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

EPC and RadiSys are registered trademarks and EPConnect is a trademark of RadiSys Corporation.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation and Windows is a trademark of Microsoft Corporation.

National Instruments is a registered trademark of National Instruments Corporation and NI-488 and NI-488.2 are trademarks of National Instruments Corporation.

IBM and PC/AT are trademarks of International Business Machines Corporation.

August 1990

Copyright © 1990, 1994 by RadiSys Corporation

All rights reserved.

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. Introducing Bus Management for DOS 1-1

1.1 How This Manual is Organized 1-2
1.2 What is Bus Management for DOS? 1-2
1.2.1 Bus Management Library and BusManager Device Driver 1-3
1.2.2 SURM 1-4
1.3 Programming, Compiling and Linking 1-4
1.3.1 Header Files 1-4
1.3.2 Programming Interface 1-5
Calling Bus Management for DOS From MS "C" and QuickC 1-6
Calling EPConnect From Borland Turbo C 1-6
Calling EPConnect from MS BASIC 1-6
Calling Bus Management for DOS From Assembly Language 1-7
1.3.3 Compiling and Linking Applications 1-7
Compiling and Linking MS BASIC Applications 1-8
1.4 What to do Next 1-8

2. Function Descriptions 2-1

2.1 Introduction 2-1
2.2 Functions by Category 2-1
2.2.1 Bus Access Functions 2-2
2.2.2 Byte-Swapping Functions 2-2
2.2.3 Block Copy Functions 2-3
2.2.4 Interrupt and Error Handling Functions 2-4
2.2.5 Bus Control Functions 2-5
2.2.6 Commander Functionality 2-6
2.2.7 Event/Response Functions 2-7
2.2.8 Servant Functionality 2-8
2.2.9 Other Functions 2-9
2.3 Functions By Name 2-10
EpcBiosVer 2-11
EpcBmVer 2-12
EpcCkBm 2-13
EpcCkIntr 2-14
EpcDisErr 2-15
EpcDisIntr 2-17

EpcEnErr 2-18
EpcEnIntr 2-20
EpcErGet 2-22
EpcErQue 2-23
EpcErRedir 2-24
EpcErServIntr 2-26
EpcErServSig 2-28
EpcErUnredir 2-29
EpcErrStr 2-30
EpcElwsCmd 2-31
EpcFromVme 2-33
EpcFromVmeAm 2-37
EpcGetAccMode 2-41
EpcGetAmMap 2-43
EpcGetError 2-45
EpcGetIntr 2-46
EpcGetSlaveAddr 2-48
EpcGetSlaveBase 2-50
EpcGetUla 2-52
EpcHwVer 2-53
EpcLwsCmd 2-54
EpcMapBus 2-56
EpcMemSwapL 2-57
EpcMemSwapW 2-58
EpcRestState 2-59
EpcSaveState 2-60
EpcSetAccMode 2-61
EpcSetAmMap 2-63
EpcSetError 2-65
EpcSetIntr 2-67
EpcSetSlaveAddr 2-70
EpcSetSlaveBase 2-72
EpcSetUla 2-74
EpcSigIntr 2-75
EpcSwapL 2-77
EpcSwapW 2-78
EpcToVme 2-79
EpcToVmeAm 2-82
EpcVmeCtrl 2-86
EpcVxiCtrl 2-88
EpcWaitIntr 2-90

EpcWsCmd 2-93
EpcWsRcvStr 2-95
EpcWsServArm 2-97
EpcWsServPeek 2-99
EpcWsServRcv 2-101
EpcWsServSend 2-103
EpcWsSndStr 2-105
EpcWsSndStrNe 2-107
EpcWsStat 2-109

3. OLRM Functions 3-1

3.1 Calling the OLRM From MS C and QuickC 3-2
3.2 Calling the OLRM From MS BASIC and QuickBASIC 3-3
3.4 Functions by Name 3-4
OLRMAllocate 3-5
OLRMDeallocate 3-7
OLRMGetBoolAttr 3-8
OLRMGetList 3-11
OLRMGetNumAttr 3-13
OLRMGetStringAttr 3-16
OLRMRename 3-18

4. Advanced Topics 4-1

4.1 Byte Ordering and Data Representation 4-1
5.1.1 Byte Swapping Functions 4-2
4.1.2 Correcting Data Structure Byte Ordering 4-2
4.2 EPConnect Handler Execution Under DOS 4-3
4.3 Writing Device Drivers 4-4
4.3.1 General Information 4-4
4.3.2 Using the VMEbus Window 4-5
4.3.3 Interrupts 4-6
Waiting for Interrupts 4-6
Interrupt Handlers 4-7
4.3.4 Building Resident Drivers 4-7
4.3.5 Writing Device Drivers In MS C and QuickC 4-7
Using the MS C EPConnect Interface 4-7
Using the MS QuickC EPConnect Interface 4-8
Example 1: Using the VMEbus Window 4-8
Example 2: Waiting for Interrupts 4-10
Example 3: Implementing Interrupt Handlers 4-11

4.3.6 Writing Device Drivers In Turbo C 4-14
Using the Turbo "C" EPConnect Interface 4-14
4.3.7 C Optimization 4-17

5. Error Messages 5-1

6. Support and Servivce 6-1

Index I-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.

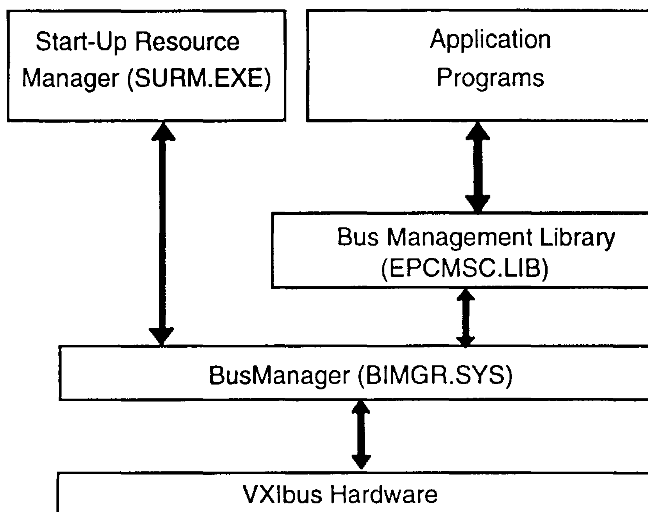


Figure 1-1. Bus Management for DOS Architecture

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

2.2.1 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

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.

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.**

2

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.**

2

EpcCkIntr

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.**

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.**

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

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

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.

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 x 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**).

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

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%*)

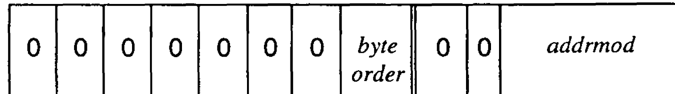
DIM src%[...]

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:



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

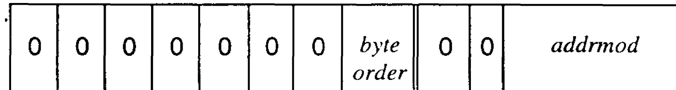
short FAR PASCAL
EpcGetAccMode(void);

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:



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

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.**

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);*

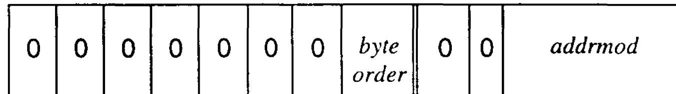
accessmode	Location where the current access mode will be placed.
busaddress	Location where the current bus window address will be placed.

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:



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.**

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

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.

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

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.**

2

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.

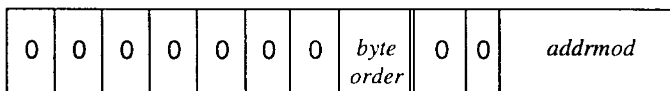
MS BASIC Synopsis

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.



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.

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&*).

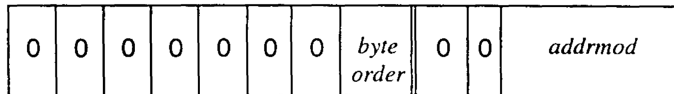
returncode% = **EpcSetAmMap**%(*accessmode%*, *busaddress&*, *mapped_ptr&*)

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*

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

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.**

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.**

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.

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.**

2

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.

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.**

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%*)

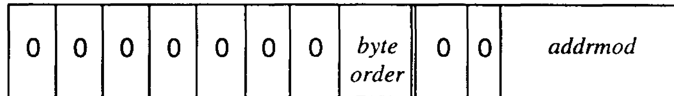
DIM *source%*[...]

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:



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 × 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.**

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.

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.**

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

2

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&*)

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.

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&*)

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*&)

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.

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

3

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

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.

3

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.

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

4.1 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.

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.

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.

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.

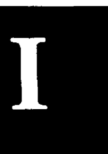
6.1.2 Bulletin Board

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

"C" optimization, 4-1

8-bit data
    no swapping needed, 4-2

A

A16, 4-5, 4-8
A24, 4-5, 4-8
A32, 4-5, 4-8
address modifiers, 2-62
advanced application programming topics, 4-1
ANSI C specification, 4-17
ANSI compatibility, Turbo C, 4-14
application development
    compiling, paths, 1-6, 1-7
Arm Command Receive, 2-102
Assembly Language, 1-6
Assembly language, 1-5
Autoexec.bat, 4-4
Automatic variables, 4-15
Auxx flag, 4-12

B

BERR, 2-35, 2-40, 2-81, 2-85
Big-endian, 4-17
BIOS version, 2-9
Block Copy Functions, 2-3
block transfer function, 4-2
bmclib.lib, 1-3
BMINT, 1-6
Borland Turbo C, 1-6
Building Resident Drivers, 4-7
Building your own drivers, 4-1
Bus Access Functions, 2-2
Bus Control Functions, 2-5
Bus interface hardware, 2-35, 2-40, 2-81, 2-85
Bus state, 2-68
Bus window, 4-5
BusManager
    Other Functions, 2-9
BusManager stack, 4-11
busmgr.h, 1-4
busmgr.inc, 1-4, 1-6
busmgr.sys, 1-3
byte ordering, 2-6, 4-1
byte ordering problems, 4-2
Byte swapping, 4-2, 4-17
byte-swapping, 4-2
    with greater data transfer widths, 4-3
Byte-swapping Functions, 2-2
byte swapping functions, 4-2

C

C Optimization, 4-17
Command size, 2-102
Compact memory model, 1-6, 4-12
compiling under C++, 1-5
compiling, applications, 1-6, 1-7
Constants, 4-7, 4-14
Control and status registers, 4-5, 4-8
Custom memory model, 4-12

D

data representation, 4-1
Data segment register, 4-15
data structure
    byte ordering, 4-3
data widths, 4-1
Definition files, 1-5
Device driver, 4-4
Direct memory access, 4-5, 4-8
Disable Interrupt, 4-6

DOS
  not reentrant, 4-12
DOS applications
  capabilities, 1-3
DOS clock, 2-102, 2-104
DOS device, 4-5
DOS interrupt, 4-5
Double-word references, 4-17

E

Enable Interrupt, 4-6
epc_obm.h, 1-4
EpcBiosVer, 2-9
  function, 2-11
EpcBmVer, 2-9
  function, 2-12
EpcCkBm, 2-9, 2-10
  function, 2-13
EpcCkIntr, 2-4
  function, 2-14
EpcDisErr, 2-4
  function, 2-15
EpcDisIntr, 2-4
  function, 2-17, 3-13
EpcEnErr, 2-4
  function, 2-18
EpcEnIntr, 2-4
  function, 2-20
EpcErrStr, 2-9
  function, 2-30
EpcFromVme, 2-3, 4-2
  function, 2-33
EpcFromVmeAm, 2-3, 4-2
  function, 2-37
EpcGetAccMode, 2-2, 3-1
  function, 2-41
EpcGetAmMap, 2-2, 3-1
  function, 2-43
EpcGetErr
  function, 2-45
EpcGetError, 2-4
EpcGetIntr, 2-4
  function, 2-46
EpcGetSlaveAddr, 2-5
  function, 2-48
EpcGetSlaveBase, 2-5
  function, 2-50
EpcGetUla, 2-5
  function, 2-52
EpcHwVer, 2-9
  function, 2-53
EpcMapBus, 2-2, 3-1
  function, 2-56
EpcMemSwapL, 2-2, 4-2
  function, 2-57
EpcMemSwapW, 2-2, 4-2
  function, 2-58
EPConnect functions, 1-5
EPConnect/VME for DOS
  what is it?, 1-2
EpcRestState, 2-2, 3-1
  function, 2-59
EpcSaveState, 2-2, 3-1
  function, 2-60
EpcSetAccMode, 2-2, 3-1
  function, 2-61
EpcSetAmMap, 2-2, 2-63
EpcSetError, 2-4, 4-4
  function, 2-65
EpcSetIntr, 2-4, 2-67, 4-4
EpcSetSlaveAddr, 2-5
  function, 2-70
EpcSetSlaveBase, 2-5
  function, 2-72
EpcSetUla, 2-5
  function, 2-74
EpcSigIntr, 2-4
  function, 2-75
epcstd.h, 1-4
EpcSwapL, 2-2, 4-2, 4-3
  function, 2-77
EpcSwapW, 2-2, 4-2, 4-3

function, 2-78
EpcToVme, 2-3, 4-2, 4-3
function, 2-79
EpcToVmeAm, 2-3, 4-2, 4-3
function, 2-82
EpcVmeCtrl, 2-5
function, 2-86
EpcWaitIntr, 2-4, 4-6
function, 2-90
EpcWsServArm, 2-8
function, 2-97
EpcWsServPeek, 2-8
function, 2-99
EpcWsServRcv, 2-8
function, 2-101
EpcWsServSend, 2-8
function, 2-103
Error Handling Functions, 2-4
error messages, 1-8, 5-1
system-level errors, 5-1
Error string, 2-9

F

Fast Copy, 2-35, 2-40, 2-81, 2-85
fully reentrant functions, 4-3
function descriptions, 1-8
Functions By Name, 2-10

H

Handler, 2-4
handler functions, 4-3
handler operations, 4-1
handlers
interrupt execution, 4-4
Hardware version, 2-9
header files, 1-4
High-level programming languages, 1-5

I

IACK, 2-91, 4-6, 4-11
Implementing Interrupt Handlers, 4-11
installation and configuration, 1-8
Intel, byte ordering, 4-1
Interface library, 1-5, 4-5
interrupt
handler execution, 4-4
Interrupt acknowledge cycle, 2-91, 4-6
Interrupt acknowledgement, 2-91
Interrupt and Error Handling Functions, 2-4
Interrupt handler, 4-13, 4-15
interrupt handler
installation, 4-4
Interrupt Handlers, 4-7
interrupt thread, 4-4
Interrupts, 4-6
Interrupts, Waiting for, 4-10
Interrupts, waiting for, 4-6

L

Large memory model, 1-6, 4-12, 4-15
library files, 1-5
Little-endian, 4-17
Locking protocol, 2-98

M

manual organization, 1-2
Master, 4-5, 4-8
Medium memory model, 1-6, 4-12
Memory model, 4-12
Memory reference optimization, 4-17
Message interrupt, 2-92
MI flag, 4-15
Motorola, byte ordering, 4-1
MS C and QuickC, 1-6
MS C and QuickC, Writing Device Drivers In, 4-7
MS C EPConnect Interface, 4-7
MS QuickC EPConnect Interface, 4-8
Multi-tasking, 4-7, 4-11

O

Odd-only, 2-35, 2-39, 2-40, 2-81, 2-84, 2-85
Optimizing memory references, 4-17
Other Functions, BusManager, 2-9

P

Pipelining, 2-35, 2-40, 2-81, 2-85
Poll, 4-6, 4-10
Program-specific drivers, 4-4, 4-16
programming interface, 4-1
Prototype, 4-7, 4-14
Prototyping, 1-6

R

Race condition, 2-92, 2-102, 4-11
RadiSys EPC controllers, 4-1
Read-modify-write, 2-34, 2-39, 2-80, 2-84
Reentrancy, 4-15
Resident device drivers, 4-7
Resident drivers, 4-4, 4-16
Response register, 2-98
Restore State, 4-8

S

Save State, 4-5, 4-8
SE option, 4-7
Segment, 4-7, 4-15
Set Access Mode and Map Bus, 4-5, 4-8
Set Interrupt Handler, 4-6
Slave address, 4-5, 4-8
Small memory model, 1-6, 4-12
Software version, 2-9
Stack checking, 4-12
State, 4-8, 4-11
Status registers, 4-5, 4-8
Strong type checking, 1-6
Structure definitions, 4-7, 4-14

T

Technical Support, 6-1
  E-mail, 6-1
  E-mail address, 6-1
  electronic bulletin board (BBS), 6-1
  FAX, 6-1
Terminate-and-stay-resident program, 4-4
Too many segments, 4-7
TSR, 4-4
Turbo C, 1-6
  ANSI compatibility, 4-14
Turbo C EPConnect Interface, 4-14
Turbo C, Writing Device Drivers In, 4-14

U

Using the VMEbus Window, 4-5

V

VMEbus interrupts, 4-6
VMEbus Window, 4-8
vmeregs.h, 1-5
Volatile, 4-17
VXIbus devices, 4-1

W

Waiting for Interrupts, 4-6, 4-10
Word and double-word references, 4-17
Word serial command, 2-104
WRDY, 2-104
Writing Device Drivers, 4-4
  General Information, 4-4