where the function stores the actual number of bytes read from the device or interface.

Remarks

This function reads *bufsize* bytes from the device or interface of the session pointed to by *id* and stores them into the buffer beginning at *buf*. *Bufsize* has a maximum value of 0x10000. It performs no formatting or data conversion.

Reading ends when *bufsize* bytes are read, an END indicator is received, a termination character is received, or a timeout occurs. Unlike the **inbread** function, this function blocks until one of these three conditions is met.

When *id* specifies a device session, data is read using interface independent communications methods. When *id* specifies an interface session, data is read in raw mode using interface specific methods.

If *actualcnt* is not null, the function stores the number of bytes read in the referenced memory location.

For VXI device sessions, the function issues BYTE REQUEST word-serial commands. The function only supports message based VXI devices; other VXI devices cause an error.

For VXI interface sessions, the function generates an **I_ERROR_PARAM** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's listen address, followed by the device's talk address. Finally, the function reads the data bytes.

For GPIB interface sessions, the function reads data from a GPIB interface without performing any addressing.

If *reason* is not null, the function stores a bit mask describing why the read terminated in the referenced memory location. These constants define valid bits in the mask pointed to by *reason*:

Constant	Description
I_TERM_CHR	Termination character received (see **itermchr**)
I_TERM_END	END indicator received
I_TERM_MAXCNT	*Bufsize* bytes read

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred during the read operation.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred during the read operation.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based, or *buf* is null.
I_ERR_TIMEOUT	A timeout occurred.

See Also **igettermchr, inbread, inbwrite, itermchr, itimeout, iwrite**

Example

```
/*
//      This example calls iread to read an instrument's
//      response
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{    INST instance;
     int returncode, reason, errornumber;
     unsigned long readcount;
     char buffer[50] = {0};
     char *sessionname = "vdev1";
```

2

```
    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    (void) iprintf(instance,"rmx\n");
    returncode = iread(instance,
                    buffer,
                    sizeof(buffer),
                    &reason,
                    &readcount);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIread failed, error = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    buffer[(short) readcount] = 0;
    fprintf(stdout,
            "The data read from <%s> is %s\n\r",
            sessionname,
            buffer);
    fprintf(stdout,"Read termination reason(s):\n\n\r");
    if (reason & I_TERM_CHR)
        fprintf(stdout,"\tI_TERM_CHR\n\r");
    if (reason & I_TERM_END)
        fprintf(stdout,"\tI_TERM_END\n\r");
    if (reason & I_TERM_MAXCNT)
        fprintf(stdout,"\tI_TERM_MAXCNT\n\r");
    exit(0);
}
```

ireadstb

Description

Reads the status byte from a device.

int PASCAL
ireadstb(INST *id*, unsigned char **statusbyte*);

id	Pointer to a device session structure.
statusbyte	Pointer to a location where the function stores the device's status byte.

Remarks

This function reads the device status byte of the device of the session pointed to by *id* and is valid only for device sessions.

For VXI device sessions, the function issues a READ STB word-serial command. The function only supports message-based VXI devices; other VXI devices cause an error.

For GPIB device sessions, the function issues a GPIB serial poll (SPOLL) command.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies an interface or commander session or a VXI device that is not message-based, or *statusbyte* is null.
I_ERR_TIMEOUT	A timeout occurred.

See Also **isetstb, itimeout**

Example

```
/*
//      This example uses ireadstb to issue a VXI
//      word serial READ STB command.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vdev1";
    unsigned char statusbyte;
```

```
    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ireadstb(instance,&statusbyte);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIreadstb failed, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    fprintf(stdout,
            "Status byte = %x",statusbyte);
    exit(0);
}
```

2

iremote

Description

Puts a device in remote mode.

int PASCAL
iremote(INST *id***);**

id	Pointer to a device session structure.

Remarks

This function places the session device pointed to by *id* into remote mode and is valid only for device sessions.

For VXI device sessions, the function issues a SET LOCK word-serial command. The function only supports message-based VXI devices; other VXI devices cause an error.

For GPIB device sessions, the function asserts the REN line then addresses the device to listen.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies an interface or commander session or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also ilocal

Example

```
/*
//      This example uses iremote to issue a SET LOCK word
//      word command.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = iremote(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIremote failed, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    exit(0);
}
```

2

iscanf

Description Reads and formats data from a device or interface.

int CDECL
iscanf(INST *id*, **char** **format* [, **argument*]...);

id	Pointer to a session structure.
format	Pointer to a format control string.
argument	Pointer(s) to locations where the function stores the formatted data.

Remarks This function reads a series of characters and values from the device or interface session pointed to by *id*. The characters and values are read into the locations pointed to by *argument*. *Format* is a string of ordinary characters that control how to format and convert characters from the specified device or interface. It can contain one or more of the following:

- The white-space characters blank (" "), tab (**\t**), or newline (**\n**). A white-space character causes **iscanf** to read, but not store, all consecutive white-space characters in the input up to the next non-white-space character. One white-space character in the *format* string matches any number (including 0) and combination of white-space characters in the input.
- Non-white-space characters, except the percent sign (%). A non-white-space character causes **iscanf** to read, but not store, a matching non-white-space character. If the read character does not match the *format* character, **iscanf** terminates.
- Format specifications. Format specifications begin with the percent sign (%) and cause **iscanf** to read and convert input characters into values of a specified type. The value is assigned to an argument in the *argument* list.

Format specifications always begin with the percent sign (%) and are read left to right. Characters outside the format specification are expected to match the sequence of characters from the device or interface. The matching characters from the device or interface are scanned but not stored. If a scanned character does not match the format specification **iscanf** terminates.

The first format specification causes the first input field from the device or interface to be converted and written to the location pointed to by the first *argument*. The second format specification causes conversion of the second input field from the device or interface to be converted and written to the location pointed to by the second *argument*, and so forth. There must be enough format specifications and arguments for the input field being read for the results to be predictable. Excess format specifications and arguments are ignored.

Format Specification Fields

There are six format specification fields. Each field is a character, a series of characters, or number signifying a format option. The following defines the form of a format specification:

%[*] [*flags*] [*width*] [*distance*] [*size*] *type*

Field	Description
type	Required character that determines whether the associated input field is interpreted as a character, string, number, or pointer.
*	Optional character that suppresses assignment of the next input field. The field is scanned but not stored.

flags	Optional character that specifies a maximum size.
width	Optional character that specifies the maximum number of characters to read.
distance	Optional character prefix that refers to the near or far object.
size	Optional character that specifies an argument size modifier.

The simplest format contains only the % sign and a *type* field character. The option fields that appear before the *type* field character control other formatting aspects.

Type Field Character

The *type* field character is the only required format field and determines whether the read data is interpreted as a character, string, number, or pointer. It also controls whether the read data terminates with a END indicator. The following describes the *type* field characters:

Character	Expected Input Type	Argument Type
d	Decimal integer in either IEEE 488.2 DECIMAL NUMERIC PROGRAM DATA (NRf) or NON-DECIMAL NUMERIC PROGRAM DATA (#H, #Q, and #B) format.	Pointer to **int**.
D	Decimal integer in either IEEE 488.2 DECIMAL NUMERIC PROGRAM DATA (NRf) or NON-DECIMAL NUMERIC PROGRAM DATA (#H, #Q, and #B) format.	Pointer to **long**

i	Decimal, octal, or hexadecimal integer.	Pointer to **int**.
I	Decimal, octal, or hexadecimal integer.	Pointer to **long**
u	Unsigned decimal integer	Pointer to **unsigned int**.
U	Unsigned decimal integers	Pointer to **long**
o	Octal integer	Pointer to **int**.
O	Octal integer	Pointer to **long**
x,X	Hexadecimal integer	Pointer to **int**.
e, E, f, g, G	Floating-point value in either IEEE 488.2 DECIMAL NUMERIC PROGRAM DATA (NRf) or NON-DECIMAL NUMERIC PROGRAM DATA (#H, #Q, and #B) format. The value consists of an optional sign (+ or –), a series of one or more decimal digits containing a decimal point, and an optional exponent (**e** or **E**) followed by an optionally signed integer value.	Pointer to **float**.
c	Character. White-space characters that are ordinarily skipped are read when **c** is specified. To read the next non-white-space character use "**%1c**".	Pointer to a **char**.

2

s	Null-terminated string where leading white-space characters are ignored and all ordinary characters are read until a white-space character is read. *Flags* can contain either an integer or #. When *flags* is an integer, it specifies the maximum string size. The string size must be large enough to hold the characters and a NULL character. When *flags* contains a #, it specifies that the next *argument* contains a pointer to the maximum size of the string. If maximum number of characters is read before a white-space character, all additional characters are read and discarded until a white-space character is found.	Pointer to a string.
S	Null-terminated string that conforms to IEEE 488.2 STRING RESPONSE DATA. Leading white-space before the required double quote is ignored, then all characters up to the next double quote are read. Two double quote characters are converted to a single quote. The beginning and ending double quotes are not inserted into the argument. *Flags* is the same as s.	Pointer to a string.

n	No input read.	Pointer to **int**, into which is stored the number of characters read so far.
p	Value in the form *xxxx.yyyy*, where *xxxx* is the segment and *yyyy* is the offset and the digits *x* and *y* are upper case hexadecimal digits	Pointer to far pointer to **void**.
b	Data block that conforms to IEEE 488.2 ARBITRARY BLOCK PROGRAM DATA. *Flags* must contains a long that specifies the number of elements in the data block or an #. If *flags* contains #, two arguments are used. The first contains a pointer to a long containing the size of the second argument, which is a pointer to the array.	Pointer to data block.
t	END indicator terminated string. *Flags* is the same as s. The stored string is null terminate. If the maximum number of characters is read before an END indicator is read, all additional characters are read and discarded until an END indicator is read.	Pointer to a string.

To read characters not delimited by white-space characters, a set of characters in brackets ([ ]) can be substituted for the s type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. Use a caret (^) to reverse the effect.

To store a string without storing the terminating null character (\0), use the specification **%nc**, where n is a decimal integer specifying the number of characters to store.

The **iscanf** function can stop converting a field for a variety of reasons:

- The specified width has been reached.
- The next character cannot be converted as specified.
- The next character conflicts with a character in the format specification string that it is supposed to match.
- The next character fails to appear in a given character set.

After reading stops, the next input field is considered to begin at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent operations.

An input field is defined as all characters up to the first white-space character, or up to the first character that can not be converted as specified, or until *width* is reached.

Flags Field Character

The *flags* character is optional.

Flag	Meaning
#	When used with *type* **b**, specifies that the next *argument* contains a pointer to a long that contains the size of the second *argument* which is a pointer to the data array.
#	When used with *type* **s**, **S**, or **t** format, specifies that the next *argument* contains a pointer to an integer that is the maximum size of the string.

Width Field Character

The *width* field is an optional field containing a positive decimal integer that controls the maximum number of characters read. No more than *width* characters are converted and stored at the corresponding *argument*. Fewer than *width* characters may be read if a white-space character or a character that can not be converted is read before *width* is reached.

Distance Field Character

The optional *distance* prefix refers to the distance to the memory location used to store the converted *argument*. The prefixes **h** and **l** refer to the size of the object begin read.

F and **N** are not part of the ANSI or SICL definition and should not be used if ANSI or SICL portability is required.

The following demonstrates the use of **F, N, h,** and **l**.

Sample Code	Action
iscanf("%Ns", &x);	Read a string into near memory.
iscanf("%Fs", &x);	Read a string into far memory.
iscanf("%Nd", &x);	Read an **int** into near memory.
iscanf("%Fd", &x);	Read an **int** into far memory.
iscanf("%Nld", &x);	Read a **long int** into near memory.
iscanf("%Fld", &x);	Read a **long int** into far memory.
iscanf("%Nhp", &x);	Read a 16-bit pointer into near memory.
iscanf("%Nlp", &x);	Read a 32-bit pointer into near memory.
iscanf("%Fhp", &x);	Read a 16-bit pointer into far memory.
iscanf("%Flp", &x);	Read a 32-bit pointer into far memory.

Floating point format types use non-reentrant C library calls; therefore, do not use **iscanf** function calls with floating point types within interrupt handlers.

Size Field Character

The *size* field character is optional and is an argument modifier. The following defines the valid *size* entries:

Character	Description
h	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a **short int** or with *type* **u** to specify a **short unsigned int**. If used with *type* **p** , it indicates a 16-bit pointer (offset only).
l	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a **long int**. Use with the *type* **u** to specify a **long unsigned int**. Use with *types* **e, E, f, g,** and **G** to specify a **double** rather than a **float**. If used with *type* **p**, it indicates a 32-bit pointer. Use with *type* **b** to specify that the argument is a pointer to an array of **long unsigned ints** (32-bits). The data block is sent as an array of 32-bit words. *Flags* must contain an integer or #. When *flags* contains a long, it specifies the maximum number of longwords to read. When *flags* contains #, it specifies that the next *argument* contains a pointer to a long containing the size of the following *argument*. For *types* **s**, **S**, **t**, and **B**, *flags* must contain a # or a *width* must be specified for types. The longwords are byte swapped and padded as necessary so that they conform to IEEE 488.2.
L	Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **long double**.

w	Use with *type* **b** to specify that the argument is a pointer to an array of **unsigned shorts** (16-bits). The data block is sent as an array of 16-bit words. *Flags* must contain a long or #. When *flags* contains a long, it specifies the maximum number of words to read. When *flags* contains #, it specifies that the next *argument* contains a pointer to a long containing the size of the following *argument*. The words are byte swapped and padded as necessary so they conform to IEEE 488.2.
z	Use with *type* **b** to specify that the argument is a pointer to an array of **floats**. The data block is read as an array of 32-bit IEEE-754 floating point numbers. *Flags* must contain a long or #. When *flags* contains a long, it specifies the maximum number of **floats** to read. When *flags* contains #, it specifies that the next *argument* contains a pointer to a long containing the size of the following *argument*
Z	Use with *type* **b** to specify that the argument is a pointer to an array of **doubles**. The data block is read as an array of 64-bit IEEE-754 floating point numbers. *Flags* must contain an integer or #. When *flags* contains an integer, it specifies the maximum number of **doubles** to read. When *flags* contains #, it specifies that the next *argument* contains a pointer to a long containing the size of the following *argument*.

Return Value

The function returns an integer indicating the actual number of format conversions performed. Conversions that require multiple *arguments* are counted as one conversion for the return value. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred during the read operation.

I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred during the read operation.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iflush, ipromptf, iread, iscanf, isetbuf**

2

Example

```
/*
//      This program illustrates input formatting with iscanf The
//      program prints to a device, EPC2, that simple echoes all
//      input. The printed value should be identical to the scanned
//      value.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sicl.h"

void main(void)
{    INST instance;
     int returncode, errornumber;
     char startstring[7] = {0}, startstring2[7] = {0};
     double seed1 = 3825.1e+7, seed2 = 0;
     char *sessionname = "EPC2";

#if !defined(I_SICL_FMTIO)
     fprintf(stderr,
            "\tFormatted I/O is not supported on this
implementation");
     exit(0);
#endif
     /*
     // Open a device session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
     }
     (void) itimeout(instance,500L);
     returncode = iprintf(instance,"%s\n",startstring);
     if (returncode != 1) {
        fprintf(stderr,"\tIprintf failed\n\r");
        exit(2);
     }
     returncode = iscanf(instance,"%s\n",&startstring2);
     if (strcmp(startstring2,startstring)) {
        fprintf(stderr,"\tUnexpected input\n\r");
        exit(3);
     }
```

2

```
        (void) iflush(instance,I_BUF_READ);
        returncode = iprintf(instance,"%le\n",seed1);
        if (returncode != 1) {
            fprintf(stderr,"\tIprintf failed\n\r");
            exit(2);
        }
        returncode = iscanf(instance,"%le",&seed2);
        errornumber = igeterrno();
        if (returncode != 1 || errornumber != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIscanf failed, error = %s (%d)\n\r",
                igeterrstr(errornumber),errornumber);
            exit(4);
        }
        fprintf(stdout,"seed2 = \t%le\n\r",seed2);
        exit(0);
    }
```

isetbuf

Description Sets the size of the formatted I/O read and/or write buffers.

int PASCAL
isetbuf(INST *id*, **int** *buffermask*, **int** *buffersize***);**

id	Pointer to a session structure.
buffermask	Buffer selection mask.
buffersize	Buffer size, in bytes.

Remarks This function sets the read buffer and/or write buffer size for the device or interface session pointed to by *id*.

Buffermask is an OR'd combination of the following buffer selection constants:

Constant	Description
I_BUF_READ	Discard the session's current read buffer and read from the device or interface of the session pointed to by *id* until and END indicator is read. Also, resynchronizes the next **iscanf** call to read until EOI is received.
I_BUF_WRITE	Writes all data in the session's current write buffer to the device or interface session pointed to by *id*.

Specifying a *buffersize* equal to zero disables buffering and all reads and writes take place directly to the device or interface.

2

Specifying a *buffersize* greater than zero creates a new buffer of the specified size. The write buffer is written to the device or interface anytime the buffer fills or when the END indicator is placed in the buffer. Read buffers retain data until explicitly flushed using **iflush**.

Specifying a *buffersize* less than zero creates a buffer of the absolute value of the specified size. The write buffer is written to the device or interface anytime the buffer fills, when the END indicator is placed in the buffer, or at the end of each **iprintf** or **ipromptf** call. Read buffers flush data at the end of every **iscanf** or **ipromptf** call.

Read and write buffers are of length zero when the session opens. Closing and reopening a session flushes the buffers and resets their length to zero.

If the function fails and the returned error is **I_ERR_NORSRC**, the buffer size for *buffermask* is set to zero.

Return Value

.The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NORSRC	The system contains insufficient resources to allocate the specified buffer.

See Also

iflush, ipromptf, iprintf, iscanf

2

Example

```
/*
//      This program illustrates the effect of the buffersize
//      on iprintf
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void
check(int returncode);

void main(void)
{    INST instance;
     int returncode, errornumber, bufferindex;
     char *startstring = "BEGIN";
     short blockresponsedata[4] = { 1, 2, 3, 4 };
     int index = 1;
     double seed = 3825.1e+15;
     char *sessionname = "EPC2";
     int buffersize[] = { -100, 0, 100 };

#if !defined(I_SICL_FMTIO)
     fprintf(stderr,
            "\tFormatted I/O is not supported on this
implementation");
     exit(0);
#endif
     /*
     // Open a device session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     for (bufferindex = 0; bufferindex < 3; bufferindex++) {
         returncode = isetbuf(instance,
                              I_BUF_WRITE,
                              buffersize[bufferindex]);
         returncode = iprintf(instance,"%s",startstring);
         check(returncode);
         returncode = iprintf(instance,"%@Hd",index);
         check(returncode);
         returncode = iprintf(instance,"%le",seed);
         check(returncode);
         returncode = iprintf(instance,"%@Bg",seed);
         check(returncode);
         returncode = iprintf(instance,"%4wB",blockresponsedata);
         check(returncode);
```

2

```
        /*
        //  For buffersize's > 0, the buffer is only flushed
        //  when the buffer is full or the END indicator
        //  is placed into the buffer.  The buffer is
        //  being implicitly flushed by placing "\n"
        //  into the buffer.
        */
        if (buffersize[bufferindex] > 0) {
            returncode = iprintf(instance,"\n");
            check(returncode);
        }
    }
    exit(0);
}

void
check(int returncode)
{   int errornumber;

    errornumber = igeterrno();
    if (returncode != 1 || errornumber != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIprintf failed, error = %s (%d)\n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
}
```

isetdata

Description Stores a pointer to the session data structure.

int PASCAL
isetdata(INST *id*, void **data*);

id	Pointer to a session structure.
data	Pointer to a data structure.

Remarks This function places a pointer to data structure and associates it with the session pointed to by *id*. The pointer can be queried with the **igetdata** function.

The session data structure is a 4-byte memory block. Its contents is application specific, but typically, it is a pointer to the application's data structure.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.

See Also **igetdata**

Example See **igetdata**.

isetintr

Description Enables and disables interrupt reception.

int PASCAL
isetintr(INST *id*, **int** *intrtype*, **long** *intrenable*);

id	Pointer to a session structure.
intrtype	Interrupt type.
intrenable	Interrupt enable flag.

Remarks This function enables or disables interrupt reception for the interrupt type specified by *intrtype* for the session pointed to by *id*.

The following are valid constants for *intrtype*:

Constant	Description
I_INTR_GPIB_IFC	Interrupt on GPIB interface clear (GPIB interface sessions only).
I_INTR_INTFACT	Interrupt when an interface becomes active (GPIB interface sessions only).
I_INTR_INTFDEACT	Interrupt when an interface deactivates (GPIB interface sessions only).
I_INTR_OFF	Disable all interrupts.
I_INTR_TRIG	Interrupt on a trigger (EPC-7 interface sessions only).
I_INTR_VXI_SIGNAL	Interrupt on a VXI signal or a VME interrupt from a servant VXI device (VXI device sessions only).
I_INTR_VXI_VME	Interrupt on a VME interrupt from a non-servant device (VXI interface sessions only).
I_INTR_VXI_UNKSIG	Interrupt on a VXI signal from a non-servant device (VXI interface sessions only).

When *intrenable* is zero, the function disables the interrupts specified by *intrtype*; a value other than zero enables the selected interrupt.

When *intrtype* is **I_INTR_TRIG** and *id* specifies a VXI interface session on an EPC-7, *intrenable* becomes a bit mask that specifies a trigger interrupt. Setting *intrenable* to zero disables the trigger interrupt. The following are valid constants for *intrenable* when *intrtype* is **I_INTR_TRIG:**

Constant	Description
I_TRIG_ALL	All valid triggers.
I_TRIG_STD	Standard trigger.
I_TRIG_EXT0	EXT trigger 0, if it is mapped as an input trigger (see **ivxitrigroute**).
I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

The trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxirigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Proper VXI trigger interrupt operation on an EPC-7 requires direct program manipulation of EPC-7 hardware, refer to Chapter 4, *Advanced Topics*, for additional information.

2

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	The hardware/software platform does not support the specified *intrtype/intrenable*.
I_ERR_PARAM	*Id* specifies a session whose type is inconsistent with the given *intrtype* or *intrenable* is invalid.

See Also **igetonintr, iintron, iintroff, ionintr**

Example See **igetonintr**.

2

isetlockwait

Description

Determines whether accessing a locked device or interface suspends the calling process or generates an error.

int PASCAL
isetlockwait(INST *id*, **int** *waitflag***);**

id	Pointer to a session structure.
waitflag	Lock-waitflag.

Remarks

When *waitflag* is non-zero (default) and the device or interface session pointed to by *id* is locked by another process, all interlocked operations using the session pointer *id* suspend the calling process until the lock is released. When *waitflag* is zero, all interlocked operations using the pointer *id* return an error.

Under DOS, a session's lock wait flag has no effect and locking conflicts always generate an **I_ERR_LOCKED** error. This error is because DOS does not support process preemption.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.

See Also

igetlockwait, ilock, iunlock

isetstb

Description

Sets this controller's status byte.

int PASCAL
isetstb(INST *id*, **unsigned char** *statusbyte);*

id	Pointer to a commander session structure.
statusbyte	Status byte.

Remarks

The SICL library supports SICL standard level 2F (support for device and interface sessions only); therefore, this function always returns an error.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_PARAM	*Id* specifies an device or interface session.

See Also

ireadstb

itermchr

Description

Specifies a session's termination character.

int PASCAL
itermchr(INST *id*, **int** *termchar*);

id	Pointer to a session structure.
termchar	Termination character.

Remarks

This function specifies the termination character for the session pointed to by *id*. The functions **inbread**, **ipromptf**, **iread**, and **iscanf** use the termination character to signal the end of a read operation.

Use the **igettermchr** function to get the current termination character.

Valid *termchr* values are −1 and 0 through 255, inclusive. The value −1 (default) indicates that no termination character is set. A value of 0 through 255 is a termination character.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function was unable to set the session's termination character.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Termchr* is invalid. Valid values are -1 and 0 through 255, inclusive.

See Also **igettermchr, inbread, ipromptf, iread, iscanf**

Example See **igettermchr**.

2

itimeout

Description Set a session's timeout value.

int PASCAL
itimeout(INST *id*, **long** *timeout***);**

id	Pointer to a session structure.
timeout	Timeout interval, in milliseconds.

Remarks This function specifies the timeout value for the session pointed to by *id*. A timeout value is the time interval to wait for an operation to complete before aborting. When an operation aborts because of a timeout, the aborted function returns an error indicating that the call timed out. Timeouts affect these SICL functions:

imap	**inbread**	**itrigger**
iclear	**inbwrite**	**ivxigettrigroute**
igpibatnctl	**iopen**	**ivxitrigoff**
igpibllo	**iprintf**	**ivxitrigon**
igpibpassctl	**ipromptf**	**ivxitrigroute**
igpibppoll	**iread**	**ivxiwaitnormop**
igpibppollconfig	**ireadstb**	**ivxiws**
igpibrenctl	**iremote**	**iwaithdlr**
igpibsendcmd	**iscanf**	**iwrite**
ilocal	**isetbuf**	**ixtrig**
ilock	**isetstb**	

The *timeout* value is in milliseconds. A *timeout* value of less than or equal to zero indicates an infinite timeout. The default *timeout* value is 0.

Use **igettimeout** to get a session's current *timeout* value.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.

See Also **igettimeout**

Example See **igettimeout**.

2

itrigger

Description Sends a trigger to a device or interface.

int PASCAL
itrigger(INST *id*);

id Pointer to a session structure.

Remarks This function sends a trigger to the device or interface of the session pointed to by *id*. When *id* specifies a device session, the trigger is sent to the device of the session and is dependent on the interface (VXI or GPIB), but the trigger is an addressed trigger. When *id* specifies an interface session, the trigger is interface specific.

For VXI device sessions, the function issues a TRIGGER word-serial command. Only message based VXI devices are supported. Other VXI devices cause an error.

For VXI interface sessions, the function generates an error.

For GPIB device sessions, the function issues an addressed Group Execute Trigger (GET) command.

For GPIB interface sessions, the function issues a broadcast Group Execute Trigger (GET) command.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a commander session, a VXI interface session, or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **itimeout,ixtrig**

Example

```
/*
//      This example uses itrigger to issue a TRIGGER word
//      word command.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{    INST instance;
     int returncode, errornumber;
     char *sessionname = "vdev1";

     /*
     //  Open a device session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
```

2

```
    returncode = itrigger(instance);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tItrigger failed, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    exit(0);
}
```

iunlock

Description Unlocks a device or interface.

int PASCAL
iunlock(INST *id*);

id	Pointer to a session structure.

Remarks This function unlocks the device or interface of the session pointed to by *id*.

Closing a session implicitly unlocks any locks held by the session.

Attempting to unlock a device or interface that is not locked generates an error.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOLOCK	*Id* specifies a device or interface that is not locked by the calling process.

See Also **ilock**

Example See **ilock**.

2

iunmap

Description Deletes an address space mapping.

int PASCAL
iunmap(INST *id*, **char** **mapaddress*, **int** *mapspace*, **unsigned int** *pagestart*, **unsigned int** *pagecnt*);

id	Pointer to a session structure.
mapaddress	Mapped address pointer.
mapspace	Mapping address space.
pagestart	Starting page number.
pagecnt	Number of mapped pages.

Remarks *Mapaddress* is a pointer returned by a previous **imap** call. Valid constants for *mapspace* are:

Constant	Description
I_MAP_A16	Unmap the A16 address space
I_MAP_A24	Unmap the A24 address space (page size 64K bytes)
I_MAP_A32	Unmap the A32 address space (page size 64K bytes)
I_MAP_VXIDEV	Unmap a VXI device's configuration registers
I_MAP_EXTEND	Unmap the A24/A32 address space that corresponds to this EPC (EPC-2 and EPC-7 only).

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_BADMAP	*Mapaddress* does not correspond to a valid mapping.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).

See Also **imap, imapinfo, iopen**

Example

```
/*
//      This example uses explicitly uses iunmap to release
//      control of a memory space.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

void main(void)
{   INST instance;
    short *vxiregisters;
    int returncode, errornumber, vxiid;
    char *sessionname = "vdev1";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
```

2

```
vxiregisters = (short *)
   imap(instance,I_MAP_VXIDEV,0,0,NULL);
if (vxiregisters == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tImap call failed\n\r");
    fprintf(stderr,
        "\tError = %s (%d) \n\r",
        igeterrstr(errornumber),errornumber);
    exit(2);
}
returncode = iwblockcopy(instance,
                    (unsigned short *) vxiregisters,
                    (unsigned short *) &vxiid,
                    1L,
                    -1);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,"\tIwblockcopy unsuccessful\n\r");
    fprintf(stderr,"\tError = %s (%d) \n\r",
        igeterrstr(returncode),returncode);
    exit(3);
}
fprintf(stdout,"Manufacturer ID of device <%s> is %d",
        sessionname,
        vxiid & 0xfff);
/*
// Remove the address space mapping
*/
returncode = iunmap(instance,(char *) vxiregisters,0,0,0);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,"\tIunmap unsuccessful\n\r");
    fprintf(stderr,"\tError = %s (%d) \n\r",
        igeterrstr(returncode),returncode);
    exit(4);
}
exit(0);
}
```

ivxibusstatus

Description Gets the VXI bus status.

int PASCAL
ivxibusstatus(INST *id*, **int** *request*, **int** **result*);

id	Pointer to a VXI interface session structure.
request	Status request.
result	Pointer to a location where the functions stores the requested status information.

Remarks This function places the VXIbus interface status information specified by *request* in the location pointed to by *result*. It is valid only for VXI interface sessions.

The following are valid constants for *request*:

Constant	Description
I_VXI_BUS_CMDR_LADDR	Return the commander device logical address of this EPC (−1 = no commander exists, either because this EPC is a top-level commander or normal operation has not be established).
I_VXI_BUS_LADDR	Return the logical address of this EPC.
I_VXI_BUS_MAN_ID	Return the manufacturer's ID of this EPC.
I_VXI_BUS_MODEL_ID	Return the model ID of this EPC.
I_VXI_BUS_NORMOP	Return normal operation status of this EPC (1 = normal, 0 = other).

I_VXI_BUS_PROTOCOL	Return the protocol register value of this EPC.
I_VXI_BUS_SERVANT_AREA	Return the servant area size of this EPC.
I_VXI_BUS_SHM_ADDR_SPACE	Return this EPC's VXI memory space. Returns 24 for A24 space or 32 for A32 space. EPC-2 and EPC-7 only.
I_VXI_BUS_SHM_PAGE	Return this EPC's VXI memory location, in pages. For A24 memory, page size is 256 bytes. For A32 memory, page size is 64K bytes. EPC-2 and EPC-7 only.
I_VXI_BUS_SHM_SIZE	Returns this EPC's VXI memory size in pages. For A24 memory, page size is 256 bytes. For A32 memory, page size is 64K bytes. EPC-2 and EPC-7 only.
I_VXI_BUS_TRIGGER	Return a bit mask of the currently asserted trigger lines (see **ivxitrigroute**). EPC-2 and EPC-7 only.
I_VXI_BUS_VXIMXI	Returns this EPC's MXI bus status. Returns 1 if this EPC is a MXI interface, 0 otherwise.
I_VXI_BUS_XPROT	Return the Read Protocol word-serial command response value of this EPC.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_PARAM	*Id* specifies a device or commander session, *request* is invalid, or *result* is null.

See Also **iopen**

Example

```
/*
//      This example calls ivxibusstatus to display
//      the VXI bus status information.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define DIM(x)          (sizeof(x)/sizeof(int))

int requests[] = { I_VXI_BUS_CMDR_LADDR,
                   I_VXI_BUS_LADDR,
                   I_VXI_BUS_MAN_ID,
                   I_VXI_BUS_MODEL_ID,
                   I_VXI_BUS_NORMOP,
                   I_VXI_BUS_PROTOCOL,
                   I_VXI_BUS_SERVANT_AREA,
                   I_VXI_BUS_SHM_ADDR_SPACE,
                   I_VXI_BUS_SHM_PAGE,
                   I_VXI_BUS_SHM_SIZE,
                   I_VXI_BUS_TRIGGER,
                   I_VXI_BUS_VXIMXI,
                   I_VXI_BUS_XPROT };
```

2

```
char *requeststrings[] = {
                    "I_VXI_BUS_CMDR_LADDR        ",
                    "I_VXI_BUS_LADDR             ",
                    "I_VXI_BUS_MAN_ID            ",
                    "I_VXI_BUS_MODEL_ID          ",
                    "I_VXI_BUS_NORMOP            ",
                    "I_VXI_BUS_PROTOCOL          ",
                    "I_VXI_BUS_SERVANT_AREA      ",
                    "I_VXI_BUS_SHM_ADDR_SPACE    ",
                    "I_VXI_BUS_SHM_PAGE          ",
                    "I_VXI_BUS_SHM_SIZE          ",
                    "I_VXI_BUS_TRIGGER           ",
                    "I_VXI_BUS_VXIMXI            ",
                    "I_VXI_BUS_XPROT             "};


void main(void)
{   INST instance;
    int returncode, errornumber, result;
    char *sessionname = "vxi";
    register short vinductor;

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
        exit(1);
    }
    for (vinductor = 0; vinductor < DIM(requests); vinductor++) {
         returncode = ivxibusstatus(instance,
                                 requests[vinductor],
                                 &result);
         if (returncode != I_ERR_NOERROR) {
             fprintf(stderr,
                 "\tUnable to execute ivxibusstatus\n\r");
             fprintf(stderr,
                 "\tRequest = %s",
                 requeststrings[vinductor]);
             fprintf(stderr,
                 "\tError = %s (%d)\n\r",
                 igeterrstr(returncode),returncode);
             exit(2);
         }
         fprintf(stdout,"%s = \t%d\n\r",
             requeststrings[vinductor],
             result);
    }
    exit(0);
}
```

ivxigettrigroute

Description

Gets a current trigger routing.

int PASCAL
ivxigettrigroute(INST *id*, **unsigned long** *intriggermask*, **unsigned long** **outtriggermask);*

id	Pointer to VXI interface session structure.
intriggermask	Input triggermask.
outtriggermask	Pointer to a location where the functions stores a trigger mask that describes the routing of the input trigger.

Remarks

This function places a mask of current trigger routing from *intriggermask* in the location pointed to by *outtriggermask*. It is valid only for VXI interface sessions.

The following are valid constants for *intriggermask*:

Constant	Description
I_TRIG_ALL	All valid triggers.
I_TRIG_STD	Standard trigger.
I_TRIG_CLK0	Internal clock trigger 0.
I_TRIG_CLK1	Internal clock trigger 1.
I_TRIG_CLK2	Internal clock trigger 2.
I_TRIG_ECL0	ECL trigger 0.
I_TRIG_ECL1	ECL trigger 1.
I_TRIG_ECL2	ECL trigger 2.
I_TRIG_ECL3	ECL trigger 3.
I_TRIG_EXT0	External trigger 0.
I_TRIG_EXT1	External trigger 1.
I_TRIG_EXT2	External trigger 2.
I_TRIG_EXT3	External trigger 3.

I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

Use **ivxitrigroute** to route triggers.

Specifying an *intriggermask* of **I_TRIG_ALL** returns a mask of all valid triggers for this EPC.

Specifying an *intriggermask* of **I_TRIG_STD** returns a mask of triggers corresponding to the **I_TRIG_STD** constant.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_PARAM	*Id* specifies a device or commander session, *intriggermask* specifies an invalid trigger bit, or *outtriggermask* is null.

See Also **ivxitrigoff, ivxitrigon, ivxitrigroute, ixtrig**

2

Example

```
/*
//      This example uses ivxigettrigroute to get the current
//      trigger routing.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

unsigned long triggermasks[] = {  I_TRIG_ALL,
                                  I_TRIG_STD,
                                  I_TRIG_CLK0,
                                  I_TRIG_CLK1,
                                  I_TRIG_CLK2,
                                  I_TRIG_ECL0,
                                  I_TRIG_ECL1,
                                  I_TRIG_ECL2,
                                  I_TRIG_ECL3,
                                  I_TRIG_EXT0,
                                  I_TRIG_EXT1,
                                  I_TRIG_EXT2,
                                  I_TRIG_EXT3,
                                  I_TRIG_TTL0,
                                  I_TRIG_TTL1,
                                  I_TRIG_TTL2,
                                  I_TRIG_TTL3,
                                  I_TRIG_TTL4,
                                  I_TRIG_TTL5,
                                  I_TRIG_TTL6,
                                  I_TRIG_TTL7
};

char *triggernames[] = {          "I_TRIG_ALL ",
                                  "I_TRIG_STD ",
                                  "I_TRIG_CLK0",
                                  "I_TRIG_CLK1",
                                  "I_TRIG_CLK2",
                                  "I_TRIG_ECL0",
                                  "I_TRIG_ECL1",
                                  "I_TRIG_ECL2",
                                  "I_TRIG_ECL3",
                                  "I_TRIG_EXT0",
                                  "I_TRIG_EXT1",
                                  "I_TRIG_EXT2",
                                  "I_TRIG_EXT3",
                                  "I_TRIG_TTL0",
                                  "I_TRIG_TTL1",
                                  "I_TRIG_TTL2",
                                  "I_TRIG_TTL3",
                                  "I_TRIG_TTL4",
                                  "I_TRIG_TTL5",
                                  "I_TRIG_TTL6",
                                  "I_TRIG_TTL7"
};

void main(void)
```

2

```
{    INST instance;
     int returncode, errornumber;
     char *sessionname = "vxi";
     unsigned long triggers;
     register int tinductor;

     /*
     // Open an interface session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     returncode = ivxigettrigroute(instance,I_TRIG_ALL,&triggers);
     if (returncode != I_ERR_NOERROR) {
         fprintf(stderr,
             "\tIvxigettrigroute failed, error = %s (%d) \n\r",
             igeterrstr(returncode),returncode);
         exit(2);
     }
     fprintf(stdout,"Default triggers:\n\r\n\r");
     for (tinductor = 0;
           tinductor < sizeof(triggermasks)/sizeof(unsigned long);
           tinductor++) {
         if (triggers & triggermasks[tinductor])
             fprintf(stdout,"%s\n\r",triggernames[tinductor]);
     }
     exit(0);
 }
```

ivxirminfo

Description

Gets VXI device information.

int PASCAL
ivxirminfo(INST *id*, **int** *ula*, **struct vxiinfo** **information*);

id	Pointer to a VXI session structure.
ula	Device unique logical address.
information	Pointer to a location where the function stores the device's VXI configuration information.

Remarks

This function places the VXI configuration information of the device at unique logical address *ula* in the location pointed to by *information*.

The function ignores *id* when *ula* specifies a valid device on a VXI interface.

For VXI device sessions only, specifying a *ula* of −1 causes the function to return the configuration of the session device pointed to by *id*.

VXI configuration information is returned in the format of a VXIINFO structure. The VXIINFO structure is defined as:

```
struct vxiinfo
{
/* Device identification. */
short laddr;                      /* Unique logical address. */
char name[16];                    /* Symbolic name (primary) */
char manuf_name[16];              /* Manufacturer name.  */
char model_name[16];              /* Model name.         */
unsigned short man_id;            /* Manufacturer ID.    */
unsigned short model;             /* Model number.       */
unsigned short devclass;   /* Device class.        */

/* Self-test status. */
short selftest;                   /* Self test status:   */
                                  /* 1 == PASSED  */
                                  /* 0 == FAILED  */

/* Location of device. */
short cage_num;                   /* Card cage number.   */
short slot;                       /* Slot number:        */
                                  /* -1 == UNKNOWN       */
                                  /* -2 == MXI    */

/* Device information. */
unsigned short protocol;          /* Value of protocol  register.*/
unsigned short x_protocol;        /* Value of extended protocol
                                        register */
unsigned short servant_area;      /* Value of servant area. */

/* Memory information. */
unsigned short addrspace;         /* Memory address space:     */
                                  /* 0  == None   */
                                  /* 24 == A24    */
                                  /* 32 == A32    */
unsigned short memsize;           /* Amount of memory, in pages
                                  (pages are 256 bytes in A24, 64K
                                  in A32).*/
unsigned short memstart;          /* Start of memory, in pages
                                  (pages are 256 bytes in A24, 64K
                                  in A32).*/

/* Miscellaneous information. */
short slot0_laddr;          /* ULA of slot 0 controller (-1 if
                            unknown). */
short cmdr_laddr:           /* ULA of commander (-1 if top* level). */

/* Interrupt information. */
short int_handler[8];       /* Array of interrupt handler flags.*/
short interrupter[8];       /* Array of interrupter flags.       */
short fill[10];             /* Unused space.        */
};
```

2

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADADDR	*Ula* does not specify a valid VXI device.
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_PARAM	*Ula* is -1 and *id* specifies an interface or commander session, or *information* is null.

See Also **iopen**

Example

```
/*
//      This example call ivxirminfo to retrieve resource
//      management configuration information
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"


void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionnames[] = { "vxi", "vdev1" };
    struct vxiinfo Vxiinfo = {0};

    /*
    // Open an interface session
    */
    instance = iopen(sessionnames[0]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[0],
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
```

2

```
    /*
    //      Get information for ULA 0
    */
    returncode = ivxirminfo(instance,0,&Vxiinfo);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute ivxirminfo\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,"Symbolic name       %s\n\r",Vxiinfo.name);
    fprintf(stdout,"Manufacturer name   %s\n\r",Vxiinfo.manuf_name);
    (void) iclose(instance);
    /*
    //      Get information for device referenced by this session id
    */
    instance = iopen(sessionnames[1]);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionnames[1],
            igeterrstr(errornumber),errornumber);
        exit(3);
    }
    returncode = ivxirminfo(instance,-1,&Vxiinfo);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute ivxirminfo\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(4);
    }
    fprintf(stdout,"Symbolic name       %s\n\r",Vxiinfo.name);
    fprintf(stdout,"Manufacturer name   %s\n\r",Vxiinfo.manuf_name);
    exit(0);
}
```

ivxiservants

2

Description Gets a list of VXI servants.

int PASCAL
ivxiservants(INST *id*, **int** *listsize*, **int** **list*);

id	Pointer to a VXI interface session structure.
listsize	Size of servant list, in entries.
list	Pointer to a location where the function stores an integer list of the ULAs of this device's servant devices.

Remarks This function places a list of the unique logical addresses (ULA) of the servants of the VXI interface pointed to by *id* in the memory location pointed to by *list*. Specifying an *id* pointing to a GPIB session or VXI device session generates an error.

Listsize specifies the maximum number of entries in *list*.

If the VXI interface has less than *listsize* servant devices, all unused entries are set to −1. If the interface has more than *listsize* servant device, only the first *listsize* ULA's are placed in *list*.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_PARAM	*Id* specifies a device or commander session, or *list* is null.

See Also **iopen**

2

Example

```
/*
//      This example uses ivxiservants to get the list
//      of VXI servants.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define MAXULA                      256

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vxi";
    unsigned short totalulas = 0;
    int ulas[MAXULA];
    register short iinductor;

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ivxiservants(instance,
        sizeof(ulas)/sizeof(int),ulas);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIvxiservants failed\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
            exit(2);
    }
```

```
    fprintf(stdout,"VXI servant ULA table :\n\r");
    for (iinductor = 0; iinductor < MAXULA; iinductor++) {
        if (ulas[iinductor] != -1) {
            fprintf(stdout,
                    "\t%d\t(0x%x)\n\r",
                    ulas[iinductor],
                    ulas[iinductor]);
            totalulas++;
        }
    }
    fprintf(stdout,"Total number of servants is %d",totalulas);
    exit(0);
}
```

2

ivxitrigoff

Description

Deasserts VXIbus trigger lines.

int PASCAL
ivxitrigoff(INST *id*, unsigned long *triggermask*);

id	Pointer to a VXI interface session structure.
triggermask	VXIbus trigger line(s) to deassert.

Remarks

This function deasserts the VXIbus trigger lines specified in *triggermask* for the VXI interface session pointed to by *id*. *Triggermask* is a bit mask that is an OR'ed combination of one or more of the following:

Constant	Description
I_TRIG_ALL	All valid triggers. (EPC-2 and EPC-7 only)
I_TRIG_ECL0	ECL trigger 0. (EPC-2 and EPC-7 only)
I_TRIG_ECL1	ECL trigger 1. (EPC-2 and EPC-7 only)
I_TRIG_EXT0	EXT trigger 0 (valid only on an EPC-7). Has no effect unless **I_TRIG_EXT0** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIG_STD	Standard trigger. (EPC-2 and EPC-7 only)
I_TRIG_TTL0	TTL trigger 0. (EPC-2 and EPC-7 only)
I_TRIG_TTL1	TTL trigger 1. (EPC-2 and EPC-7 only)
I_TRIG_TTL2	TTL trigger 2. (EPC-2 and EPC-7 only)
I_TRIG_TTL3	TTL trigger 3. (EPC-2 and EPC-7 only)
I_TRIG_TTL4	TTL trigger 4. (EPC-2 and EPC-7 only)
I_TRIG_TTL5	TTL trigger 5. (EPC-2 and EPC-7 only)
I_TRIG_TTL6	TTL trigger 6. (EPC-2 and EPC-7 only)
I_TRIG_TTL7	TTL trigger 7. (EPC-2 and EPC-7 only)

Use **ivxigettrigroute** to get the trigger mask bits corresponding to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

The trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Use **ixtrig** to assert a trigger line then immediately deassert it.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_NOTSUPP	The hardware/software platform does not support the specified *triggermask* bits.
I_ERR_PARAM	*Id* specifies a device or commander session, or *triggermask* specifies an invalid trigger bit.

See Also **ivxigettrigroute, ivxitrigon, ivxitrigroute, ixtrig**

2

ivxitrigon

Description Asserts VXIbus trigger lines.

int PASCAL
ivxitrigon(INST *id*, **unsigned long** *triggermask*);

id	Pointer to a VXI interface session structure.
triggermask	VXIbus trigger line(s) to assert.

Remarks This function asserts the VXIBus trigger lines specified in *triggermask* for the VXI interface session pointed to by *id*. *Triggermask* is a bit mask that is an OR'ed combination of one or more of the following:

Constant	Description
I_TRIG_ALL	All valid trigger. (EPC-2 and EPC-7 only)
I_TRIG_ECL0	ECL trigger 0. (EPC-2 and EPC-7 only)
I_TRIG_ECL1	ECL trigger 1. (EPC-2 and EPC-7 only)
I_TRIG_EXT0	EXT trigger 0 (valid only on an EPC-7). Has no effect unless **I_TRIG_EXT0** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIG_STD	Standard trigger. (EPC-2 and EPC-7 only)
I_TRIG_TTL0	TTL trigger 0. (EPC-2 and EPC-7 only)
I_TRIG_TTL1	TTL trigger 1. (EPC-2 and EPC-7 only)
I_TRIG_TTL2	TTL trigger 2. (EPC-2 and EPC-7 only)
I_TRIG_TTL3	TTL trigger 3. (EPC-2 and EPC-7 only)
I_TRIG_TTL4	TTL trigger 4. (EPC-2 and EPC-7 only)
I_TRIG_TTL5	TTL trigger 5. (EPC-2 and EPC-7 only)
I_TRIG_TTL6	TTL trigger 6. (EPC-2 and EPC-7 only)
I_TRIG_TTL7	TTL trigger 7. (EPC-2 and EPC-7 only)

Use **ivxigettrigroute** to get the triggermask bits that correspond to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

The trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Use **ixtrig** to assert a trigger line then immediately deassert it.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_NOTSUPP	The hardware/software platform does not support the specified *triggermask* bits.
I_ERR_PARAM	*Id* specifies a device or commander session, or *triggermask* specifies an invalid trigger bit.

See Also

ivxigettrigroute, ivxitrigoff, ivxitrigroute, ixtrig

2

Example

```
/*
//      This example asserts, checks and then deasserts the
//      standard trigger on VXI.
*/

#include <stdio.h>
#include <stdlib.h>
#include "busmgr.h"
#include "sicl.h"

void main(void)
{   INST instance;
    int returncode, errornumber, result;
    char *sessionname = "vxi";

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            ,igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ivxitrigon(instance,I_TRIG_TTL0);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIvxitrigon failed\n\r");
        fprintf(stderr,
            "\terror = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    returncode = ivxibusstatus(instance,
                        I_VXI_BUS_TRIGGER,
                        &result);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to execute ivxibusstatus\n\r");
        fprintf(stderr,
            "\tError = %s (%d)\n\r",
             igeterrstr(returncode),returncode);
        exit(3);
    }
    if (result & I_TRIG_TTL0 == 0){
        fprintf(stderr,
            "\tI_TRIG_TTL0 is not asserted\n\r");
        exit(4);
    }
```

```
        returncode = ivxitrigoff(instance,I_TRIG_TTL0);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tIvxitrigoff failed\n\r");
            fprintf(stderr,
                "\terror = %s (%d)\n\r",
                igeterrstr(returncode),returncode);
            exit(5);
        }
        returncode = ivxibusstatus(instance,
                            I_VXI_BUS_TRIGGER,
                            &result);
        if (returncode != I_ERR_NOERROR) {
            fprintf(stderr,
                "\tUnable to execute ivxibusstatus\n\r");
            fprintf(stderr,
                "\tError = %s (%d)\n\r",
                 igeterrstr(returncode),returncode);
            exit(6);
        }
        if (result & I_TRIG_TTL0 != 0) {
            fprintf(stderr,
                "\tI_TRIG_TTL0 is asserted\n\r");
            exit(7);
        }
        exit(0);
    }
```

ivxitrigroute

Description Routes VXIbus trigger lines.

int PASCAL
ivxitrigroute(INST *id*, unsigned long *intrigger*, unsigned long *outtriggermask*);

id	Pointer to a VXI interface session structure
intrigger	Input trigger
outtriggermask	Output trigger mask

Remarks This function routes the VXIbus input trigger line *intrigger* to the VXIbus output trigger lines *outtriggermask* for the VXI interface of the session pointed to by *id*. Asserting an input trigger line causes assertion of all the routed output trigger lines.

Intrigger is a constant specifying the input trigger to route. *Outtriggermask* is an OR'ed combination of constants specifying the routed trigger(s). Valid combinations of *intrigger* and *outtriggermask* are:

intrigger	*outtriggermask*	Description
I_TRIG_STD	**I_TRIG_ALL** **I_TRIG_ECL0** to **ECL1** **I_TRIG_EXT0** **I_TRIG_STD** **I_TRIG_TTL0** to **TTL7**	Defines one or more triggers corresponding to the **I_TRIG_STD** constant. An *outtriggermask* containing the **I_TRIG_EXT0** bit is valid only on an EPC-7, and only has an effect if **I_TRIG_EXT0** is routed as an output trigger.
I_TRIG_TTL0 through **I_TRIG_TTL7**	**I_TRIG_EXT0**	Defines **I_TRIG_EXT0** as an output of another trigger. Valid only on an EPC-7.
I_TRIG_EXT0	**I_TRIG_TTL0** through **I_TRIG_TTL7**	Defines **I_TRIG_EXT0** as the input to one or more triggers. Valid only on an EPC-7.

If *intrigger* is **I_TRIG_STD**, then *outtriggermask* defines which triggers are affected when a subsequent **isetintr**, **ivxitrigon**, **ixtrig**, or **ivxitrigoff** function call executes with the **I_TRIG_STD** constant specified.

Calls to **ivxitrigroute** override previous routings. For example, routing **I_TRIG_STD** to **I_TRIG_TTL7** invalidates the default routing for **I_TRIG_STD**.

On an EPC-7, **I_TRIG_EXT0** must be routed as either an output from another trigger or as an input to exactly one trigger. It cannot be routed as an output trigger and an input trigger simultaneously. Also, **I_TRIG_EXT0** routing can never be disabled. At power-up, **I_TRIG_EXT0** is routed as an input to **I_TRIG_TTL0**.

Use **ivxigettrigroute** to get the trigger mask bits that correspond to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

2

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_NOTSUPP	The hardware/software platform does not support the specified *intrigger* and/or *outtriggermask* bits.
I_ERR_PARAM	*Id* specifies a device or commander session, or *intrigger* and/or *outtriggermask* specifies an invalid trigger bit.

See Also **isetintr, ivxigettrigroute, ivxitrigoff, ivxitrigon, ixtrig**

2

Example

```
/*
//      This example uses ivxitrigroutne to set a trigger
//      routing.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

unsigned long triggermasks[] = {  I_TRIG_ALL,
                                  I_TRIG_STD,
                                  I_TRIG_CLK0,
                                  I_TRIG_CLK1,
                                  I_TRIG_CLK2,
                                  I_TRIG_ECL0,
                                  I_TRIG_ECL1,
                                  I_TRIG_ECL2,
                                  I_TRIG_ECL3,
                                  I_TRIG_EXT0,
                                  I_TRIG_EXT1,
                                  I_TRIG_EXT2,
                                  I_TRIG_EXT3,
                                  I_TRIG_TTL0,
                                  I_TRIG_TTL1,
                                  I_TRIG_TTL2,
                                  I_TRIG_TTL3,
                                  I_TRIG_TTL4,
                                  I_TRIG_TTL5,
                                  I_TRIG_TTL6,
                                  I_TRIG_TTL7
};

char *triggernames[] = {          "I_TRIG_ALL ",
                                  "I_TRIG_STD ",
                                  "I_TRIG_CLK0",
                                  "I_TRIG_CLK1",
                                  "I_TRIG_CLK2",
                                  "I_TRIG_ECL0",
                                  "I_TRIG_ECL1",
                                  "I_TRIG_ECL2",
                                  "I_TRIG_ECL3",
                                  "I_TRIG_EXT0",
                                  "I_TRIG_EXT1",
                                  "I_TRIG_EXT2",
                                  "I_TRIG_EXT3",
                                  "I_TRIG_TTL0",
                                  "I_TRIG_TTL1",
                                  "I_TRIG_TTL2",
                                  "I_TRIG_TTL3",
                                  "I_TRIG_TTL4",
                                  "I_TRIG_TTL5",
                                  "I_TRIG_TTL6",
                                  "I_TRIG_TTL7"
};

void main(void)
```

2

```
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vxi";
    unsigned long triggers;
    register int tinductor;

    /*
    //  Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    /*
    //  The following command will fire TTL1 & TTL5 whenever EXT0 is
    //  fired
    */
    returncode = ivxitrigroute(instance,
                         I_TRIG_EXT0,
                         I_TRIG_TTL1);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIvxitrigroute failed, error = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    /*
    //  Get trigger routing for I_TRIG_EXT0
    */
    returncode = ivxigettrigroute(instance,
                            I_TRIG_EXT0,
                            &triggers);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIvxigettrigroute failed, error = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(3);
    }
    fprintf(stdout,"I_TRIG_EXT0 mapping:\n\r\n\r");
    for (tinductor = 0;
         tinductor < sizeof(triggermasks)/sizeof(unsigned long);
         tinductor++) {
        if (triggers & triggermasks[tinductor])
            fprintf(stdout,"%s\n\r",triggernames[tinductor]);
    }
    exit(0);
}
```

ivxiwaitnormop

Description Waits for a normal operation of a VXI interface.

int PASCAL
ivxiwaitnormop(INST *id*);

id	Pointer to a VXI interface session structure.

Remarks If the VXIbus interface specified by *id* is active, the function returns immediately.

If the interface is inactive, the function waits until normal operation is established unless a timeout limit has been set by **itimeout**. Then, it waits for the timeout limit and generates an error.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iopen, itimeout**

2

Example

```
/*
//      This example call ivxiwaitnormop to wait for an
//      instrument to establish normal operation.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"


void main(void)
{    INST instance;
     int returncode, errornumber;
     char *sessionname = "vxi";

     /*
     // Open an interface session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     returncode = ivxiwaitnormop(instance);
     if (returncode != I_ERR_NOERROR) {
         fprintf(stderr,
             "\tUnable to execute ivxiwaitnormop\n\r");
         fprintf(stderr,
             "\tError = %s (%d)\n\r",
             igeterrstr(returncode),returncode);
         exit(2);
     }
     exit(0);
}
```

ivxiws

Description

Sends a word-serial command to a VXI device.

int PASCAL
ivxiws(INST *id*, **unsigned short** *command*, **unsigned short** **reply*,
unsigned short **error*);

id	Pointer to a session structure.
command	Word-serial command to send.
reply	Pointer to a location where the function stores the word-serial response.
error	Pointer to a location where the function stores the response to a READ PROTOCOL ERROR word-serial command.

Remarks

This function sends the word-serial command specified by *command* to the VXI device session pointed to by *id*.

If *reply* is not null, a word-serial response is read and stored in the location pointed to by *reply*.

If *error* is not null and a word-serial protocol error is detected, a READ PROTOCOL ERROR word-serial command is sent to the device and the response is placed in the location pointed to by *error*.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred.
I_ERR_IO	A VXI word-serial protocol error occurred.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOINTF	*Id* specifies a non-VXI interface type.
I_ERR_PARAM	*Id* specifies an interface or commander session or a VXI device that is not message-based.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iclear, ilocal, inbread, inbwrite, iread, ireadstb, iremote, itimeout, itrigger, iwrite**

Example

```
/*
//     This example uses ivxiws to send a word serial
//     command to a device.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define WSCOMMAND  0xdfff
```

```
void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "vdevl";
    unsigned short readerror, reply;

    /*
    //  Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = ivxiws(instance,WSCOMMAND,&reply,&readerror);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIvxiws failed, error = %s (%d) \n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,"Reply = %d, Readerror = %d",reply,readerror);
    exit(0);
}
```

2

2

iwaithdlr

Description Waits for a SRQ or interrupt handler function to execute.

int PASCAL
iwaithdlr(long *timeout***);**

timeout	Timeout time period.

Remarks This function waits for *timeout* milliseconds for a SRQ or interrupt handler function to execute. If *timeout* is less than or equal to zero, processing suspends indefinitely until a SRQ or interrupt event handler completes execution. If *timeout* is greater than zero, processing suspends for up to the specified time.

This function ignores the state of event processing as set by **iintron** and **iintroff**.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_NOERROR	Successful function completion.
I_ERR_TIMEOUT	A timeout occurred.

See Also **iintron**, **iintroff**, **ionintr**, **ionsrq**, **isetintr**

Example See **ionintr**.

iwblockcopy

Description

Copies blocks of 16-bit words from one set of sequential memory locations to another.

int PASCAL
iwblockcopy(INST *id*, **unsigned short** **src*, **unsigned short** **dest*, **unsigned long** *count*, **int** *swap*);

id	Pointer to a session structure.
src	Source pointer.
dest	Destination pointer.
count	Number of 16-bit words to copy.
swap	Byte swap flag.

Remarks

This function copies 16-bit words from successive memory locations beginning at *src* into successive memory locations beginning at *dest*. *Count* specifies the number of 16-bit words to transfer and has a maximum value of 0x8000. *Id* specifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

When *swap* is non-zero and a VXIbus access is made, the function byte-swaps the 16-bit words to or from Motorola byte ordering as necessary. When *swap* is zero, no byte swapping occurs. The following table lists the possible scenarios when accessing EPC and VXIbus memory:

2

src	*dest*	*swap*	Result
EPC	EPC	0	No byte-swapping
EPC	EPC	Non-zero	No byte-swapping
EPC	VXI	0	No byte-swapping
EPC	VXI	Non-zero	One byte-swap
VXI	EPC	0	No byte-swapping
VXI	EPC	Non-zero	One byte-swap
VXI	VXI	0	No byte-swapping
VXI	VXI	Non-zero	Two byte-swaps (equivalent to no byte-swap)

For 16-bit byte-swapping to execute properly, all VXI bus access must be aligned on 16-bit boundaries.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Src* and/or *dest* is null.

See Also **ibblockcopy**, **ilblockcopy**, **imap**, **iwpeek**, **iwpoke**, **iwpopfifo**, **iwpushfifo**,

Example

```
/*
//      This example uses iwblockcopy to read the VXI register of
//      the device configured as ULA 0.  The bit encodings of this
//      register id defined by the VXI specification.  For this
//      particular example, the program is using the manufacture ID
//      bits.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define     VXIREGISTEROFFSET       0xc000

void main(void)
{   INST instance;
    int *vxiregisters;
    int returncode, errornumber;
    char deviceclass;
    char *dclass[] = { "Memory",
                  "Extended",
                  "Message Based",
                  "Register Based" };
    char *sessionname = "vxi";

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxiregisters = (int *) imap(instance,I_MAP_A16,0,0,NULL);
    if (vxiregisters == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = %s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
```

2

```
    returncode = iwblockcopy(instance,
                        (unsigned short *)
                        ((unsigned long) vxiregisters +
                        (unsigned long) VXIREGISTEROFFSET),
                        (unsigned short *) &deviceclass,
                        1L,
                        0);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to copy ID register, error = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(3);
    }
    fprintf(stdout,
            "Class of device at ULA 0 is %s.",
             dclass[(deviceclass >> 6) & 0x3]);
    exit(0);
}
```

iwpeek

Description Reads a 16-bit word stored at an address.

volatile unsigned short PASCAL
iwpeek(volatile unsigned short **addr***);**

addr	Address of a 16-bit word.

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

Return Value The function returns the 16-bit word contained at *addr*.

See Also **ibpeek, ilpeek, imap, iwpoke**

Example

```
/*
//      This example uses iwpeek to read our own slave
//      memory thru the VXIbus.
*/

#include <stdlib.h>
#include <stdio.h>
#include "busmgr.h"
#include "sicl.h"

void main(void)
{   INST instance;
    int errornumber, returncode, result;
    char *lowpage;
    unsigned short lowmemory;
    char *sessionnames[] = { "vxi", "vdevl" };
    unsigned short *baseoffset = 0x400;
```

2

2

```
/*
//  Open an interface session
*/
instance = iopen(sessionnames[0]);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionnames[0],
        igeterrstr(errornumber),errornumber);
    exit(1);
}
/*
//      Find where our memory begins
*/
returncode = ivxibusstatus(instance,
                      I_VXI_BUS_SHM_PAGE,
                      &result);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,
        "\tUnable to execute ivxibusstatus\n\r");
    fprintf(stderr,
        "\tError = %s (%d) \n\r",
        igeterrstr(returncode),returncode);
    exit(2);
}
(void) iclose(instance);
/*
//  Open a device session
*/
instance = iopen(sessionnames[1]);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionnames[1],
        igeterrstr(errornumber),errornumber);
    exit(3);
}
/* Map in A24 space */
lowpage = imap(instance,I_MAP_A24,result >> 8,1,NULL);
if (lowpage == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to map in A24 space, error = %s (%d) \n\r",
        igeterrstr(errornumber),errornumber);
    exit(4);
}
```

2

```
    /*
    //      Reading the 400th word of VME memory at our base address
    //      should return the same value as reading 0:400 through PC
    //      memory
    */
    lowmemory = iwpeek((unsigned short *)
                 ((unsigned long) lowpage+
                 (unsigned long) baseoffset));
    EpcMemSwapW(&lowmemory,1);
    if (lowmemory != *baseoffset) {
        fprintf(stderr,
            "\tVME memory at page %x longword offset %x ",
            result >> 8,baseoffset);
        fprintf(stderr,"= %04.4x\n\r",lowmemory);
        fprintf(stderr,"\tExpected %04.4x\n\r",*baseoffset);
        exit(5);
    }
    fprintf(stdout,"VME memory at page %x longword offset %x = ",
             result >> 8,baseoffset);
    fprintf(stdout,"%04.4x\n\r",lowmemory);
    exit(0);
}
```

2

iwpoke

Description Writes a 16-bit word to an address.

void PASCAL
ibpoke(volatile unsigned short **dest*, **unsigned short** *value*);

dest	Destination address.
value	16-bit word to write.

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

Return Value The function returns no value.

See Also **ibpoke, ilpoke, imap, iwpeek**

Example

```
/*
//      This example uses iwpoke to write into
//      DOS's communcation area via VME memory.
*/

#include <stdlib.h>
#include <stdio.h>
#include "sicl.h"
#include "busmgr.h"

#define FOOTPRINT          0x1234

void main(void)
{   INST instance;
    int errornumber, returncode, result;
    char *lowpage;
    short *doscom = (short *) 0x4f0L;
    char *sessionnames[] = { "vxi", "vdev1" };
```

```
/*
//  Open an interface session
*/
instance = iopen(sessionnames[0]);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionnames[0],
        igeterrstr(errornumber),errornumber);
    exit(1);
}
/*
//      Find where our memory begins
*/
returncode = ivxibusstatus(instance,
                        I_VXI_BUS_SHM_PAGE,
                        &result);
if (returncode != I_ERR_NOERROR) {
    fprintf(stderr,
        "\tUnable to execute ivxibusstatus\n\r");
    fprintf(stderr,
        "\tError = %s (%d) \n\r",
        igeterrstr(returncode),returncode);
    exit(2);
}
(void) iclose(instance);
/*
//  Open a device session
*/
instance = iopen(sessionnames[1]);
if (instance == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to open <%s>, error = %s (%d)\n\r",
        sessionnames[1],
        igeterrstr(errornumber),errornumber);
    exit(3);
}
/* Map in A24 space */
lowpage = imap(instance,I_MAP_A24,result >> 8,1,NULL);
if (lowpage == NULL) {
    errornumber = igeterrno();
    fprintf(stderr,
        "\tUnable to map in A24 space, error = %s (%d) \n\r",
        igeterrstr(errornumber),errornumber);
    exit(4);
}
```

2

```
    /*
    //      Write into DOS's communication area at PC address
    //      4f0:0
    */
    iwpoke((unsigned short *)
           ((unsigned long) lowpage+(unsigned long) doscom),
           FOOTPRINT);
    EpcMemSwapW((unsigned short *) doscom,1);
    if (*doscom != FOOTPRINT) {
        fprintf(stderr,
            "\tVME memory at page %x longword offset %x ",
            result >> 8,doscom);
        fprintf(stderr,"= %04.4x\n\r",*doscom);
        fprintf(stderr,"\tExpected %04.4x\n\r",FOOTPRINT);
        exit(5);
    }
    fprintf(stdout,"VME memory at page %x longword offset %x = ",
                result >> 8,doscom);
    fprintf(stdout,"%04.4x\n\r",*doscom);
    exit(0);
}
```

iwpopfifo

Description

Copies 16-bit words from a single memory location (FIFO register) to sequential memory locations.

int PASCAL
ibpopfifo(INST *id,* **unsigned short** **fifo*, **unsigned short** **dest*,
unsigned long *count*, **int** *swap*);

id	Pointer to a session structure.
fifo	FIFO pointer.
dest	Destination address.
count	Number of 16-bit words to copy.
swap	Byte swap flag.

Remarks

This function copies *count* 16-bit words from *fifo* into sequential memory locations beginning at *dest*. *Count* specifies the number of 16-bit words to transfer and has a maximum value of 0x8000. *Id* identifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

When *swap* is non-zero and a VXIbus access is made, the function byte-swaps the 16-bit words to or from Motorola byte ordering as necessary. When *swap* is zero, no byte swapping occurs. The following table lists the possible scenarios when accessing EPC and VXIbus memory:

2

src	*dest*	*swap*	Result
EPC	EPC	0	No byte-swapping
EPC	EPC	Non-zero	No byte-swapping
EPC	VXI	0	No byte-swapping
EPC	VXI	Non-zero	One byte-swap
VXI	EPC	0	No byte-swapping
VXI	EPC	Non-zero	One byte-swap
VXI	VXI	0	No byte-swapping
VXI	VXI	Non-zero	Two byte-swaps (equivalent to no byte-swap)

For 16-bit byte-swapping to execute properly, all VXI bus access must be aligned on 16-bit boundaries.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Fifo* and/or *dest* is null.

See Also **ibpopfifo, ilpopfifo, imap, iwpushfifo**

Example

```
/*
//      This example uses iwpopfifo to read from a
//      hypothetical VXI fifo at offset 0.
*/

#include <stdlib.h>
#include <stdio.h>
#include "sicl.h"

#define NOSWAP          0       /* 0 indicates no byte swapping */
```

```
void main(void)
{   INST instance;
    unsigned short *vxi;
    int returncode, errornumber;
    unsigned short datafifo[5];
    char *sessionname = "vxi";

    /*
    // Open an interface session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxi = (unsigned short *) imap(instance,I_MAP_A16,0,0,NULL);
    if (vxi == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = ");
        fprintf(stderr,
            "%s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    /*
    // Read the Fifo 5 times, storing the values into datafifo[]
    */
    returncode = iwpopfifo(instance,
                        vxi,
                        datafifo,
                        (long) sizeof(datafifo)/sizeof(short),
                        NOSWAP);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to read the fifo at address ");
        fprintf(stderr,
            "%p\n\r\tError = %s (%d) \n\r",
            vxi,
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    exit(0);
}
```

iwpushfifo

Description

Copies 16-bits words from sequential memory locations to a single memory location (FIFO register).

int PASCAL
ibpushfifo(INST *id*, **unsigned short** **src*, **unsigned short** **fifo*,
unsigned long *count*);

id	Pointer to a session structure.
src	Source address.
fifo	FIFO pointer.
count	Number of 16-bit words to copy.
swap	Byte swap flag.

Remarks

This function copies *count* 16-bit words from the sequential memory locations beginning at *src* into the FIFO at *fifo*. *Count* specifies the number of 16-bit words to transfer and has a maximum value of 0x8000. *Id* specifies the interface to use for the transfer.

The function is valid only for VXI interfaces. It does not detect segment wrap around conditions or detect bus errors caused by its use.

This function allows any address (VXI via **imap** address or EPC) to any address (VXI via **imap** address or EPC) copies.

When *swap* is non-zero and a VXIbus access is made, the function byte-swaps the 16-bit words to or from Motorola byte ordering as necessary. When *swap* is zero, no byte swapping occurs. The following table lists the possible scenarios when accessing EPC and VXIbus memory:

src	*dest*	*swap*	Result
EPC	EPC	0	No byte-swapping
EPC	EPC	Non-zero	No byte-swapping
EPC	VXI	0	No byte-swapping
EPC	VXI	Non-zero	One byte-swap
VXI	EPC	0	No byte-swapping
VXI	EPC	Non-zero	One byte-swap
VXI	VXI	0	No byte-swapping
VXI	VXI	Non-zero	Two byte-swaps (equivalent to no byte-swap)

For 16-bit byte-swapping to execute properly, all VXI bus access must be aligned on 16-bit boundaries.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	*Id* specifies an interface type that does not support address mapping (e.g., GPIB).
I_ERR_PARAM	*Src* and/or *fifo* is null.

See Also

ibpushfifo, ilpushfifo, imap, iwpopfifo

Example

```
/*
//      This example uses ilpushfifo to write values
//      to a hypothetical VXI fifo at offset 0.
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define NOSWAP          0       /* 0 indicates no byte swapping */
```

2

```
void main(void)
{   INST instance;
    char FAR *vxi;
    int returncode, errornumber;
    unsigned short datafifo[] = { 0x1000,
                                  0x2000,
                                  0x3000,
                                  0x4000,
                                  0x5000 };
    char *sessionname = "vxi";

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
            sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    vxi = imap(instance,I_MAP_A16,0,0,NULL);  /* Map in A16 space */
    if (vxi == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to map in A16 space, error = ");
        fprintf(stderr,
            "%s (%d) \n\r",
            igeterrstr(errornumber),errornumber);
        exit(2);
    }
    /*
    // Write to the fifo 5 times, storing 0x1000, 0x2000, 0x3000,
    // 0x4000, 0x5000
    */
    returncode = iwpushfifo(instance,
                     (unsigned short *) vxi,
                     datafifo,
                     (unsigned long) sizeof(datafifo)/sizeof(short),
                     NOSWAP);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tUnable to write to the fifo at address ");
        fprintf(stderr,
            "%p\n\r\tError = %s (%d) \n\r",
            vxi,
            igeterrstr(returncode),
            returncode);
        exit(3);
    }
    exit(0);
}
```

iwrite

Description Writes data to a device or interface.

int PASCAL
iwrite(INST *id*, **char** **buf*, **unsigned long** *bufsize*, **int** *end*,
unsigned long **actualcnt*);

id	Pointer to a session structure.
buf	Pointer to the data buffer.
bufsize	Length, in bytes, of data buffer.
end	END indicator flag.
actualcnt	Pointer to a location where the function stores the actual number of bytes written.

Remarks This function writes the *bufsize* bytes at *buf* to the device or interface of the session pointed to by *id*. *Bufsize* has a maximum value of 0x10000. It performs no formatting or data conversion.

Writing ends when *bufsize* bytes are written or a timeout occurs. Unlike the **inbwrite** function, this function blocks until one of these two conditions is met.

When *id* specifies a device session, the function writes data using interface dependent communication methods. When *id* specifies an interface session, the function writes data in raw mode using interface specific methods.

If *end* is non-zero, the function writes an END indicator with the last data byte. If *end* is zero, the function does not write an END indicator with the last data byte.

If *actualcnt* is not null, the function stores the number of data bytes written in the referenced memory location.

For VXI device sessions, the function issues BYTE AVAILABLE word-serial commands and supports only message based VXI devices. Other VXI devices generate an error.

2

For VXI interface sessions, the function generates an error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's talk address, followed by the device's listen address. Finally, the function writes the data.

For GPIB interface sessions, the function writes bytes directly to the interface without performing any addressing. The ATN line state determines whether the bytes are interpreted as data or command bytes.

Return Value The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_DATA	A VXIbus error occurred during the write operation.
I_ERR_IO	A GPIB protocol error or VXI word-serial protocol error occurred during the write operation.
I_ERR_LOCKED	*Id* specifies a device or interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_PARAM	*Id* specifies a VXI interface or a VXI device that is not message-based, or *buf* is null.
I_ERR_TIMEOUT	A timeout occurred.

See Also **inbread, inbwrite, iread, itimeout**

2

Example

```
/*
//      This program illustrates serial IO using iwrite
*/

#include <stdio.h>
#include <stdlib.h>
#include "sicl.h"

#define EOI                 -1

void main(void)
{   INST instance;
    int returncode, errornumber;
    char *sessionname = "EPC2";
    unsigned long actualcount;

    /*
    // Open a device session
    */
    instance = iopen(sessionname);
    if (instance == NULL) {
        errornumber = igeterrno();
        fprintf(stderr,
            "\tUnable to open <%s>, error = %s (%d)\n\r",
          , sessionname,
            igeterrstr(errornumber),errornumber);
        exit(1);
    }
    returncode = iwrite(instance,"rmx\n",4L,EOI,&actualcount);
    if (returncode != I_ERR_NOERROR) {
        fprintf(stderr,
            "\tIwrite failed, error = %s (%d)\n\r",
            igeterrstr(returncode),returncode);
        exit(2);
    }
    fprintf(stdout,
            "%ld bytes written to <%s>",
             actualcount,
             sessionname);
    exit(0);
}
```

ixtrig

Description Asserts and deasserts one or more triggers to an interface.

int PASCAL
ixtrig(INST *id*, **unsigned long** *triggermask*);

id	Pointer to an interface session structure.
triggermask	Trigger mask to assert.

Remarks For GPIB interface session, the function issues a broadcast Group Execute Trigger (GET) command. The *triggermask* argument must be **I_TRIG_STD**.

For VXI interface sessions, the function asserts and immediately deasserts the VXIbus triggers specified by the *triggermask* argument. *Triggermask* is a bit mask that is an OR'd combination of one or more of the following:

Constant	Description
I_TRIG_ALL	All valid triggers. (EPC-2 and EPC-7 only)
I_TRIG_ECL0	ECL trigger 0. (EPC-2 and EPC-7 only)
I_TRIG_ECL1	ECL trigger 1. (EPC-2 and EPC-7 only)
I_TRIG_EXT0	EXT trigger 0 (valid only on an EPC-7). Has no effect unless **I_TRIG_EXT0** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIG_STD	Standard trigger. (EPC-2 and EPC-7 only)
I_TRIG_TTL0	TTL (EPC-2 and EPC-7 only)trigger 0.
I_TRIG_TTL1	TTL trigger 1. (EPC-2 and EPC-7 only)
I_TRIG_TTL2	TTL trigger 2. (EPC-2 and EPC-7 only)
I_TRIG_TTL3	TTL trigger 3. (EPC-2 and EPC-7 only)
I_TRIG_TTL4	TTL trigger 4. (EPC-2 and EPC-7 only)
I_TRIG_TTL5	TTL trigger 5. (EPC-2 and EPC-7 only)
I_TRIG_TTL6	TTL trigger 6. (EPC-2 and EPC-7 only)
I_TRIG_TTL7	TTL trigger 7. (EPC-2 and EPC-7 only)

Use **ivxigettrigroute** to get the VXIbus trigger mask bits corresponding to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

The VXIbus triggers corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Return Value

The function returns an integer to indicate its success or failure. Possible errors are:

Constant	Description
I_ERR_BADID	Invalid *id* session pointer.
I_ERR_IO	The function was unable to generate the specified interface trigger.
I_ERR_LOCKED	*Id* specifies an interface that is locked by another process.
I_ERR_NOERROR	Successful function completion.
I_ERR_NOTSUPP	The hardware/software platform does not support the specified *triggermask* bits.
I_ERR_PARAM	*Id* specifies a device or commander session or *triggermask* specifies an invalid trigger bit.
I_ERR_TIMEOUT	A timeout occurred.

See Also

itimeout, itrigger, ivxigettrigroute, ivxitrigoff, ivxitrigon, ivxitrigroute

2

Example

```
/*
//      This example asserts and deasserts the standard
//      trigger on GPIB.
*/

#include <stdio.h>
#include <stdlib.h>
#include "busmgr.h"
#include "sicl.h"

void main(void)
{    INST instance;
     int returncode, errornumber;
     char *sessionname = "gpib";

     /*
     //  Open an interface session
     */
     instance = iopen(sessionname);
     if (instance == NULL) {
         errornumber = igeterrno();
         fprintf(stderr,
             "\tUnable to open <%s>, error = %s (%d)\n\r",
             sessionname,
             igeterrstr(errornumber),errornumber);
         exit(1);
     }
     returncode = ixtrig(instance,I_TRIG_STD);
     if (returncode != I_ERR_NOERROR) {
         fprintf(stderr,
             "\tIxtrig failed\n\r");
         fprintf(stderr,
             "\terror = %s (%d)\n\r",
             igeterrstr(returncode),returncode);
         exit(2);
     }
     exit(0);
}
```

3. Advanced Topics

3

This chapter discusses topics of interest to advanced application programmers. Topics include:

- Byte Ordering and Data Representation
- Correcting Data Structure Byte Ordering
- SRQ, Interrupt, and Error Handler Execution
- Handler Operations Under DOS
- VXI TTL Trigger Interrupts on an EPC-7
- Microsoft Quick C
- Borland C
- Interfacing to Other Language Environments
- Terminating GPIB communication

Application cleanup

SICL has defined a special function, **_siclcleanup()**, to ensure that Windows performs the necessary clean-up required when a SICL program completes execution. Each SICL application should call **siclcleanup()** before exiting or posting a WM_QUIT message in order to release resources allocated for the application by the SICL library. Without this call, you may experience difficulty in executing your application, especially form within debuggers.

Note that the **I_ERROR_EXIT** handler calls **siclcleanup()** automatically before it exits.

Memory Models

We strongly recommend that you use the large memory model when designing applications that call SICL functions. This is because SICL requires all pointer parameters to be "**far**" pointers. Most SICL function prototypes in the **sicl.h** header file explicitly declare all pointer parameters to be **far**. However, there is no way to declare pointer types for functions that take a variable number of arguments (such as SICL's formatted I/O routines), and your compiler will not be able to properly check or cast types for these functions.

3.1 Byte Ordering and Data Representation

Byte ordering adds complexity to the VXIbus interface. Many VXIbus devices use the data formats of Motorola microprocessors. Others, including RadiSys EPC controllers, use the data format of Intel microprocessors. Although the Motorola and Intel microprocessors use the same data types, the hardware representations of these data types differ.

Figure 3-1 shows how the same sequence of bytes in memory is interpreted by Intel and Motorola microprocessors. Memory value 11 is the lowest address and memory value AA is the highest address. The data widths shown correspond to the data operand sizes found on both microprocessors.

Memory Value	Intel Order	Data Width	Motorola Order
11	11	8 bits	11
22	2211	16 bits	1122
33			
44	44332211	32 bits	11223344
55			
66			
77			
88	8877665544332211	64 bits	1122334455667788
99			
AA	AA99887766554433221 1	80 bits	112233445566778899AA

Figure 3-1. Byte Order Example

Byte Swapping Functions

The following functions, which are not part of the SICL library, convert 16-bit, 32-bit, 64-bit, and 80-bit data between Intel and Motorola byte orders (8-bit data does not require conversion).

3

Swap16 is a function that takes a pointer to a 16-bit value as a parameter and byte-swaps the value in place:

```
void Swap16(char *value)
{
        char temp;

        temp = value[0]; value[0] = value[1]; value[1] = temp;
}
```

Swap32 is a function that takes a pointer to a 32-bit value as a parameter and byte-swaps the value in place:

```
void Swap32(char *value)
{
        char temp;

        temp = value[0]; value[0] = value[3]; value[3] = temp;
        temp = value[1]; value[1] = value[2]; value[2] = temp;
}
```

Swap64 is a function that takes a pointer to a 64-bit value as a parameter and byte-swaps the value in place:

```
void Swap64(char *value)
{
        char temp;

        temp = value[0]; value[0] = value[7]; value[7] = temp;
        temp = value[1]; value[1] = value[6]; value[6] = temp;
        temp = value[2]; value[2] = value[5]; value[5] = temp;
        temp = value[3]; value[3] = value[4]; value[4] = temp;
}
```

Swap80 is a function that takes a pointer to an 80-bit value as a parameter and byte-swaps the value in place:

```
void Swap80(char *value)
{
        char temp;

        temp = value[0]; value[0] = value[9]; value[9] = temp;
        temp = value[1]; value[1] = value[8]; value[8] = temp;
        temp = value[2]; value[2] = value[7]; value[7] = temp;
        temp = value[3]; value[3] = value[6]; value[6] = temp;
        temp = value[4]; value[4] = value[5]; value[5] = temp;
}
```


The SICL 16-bit peek and poke functions (**iwpeek** and **iwpoke**) and 32-bit peek and poke functions (**ilpeek** and **ilpoke**) always perform byte-swapping. The peek functions assume the data at the specified address is in Motorola byte order, and byte-swaps the data to Intel byte order after reading it. Conversely, the SICL poke functions assume the specified data is in Intel byte order, and byte-swaps the data to Motorola byte order before writing it to the specified address.

The SICL 16-bit block transfer functions (**iwblockcopy**, **iwpopfifo**, and **iwpushfifo**) and 32-bit block transfer functions (**ilblockcopy**, **ilpopfifo**, and **ilpushfifo**) conditionally perform byte-swapping. Unless specifically directed to perform byte-swapping, the SICL block transfer functions assume that both the source and destination addresses of the transfer use Intel byte order.

3

Correcting Data Structure Byte Ordering

The SICL 16-bit and 32-bit peek and poke (**ilpeek**, **iwpeek**, **ilpoke**, and **iwpoke**) and block transfer functions (**ilblockcopy** and **iwblockcopy**) do not solve all byte ordering problems. Even if byte-swapping occurs during a SICL block transfer function, byte ordering problems occur when Motorola-ordered data is copied to EPC memory using a different data width than the width of the operand itself. This situation occurs when a data structure containing mixed-type fields is copied in a single operation. The following code fragment illustrates how to correct the byte order in the local copy of the data structure:

```
struct DataStructure
{
       char       field8;
       short      field16;
       long       field32;
       double     field64;
       char       field80[10];
} data;

/* Copy the data structure to local memory from the VMEbus. */

ibblockcopy(ID, VMEADDR, &data, sizeof(struct DataStructure));

/* Byte-swap the individual structure fields (data.field8 is an
8-bit field, so it is already correct).
*/

Swap16(&data.field16);
Swap32(&data.field32);
Swap64(&data.field64);
Swap80(data.field80);
```

In the above example, the data structure was copied from VXIbus memory one byte at a time. To copy data from EPC memory to Motorola-ordered memory, byte-swap the fields of the structure in local memory (using the above byte swapping functions) and copy the data using the SICL **ibblockcopy** function.

It is usually more efficient to copy blocks of data using data transfer width greater than the expected data width. If you use a greater data transfer width to copy data structures containing mixed-type fields to/from Motorola-order memory, do not use the SICL function byte-swapping feature. Swap the data structure fields individually.

3.2 SRQ Handler Execution

These conditions must be true before an application's SICL SRQ handlers can execute:

- The application must call **ionsrq** to install a session's SRQ handler.
- A SRQ must occur.
- The application must call **iwaithdlr** or enable asynchronous event processing by calling **iintron**.

SICL discards all SRQ events that occur before the application installs a SRQ handler.

When an application installs a SRQ handler and enables asynchronous event processing, the SRQ handler processes SRQ events as soon as they are received. Under DOS, the installed handler executes as part of an interrupt thread, with processor interrupts enabled, and using the SICL driver's interrupt stack.

When an application installs a SRQ handler and does not enable asynchronous event processing, SICL queues SRQ events as they are received. The number of events to queue is set by the *eventqueuesize* variable in the SICLIF file. The SRQ handler will process the queued events when the application enables asynchronous event processing or calls **iwaithdlr**. If the application removes the installed SRQ handler before processing the queued events, the handler discards the events. Under DOS, the installed SRQ handler executes as part of the application's thread, with processor interrupts in a state defined by the application, and using the application's stack.

3

3.3 Interrupt Handler Execution

These conditions must be true before an application's SICL interrupt handlers can execute:

- The application must use **ionintr** to install an interrupt handler.
- The application must use **isetintr** to enable interrupt reception.
- An interrupt must occur.
- The application must call **iwaithdlr** or enable asynchronous event processing by calling **iintron**.

SICL discards all interrupt events that occur before the application installs an interrupt handler and enables interrupt reception.

When an application installs an interrupt handler, enables interrupt reception, and enables asynchronous event processing, the interrupt handler processes interrupts as soon as they are received. Under DOS, the installed interrupt handler executes as part of an interrupt thread, with processor interrupts enabled, and using a SICL driver's interrupt stack.

When an application installs an interrupt handler, enables interrupt reception, and does not enable asynchronous event processing, SICL queues the interrupts as they are received. The number of events to queue is set by the *eventqueuesize* variable in the SICLIF file. The interrupt handler will process the interrupts when the application enables asynchronous event processing or calls **iwaithdlr**. If the application removes the interrupt handler before processing the queued interrupts, the handler discards the interrupts. Under DOS, the installed interrupt handler executes as part of the application's thread, with processor interrupts in a state defined by the application, and using the application's stack.

3.4 Error Handler Execution

These conditions must be true before an application's SICL error handler can execute:

- The application must use **ionerror** to install the error handler.
- A SICL error must occur.

SICL discards all errors that occur before the application installs an error handler.

When an application has installed an error handler, and an error occurs, and if the handler is not already executing as part of one of the application's other threads, the error handler processes the error.

When an application has installed an error handler and an error occurs and the handler is already executing as part of one of the application's other threads, the SICL queues the error. The number of events to queue is set by the *errorqueuesize* variable in the SICLIF file. The error handler process the queued error when it finishes its current execution.

It is possible for error handlers to execute either as part of the application's thread or as part of an interrupt thread because errors can occur as part of a SRQ handler, a interrupt handler, or the main program.

Enabling or disabling asynchronous event processing does not affect error handler execution.

3.5 Handler Operations Under DOS

SRQ, interrupt, and error handlers can execute as part of an interrupt thread under DOS. This feature implies that a SICL handler can only call fully reentrant C library and SICL library functions. Also, a SICL handler can only invoke fully reentrant DOS and BIOS support functions, and cannot execute unprotected floating point instructions under DOS.

The following C library functions are reentrant under Microsoft C Version 6.0, and may be called from a SICL handler or any application code that executes as part of an interrupt thread (it is likely that this list is different for other releases of the Microsoft C compiler and for compilers from other vendors):

abs	**memccpy**	**strcat**	**strnset**
atoi	**memchr**	**strchr**	**strrchr**
atol	**memcmp**	**strcmp**	**strrev**
bsearch	**memcpy**	**strcmpi**	**strset**
chdir	**memicmp**	**strcpy**	**strstr**
getpid	**memmove**	**stricmp**	**strupr**
halloc	**memset**	**strlen**	**swab**
hfree	**mkdir**	**strlwr**	**tolower**
itoa	**movedata**	**strncat**	**toupper**
labs	**putch**	**strncmp**	
lfind	**rmdir**	**strncpy**	
lsearch	**segread**	**strnicmp**	

All the SICL library functions except **iopen**, **iclose**, **imap**, **iunmap**, **iprintf**, **iscanf**, **ipromptf**, and **isetbuf** are fully reentrant, and may be called from a SICL handler or any application code that executes as part of an interrupt thread. These eight functions execute non-reentrant floating point, dynamic memory management, file I/O, and task management functions. This is a departure from the SICL specification, which states that **iprintf**, **iscanf**, and **ipromptf** can be called from a SICL handler. In the DOS implementation **iprintf**, **iscanf**, and **ipromptf** functions are reentrant only when performing formatted I/O that does not include the conversion of floating point values.

Not all DOS and BIOS functions are fully reentrant. However, mechanisms exist (the "InDos" and "CriticalError" flags) for avoiding DOS reentrancy by delaying background processing until DOS is not in use.

Under DOS, floating point operations and standard floating point libraries provided with ANSI compilers are fully reentrant.

3.6 VXI TTL Trigger Interrupts on an EPC-7

Receiving and processing VXI TTL trigger interrupts on an EPC-7 requires software intervention.

EPC-7 hardware generates a VXI TTL trigger interrupt when all of the following conditions are true:

- A bit in the TTL trigger interrupt enable register is set. The SICL function **isetintr** sets one or more of these bits when enabling the reception of **I_INTR_TRIG** interrupts for a VXI interface session.
- The corresponding bit in the TTL trigger latch register is clear.
- The corresponding TTL trigger line is asserted for at least 30 nanoseconds.

The main complication to this scenario is that the TTL trigger latch register cannot be cleared until a TTL trigger is deasserted. In order to clear a bit in the register, the register must be read while the corresponding TTL trigger is deasserted. A TTL trigger assertion is not necessarily under EPC control.

The operation of the EPC-7 TTL trigger latch register has two potential side effects for SICL software:

- If the TTL trigger latch register is not cleared before **isetintr** enables the reception of **I_INTR_TRIG** interrupts for a VXI interface session, it is possible to receive one or more interrupts for a TTL trigger that was asserted, latched, and deasserted long before **isetintr** was called.

- If the TTL trigger latch register is not cleared after an **I_INTR_TRIG** interrupt is signaled to a VXI interface session, the EPC will not latch subsequent TTL trigger assertions and, therefore, will miss subsequent **I_INTR_TRIG** interrupts.

The following function, **WaitForTriggerDeassert**, clears the EPC-7 TTL trigger latch register.

```
#define EPC2        1
#include <conio.h>
#include "sicl.h"
#include "vmeregs.h"

int PASCAL
WaitForTriggerDeassert(    long TriggerMask,long RetryCount)
{
        long index;

        /*
         * Wait for the desired TTL latch register bits
         * to clear, indicating that the trigger(s) have
         * been deasserted.  Return an error if the
         * trigger(s) are not deasserted.
         */

        for (       index = 0;
                    ((long) INPORT(BTTL) & TriggerMask) != 0;
                    index += 1)
        {
            if (index == RetryCount)
            {
                    return (I_ERR_IO);
            }
        }
        return (I_ERR_NOERROR);
}
```

To avoid the problem of receiving extraneous SICL TTL trigger interrupts, execute **WaitForTriggerDeassert** before calling **isetintr** to enable a **I_INTR_TRIG** interrupts for a VXI interface session. To avoid the problem of missing **I_INTR_TRIG** interrupts, execute **WaitForTriggerDeassert** as soon as possible after receiving each trigger interrupt, preferably as part of the interrupt handler routine itself.

Reading the TTL trigger latch register (as in **WaitForTriggerDeassert**) clears all previously latched and deasserted TTL triggers, not just one particular trigger. To avoid the loss of TTL trigger interrupts, the TTL trigger latch register should only be read with processor interrupts enabled.

3.7 Microsoft Quick C

SICL supports Microsoft's Quick C version 2.5 and above. Quick C can link with the standard Microsoft C SICL library, MSSICL.LIB, to create Quick C applications. The following is an example of a typical Quick C compiler and linker invocation.

qc /G2s /Ox /W4 /AL /Ic:\epconnec\include *application*

qlink /B /NOD *application*,,,**c:\epconnec\lib\mssicl**
+c:\epconnec\epcmsc+llibce.lib;

See the Microsoft Quick C documentation for specific details about the Quick C compiler and linker.

3.8 Borland C or C++

SICL supports Borland C and C++ version 2.0 and above. Borland C users must link with the Borland SICL library, BSICL.LIB, to create their application. The following is an example of a typical Borland C compiler and linker invocation..

3

bcc -2 -Ox -c -M -ml -w -ic:\epconnec\include *application*

tlink \bc\bin\c0l+ *application*,,,**c:\epconnec\lib\bsicl+c:\epconnec\epcmsc\ +\bc\bin\emu+\bc\bin\mathl+\bc\bin\cl;**

See the Borland C Tools and Utilities guide and Users guide for specific details on the Borland C/C++ compiler and linker.

3.9 Interfacing to Other Language Environments

The MSSICL.LIB uses Microsoft's C runtime library and BSICL.LIB uses Borland's C runtime library. If you need to use another compiler or language than those discussed earlier, that compiler must be able to interpret either Microsoft or Borland object formats. Linking applications with other compilers or runtime libraries requires resolution of bindings required by the SICL library and resolution of bindings introduced by the application. In addition, the compiler must be capable of generating code in the Pascal calling convention and in CDECL format for formatted I/O. Failure to resolve binding results in unresolved externals during the linking process.

3.10 Terminating GPIB Communication

When using National Instruments GPIB drivers with SICL for DOS, the EOI message is not recognized to end communications. You must do one of the following:

1) Wait for the buffer to fill. This is the default.

2) Use **itermchr** to specify a termination character. The default is not to use a terminating character.

3) Use **itimeout** to specify a timeout period. The default is infinite time.

NOTES

4. Error Messages

4

This chapter contains an alphabetic listing of error messages that may be returned when installing the following SICL drivers:

- **BIMGR.SYS**
- **SICLGPIB.SYS**
- **SICLVXI.SYS**

Accompanying each error message is the probable cause of the error, a suggested action to take to correct the error , and the source of the error.

All three device drivers are installed by the **CONFIG.SYS** file in the root directory. If you make changes to **CONFIG.SYS**, be sure to reboot your system so the change will take effect.

Bad parameter */parameter* -- Missing "=" or ":"

Cause	*Parameter* specified on the **BIMGR.SYS** installation line of the **CONFIG.SYS** file is incorrectly formatted. **BIMGR.SYS** was not installed.
Corrective Action	Correct *parameter* format (refer to *EPConnect/VXI for DOS Programmer's Reference* for a list of valid options) and reboot.
Source	**BIMGR.SYS**

Bad value for parameter */parameter* -- should be *valid_value*

Cause — The value of *parameter* on the **BIMGR.SYS** installation line in the **CONFIG.SYS** file is not valid. **BIMGR.SYS** was not installed.

Corrective Action — Change value of *parameter* to *valid_value* and reboot.

Source — **BIMGR.SYS**

*** BIMGR.SYS is not installed ***
*** SICLVXI.SYS installation aborted ***

Cause — The **SICLVXI.SYS** device driver was installed before the **BIMGR.SYS** device driver.

Corrective Action — Edit the **CONFIG.SYS** file so that **SICLVXI.SYS** is loaded after **BIMGR.SYS** and reboot.

Source — **SICLVXI.SYS**

*** Device name parameter syntax error -- default used ***

Cause — The device name parameter specified on the **SICLGPIB.SYS** installation line of the **CONFIG.SYS** file is not syntactically correct.

Corrective Action — Correct device name. Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide*, for SICLGPIB device names. The default device name EPCDEV1 was used to complete the device driver installation.

Source — **SICLGPIB.SYS**

*** Driver name parameter syntax error -- default used ***

Cause	The driver name parameter specified on the device driver installation line of the **CONFIG.SYS** file is not syntactically correct.
Corrective Action	Correct driver name. Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for driver name parameter syntax.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Duplicate device driver name ***
*** SICLGPIB.SYS installation aborted ***

Cause	**CONFIG.SYS** tried to install **SICLGPIB.SYS** more than once.
Corrective Action	Remove redundant **SICLGPIB.SYS** installation lines from the **CONFIG.SYS** file.
Source	**SICLGPIB.SYS**

*** Duplicate device driver name ***
*** SICLVXI.SYS installation aborted ***

Cause	**CONFIG.SYS** tried to install **SICLVXI.SYS** more than once.
Corrective Action	Remove redundant **SICLVXI.SYS** installation lines from the **CONFIG.SYS** file.
Source	**SICLVXI.SYS**

*** EPConnect BusManager NOT INSTALLED due to configuration errors ***

Cause	One or more parameters on the **BIMGR.SYS** installation line of the **CONFIG.SYS** file is not valid.
Corrective Action	Correct invalid parameter (refer to *EPConnect/VXI for DOS Programmer's Reference* for a list of valid options) and reboot.
Source	**BIMGR.SYS**

ERROR: Unknown EPC Hardware!

Cause	**BIMGR.SYS** does not recognize the EPC hardware. **BIMGR.SYS** was not installed.
Corrective Action	Verify that **BIMGR.SYS** version supports EPC model number. Install correct **BIMGR.SYS** version, update **CONFIG.SYS** installation line, and reboot.
Source	**BIMGR.SYS**

ERROR: VXI hardware not responding!

Cause	**CONFIG.SYS** tried to load **BIMGR.SYS** on a non-EPC computer, or there is a problem with the VXIbus interface registers on the EPC. **BIMGR.SYS** was not installed.
Corrective Action	Verify the state of the hardware by rebooting the system and checking the EPC power-on self-test (POST) results.
Source	**BIMGR.SYS**

Interrupt Stack Overflow Detected in BusManager *** --Hit CTRL-ALT-DEL to reboot

Cause	**BIMGR.SYS** detected an overflow in the **BIMGR.SYS** stack.
Corrective Action	Correct nesting error in **BIMGR.SYS** calls by user-installed VXIbus interrupt handlers.
Source	**BIMGR.SYS**

*** Not enough memory to allocate stacks *** *** SICLGPIB.SYS installation aborted ***

Cause	128 KB of DOS memory would not be available after **SICLGPIB.SYS** installation.
Corrective Action	Decrease the number of device drivers and/or their memory usage by editing the **CONFIG.SYS** file and reboot.
Source	**SICLGPIB.SYS**

*** Not enough memory to allocate stacks *** *** SICLVXI.SYS installation aborted ***

Cause	128 KB of DOS memory would not be available after **SICLVXI.SYS** installation.
Corrective Action	Decrease the number of device drivers and/or their memory usage by editing the **CONFIG.SYS** file and reboot.
Source	**SICLVXI.SYS**

*** Parameter syntax error -- parameter ignored ***

Cause	The parameter specified on the device driver installation line of the **CONFIG.SYS** file is not syntactically correct.
Corrective Action	Correct parameter syntax. Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for driver name parameter syntax.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

4

*** Process count parameter invalid -- maximum used ***

Cause	The process count parameter specified on the device driver installation line of the CONFIG.SYS is too large. Device driver was installed using the maximum process count of 16
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver process count parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Process count parameter invalid -- minimum used ***

Cause	The process count parameter specified on the device driver installation line of the **CONFIG.SYS** is too small. Device driver was installed using the minimum process count of 1
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver process count parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Process count parameter syntax error -- default used ***

Cause	The process count parameter specified on the device driver installation line of the **CONFIG.SYS** file is not syntactically correct. Device driver was installed using the default process count of 4.
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver process count parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Session count parameter invalid -- maximum used ***

Cause	The session count parameter specified on the device driver installation line of the **CONFIG.SYS** is too large. Device driver was installed using the maximum session count of 256.
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver session count parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Session count parameter invalid -- minimum used ***

Cause	The session count parameter specified on the device driver installation line of the **CONFIG.SYS** is too small. Device driver was installed using the minimum session count of 1.
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver session count parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Session count parameter syntax error -- default used ***

Cause	The session count parameter specified on the device driver installation line of the **CONFIG.SYS** file is not syntactically correct. Device driver was installed using the default session count of 16.
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver session count parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Stack count parameter invalid -- maximum used ***

Cause — The stack count parameter specified on the device driver installation line of the **CONFIG.SYS** is too large. Device driver was installed using the maximum stack count of 256.

Corrective Action — Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver stack count parameter values.

Source — **SICLGPIB.SYS** or **SICLVXI.SYS**

*** Stack count parameter invalid -- minimum used ***

Cause — The stack count parameter specified on the device driver installation line of the **CONFIG.SYS** is too small. Device driver was installed using the minimum stack count of 1.

Corrective Action — Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for device driver stack count parameter values.

Source — **SICLGPIB.SYS** or **SICLVXI.SYS**

*** Stack parameter syntax error -- default used ***

Cause — The stack parameter specified on the device driver installation line of the **CONFIG.SYS** file is not syntactically correct. Device driver was installed using the default values of four 1 KB stacks.

Corrective Action — Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver stack size parameter values.

Source — **SICLGPIB.SYS** or **SICLVXI.SYS**

*** Stack size parameter invalid -- maximum used ***

Cause	The stack size parameter specified on the device driver installation line of the **CONFIG.SYS** is too large. Device driver was installed using the maximum stack size of 64 KB.
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver stack size parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

*** Stack size parameter invalid -- minimum used ***

Cause	The stack size parameter specified on the device driver installation line of the **CONFIG.SYS** is too small. Device driver was installed using the minimum stack size of 256 bytes.
Corrective Action	Refer to Chapter 2, *Installation and Configuration*, in the *EPConnect/VXI for DOS and Windows User's Guide* for valid device driver stack size parameter values.
Source	**SICLGPIB.SYS** or **SICLVXI.SYS**

***** Unable to initialize GPIB interface *****
***** SICLGPIB.SYS installation aborted *****

Cause

SICLGPIB.SYS was unable to complete GPIB interface initialization for one or more of the following reasons:

1. GPIB hardware is not present or is improperly installed in the system (EPC-7 only).

2. The GPIB.COM device driver was not installed before the SICLGPIB.SYS device driver.

3. The GPIB.COM driver does not recognize the GPIB board name "GPIB0".

4. The device name parameter specified on the **SICLGPIB.SYS** installation line of the **CONFIG.SYS** file does not match any of the configured GPIB devices and/or the **GPIB.COM** driver does not recognize the default GPIB device name "EPCDEV1".

Corrective Action

1. (EPC-7 only) Verify that each EXM-4 module is properly seated in it's slot and verify the EXM's configuration. If the system reports EXM configuration errors at boot time or if DMA channel, IRQ, or I/O base address conflicts exist, EXM configuration is not correct. See the appropriate EXM hardware reference manual(s) for details.

2. Edit the **CONFIG.SYS** file so that **SICLGPIB.SYS** is loaded after **GPIB.COM** and reboot.

3. Execute the program **IBCONF.EXE** and ensure that the GPIB board name "GPIB0" exists and reboot.

4. Execute the program **IBCONF.EXE** and ensure that the GPIB device name "EPCDEV1" exists, edit the **CONFIG.SYS** file so that no device name parameter is present on the **SICLGPIB.SYS** installation line, and reboot the system.

Source

SICLGPIB.SYS

Unrecognized flag: */flag_value*

Cause	*Flag_value* specifies an unrecognized **BIMGR.SYS** installation parameter in the **CONFIG.SYS** file. **BIMGR.SYS** was not installed.
Corrective Action	Correct or delete *flag_value* (refer to *EPConnect/VXI for DOS Programmer's Reference* for a list of valid options) and reboot.
Source	**BIMGR.SYS**

NOTES

5. Support and Service

5.1 In North America

5.1.1 Technical Support

RadiSys maintains a technical support phone line at (503) 646-1800 that is staffed weekdays (except holidays) between 8 AM and 5 PM Pacific time. If you have a problem outside these hours, you can leave a message on voice-mail using the same phone number. You can also request help via electronic mail or by FAX addressed to RadiSys Technical Support. The RadiSys FAX number is (503) 646-1850. The RadiSys E-mail address on the Internet is *support@radisys.com*. If you are sending E-mail or a FAX, please include information on both the hardware and software being used and a detailed description of the problem, specifically how the problem can be reproduced. We will respond by E-mail, phone or FAX by the next business day.

5

Technical Support Services are designed for customers who have purchased their products from RadiSys or a sales representative. If your RadiSys product is part of a piece of OEM equipment, or was integrated by someone else as part of a system, support will be better provided by the OEM or system vendor that did the integration and understands the final product and environment.

5.1.2 Bulletin Board

RadiSys operates an electronic bulletin board (BBS) 24 hours per day to provide access to the latest drivers, software updates and other information. The bulletin board is not monitored regularly, so if you need a fast response please use the telephone or FAX numbers listed above.

The BBS operates at up to 14400 baud. Connect using standard settings of eight data bits, no parity, and one stop bit (8, N, 1). The telephone number is (503) 646-8290.

5.2 Other Countries

Contact the sales organization from which you purchased your RadiSys product for service and support.

Index