

# KAV30

---

digital

## Programmer's Reference Information

# KAV30

---

## Programmer's Reference Information

Order Number: AA-PEYCA-TE

**July 1991**

This guide describes the KAV30 software and describes how to develop real-time applications for the KAV30.

**Revision Information:** This is a new guide.

**Operating System and Version:** VMS Version 5.0 or higher,  
VAXELN Version 4.2 or higher

**Software Version:** VAXELN KAV Toolkit Extensions for VMS  
Version 1.0

**Digital Equipment Corporation  
Maynard, Massachusetts**

---

**July 1991**

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1991.

All Rights Reserved.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation: CVAX, DEC, DECnet, DECwindows, Digital, rtVAX, ULTRIX, VAX, VAX Ada, VAX C, VAX FORTRAN, VAXELN, VAXELN Pascal, VMS, and the DIGITAL logo.

68000 and Motorola are registered trademarks of Motorola, Inc. Intel is a trademark of Intel Corporation.

This document was prepared using VAX DOCUMENT, Version 2.0.

---

# Contents

<b>Preface</b> .....	ix
<b>1 KAV30 Overview</b>	
1.1 KAV30 Hardware .....	1-1
1.2 VAXELN Toolkit .....	1-2
1.3 KAV30 Software .....	1-4
1.3.1 Naming Conventions .....	1-4
1.3.2 KAV30 System Services .....	1-5
<b>2 KAV30 Functionality</b>	
2.1 VMEbus Master Functionality .....	2-1
2.2 VMEbus Slave Functionality .....	2-2
2.3 VSB Master Functionality .....	2-3
2.4 VMEbus Arbiter Functionality .....	2-3
2.5 VSB Arbiter Functionality .....	2-4
2.6 VMEbus Deadlock .....	2-4
2.7 VMEbus Utility Bus Signals .....	2-4
2.8 DAL Bus Timeouts .....	2-5
2.9 Parity Errors .....	2-6
2.10 VMEbus Interrupt Handler Functionality .....	2-6
2.10.1 Handling Vectored Interrupts .....	2-6
2.10.2 Handling Autovectored Interrupts .....	2-8
2.11 VMEbus Interrupt Requester Functionality .....	2-10
2.12 VSB Interrupt Handler Functionality .....	2-12
2.13 KAV30 Interrupt Priority .....	2-12



### 3 KAV30 Kernel

3.1	Asynchronous System Trap Processing . . . . .	3-1
3.1.1	AST Delivery . . . . .	3-2
3.1.2	AST Data Structures . . . . .	3-3
3.2	Timers . . . . .	3-5
3.3	Calendar/clock . . . . .	3-6
3.4	FIFO Buffers . . . . .	3-9
3.5	Battery Backed-Up RAM . . . . .	3-10
3.6	Scatter-Gather Map . . . . .	3-10
3.6.1	Outgoing SGM . . . . .	3-10
3.6.2	Incoming SGM . . . . .	3-14
3.6.3	Byte Swapping During SGM Operations . . . . .	3-18
3.7	Communicating with Another KAV30 . . . . .	3-21
3.8	KAV30 Error Logging Support . . . . .	3-24

### 4 KAV30 System Services

KAV\$BUS_BITCLR . . . . .	4-2
KAV\$BUS_BITSET . . . . .	4-10
KAV\$BUS_READ . . . . .	4-16
KAV\$BUS_WRITE . . . . .	4-23
KAV\$CHECK_BATTERY . . . . .	4-30
KAV\$CLR_AST . . . . .	4-34
KAV\$DEF_AST . . . . .	4-37
KAV\$FIFO_READ . . . . .	4-41
KAV\$FIFO_WRITE . . . . .	4-46
KAV\$GATHER_KAV_ERRORLOG . . . . .	4-51
KAV\$IN_MAP . . . . .	4-58
KAV\$INT_VME . . . . .	4-67
KAV\$LIFO_WRITE . . . . .	4-74
KAV\$NOTIFY_FIFO . . . . .	4-78
KAV\$OUT_MAP . . . . .	4-84
KAV\$QUEUE_AST . . . . .	4-96
KAV\$RTC . . . . .	4-100
KAV\$RW_BBRAM . . . . .	4-116
KAV\$SET_AST . . . . .	4-122
KAV\$SET_CLOCK . . . . .	4-127
KAV\$TIMERS . . . . .	4-131
KAV\$UNMAP . . . . .	4-144

KAV\$VME_SETUP .....	4-150
----------------------	-------

## 5 Developing KAV30 Applications

5.1	Design Guidelines .....	5-1
5.1.1	Accessing the VMEbus and VSB Address Space .....	5-1
5.1.1.1	Directly Accessing the VMEbus and VSB Address Space .....	5-2
5.1.1.2	Using the KAV\$BUS_READ and KAV\$BUS_WRITE Services .....	5-3
5.1.2	Writing Asynchronous System Trap Routines .....	5-3
5.2	Coding Guidelines .....	5-3
5.2.1	VAX Ada .....	5-4
5.2.1.1	Coding Asynchronous System Trap Routines in VAX Ada .....	5-4
5.2.2	VAX C .....	5-7
5.2.3	VAX FORTRAN .....	5-8
5.2.4	VAXELN Pascal .....	5-9
5.2.4.1	Coding AST Routines in VAXELN Pascal .....	5-9
5.3	Compiling and Linking KAV30 Applications .....	5-12
5.4	Building KAV30 System Images .....	5-13
5.4.1	Configuring the VMEbus and VSB .....	5-14
5.5	Loading and Running KAV30 System Images .....	5-20
5.6	Debugging KAV30 Applications .....	5-20
5.7	Developing SCSI Class Drivers .....	5-22
5.8	Building a SCSI Class Driver into an Application .....	5-23

## A Initial KAV30 Configuration

A.1	Hardware Configuration .....	A-1
A.2	Software Settings .....	A-2

## B Example Programs—Interprocessor Communication

B.1	FIFO Producer .....	B-1
B.2	FIFO Consumer .....	B-5

## **C Example Programs—MVME335 Device Driver**

C.1	Device Driver . . . . .	C-1
C.2	Interrupt Service Routine . . . . .	C-16

## **D Example Programs—VDAD Device Driver**

D.1	Device Driver . . . . .	D-1
D.2	Definitions File . . . . .	D-17
D.3	Test Program . . . . .	D-21
D.4	Build File . . . . .	D-25
D.5	Data File . . . . .	D-25

## **Glossary**

## **Index**

## **Figures**

1-1	Host and Run-Time System Software . . . . .	1-3
2-1	Converting VMEbus Interrupt Vectors into VAX Interrupt Vectors . . . . .	2-7
2-2	Constructing an 8-bit VMEbus Interrupt Vector . . . . .	2-11
3-1	ASB Fields . . . . .	3-4
3-2	AST Queue . . . . .	3-4
3-3	Calendar/clock Address Map . . . . .	3-8
3-4	KAV30 as Producer and Consumer . . . . .	3-11
3-5	KAV30 as Neither Producer or Consumer . . . . .	3-12
3-6	Outgoing SGM Conversion to VMEbus or VSB A32 Addresses . . . . .	3-13
3-7	Outgoing SGM Conversion to VMEbus or VSB A24 Addresses . . . . .	3-14
3-8	Outgoing SGM Conversion to VMEbus or VSB A16 Addresses . . . . .	3-15
3-9	A32 Incoming VMEbus Address . . . . .	3-16
3-10	A24 Incoming VMEbus Address . . . . .	3-17
3-11	Incoming SGM Conversion of A32 VMEbus Addresses . . . . .	3-17
3-12	Incoming SGM Conversion of A24 VMEbus Addresses . . . . .	3-18
3-13	Little-Endian Storage Format . . . . .	3-19

3-14	Big-Endian Storage Format .....	3-19
3-15	Mode 0 Byte Swapping .....	3-20
3-16	Mode 2 Byte Swapping .....	3-22
3-17	Mode 3 Byte Swapping .....	3-23
3-18	Sample Master Error Log Entry .....	3-25
3-19	Sample Slave Error Log Entry .....	3-25
4-1	Programming the Real-Time Clock .....	4-101
5-1	A Remote Debugging Configuration .....	5-21
5-2	A Local Debugging Configuration .....	5-22
5-3	Sample Add Device Description Menu .....	5-24

## Tables

1-1	KAV30 System Services .....	1-5
2-1	Interrupt Source Codes .....	2-7
2-2	System Control Block Layout .....	2-8
2-3	SCB Vector Offsets for Autovectorized ISRs .....	2-10
2-4	VMEbus Address Lines A<3..1> .....	2-11
2-5	KAV30 Interrupt Pins .....	2-13
2-6	KAV30 Interrupt Priorities .....	2-14
3-1	Internal Master Error Code .....	3-27
3-2	Internal Slave Error Code .....	3-28
5-1	Compiling and Linking Commands .....	5-12



---

# Preface

This guide describes the KAV30 software and describes how to develop real-time applications for the KAV30.

## Who Should Read This Guide

This guide is for programmers who want to develop real-time applications for the KAV30. This guide assumes that the reader is familiar with the VAXELN™ Toolkit and that the reader can program in one of the following computer languages:

- VAX Ada™
- VAX C™
- VAX FORTRAN™
- VAXELN Pascal™

## Structure of This Guide

This guide is divided into five chapters, four appendixes, a glossary, and an index:

- Chapter 1 gives an overview of the KAV30 hardware and software.
- Chapter 2 describes the KAV30 bus functionality and related topics.
- Chapter 3 describes how the KAV30 hardware relates to the KAV30 software.
- Chapter 4 describes the KAV30 system services.
- Chapter 5 describes how to develop applications for the KAV30.
- Appendix A describes the initial KAV30 hardware and software configuration.
- Appendix B lists source code which demonstrates interprocessor communication between two KAV30s.

- Appendix C lists source code which implements an MVME 335 device driver.
- Appendix D lists source code which implements a VDAD device driver.
- The glossary defines some important terms used in this guide.

## Associated Documents

For more information, see the following documents:

- *KAV30 Software Cover Letter* (AV-PEYFA-TE)
- *KAV30 Software Product Description* (AE-PFB5A-TE)
- *KAV30 System Support Addendum* (AE-PFB6A-TE)
- *KAV30 Software Installation and System Testing Information* (AA-PEYAA-TE)
- *KAV30 Hardware Cover Letter* (AV-PFSSA-TE)
- *KAV30 Hardware Installation and User's Information* (AA-PFM6A-TE)

## Related Documents

For additional information, see the following documents:

- *VMEbus Specification, Revision C.1* (PRINTEX, Phoenix, AZ, USA)
- *The VME Subsystem Bus (VSB) Specification, Revision B.1* (Motorola®, Phoenix, AZ, USA)
- *rtVAX 300 Hardware User's Guide*
- *rtVAX 300 Programmer's Guide*
- *Introduction to VAXELN*
- *VAXELN rtVAX 300 Supplement*
- *VAXELN Ada User's Guide*
- *VAXELN Application Design Guide*
- *VAXELN C Runtime Library Reference Manual*
- *VAXELN C Reference Manual*
- *VAXELN Development Utilities Guide*
- *VAXELN FORTRAN Runtime Library Reference Manual*
- *VAXELN Guide to DECwindows*

- *VAXELN Installation Guide*
- *VAXELN Internals Manual*
- *VAXELN Master Index and Glossary*
- *VAXELN Messages Manual*
- *VAXELN Pascal Language Reference Manual*
- *VAXELN Pascal Runtime Library Reference Manual*
- *VAXELN Pocket Reference*
- *VAXELN Release Notes*
- *VAXELN Runtime Facilities Guide*

For detailed information about VAXELN, Digital Equipment Corporation recommends the *VAXELN Internals and Data Structures* manual. The *VAXELN Internals and Data Structures* manual describes the internal data structures and operations of the VAXELN Kernel and its associated subsystems.

For information about the VAX architecture, Digital recommends the following documents:

- Henry M. Levy and Richard H. Eckhouse, Jr., *Computer Programming and Architecture: The VAX*, Second Edition, Bedford, (Massachusetts): The Digital Press, 1989
- Timothy E. Leonard, editor, *The VAX Architecture Reference Manual*, Bedford (Massachusetts): The Digital Press, 1987

## Conventions

The following conventions are used in this guide:

Convention	Description
<b>Note</b>	A note contains information that is of special importance to the reader.



Convention	Description
UPPERCASE	Words in uppercase indicate the following: <ul style="list-style-type: none"> <li>• VMS™ reserved words and identifiers</li> <li>• VAXELN reserved words and identifiers</li> <li>• KAV30 reserved words and identifiers</li> <li>• VAX signal lines</li> <li>• VMEbus signal lines</li> </ul>
<i>italic type</i>	Italic type emphasizes important information and indicates the complete titles of manuals.
<b>boldface type</b>	Boldface type indicates the first occurrence of terms defined either in text, in the glossary, or both.
Monospace Type	Monospace type indicates system displays and user input.
[ ]	In system service format descriptions, brackets enclose optional system service arguments. Brackets are also used in the syntax of a directory name in a VMS file specification.
...	Horizontal ellipsis points indicate that you repeat the preceding item one or more times.
. . .	Vertical ellipsis points in a figure or example indicate that not all the information the system displays is shown.
<i>n.nn</i>	A period in numerals signals the decimal point indicator. For example, <i>1.75</i> equals <i>one and three-fourths</i> .
<i>nn nnn.nnn nn</i>	A space character separates groups of 3 digits in numerals with 5 or more digits. For example, <i>10 000</i> equals <i>ten thousand</i> .
< <i>n..n</i> >	Three or more consecutive signal line numbers are enclosed in angle brackets, with the first line number separated from the last line number with two periods (..). For example, signal lines < <i>1..4</i> > represent signal lines <i>1, 2, 3, and 4</i> .

---

# KAV30 Overview

This chapter gives an overview of the KAV30 hardware and software. It briefly describes the following:

- KAV30 hardware
- VAXELN Toolkit
- KAV30 software

The KAV30 software and hardware operate in a host and target system environment. You use the KAV30 software to develop and build VAXELN applications on a VMS (host) system. Then, you down-line load, run, and debug the applications on the KAV30 (target) system. The KAV30 is a single-board computer that allows VAXELN applications to interface with VMEbus and VME subsystem bus (VSB) devices.

The KAV30 software is a VMS layered product that forms an extension to the VAXELN Toolkit. The VAXELN Toolkit is a set of development tools that allows you to develop real-time applications quickly and easily for the VAX™ family of computers.

## 1.1 KAV30 Hardware

This section describes the KAV30 hardware. The KAV30 is a single-board computer that allows VAXELN applications to interface with VMEbus and VSB devices. The KAV30 contains the following hardware:

- An rtVAX™ 300 real-time processor, which includes a CVAX™ microprocessor, a floating point coprocessor, and a second-generation Ethernet controller
- Logic circuitry that implements a scatter-gather map (SGM)
- 4M or 16M bytes of system random-access memory (RAM)
- Up to 1M bytes of user read-only memory (ROM)
- Four 255-longword first-in/first-out (FIFO) buffers

## Introduction

- 32K bytes of battery backed-up RAM (programmers can access 22K bytes)
- A calendar/clock
- Counter/timers
- A second-generation small computer systems interface (SCSI) controller
- Two serial line ports: the console port and the auxiliary port
- A watchdog timer
- Logic circuitry that implements VMEbus arbiter and bus request functionality
- Logic circuitry that implements VSB arbiter and bus request functionality
- Logic circuitry that implements VMEbus interrupt request and handler functionality
- Logic circuitry that implements VSB interrupt request and handler functionality

## 1.2 VAXELN Toolkit

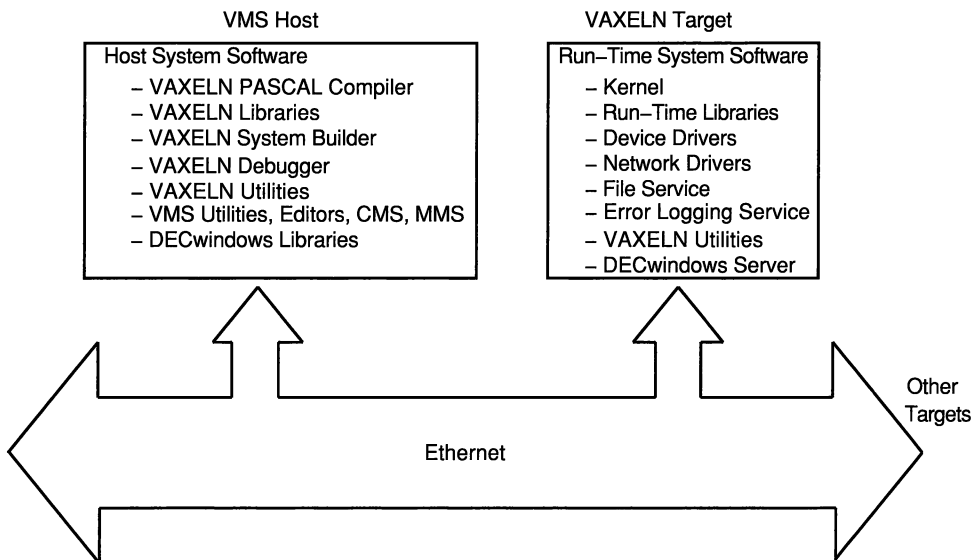
This section describes the VAXELN Toolkit. The VAXELN Toolkit is a set of development tools that allows you to develop real-time applications quickly and easily for the VAX family of computers. The VAXELN Toolkit consists of the following:

- VAXELN Host System Software, which consists of the following:
  - High-level language compiler (supported languages are VAXELN Ada, VAX C, VAX FORTRAN, and VAXELN Pascal)
  - Object module libraries
  - System Builder
  - Debugger
- VAXELN Run-Time System Software, which consists of the following:
  - VAXELN kernel
  - Run-time libraries
  - Network and file services
  - Device drivers
  - VAXELN DECwindows™ client functionality

- Error-logging support
- VAXELN Utilities, which consist of the following:
  - VAXELN Performance Utility (EPA)
  - VAXELN Display Utility (EDISPLAY)
  - VAXELN Command Language Utility (ECL)
  - Local Area Transport (LAT) Control Program (LATCP)
  - Outbound remote terminal utility (SET\_HOST)

Figure 1–1 shows how the components of the VAXELN Toolkit are distributed between the host and target systems. The figure also shows the host and target connected by an Ethernet.

**Figure 1–1 Host and Run-Time System Software**



## Introduction

See the *Introduction to VAXELN* for information on the VAXELN Toolkit. For more detailed information, see the *VAXELN Development Utilities Guide* and the *VAXELN Runtime Facilities Guide*.

### 1.3 KAV30 Software

This section describes the KAV30 software. The KAV30 software is a VMS layered product that forms an extension to the VAXELN Toolkit. It simplifies the development of VAXELN applications that use the KAV30. The KAV30 software consists of the following:

- The KAV30 kernel
- A SCSI port driver
- Automated initialization of the KAV30

You can specify initialization parameters when you build the VAXELN system.

- Sample applications, including a device driver for a VMEbus device
- The KAV30 system exerciser

The KAV30 kernel is a VAXELN kernel image that is designed specifically for the KAV30. It has the following features:

- System services that you can use to set up and control the KAV30
- Asynchronous notification of events by means of asynchronous system traps (ASTs) for KAV30 kernel services and user-written device drivers
- Error logging to battery backed-up RAM
- VMEbus and VSB exception handling

#### 1.3.1 Naming Conventions

The KAV30 software conforms to the naming conventions set out in the *Guide to Creating VMS Modular Procedures*. The names are derived from the facility prefix. The facility prefix for the KAV30 software is KAV\$. All the KAV30 software names (such as system services, global variables, and status codes) begin with KAV\$.

---

#### Note

---

Because the dollar sign (\$) is not part of the VAX Ada character set, KAV30 kernel services in VAX Ada have the facility prefix KAV\_ instead of KAV\$.

---

### 1.3.2 KAV30 System Services

The KAV30 system services allow you to do the following tasks:

- Initialize the KAV30
- Set up and control access to the devices on the VMEbus and VSB
- Set up and control the KAV30 FIFO buffers
- Set up and control the calendar/clock and counter/timers
- Exchange data with VMEbus and VSB devices
- Read and write the battery backed-up RAM on the KAV30
- Gather error information from the battery backed-up RAM
- Use ASTs in user-written device drivers

Table 1–1 summarizes the KAV30 system services. See Chapter 4 for more detailed information on each system service.

**Table 1–1 KAV30 System Services**

System Service	Description
KAV\$BUS_BITCLR	Clears the bits at a VMEbus or VSB address
KAV\$BUS_BITSET	Sets the bits at a VMEbus or VSB address
KAV\$BUS_READ	Reads the contents of a VMEbus or VSB address
KAV\$BUS_WRITE	Writes data to a VMEbus or VSB address
KAV\$CHECK_BATTERY	Checks the power supply to the battery backed-up RAM and the calendar/clock
KAV\$CLR_AST	Clears a device's AST queue
KAV\$DEF_AST	Creates an asynchronous system block (ASB) for an event on a VMEbus or VSB device
KAV\$FIFO_READ	Reads data from a KAV30 FIFO buffer
KAV\$FIFO_WRITE	Writes data to a KAV30 FIFO buffer
KAV\$GATHER_KAV_ERRORLOG	Reads error log information from the KAV30 battery backed-up RAM
KAV\$IN_MAP	Maps a 64K byte page of VMEbus address space to the KAV30 process address space

(continued on next page)

## Introduction

**Table 1–1 (Cont.) KAV30 System Services**

<b>System Service</b>	<b>Description</b>
KAV\$INT_VME	Generates vectored VMEbus interrupts
KAV\$LIFO_WRITE	Writes data to a KAV30 last-in/first-out (LIFO) buffer
KAV\$NOTIFY_FIFO	Delivers an AST when a specified event occurs in a KAV30 FIFO buffer
KAV\$OUT_MAP	Maps KAV30 system I/O space to the VMEbus or VSB address space, in 64K byte pages
KAV\$QUEUE_AST	Queues an AST for delivery to a process
KAV\$RTC	Performs real-time clock functions, using the KAV30 calendar/clock
KAV\$RW_BBRAM	Reads or writes the KAV30 battery backed-up RAM
KAV\$SET_AST	Places an entry in a device's AST queue
KAV\$SET_CLOCK	Sets the KAV30 system clock and the calendar /clock
KAV\$TIMERS	Sets a counter/timer and delivers an AST when the timer interval expires
KAV\$UNMAP	Unmaps VMEbus address space from KAV30 system RAM, or KAV30 system RAM from the VMEbus address space
KAV\$VME_SETUP	Configures the VMEbus and VSB interrupt delivery mechanism

---

# KAV30 Functionality

This chapter describes the KAV30 bus functionality and related topics. It gives information on the following:

- VMEbus master functionality
- VMEbus slave functionality
- VSB master functionality
- VMEbus arbiter functionality
- VSB arbiter functionality
- VMEbus deadlock
- VMEbus utility bus signals
- Data and address lines (DAL) bus timeouts
- Parity errors
- VMEbus interrupt handler functionality
- VMEbus interrupt requester functionality
- VSB interrupt handler functionality
- KAV30 interrupt priority

## 2.1 VMEbus Master Functionality

This section describes the KAV30 VMEbus master functionality. The KAV30 includes logic circuitry that implements VMEbus master functionality. That is, the KAV30 can start read/write operations between itself and other devices on the VMEbus. To start read/write operations on the VMEbus, the KAV30 must first get control of the VMEbus. To do this, the KAV30 can use any one of the four VMEbus bus request (BR) lines. When it has control of the VMEbus, the KAV30 VMEbus master logic circuitry can perform the following transfers:

- VMEbus A16, A24, and A32 data transfers



## KAV30 Functionality

- Read-modify-write transfers

See *The VMEbus Specification* for more information about VMEbus data transfers.

The KAV30 VMEbus master logic circuitry can operate in the following modes:

- Release-when-done (RWD) or release-on-request (ROR)
- Fair or not fair
- Hidden or not hidden

---

### Note

---

The KAV30 can operate in hidden mode only when the other masters on the VMEbus system operate in hidden mode.

---

The KAV30, by default, issues bus requests to the VMEbus, using BR line 3 while operating in the ROR, not fair, and not hidden modes. See *The VMEbus Specification* for more information about the VMEbus master operation modes.

## 2.2 VMEbus Slave Functionality

This section describes the KAV30 VMEbus slave functionality. The KAV30 includes logic circuitry that implements the VMEbus slave functionality. That is, another master module on the VMEbus system can start read/write operations between itself and the KAV30. The KAV30 VMEbus slave logic circuitry can access the following devices on the KAV30:

- System RAM
- FIFO buffers
- VMEbus reset register

The KAV30 VMEbus slave logic circuitry can process the following transfers:

- VMEbus A16, A24, and A32 data transfers
- VMEbus D08, D16, D32, and block mode data transfers
- Read-modify-write transfers

See *The VMEbus Specification* for more information about VMEbus data transfers.

The VMEbus slave logic circuitry requests control of the DAL bus when it wants to respond to a VMEbus cycle. The KAV30 central processing unit (CPU), when appropriate, indicates that the VMEbus slave logic circuitry can use the DAL bus. When the VMEbus slave logic circuitry finishes using the bus it signals the KAV30 CPU and returns control of the bus to the CPU. However, when the KAV30 wants to perform a read-modify-write cycle, the VMEbus slave data interface logic circuitry does not return control of the DAL bus until it completes the read and write cycles.

---

### Note

---

The devices that transfer data into the KAV30 by direct memory access (DMA) can use a block size of up to 4 longwords.

---

If a VMEbus device uses a block size that is greater than 4 longwords, a bus timeout occurs on the VMEbus device.

## 2.3 VSB Master Functionality

This section describes the KAV30 VSB master functionality. The KAV30 includes logic circuitry that implements VSB master functionality. That is, the KAV30 can start read/write operations between itself and other devices on the VSB. The KAV30 supports VSB ALTERNATE, SYSTEM, and I/O address spaces. See the *VME Subsystem Bus (VSB) Specification* for more information about the VSB address spaces.

---

### Note

---

The VSB master logic circuitry does not assert the VSB LOCK signal when the KAV30 CPU performs a read-modify-write cycle.

---

## 2.4 VMEbus Arbiter Functionality

This section describes the KAV30 VMEbus arbiter functionality. The KAV30 includes logic circuitry that implements VMEbus arbiter functionality. The KAV30 can perform prioritized or round-robin arbitration, using the four VMEbus bus request and bus grant levels. Use the VMEbus arbiter switch to enable the VMEbus arbiter functionality. Use the VAXELN System Builder to configure the VMEbus arbiter functionality. See the *KAV30 Hardware Installation and User Information* for more information about the VMEbus

## KAV30 Functionality

arbiter switch. See Section 5.4.1 for more information about configuring the VMEbus arbiter functionality.

Only one VMEbus arbiter can be on a VMEbus system, and that arbiter must reside in the leftmost slot of the VMEbus system (slot 1). When you use the KAV30 as a VMEbus arbiter, the KAV30 can also provide the VMEbus system clock and logic circuitry to drive the VMEbus SYSRESET signal.

## 2.5 VSB Arbiter Functionality

This section describes the KAV30 VSB arbiter functionality. The KAV30 includes logic circuitry that implements VSB arbiter functionality. You can use software to enable and configure the VSB arbiter functionality (see Section 5.4.1 for more information).

Only one VSB arbiter can be on a VSB system, and that arbiter must reside in the leftmost slot of the VSB system (slot 0).

## 2.6 VMEbus Deadlock

This section describes how the KAV30 deals with VMEbus deadlock. A deadlock can occur on the KAV30 when the KAV30 VMEbus slave logic circuitry requests control of the DAL bus at the same time as the KAV30 CPU requests control of the VMEbus. In such a case, the VMEbus slave logic circuitry is using the VMEbus and is requesting ownership of the DAL bus, while the KAV30 CPU is using the DAL bus and is requesting ownership of the VMEbus.

Such deadlocks are handled by the hardware. When this deadlock situation occurs, the KAV30 postpones the KAV30 CPU's request for the VMEbus and allows the VMEbus slave logic circuitry to own the DAL bus. When the VMEbus slave logic circuitry returns control of the DAL bus, the postponed CPU request for the VMEbus is resumed.

## 2.7 VMEbus Utility Bus Signals

This section describes the KAV30 VMEbus utility bus signals. The KAV30 uses the following VMEbus utility bus signals:

- **SYSRESET**  
When enabled, the SYSRESET signal can generate a local reset pulse on the KAV30 with the same duration as the SYSRESET signal. This pulse allows the system to initialize the KAV30 at the same time as the other modules on the VMEbus system.

The VMEbus SYSRESET signal jumper controls the interaction between the KAV30 and the SYSRESET signal. See the *KAV30 Hardware Installation and User's Information* for more information about the VMEbus SYSRESET signal jumper.

- **VMEbus Global Reset Register**  
A VMEbus write access via the SGM to the VMEbus global reset register causes a 10 microsecond ( $\mu$ s) local reset pulse.
- **ACFAIL**  
The KAV30 interrupts its CPU at interrupt priority level 1E when the VMEbus ACFAIL signal is asserted.
- **SYSFAIL**  
The KAV30 VMEbus master interface logic circuitry can assert the SYSFAIL signal and respond to assertions of the SYSFAIL signal.

When the KAV30 detects an assertion of the SYSFAIL signal, it performs one of the following actions:

- The KAV30 delivers an AST  
To deliver an AST, call the KAV\$SET\_AST routine with the KAV\$K\_VME\_SYSFAIL device code and AST routine address as arguments.
- The KAV30 calls an interrupt service routine (ISR) at vector 540<sub>16</sub>

The KAV30 performs one of these two actions. The KAV30 cannot ignore the assertion of the SYSFAIL signal. The action that the KAV30 performs depends on the setting of the VAXELN System Builder System Parameter 1. See Section 5.4.1 for more information.

See *The VMEbus Specification* for more information about the VMEbus utility signals.

## 2.8 DAL Bus Timeouts

This section describes DAL bus timeouts. The KAV30 CPU, SCSI controller, and master logic circuitry can act as DAL bus masters. The DAL bus generates an error when it times out. The default DAL bus timeout period is approximately 20  $\mu$ s. You can use the KAV\$TIMERS service to change the DAL bus timeout period. However, Digital™ strongly recommends that you do not change this value.

### 2.9 Parity Errors

This section describes parity errors. When the KAV30 CPU stores data in its system RAM, it sends one parity bit with each byte of data. It sends an even parity bit when a byte has an even address, and an odd parity bit when a byte has an odd address. When the CPU reads a byte of system RAM, it checks the parity bit.

### 2.10 VMEbus Interrupt Handler Functionality

This section describes the KAV30 VMEbus interrupt handler functionality. The KAV30 includes logic circuitry that implements VMEbus interrupt handler functionality. The VMEbus interrupt handler logic circuitry can handle interrupt-requests (IRQs) that it receives from the devices on the VMEbus.

The KAV30 VMEbus interrupt handler logic circuitry can receive IRQs on the VMEbus lines IRQ<1..7> and the POWER\_FAIL line. When it receives more than one IRQ, the interrupt handler logic circuitry assigns priorities to the requests depending on the line on which it receives the requests. It handles requests in the order of the highest priority to the lowest priority. The IRQs on the POWER\_FAIL line have the highest priority. The IRQs on the IRQ7 line have the next highest priority, and so on to the IRQs on the IRQ1 line, which have the lowest priority.

VMEbus autovectored interrupts occur when a module asserts a VMEbus IRQ line but does not provide an interrupt vector. Often VMEbus systems use the VMEbus IRQ 2 line for autovectored interrupts. The KAV30 hardware can handle only autovectored interrupts on the VMEbus IRQ7 line. However, an application program can emulate a vectored VMEbus IRQ 7 interrupt by forcing a VMEbus interrupt-acknowledge (IACK) cycle from the software.

---

#### Note

---

Do not use VMEbus IRQ lines for vectored and autovectored interrupts.

---

#### 2.10.1 Handling Vectored Interrupts

The KAV30 receives 8-bit interrupt vectors from the VMEbus. However, the KAV30 CPU expects 16-bit interrupt vectors. Therefore, the KAV30 interrupt handler logic circuitry must convert the 8-bit interrupt vectors it receives into 16-bit interrupt vectors that the KAV30 CPU can process. Figure 2–1 shows this conversion.

The VAX interrupt vector consists of the following data:

- Bits <0,1> and <12..15> contain the value 0
- Bits <2..9> contain the 8-bit VMEbus interrupt vector
- Bits <10,11> contain the interrupt source code (see Table 2–1)

**Figure 2–1 Converting VMEbus Interrupt Vectors into VAX Interrupt Vectors**

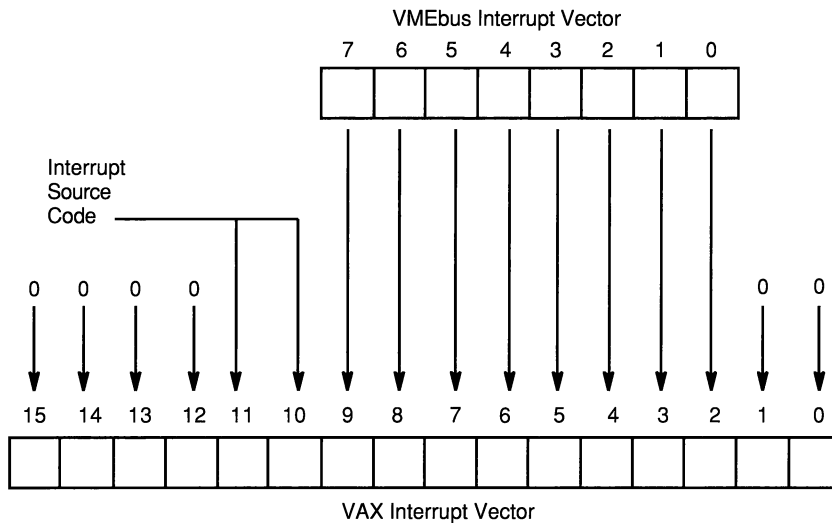


Table 2–1 describes the interrupt source codes. The KAV30 CPU uses these interrupt source codes to determine the source of an interrupt.

**Table 2–1 Interrupt Source Codes**

Bit 11	Bit 10	Type of Interrupt
0	0	UART interrupts
0	1	Local interrupts (including VMEbus and VSB autovectored interrupts)

(continued on next page)

**Table 2–1 (Cont.) Interrupt Source Codes**

Bit 11	Bit 10	Type of Interrupt
1	0	VMEbus vectored interrupts
1	1	Not used

The KAV30 CPU uses bits <2..15> of the VAX interrupt vector as a pointer to a longword in the system control block (SCB). This longword contains the address of the interrupt service routine for that interrupt. Table 2–2 describes the layout of the SCB.

**Table 2–2 System Control Block Layout**

Address Range	Contents
0000 <sub>16</sub> to 03FC <sub>16</sub>	Interrupt vectors for various system exceptions and software interrupts such as power failure, access violation, and so on
0400 <sub>16</sub> to 07FC <sub>16</sub>	KAV30 interrupt vectors
0800 <sub>16</sub> to 0BFC <sub>16</sub>	VMEbus interrupt vectors for vectored VMEbus interrupts
0C00 <sub>16</sub> to FFFC <sub>16</sub>	Not Used

The low-order two bits of each SCB vector determine the stack on which the interrupt is to be serviced. For all the KAV30, VMEbus, and VSB interrupts these two bits have the value 1, which means that the processor services the interrupts on the interrupt stack. The remaining bits contain the address of the ISR.

### 2.10.2 Handling Autovectored Interrupts

The KAV30 can handle autovectored interrupts on any of the seven VMEbus IRQ<1..7> lines. However, the KAV30 hardware can take longer to handle autovectored interrupts that it receives on VMEbus IRQ lines 1, 2, or 3. Using VMEbus IRQ lines 1, 2, or 3 for autovectored interrupts can take 20  $\mu$ s longer, before the appropriate ISR executes, and 40  $\mu$ s longer, after the ISR executes.

When another VMEbus module generates an autovectored interrupt on one of the VMEbus IRQ lines, the KAV30 hardware performs the following actions:

1. It interrupts the CPU.
2. The CPU requests an interrupt vector.

If the IRQ is on the VMEbus IRQ lines 1, 2, or 3, the KAV30 hardware sometimes performs the following actions:

- a. Generates a VMEbus IACK cycle.
  - b. Because the interrupting module is generating an autovectorized interrupt, it does not respond with an interrupt vector. The cycle times out after approximately 20  $\mu$ s.
  - c. The KAV30 hardware asserts the VAX ERR signal, which causes the CPU to start executing a passive release ISR.
  - d. Because the other VMEbus module still asserts the VMEbus IRQ line, the CPU again requests an interrupt vector. This request interrupts the process of starting the passive release ISR.
3. The KAV30 hardware returns an interrupt vector that was previously programmed into the hardware by the KAV30 kernel.
  4. The CPU starts executing the appropriate ISR. The CPU uses the interrupt vector to determine which ISR it executes.
  5. The ISR accesses the other VMEbus module and causes it to stop asserting the VMEbus IRQ line.
  6. The CPU finishes executing the ISR.
  7. When the KAV30 hardware performs steps A to D, it now causes the CPU to resume executing the passive release ISR. The passive release ISR writes an error log entry and finishes executing. This process takes approximately 40  $\mu$ s.

Because autovectorized interrupts received on the VMEbus IRQ lines 1, 2, and 3 can take 60  $\mu$ s longer to process by the KAV30 hardware, Digital recommends that you use the VMEbus IRQ lines 4, 5, 6, and 7 only for autovectorized interrupts.

When autovectorized interrupts occur, the KAV30 hardware gives the interrupt vector to the CPU. The KAV30 hardware was programmed by the KAV30 kernel with the vectors that the kernel gives in response to autovectorized IRQs. Table 2–3 lists these vectors. This table describes the condition that causes the interrupt and gives its offset (in hexadecimal) into the SCB.



**Table 2–3 SCB Vector Offsets for Autovectored ISRs**

Description	Offset into SCB
Autovectored VMEbus IRQ1	500 <sub>16</sub>
Autovectored VMEbus IRQ2	504 <sub>16</sub>
Autovectored VMEbus IRQ3	508 <sub>16</sub>
Autovectored VMEbus IRQ4	50C <sub>16</sub>
Autovectored VMEbus IRQ5	510 <sub>16</sub>
Autovectored VMEbus IRQ6	514 <sub>16</sub>
Autovectored VMEbus IRQ7	518 <sub>16</sub>
Autovectored VSB IRQ	54C <sub>16</sub>

**Note**

The SCB also contains vectors for SCSI IRQs and the VMEbus SYSFAIL signal. The vector for SCSI IRQs is at an offset of 550<sub>16</sub>. The vector for SCSI ERR interrupts is at an offset of 554<sub>16</sub>. The vector for the VMEbus SYSFAIL signal is at an offset of 540<sub>16</sub>.

See *The VMEbus Specification* for more information on VMEbus interrupt handler functionality.

**2.11 VMEbus Interrupt Requester Functionality**

This section describes the KAV30 VMEbus interrupt requester functionality. The KAV30 includes logic circuitry that implements VMEbus interrupt requester functionality. The VMEbus interrupt requester logic circuitry can generate vectored IRQs, which any module on the VMEbus, including the KAV30 VMEbus interrupt handler logic circuitry, can handle.

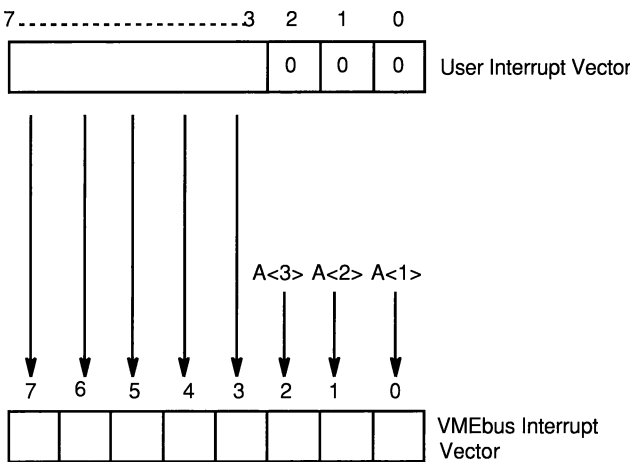
The KAV30 VMEbus interrupt requester logic circuitry can request interrupts, using VMEbus lines IRQ<1..7>. It places IRQs on these VMEbus IRQ lines according to the priority of the IRQ. The KAV30 VMEbus interrupt requester logic circuitry asserts an IRQ line until an interrupt handler acknowledges the request. When an interrupt handler acknowledges the request, the KAV30 stops asserting the IRQ lines and sends an interrupt vector to the interrupt handler.

When the KAV30 requests an interrupt it generates an 8-bit interrupt vector, which it places on the VMEbus. Figure 2–2 shows how the KAV30 CPU constructs the 8-bit VMEbus interrupt vector.

The VMEbus interrupt vector consists of the following data:

- Bits <2..0> contain VMEbus address lines A<3..1> respectively. These address lines represent the IRQ level being acknowledged. Table 2–4 explains the contents of these three bits.
- Bits <7..3> contain bits <8..4> of the interrupt vector that the user provides

Figure 2–2 Constructing an 8-bit VMEbus Interrupt Vector



See *The VMEbus Specification* for more information on VMEbus interrupt requester functionality.

Table 2–4 VMEbus Address Lines A<3..1>

Bit 3	Bit 2	Bit 1	VMEbus IRQ line
0	0	0	Not used
0	0	1	IRQ1
0	1	0	IRQ2
0	1	1	IRQ3

(continued on next page)

**Table 2–4 (Cont.) VMEbus Address Lines A<3..1>**

Bit 3	Bit 2	Bit1	VMEbus IRQ line
1	0	0	IRQ4
1	0	1	IRQ5
1	1	0	IRQ6
1	1	1	IRQ7

## 2.12 VSB Interrupt Handler Functionality

This section describes the KAV30 VSB interrupt handler functionality. The KAV30 includes logic circuitry that implements VSB interrupt handler functionality. The VSB interrupt handler logic circuitry can handle IRQs it receives from devices on the VSB. Although the VSB specification defines the handling of vectored as well as autovectored interrupts, the KAV30 VSB interrupt handler logic circuitry can handle only autovectored interrupts.

See the *VME Subsystem Bus (VSB) Specification* for more information on VSB interrupt handler functionality.

## 2.13 KAV30 Interrupt Priority

This section describes the KAV30 interrupt priority scheme. The KAV30 CPU receives IRQs on seven interrupt pins, as follows:

- The DAL bus error pin
- The VAX HALT pin
- The VAX POWER\_FAIL pin
- The VAX IRQ pins<3..0>

The KAV30 CPU assigns interrupt priority levels (IPLs) to these pins as shown in Table 2–5.

**Table 2–5 KAV30 Interrupt Pins**

Interrupt Pin	Interrupt Priority Level (Hexadecimal)
DAL bus error	1F (exception)
VAX HALT	1F
VAX POWER_FAIL	1E
VAX IRQ 3	17
VAX IRQ 2	16
VAX IRQ 1	15
VAX IRQ 0	14

When the KAV30 CPU receives more than one IRQ, it services the IRQs in the order of highest priority to lowest. The IRQs on the 1F (exception) level have the highest priority, IRQs on IPL 1F have the next highest priority, and so on to the IRQs on IPL 14, which have the lowest priority.

The KAV30 CPU also prioritizes interrupts within each IPL. Table 2–6 describes this priority scheme. In this table, the priority scheme is shown as follows:

- An en dash (–) prefixes sources that have an equal priority.
- A number prefixes sources that have an unequal priority. The magnitude of the number is inversely related to the priority of the source. That is, the number 1 prefixes the source with the highest priority.

## KAV30 Functionality

**Table 2–6 KAV30 Interrupt Priorities**

IPL	Interrupting Condition
IPL 1F:	<ul style="list-style-type: none"><li>– Occurrence of bus timeouts</li><li>– Occurrence of an IPL 1F control and status register (CSR) bit interrupting conditions, when the relevant CSR bit is configured to cause an interrupt on IPL 1F</li><li>– Issuing of the break command from a device connected to the KAV30 serial line ports</li><li>– Setting of the KAV30 reset/halt switch to the halt position</li><li>– Receiving a trigger boot message from a device on an Ethernet network</li></ul>
IPL 1E:	<ul style="list-style-type: none"><li>– Occurrence of an IPL 1E CSR bit interrupting conditions, when the relevant CSR bit is configured to cause an interrupt on IPL 1E</li><li>– Assertion of the VMEbus ACFAIL signal</li><li>– Receiving an autovector VMEbus IRQ on the IRQ7 line</li></ul>
IPL 17:	<ol style="list-style-type: none"><li>1. Occurrence of IPL 17 CSR bit interrupting conditions, including the following:<ul style="list-style-type: none"><li>– Receiving interrupts at IPL 17</li><li>– Occurrence of FIFO buffer 3 full and empty errors</li><li>– Occurrence of FIFO buffer 2 full and empty errors</li><li>– Receiving VMEbus autovector IRQs on the IRQ5 or IRQ6 line</li></ul></li><li>2. Receiving a vectored IRQ on the VMEbus IRQ5 and IRQ6 lines</li></ol>

(continued on next page)

**Table 2–6 (Cont.) KAV30 Interrupt Priorities**

<b>IPL</b>	<b>Interrupting Condition</b>
IPL 16:	<ol style="list-style-type: none"> <li>1. Occurrence of rtVAX 300 INTIM 10 milliseconds (ms) timer input</li> <li>2. Occurrence of IPL 16 CSR bit interrupting conditions, including the following: <ul style="list-style-type: none"> <li>– Receiving interrupts at IPL 16</li> <li>– Receiving SCSI interrupts</li> <li>– Receiving VSB interrupts</li> <li>– Receiving VMEbus autovectored IRQs on the IRQ3 or IRQ4 line</li> </ul> </li> <li>3. Receiving vectored IRQs on the VMEbus IRQ4 line</li> </ol>
IPL 15:	<ol style="list-style-type: none"> <li>1. Occurrence of Ethernet controller interrupts</li> <li>2. Occurrence of IPL 15 CSR bit interrupting conditions, including the following: <ul style="list-style-type: none"> <li>– Receiving interrupts at IPL 15</li> <li>– Occurrence of FIFO buffer 1 full and empty errors</li> <li>– Occurrence of FIFO buffer 0 full and empty errors</li> <li>– Receiving of VMEbus autovectored IRQs on the IRQ1 or IRQ2 line</li> </ul> </li> <li>3. Receiving of vectored IRQs on the VMEbus IRQ3 line</li> </ol>
IPL 14:	<ol style="list-style-type: none"> <li>1. Receiving of an interrupt from the KAV30 UART</li> <li>2. Occurrence of IPL 14 CSR bit interrupting conditions</li> <li>3. Receiving of vectored IRQs on the VMEbus IRQ1 and IRQ2 lines</li> </ol>



This chapter describes how the KAV30 hardware relates to the KAV30 software. It gives information on the following:

- Asynchronous system trap processing
- Timers
- Calendar/clock
- FIFO buffers
- Battery backed-up RAM
- Scatter-gather map
- Communicating with another KAV30
- KAV30 error logging support

### 3.1 Asynchronous System Trap Processing

This section describes how the KAV30 processes ASTs. In real-time systems, a process must be able to respond to events that occur asynchronously to the execution of the process. These events can result from actions by other processes in the system, by peripheral devices, or by the operating system itself. When a user program starts an event that can complete asynchronously (for example, an analog-to-digital conversion), it can specify the address of an AST routine. An **AST routine** is a procedure in the user program that the operating system calls when a particular event occurs.

The operating system maintains a queue of ASBs for each process. Each entry in the queue describes one requested AST and contains the address of the AST routine to be called when a specified event occurs.

If the user process is active when the system delivers an AST, the system interrupts the process and transfers control to the first AST routine in the queue. Each AST routine in the queue executes in turn. When the last AST routine in the queue returns, the user process resumes where it stopped.



## KAV30 Kernel

If the user process is inactive when the system delivers an AST, the process wakes up for the execution of the delivered ASTs. After the last AST is delivered the process returns to the original state.

When the programs specify an AST routine, they can also specify an associated argument called the AST parameter, which will be passed to the AST routine.

The KAV30 kernel queues ASTs to a process in the current access mode (user or kernel). The kernel mode ASTs have higher priority than the user mode ASTs, and the KAV30 kernel places them ahead of user mode ASTs in the queue.

### 3.1.1 AST Delivery

The VAXELN applications can include device drivers for the devices on the VMEbus or VSB that the KAV30 interacts with.

Each device driver that uses ASTs contains initialization code that calls the KAV\$DEF\_AST service to set up a queue called an AST queue. Each entry in the AST queue is a data structure called an ASB. The ASB contains information about the AST routine for a particular event relating to the device. (The KAV\$DEF\_AST service returns a code that the KAV\$CLR\_AST, KAV\$SET\_AST and KAV\$QUE\_AST services subsequently use to identify this ASB.)

Each device driver that uses ASTs also includes an input/output (I/O) section, in which the program calls the KAV\$SET\_AST routine to place data in an ASB. See Section 3.1.2 for information about ASBs and ASB queues.

After the KAV30 starts I/O, the relevant device driver returns control to the main program. When the device finishes performing I/O, it sends an IRQ signal to the KAV30.

If the IRQ is a vectored IRQ, the KAV30 processor sends an IACK signal to the device, which then sends an interrupt vector back to the module. The KAV30 processor uses this vector to specify an offset into the SCB. At the specified offset in the SCB, there is a longword vector that contains the address of an ISR.

If the IRQ is an autovectored IRQ, the KAV30 processor does not acknowledge the interrupt. Instead the KAV30 transfers control directly to the ISR whose address is contained in a predefined SCB vector. See Section 2.10 for more information.

Regardless of whether the IRQ is vectored or autovectored, the ISR executes at a device IPL.

There are 32 interrupt priority levels, in increasing order of priority from 0 to 31. The IPLs 16 to 31 are hardware IPLs, and the IPLs 0 to 15 are software IPLs. There are four device IPLs: IPL 20 to IPL 23. User processes execute at IPL 0, the lowest priority level. AST delivery executes at IPL 2, and AST routines execute at IPL 0.

The KAV30 kernel calls an ISR, which can include a call to the KAV\$QUE\_AST service (any ISRs executing at a higher IPL end before this ISR completes). The KAV30 kernel queues an AST to a process when the corresponding event occurs.

### 3.1.2 AST Data Structures

Several ASTs can be outstanding for a process at any time. The KAV30 kernel stores the ASTs in a FIFO queue. The entries in the queue are ASBs, which contain the following fields:

- **ASB type code**  
This field indicates whether the ASB is pending or free. The ASB is pending if it contains information about an AST that the KAV30 kernel has not yet delivered. The ASB is free if it does not contain information about an AST — this arises if the KAV\$SET\_AST service has not yet placed information in the ASB, or if the KAV30 kernel has delivered the AST. Using the KAV\$CLR\_AST service frees all the pending ASBs.  
The ASB type code field contains either the value ASB\$K\_ASBPEND or the value ASB\$K\_ASBFREE, which indicates whether the ASB is pending or free.
- **PCB address**  
This field contains the address of the PCB for the process that receives the AST.
- **AST address**  
This field contains the address of the AST routine.
- **AST parameter**  
This field contains an optional parameter to the AST routine. This parameter can specify a data value or the address of a block of data values.
- **AST flag**  
If this flag is set, the KAV30 kernel requeues the ASB to the AST pending queue for the device event immediately after delivery. If the flag is cleared, the KAV30 kernel clears the ASB after it delivers the AST.

Figure 3–1 ASB Fields

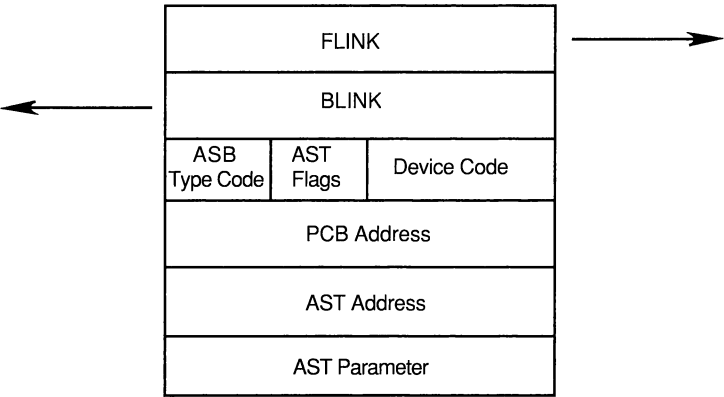
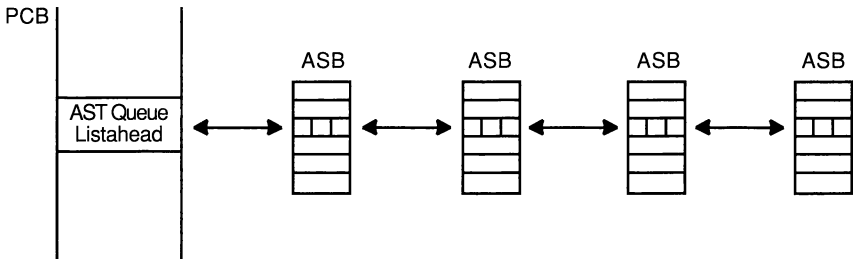


Figure 3–1 illustrates the fields in the ASB.

The head of the AST queue is located in the Process Control Block (PCB). The KAV30 kernel modifies the standard VAXELN PCB by adding two longword fields, PCB\$A\_ASTFLK and PCB\$A\_ASTBLK to the end of the PCB. These fields are pointers to the AST queue for that process, as shown in Figure 3–2.

See the *Introduction to VMS System Services* guide for more information. The *Introduction to VMS System Services* is part of the VMS programming documentation.

Figure 3–2 AST Queue



## 3.2 Timers

This section describes the KAV30 timers. There are five 32-bit timers and two 16-bit timers on the KAV30. The 32-bit timers are for general-purpose use. One 16-bit timer is a watchdog timer and the other is the local bus timeout timer.

Each timer operates as follows: you use the KAV\$TIMERS service to load a value into the timer register. The value then decrements on each clock cycle until it reaches zero. When the value reaches zero, the KAV30 delivers an AST. To calculate the value that you must load, divide the required timer interval by the clock period. The clock period is 400 nanoseconds (ns).

A 16-bit timer can time intervals up to  $(2^{16} - 1)$  clock cycles, so the maximum interval that you can use is as follows:

$$\begin{aligned} 2^{16} - 1 &= 65\,535 \\ \text{Maximum interval} &= 65\,535 \times 400 \text{ ns} \\ &= 26.214 \text{ ms} \end{aligned}$$

The 32-bit timers are made up of two 16-bit timers. The low-order word in the timer acts as a prescaler. Every time the prescaler decrements to zero, the high-order word decrements by one. The minimum value that you can specify for the prescaler is two. A 32-bit timer can time intervals up to  $(2^{16})$  times longer than a 16-bit timer can, that is, intervals of up to  $(2^{32} - 1)$  clock cycles, so the maximum interval that you can use is as follows:

$$\begin{aligned} 2^{32} - 1 &= 4\,294\,967\,295 \\ \text{Maximum interval} &= 4\,294\,967\,295 \times 400 \text{ ns} \\ &= 28.633 \text{ minutes}(\text{min}) \end{aligned}$$

A convenient way to program a timer is to load the value 2500 into the prescaler. Because the clock period is 400 ns, this value causes the prescaler to decrement to zero in 1 ms. Then, to time intervals in multiples of 1 ms, load the multiple into the high-order word. However, the timer loads the multiple into the high-order word after the prescaler first decrements to zero. Therefore, to ensure that the timer expires after the correct number of decrements, subtract one from the value that you load into the high-order word. For example, to set a timer to expire after 1 s, load the value 2500 into the prescaler and 1000 minus one into the high-order word. The value you load into the timer is as follows:

$$(65\,536 \times (1000 - 1)) + 2500 = 65\,472\,964 = 03E709C4_{16}$$

The KAV30 also has a watchdog timer and a local bus timeout timer. These are 16-bit timers. If the watchdog timer expires, a system reset occurs. The local bus timeout timer is used by the rtVAX 300 to monitor the DAL bus. Digital strongly recommends that you do not change the value of the local bus timeout timer, because this can lead to unpredictable results and VMEbus or VSB errors.

You can program the timers, using the KAV\$TIMERS service. See the description of the KAV\$TIMERS service for more information.

### 3.3 Calendar/clock

This section describes the calendar/clock on the KAV30. The calendar/clock maintains the time and date in units ranging from one-hundredth of a second to a year and leap year, as well as providing counters for the day of the week, day of the month, and day of the year. The calendar/clock keeps the time and date in binary coded decimal (BCD) format.

The calendar/clock has the following features:

- Alarm  
You can set the calendar/clock to interrupt the KAV30 at a specified time. You can also set it to interrupt after a specified interval.
- Timesave RAM  
The calendar/clock has a timesave area, in which it stores the contents of the clock in the event of a power failure.
- Twelve-hour and 24-hour clock  
The calendar/clock can operate in 12-hour mode or 24-hour mode. In 12-hour mode, you can specify A.M. and P.M.

- Julian date

The calendar/clock also provides the date in Julian format. The Julian date is the number of elapsed days in the year. For example, the Julian format for March 17, 1991 is 076 (because March 17 is the seventy-sixth day of the year). The Julian format for March 17, 1992 is 077 (because 1992 is a leap year and March 17 is the seventy-seventh day).

- Device RAM

The calendar/clock contains 31 bytes of general purpose RAM, which you can read from or write to using the KAV\$RTC system service. Either a battery or the VMEbus standby power supply backs up the device RAM.

- Timers

The calendar/clock has two 16-bit timers. The VMEbus uses one of these timers as a bus timeout timer. This timer controls the length of time within which the VMEbus must respond to an attempt by the KAV30 to gain control of the bus. Digital strongly recommends that you do not change the value of this timer because this can lead to unpredictable results.

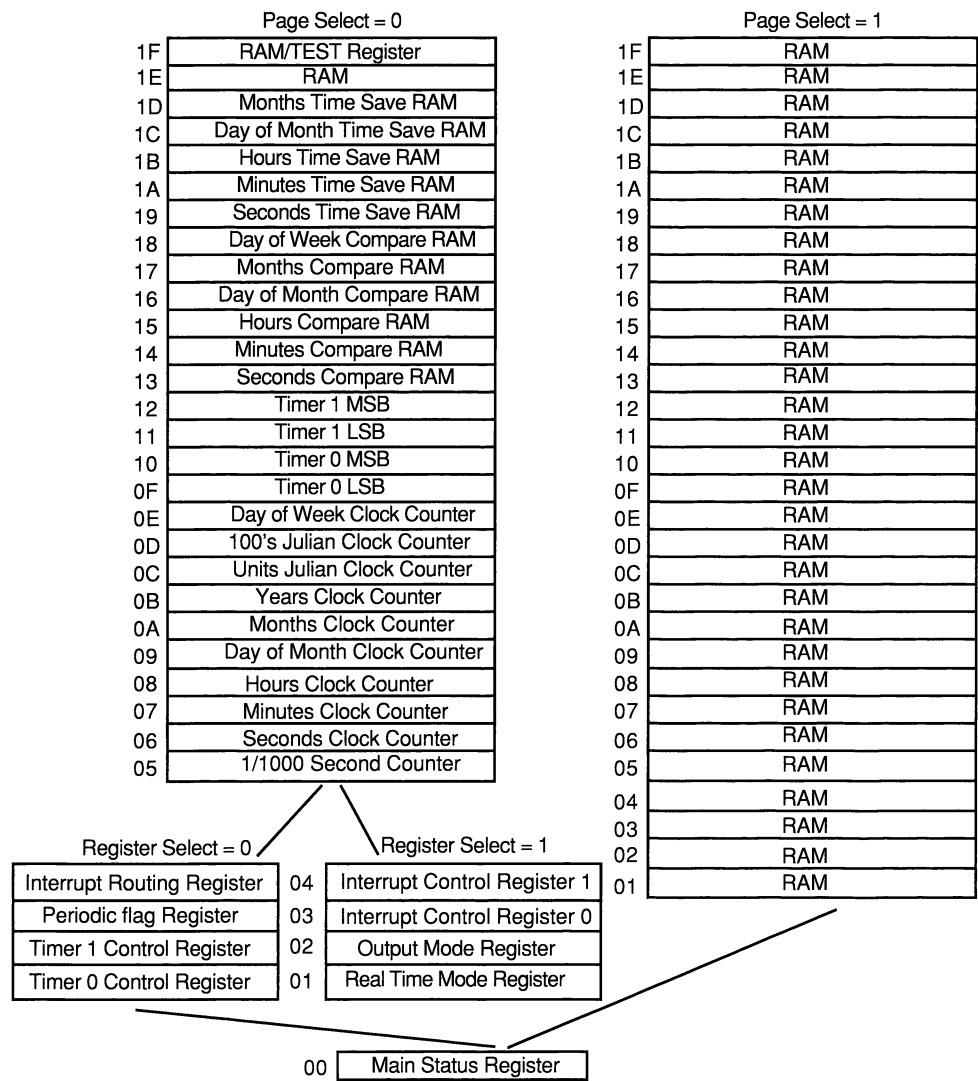
The second 16-bit timer is for general use. You can program this timer in the same way as the other KAV30 timers. See Section 3.2 for more information.

Figure 3–3 shows the address map of the calendar/clock. The first 31 bytes contain time and date information. The second 31 bytes are general-purpose RAM.

You can use the KAV\$RTC system service to carry out the following operations on the calendar/clock:

- Interrupt the KAV30 at a specified time (alarm)
- Interrupt the KAV30 when the interval you specify elapses (periodic alarm)
- Read the time at which the calendar/clock is set to interrupt the KAV30
- Set the time at which the calendar/clock interrupts the KAV30
- Read the calendar date
- Set the calendar date
- Read data from the calendar/clock RAM
- Write data to the calendar/clock RAM
- Read data from the calendar/clock timesave area

Figure 3–3 Calendar/clock Address Map



- Write data to the calendar/clock timesave area
- See the description of the KAV\$RTC service for more information.

## 3.4 FIFO Buffers

This section describes the FIFO buffers on the KAV30. The KAV30 contains four independently operating FIFO buffers. The purpose of the FIFO buffers is to enable an intelligent device on the VMEbus to exchange data with the KAV30. The VMEbus devices can write data into the FIFO buffers, and the KAV30 can then read the data from the buffers. Similarly, the KAV30 can write data into the FIFO buffers, and the VMEbus device can then read the data from the buffers.

Each FIFO buffer is organized into 255 longwords. However, it is possible to perform longword, quadword, and octaword operations on the FIFO buffers. When you perform quadword and octaword operations on the FIFO buffers, the FIFO logic circuitry writes or reads the message as an atomic collection of longwords.

Any device on the VMEbus (including the KAV30) can read from and write to the FIFO buffers in FIFO mode, or write to the buffers in LIFO mode.

A device that writes the data into the KAV30 FIFO buffers is called the **producer**. A device that reads the data from the buffers is called the **consumer**. The KAV30 can act as a producer or as a consumer, as shown in Figure 3–4. The KAV30 can also act neither as the producer or the consumer. In that case, two devices on the VMEbus act as producer and consumer, as shown in Figure 3–5.

The FIFO logic circuitry indicates an error when you read a message from an empty FIFO buffer, or write a message to a full FIFO buffer.

You can read/write the FIFO buffers, using the KAV\$FIFO\_READ, KAV\$FIFO\_WRITE, and KAV\$LIFO\_WRITE system services. See Chapter 4 for more information.

You can configure the KAV30 to notify you when a FIFO buffer changes its state under one or more of the following circumstances:

- When the state changes from empty to not-empty
- When the state changes from not-empty to empty
- When the state changes from not-empty to full

See the description of the KAV\$NOTIFY\_FIFO for more information.



### 3.5 Battery Backed-Up RAM

This section describes the battery backed-up RAM on the KAV30. The battery backed-up RAM allows you to store information that you want to protect in the event of a system or power failure. For example, you can write error messages to the battery backed-up RAM. If the system fails, the error information will still be in the RAM after the system is rebooted. You can use the error information to analyze the cause of the failure. However, you must install the battery jumper during the hardware installation to enable this functionality. See the *KAV30 Hardware Installation and User's Information* for more information.

Programs can use 22 of the 32K bytes of battery backed-up RAM. Of the remaining 10K bytes, 8K bytes are reserved for use by the KAV30 kernel and 2K bytes are reserved for future use by Digital.

You can read from and write to the battery backed-up RAM, using the KAV\$RW\_BBRAM service. See the description of the KAV\$RW\_BBRAM service for more information.

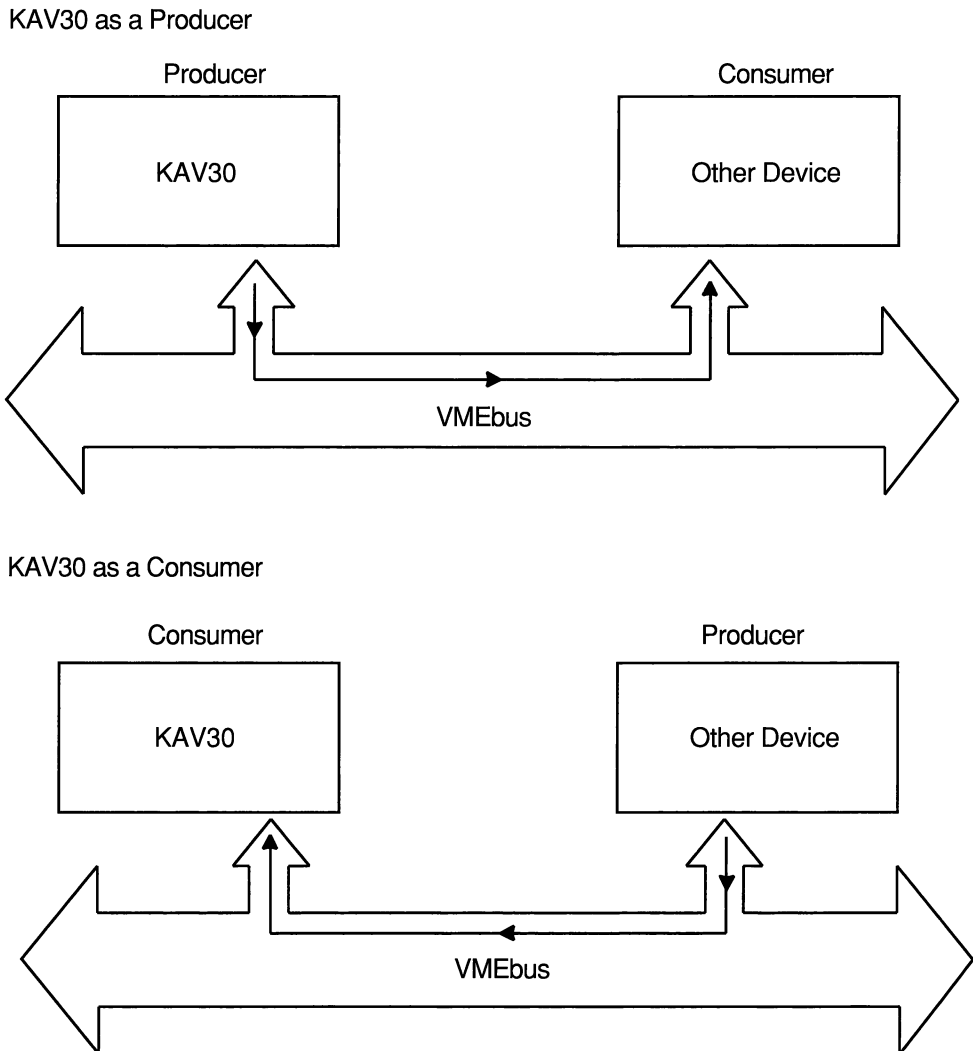
### 3.6 Scatter-Gather Map

This section describes the SGM. The SGM is the part of the KAV30 hardware that allows the devices on the VMEbus to access the KAV30 and allows the KAV30 to access the devices on the VMEbus or VSB. The SGM has two parts: the outgoing SGM and the incoming SGM.

The KAV30 uses the outgoing SGM while operating in master mode, that is, when the KAV30 accesses the devices on the VMEbus or VSB. It uses the incoming SGM while operating in slave mode, that is, when the devices on the VMEbus access the KAV30.

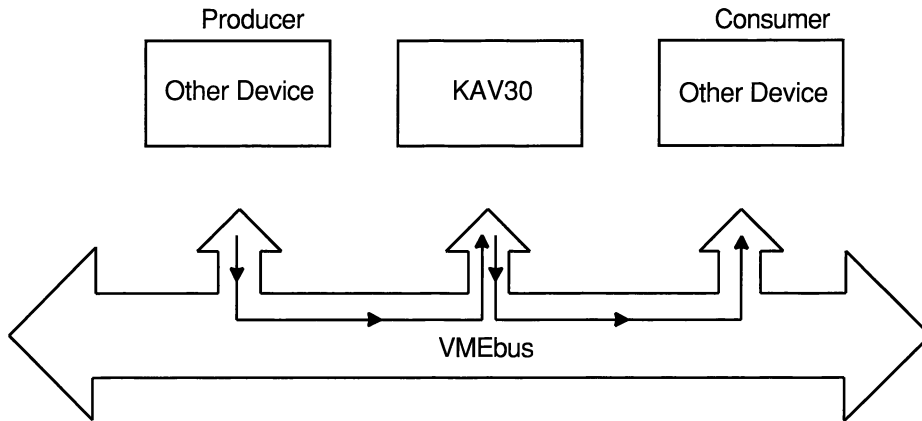
#### 3.6.1 Outgoing SGM

The outgoing SGM maps KAV30 system (S0) virtual address space to the address space of a target device on the VMEbus or VSB. This makes the address space of a target device visible to the KAV30, and enables the KAV30 kernel to access the target device address space.

**Figure 3–4 KAV30 as Producer and Consumer**

When you configure the KAV30, you pass the base address of the device on the VMEbus or VSB to the KAV\$OUT\_MAP kernel service. This service returns the KAV30 S0 space virtual address that corresponds to the base address of the VMEbus or VSB device. To write data to an offset in the address space of the

**Figure 3–5 KAV30 as Neither Producer or Consumer**



VMEbus or VSB device, add the offset to the virtual address returned by the KAV\$OUT\_MAP service, and write to the resulting address.

The outgoing SGM can map 230M bytes of KAV30 I/O space to the VMEbus or VSB address space. The size of the VMEbus or VSB address space depends on the type of addressing you use, as follows:

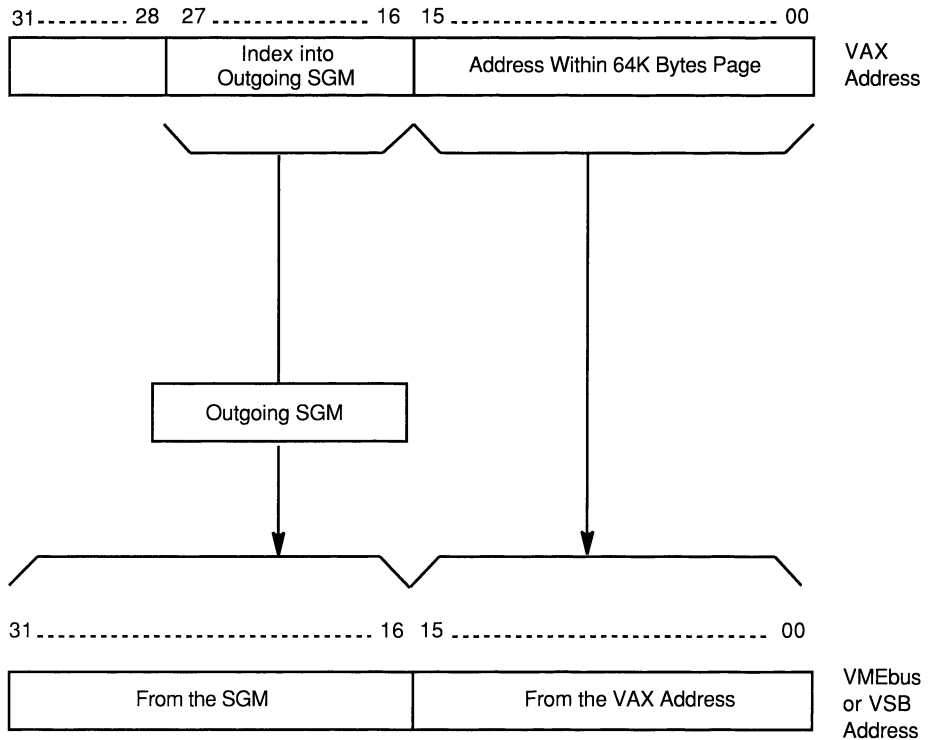
- When you use VMEbus or VSB A32 addressing you can access 4G bytes of address space
- When you use VMEbus or VSB A24 addressing you can access 14M bytes of address space
- When you use VMEbus or VSB A16 addressing you can access 64K bytes of address space

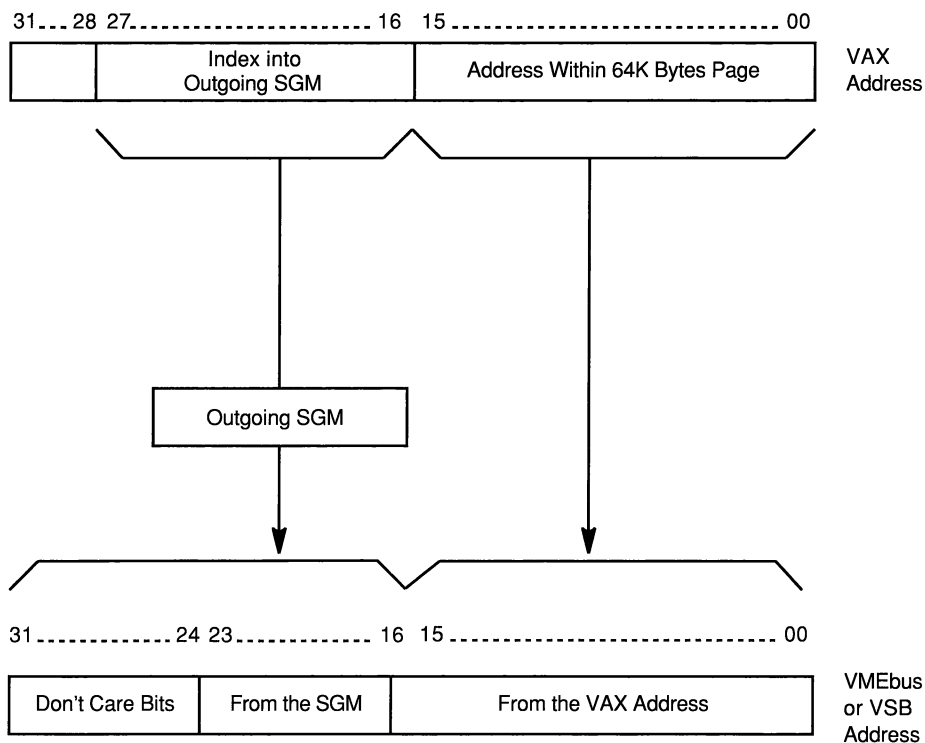
The outgoing SGM logic circuitry uses one SGM entry to map each 64K byte-page of KAV30 I/O space to the VMEbus or VSB address space. However, the first SGM entry for a device specifies the base VMEbus or VSB address of its memory-mapped I/O space. You can map up to 3584 64K byte-pages of KAV30 S0 space to the VMEbus or VSB address space. See the description of the KAV\$OUT\_MAP service for more information.

When the KAV30 sends a VAX address, bits <15..0> of the VAX address remain unchanged in the VMEbus or VSB address, and the outgoing SGM logic circuitry uses the remainder of the VAX address as an index into the outgoing SGM. Figure 3–6 illustrates the conversion of a VAX address into an A32 VMEbus or VSB address. Figure 3–7 illustrates the conversion of a

VAX address into an A24 VMEbus or VSB address. Figure 3–8 illustrates the conversion of a VAX address into an A16 VMEbus or VSB address.

**Figure 3–6 Outgoing SGM Conversion to VMEbus or VSB A32 Addresses**

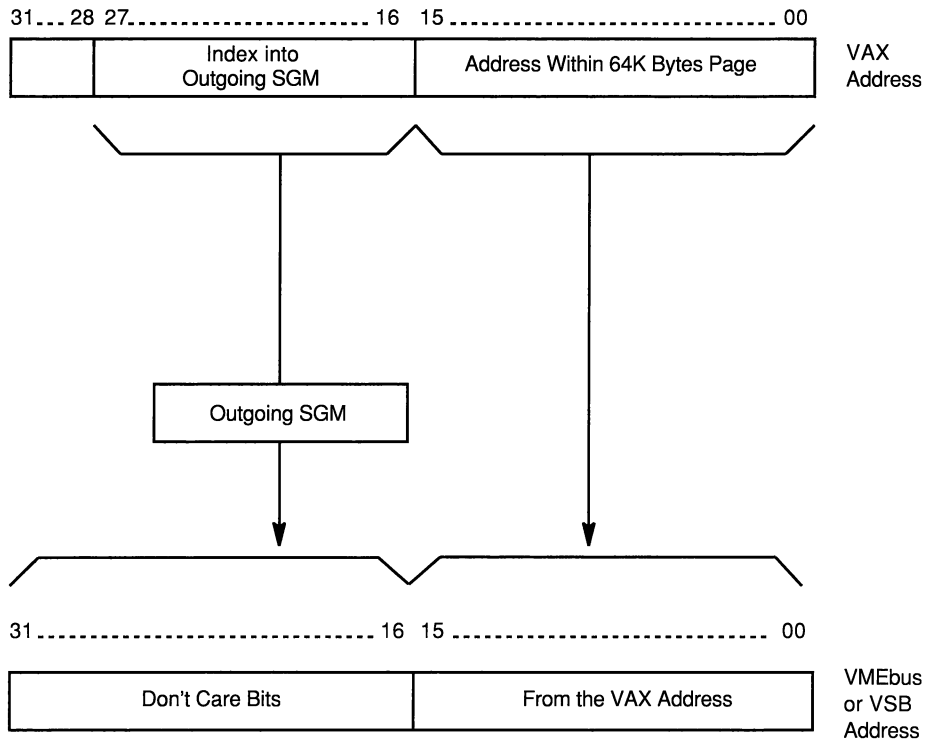


**Figure 3–7 Outgoing SGM Conversion to VMEbus or VSB A24 Addresses**

### 3.6.2 Incoming SGM

The incoming SGM maps one or more 64K byte-pages of VMEbus address space to the KAV30 process (P0) virtual address space. The size of the VMEbus address space depends on the size of the VMEbus address. The VMEbus A24 addresses can access up to 1M byte, or sixteen 64K byte-pages of address space. The VMEbus A32 addresses can access up to 4M bytes, or 256 64K byte-pages of address space.

To program the incoming SGM, call the KAV\$IN\_MAP service and specify an entry number in the incoming SGM. This entry number indicates the 64K byte-page of VMEbus address space that you want to map into KAV30 P0 space. The KAV\$IN\_MAP service returns the virtual address to which the incoming SGM maps the base address of the 64K byte-page of VMEbus address space.

**Figure 3–8 Outgoing SGM Conversion to VMEbus or VSB A16 Addresses**

When the KAV30 maps a page of VMEbus address space to a page of P0 space, it sets both the page boundaries where the low-order 16 bits are all 0.

The following list describes the VMEbus addresses for each type of addressing:

- A32 addressing

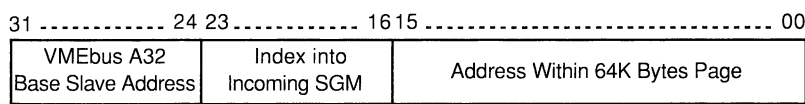
The VMEbus physical address for A32 addressing consists of the following:

- Bits 0 to 15 represent an address within a 64K byte-page
- Bits 16 to 23 represent an index into the incoming SGM
- Bits 24 to 31 are taken from the setting of the KAV30 VMEbus A32 base slave address register

The KAV30 base address specifies the part of the VMEbus address space allocated to the module. For A32 addressing, each device has 16M bytes of VMEbus address space. You can set bits 24 to 31 by calling the KAV\$VME\_SETUP service to set up the VMEbus system.

Figure 3–9 shows the A32 incoming VMEbus address.

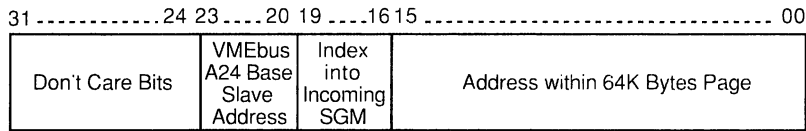
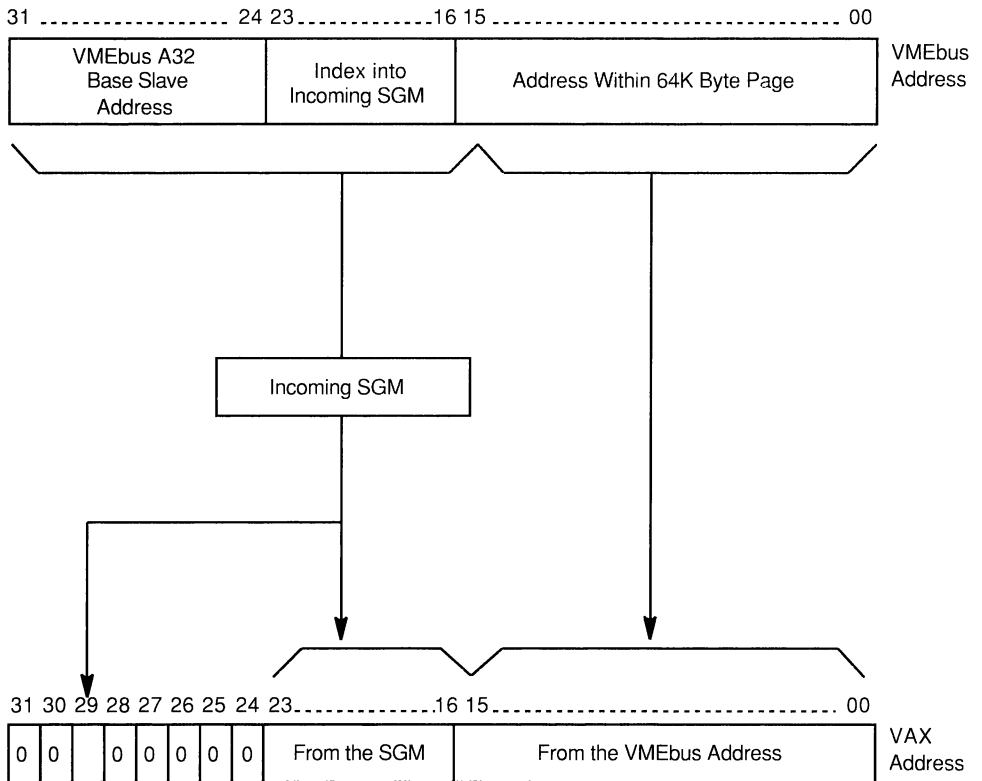
Figure 3–9 A32 Incoming VMEbus Address



- A24 addressing
- The VMEbus physical address for A24 addressing consists of the following:
- Bits 0 to 15 represent an address within a 64K byte-page
  - Bits 16 to 19 represent the 4 low-order bits of an SGM entry
- In A24 addressing, the incoming SGM can map up to sixteen 64K byte-pages of VMEbus address space into P0 space, so the SGM can have a maximum of 16 entries.
- Bits 20 to 23 are taken from the setting of the KAV30 rotary switch. This switch specifies the KAV30 VMEbus base slave address.
- The KAV30 base address specifies the part of the VMEbus address space allocated to the module. For A24 addressing mode, each device has 1M bytes of VMEbus address space. You can set bits 20 to 23 by using the KAV30 rotary switch. See the *KAV30 Hardware Installation and User's Information* for more information.
- Bits 24 to 31 are don't care bits.

Figure 3–10 shows the A24 incoming VMEbus address.

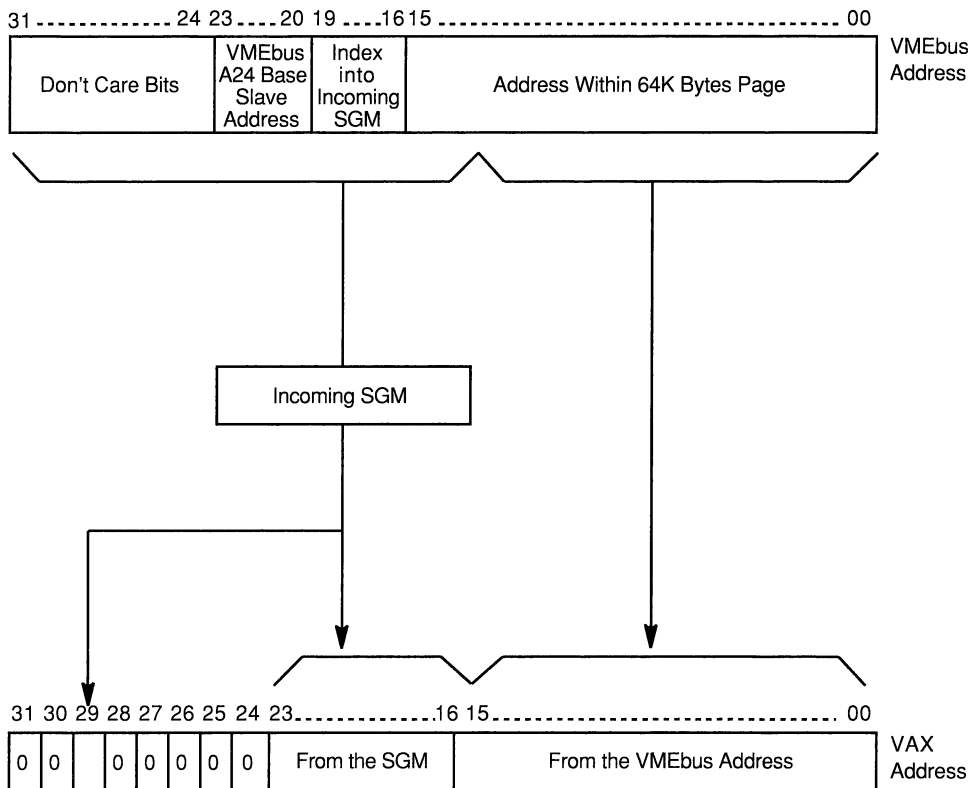
When the KAV30 receives a VMEbus address, the conversion process it uses differs depending on the type of VMEbus address that it receives. Figure 3–11 shows how it converts A32 VMEbus addresses. Figure 3–12 shows how it converts A24 VMEbus addresses.

**Figure 3–10 A24 Incoming VMEbus Address****Figure 3–11 Incoming SGM Conversion of A32 VMEbus Addresses**

The SGM can map the VMEbus address space into the KAV30 FIFO buffers instead of into P0 space. Another device on the VMEbus can then access the information in the FIFO buffers.



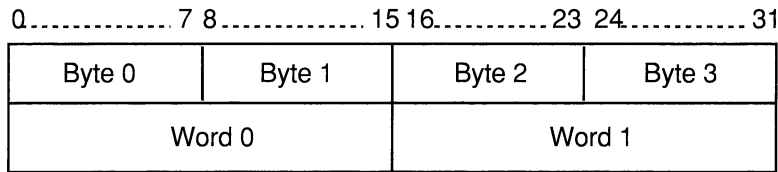
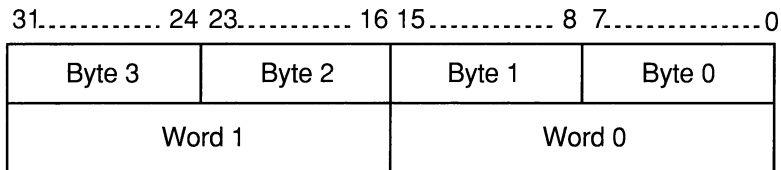
### Figure 3–12 Incoming SGM Conversion of A24 VMEbus Addresses



See the description of the KAV\$IN\_MAP service for more information.

### 3.6.3 Byte Swapping During SGM Operations

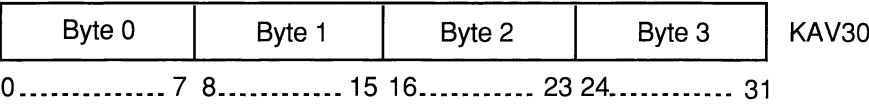
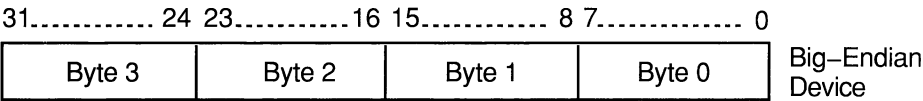
The KAV30 arranges the bytes in each longword it uses in little-endian format. However, VMEbus and VSB devices arrange the bytes in a longword in big-endian format. Therefore, when you want to transfer information between the KAV30 and a big-endian device, swap the bytes in the longword to preserve the order of the data. Figure 3-13 shows the little-endian data format. Figure 3-14 shows the big-endian data format.

**Figure 3–13 Little-Endian Storage Format****Figure 3–14 Big-Endian Storage Format**

The KAV30 can perform the following types of byte swapping:

- **Mode 0 swapping**  
Two terms are commonly used to describe mode 0 byte swapping: *no swap*, and *byte and word swap*. The term *no swap* refers to the relative position of the bytes in the two formats. That is, both devices process the data in the same order. For example, the byte at bit 0 in the big-endian format is the byte at bit 0 in the little-endian format. The term *byte and word swap* refers to the hardware operation that the byte swap logic circuitry performs on the data. That is, the byte swap logic swaps the low-order and high-order bytes in each word, and then swaps the low-order and high-order words in the longword. Figure 3–15 shows mode 0 byte swapping.
- **Mode 1 swapping**  
Mode 1 swapping is Digital reserved.

Figure 3–15 Mode 0 Byte Swapping



- Mode 2 swapping

---

**Note**

---

When you use mode 2 swapping, you can perform only word-aligned word accesses and longword-aligned longword accesses.

---

Mode 2 swapping is also referred to as *word swapping*. This is because the low-order and high-order words in the longword are swapped. Figure 3–16 shows mode 2 swapping.

- Mode 3 swapping

---

**Note**

---

When you use mode 3 swapping, you can perform only word-aligned word accesses and longword-aligned longword accesses.

---

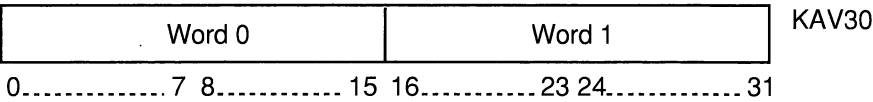
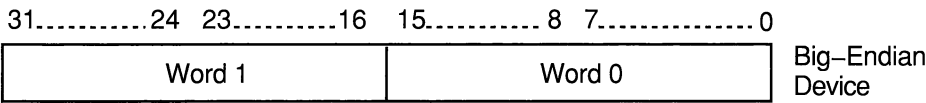
Two terms are commonly used to describe mode 3 byte swapping: byte and word swap, and no swap. The term *byte and word swap* refers to the relative position of the bytes in the two formats. That is, although the two formats receive identical data, big-endian devices process the data in a different manner than the little-endian devices. For example, the byte at bit 0 in the big-endian format is identical to the byte at bit 24 in the little-endian format. The term *no swap* refers to the hardware operation that the byte swap logic circuitry performs on the data. That is, the byte swap logic circuitry does not perform any hardware operation on the data. Figure 3–17 shows mode 3 byte swapping.

## 3.7 Communicating with Another KAV30

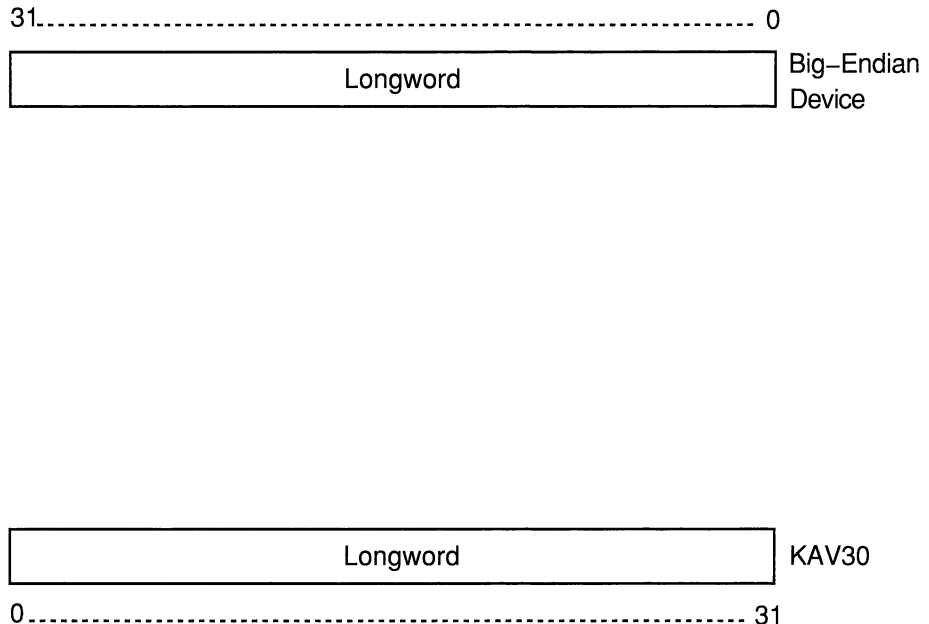
This section describes how two KAV30s can communicate with each other. A KAV30 can communicate with another KAV30 in two ways:

- By using shared memory pages to access the system RAM of the other KAV30

Figure 3–16 Mode 2 Byte Swapping



Two KAV30s in a VMEbus system can communicate using shared memory pages. The shared memory pages map physical pages from one of the modules into a virtual address space on both modules. The module whose physical page the shared memory page uses is the slave module. However, it is possible to configure the system so that the physical memory of both modules has a shared memory page.

**Figure 3–17 Mode 3 Byte Swapping**

The shared memory pages provide the fastest method for one KAV30 to pass information to another KAV30, because the modules have the data area mapped into their virtual address space.

When two KAV30s use shared memory pages to communicate, the application program must provide proper synchronization methods to ensure data integrity.

- By reading and writing the FIFO buffers on the other module
- Two KAV30s can also communicate using the KAV30 FIFO buffers. To write data to the FIFO buffer of another KAV30, the first module uses the KAV\$BUS\_WRITE kernel service with the KAV\$M\_FIFO\_ACCESS modifier to write the data to the VMEbus address space. The second module uses its incoming SGM to map the FIFOs onto the VMEbus address space. The second module then uses the KAV\$FIFO\_READ service to read the data in the FIFO buffers.

## KAV30 Kernel

The FIFO buffers method has the advantage that the sender does not have to wait for the receiver to process data before it sends more data to the receiver.

Communicating using the KAV30 FIFO buffers is preferable, except when you want to transfer large amounts of data. In this case, the shared memory pages method is preferable.

In VMEbus systems containing two KAV30s, ensure that you do not configure the modules at the same VMEbus base address.

---

### Note

---

When a VMEbus system contains two KAV30s with the same VMEbus base address, unpredictable results occur.

---

## 3.8 KAV30 Error Logging Support

This section describes the KAV30 error logging support. The KAV30 logs errors that you generate in its battery backed-up RAM. When you build error-logging into the system, the KAV30 also logs errors in the KAV30 error log file. The KAV30 kernel creates the error log file and sends it to the host system if a DECnet™ connection to the host is available.

The KAV30 logs the following errors:

- Bus errors and timeouts
- Invalid SGM entries
- SGM access violations

You can analyze the KAV30 error log file, using the VMS Error Log Utility. However, the reports that this utility generates are primarily intended to assist Digital Customer Services personnel. (See the *VAXELN Development Utilities Guide* for more information on VAXELN error logging.)

---

### Note

---

You might want to build two versions of a system: one with error logging support and the other without error logging support. Then, if problems arise, you can run the version with error logging to analyze the problem.

---

Error log-reports contain two sections: the identification section and the device-dependent data section. The identification section consists of the first four lines of the report. The device-dependent data section, which follows the identification section, contains information about the selected error log entries. Each line in this section gives a hexadecimal value and a short description of what the value signifies. See the *VMS Error Log Utility Manual* for more information.

There are two types of error log entry: error log entries for errors that occur while the KAV30 acts in master mode, and error log entries for errors that occur while the KAV30 acts in slave mode. Figure 3–18 shows a sample error log report entry for an error that occurred while the KAV30 was acting in master mode. Figure 3–19 shows a sample error log report entry for an error that occurred while the KAV30 was acting in slave mode.

**Figure 3–18 Sample Master Error Log Entry**

```
***** ENTRY          644. *****
ERROR SEQUENCE 20479.          LOGGED ON:          SID 0A000006
DATE/TIME 1-FEB-1991 21:30:31.53          SYS_TYPE 09100002
SCS NODE: KAV30E
❶

$SNDERR MESSAGE KA300 CPU REV# 7. FW REV# 1.0          ❷
MESSAGE TEXT
00007E0C Status Code ❸
3944000B VME/VSF Master AM & Error Code & Retry Count ❹
00F03038 VME/VSF Address Accessed ❺
8000B7C0 PC ❻
00C80009 PSL ❼

*****
```

**Figure 3–19 Sample Slave Error Log Entry**

```
***** ENTRY          24. *****
ERROR SEQUENCE 2.          LOGGED ON:          SID 0A000006
DATE/TIME 28-Jan-1991 15:15:30.87          SYS_TYPE 09100002
SCS NODE: KAV30D
❶

$SNDERR MESSAGE KA300 CPU REV# 7. FW REV# 1.0          ❷
```

(continued on next page)



**Figure 3–19 (Cont.) Sample Slave Error Log Entry**

```
MESSAGE TEXT
00007E3C   Status Code   ❸
00000084   VME Slave Error Status  ❹
00000001   VME Slave Error count  ❺
*****
```

The following list explains the information labeled with callouts in Figure 3–18 and Figure 3–19.

- ❶ The first four lines make up the identification section. See the *VMS Error Log Utility Manual* for a description of the information contained in the identification section.
- ❷ This line gives the following information:
  - The mechanism used to write the error log entry into the error log file
  - The CPU type
  - Hardware and firmware revision levels
- ❸ This line gives the KAV30 status code for the error. The status codes returned by KAV30 services are similar to those returned by VAXELN kernel procedures. All of these codes follow the VAX convention for status codes. A message is associated with each status code. You can use the `ELN$GET_STATUS_TEXT` VAXELN message-processing routine to retrieve the message text associated with a specified status code. An application can use these routines to retrieve a message text from the system message files or user-created message files. See the *VAXELN Runtime Facilities Guide* for more information.
- ❹ The contents of this line depend on whether the error occurred while the KAV30 was acting in master mode or slave mode:
  - If the error occurred while the KAV30 was acting in master mode, this line provides the following:
    - The VMEbus or VSB Address Modifier (AM) bits
    - The internal master error code
    - The number of times the KAV30 kernel retried to gain control of the VMEbus or VSB

The high-order byte of the status code provides the AM bits that the KAV30 was using when the error occurred.

The next byte gives the internal master error code. Table 3–1 explains the bits of the internal master error code.

**Table 3–1 Internal Master Error Code**

Bit	Value	Meaning
0	0	The SGM entry for the VMEbus or VSB address is valid.
	1	The SGM entry for the VMEbus or VSB address is invalid.
1	0	The SGM entry for the VMEbus or VSB address is not write protected.
	1	The SGM entry for the VMEbus or VSB address is write protected.
2	0	The KAV30 local bus timeout timer has not expired.
	1	The KAV30 local bus timeout timer has expired.
3	0	The KAV30 was trying to gain control of the VMEbus when the error occurred.
	1	The KAV30 was trying to gain control of the VSB when the error occurred.
4	0	The KAV30 did not access one of the FIFO buffers on the module when the error occurred.
	1	The KAV30 accessed one of the FIFO buffers on the module when the error occurred.
5	0	The VMEbus IACK cycle did not fail.
	1	The VMEbus IACK cycle failed.
6	0	The KAV30 did not have control of the VMEbus or VSB when the error occurred.
	1	The KAV30 had control of the VMEbus or VSB when the error occurred.
7	0	The KAV30 was performing a write operation when the error occurred.
	1	The KAV30 was performing a read operation when the error occurred.

The low-order word contains the number of times the KAV30 kernel retried to gain control of the bus (this does include the retries performed by the KAV30 hardware). In this sample master error log entry the KAV30 kernel performed eight software retries.

- If the error occurred while the KAV30 was acting in slave mode, this line gives the internal slave error code. Table 3–2 explains the bits of the internal slave error code.

**Table 3–2 Internal Slave Error Code**

Bit	Value	Meaning
0	0	The SGM entry for the VMEbus address is valid.
	1	The SGM entry for the VMEbus address is invalid.
1	0	The SGM entry for the VMEbus address is not write protected.
	1	The SGM entry for the VMEbus address is write protected.
2	0	The KAV30 local bus timeout timer has not expired.
	1	The KAV30 local bus timeout timer has expired.
3	0	The KAV30 was accessing system RAM when the error occurred.
	1	The KAV30 was accessing the FIFO buffers when the error occurred.
4	0	When the bit 3 has the value 1, the KAV30 was accessing a FIFO port when the error occurred.
	1	When the bit 3 has the value 1, the KAV30 was accessing FIFO memory when the error occurred.
5	0	The KAV30 was performing a write operation when the error occurred.
	1	The KAV30 was performing a read operation when the error occurred.
6		When bit 3 has the value 1, bit 6 contains the value of the least significant bit of the FIFO port number that you were accessing when the error occurred.
7		When bit 3 has the value 1, bit 7 contains the value of the most significant bit of the FIFO port number that you were accessing when the error occurred.

- ⑤ The contents of this line depend on whether the error occurred while the KAV30 was acting in master mode or slave mode:
- If the error occurred while the KAV30 was acting in master mode, this line gives the VMEbus or VSB address that the KAV30 tried to access. In this case, the address was 00F0 3038<sub>16</sub>. The address is always longword aligned.
  - If the error occurred while the KAV30 was acting in slave mode, this line gives the address being accessed when the error occurred.
- ⑥ This line is displayed in the error log entry only when the error occurred while the KAV30 was acting in master mode. This line gives the value of the Program Counter (PC) when the error occurred. If the KAV30 was trying to perform a read operation, the PC points to the failing instruction. If the module was trying to perform a write operation by accessing the VMEbus or VSB directly rather than using the KAV\$BUS\_

READ or KAV\$BUS\_WRITE kernel services, this PC is of no value. This is because the rtVAX 300 processor performs disconnected write operations (see Section 5.1.1 for more information).

- ⑦ This line is displayed in the error log entry only when the error occurred while the KAV30 was acting in master mode. This line lists the value of the processor status longword when the error occurred.



---

## KAV30 System Services

This chapter describes the KAV30 system services. Each service description has the following format:

- An overview of the service
- The call format for the service in each supported language
- A list of the arguments for the service
- A list of the status values returned by the service
- A list of related services
- Examples

---

## KAV\$BUS\_BITCLR

Clears the bits at a specified VMEbus or VSB address according to a specified bit mask.

---

### Note

---

This service performs read-modify-write cycles on the VMEbus or VSB.

---

The VMEbus or VSB address that you specify must be mapped to the KAV30 system virtual address (S0) space via the incoming SGM.

You can clear the bits in a byte, a word, or a longword. However, the VMEbus or VSB device containing the bits that you want to clear must allow the type of access that you specify. Also, when you want to clear the bits in a word or a longword, ensure that the address you specify is aligned to a word or longword.

The bit mask indicates the bits that you want to clear. For each bit that is set in the bit mask, this service clears the corresponding bit at the VMEbus or VSB address.

Calls to this service can result in VMEbus or VSB errors such as arbitration failures and bus timeouts. When the KAV\$M\_NO\_RETRY flag in the KAV\$OUT\_MAP service is not set, the KAV30 retries 29 times to clear the bits. However, the KAV30 kernel can direct the module to perform additional retries—you can specify this when you build the system. See Section 5.4 for more information.

## Ada Call Format

```
WITH KAVDEF;

KAV_BUS_BITCLR ([STATUS => status,]
                DATA_TYPE => data_type,
                VIRTUAL_ADDRESS => virtual_address,
                MASK => mask);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>data_type</i> :	in	INTEGER;
<i>virtual_address</i> :	in	SYSTEM.ADDRESS;
<i>mask</i> :	in	INTEGER;

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$bus_bitclr ([status],
                   data_type,
                   virtual_address,
                   mask)
```

### argument information

int	<i>*status</i> ;
int	<i>data_type</i> ;
void	<i>*virtual_address</i> ;
int	<i>mask</i> ;



---

**FORTRAN Call Format**

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$BUS_BITCLR ([status],
                    %VAL(data_type),
                    virtual_address,
                    %VAL(mask))
```

**argument information**

INTEGER*4	<i>status</i>
INTEGER*4	<i>data_type</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>mask</i>

---

**Pascal Call Format**

```
INCLUDE $KAVDEF;

KAV$BUS_BITCLR ([STATUS := status,]
               data_type,
               virtual_address,
               mask)
```

**argument information**

<i>status</i> :	INTEGER;
<i>data_type</i> :	INTEGER;
<i>virtual_address</i> :	^ANYTYPE;
<i>mask</i> :	INTEGER;

---

## Arguments

***status***

Usage: Longword (unsigned)  
 VAX Type: cond\_value  
 Access: Write only  
 Mechanism: Reference

Receives the completion status.

***data\_type***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies whether this service clears the bits in a byte, a word, or a longword. Specify one of the following values:

KAV\$K_BYTE	Clears the bits in a byte
KAV\$K_LONGWORD	Clears the bits in a longword
KAV\$K_WORD	Clears the bits in a word

The VMEbus or VSB device containing the bits that you want to clear must allow accesses of the type that you specify.

***virtual\_address***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the virtual address, in S0 address space, of the bits that you want to clear.

To calculate the VMEbus or VSB address, follow these steps:

1. Calculate the base address of the VMEbus or VSB device containing the bits that you want to clear. The KAV\$OUT\_MAP service returns the base VMEbus or VSB address of a device. See the description of the KAV\$OUT\_MAP service for more information.
2. Calculate the offset, into the device's address space, of the bits that you want to clear.

# KAV\$BUS\_BITCLR

3. Add the base address to the offset.

**mask**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the bit mask that indicates the bits to clear. The bit mask is the type that you specify in the *data\_type* argument. For example, if you specify KAV\$K\_WORD for the *data\_type* argument, the bit mask is in the low-order 16 bits of the *mask* address.

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_BUS_ARB_ERROR	A VMEbus or VSB arbitration failure occurred.
KAV30\$_BUS_RD_ERROR	A VMEbus or VSB read error occurred.
KAV30\$_BUS_WRT_ERROR	A VMEbus or a VSB write error occurred.
KAV30\$_INVALID_SG_ENTRY	You specified an invalid SGM entry.
KAV30\$_NO_BUS_RD_RESP	There was no read response from the device.
KAV30\$_NO_BUS_WRT_RESP	There was no write response from the device.
KAV30\$_WRPROT_SG_ENTRY	You attempted to write to a write-protected SGM entry.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service could not access an item.
KER\$_SUCCESS	The operation was successful.

---

## Related Services

KAV\$BUS\_BITSET

KAV\$BUS\_WRITE

KAV\$BUS\_READ

---

## Examples

The following code is an example program that calls the KAV\$BUS\_BITCLR service:

```

/*
 * Facility:      KAV30 VAXELN System Services programming example.
 *
 * Description:   This is an example program demonstrating the calling
 *                procedures for the following KAV System Services:
 *                1. KAV$VME_SETUP      (Configure VMEbus interrupting,...)
 *                2. KAV$OUT_MAP        (Map onto VMEbus address)
 *                3. KAV$BUS_BITSET     (Set a bit at a VMEbus address)
 *                4. KAV$BUS_BITCLR     (Clear ....)
 *
 * Abstract:      This program assumes that there is a VME-bus device located
 *                on the bus, at the address specified below.
 *                It performs a bit-clear and a bit-set on two of this
 *                device's registers.
 *
 * Language:      Vax C; Version 3.1
 *
 * Notes: (1)     If there is no device located on the VME-bus at the
 *                specified address, then the KAV System Service routines
 *                will return errors (in the 'status' variable).
 *                (2) The device is assumed to be set up for 24-bit addressing.
 *                (3) In the interests of program clarity, no error checking has
 *                been included.
 */

#include  stdio
#include  $vaxelnc
#include  <eln$:kavdef.h>      /* KAV30 definitions file.      */

```

## KAV\$BUS\_BITCLR

```
/*
 * The following definitions are device-specific.
 */
#define DEVICE_ADDRESS 0xfe0000 /* Base address of the device. */
#define REGISTER_BASE 0xe00 /* The rotary switch on the KAV */
/* module must be set up to */
/* agree with this addressing. */
#define REGISTER1_OFFSET 0x01 ; /* Offset of first register. */
#define REGISTER2_OFFSET 0x03 ; /* Offset of second register. */

main()
{
    unsigned long    am_code,
                    setup_functions,
                    buffer,
                    entry,
                    vir_addr,
                    phys_addr ;
    unsigned long    *register1_address, /* Note these are address POINTERS. */
                    *register2_address ;
    int              status,
                    page_count,
                    map_functions ;
    unsigned char    bit0 = 0x01, /* Bit-0 (least significant bit in byte). */
                    bit7 = 0x80 ; /* Bit-7 (most .....). */

    /*
     * Setup the VME functions to disable the VME-device from interrupting.
     * =====
    */
    buffer          = 0x00000000; /* No IRQ allowed by VME-device */
    setup_functions = KAV$K_ALLOW_VME_IRQ;
    KAV$VME_SETUP( &status, setup_functions, &buffer );

    /*
     * Map into the device register region.
     * =====
    */
    page_count      = 1 ; /* Number of 64K pages. */
    phys_addr       = DEVICE_ADDRESS ;
    am_code         = KAV$K_USER_24 ; /* Standard User Mode addressing.*/
    map_functions    = KAV$M_VME + KAV$M_MODE_0_SWAP; /* Byte/Word Swapping. */
    KAV$OUT_MAP( &status, &entry, page_count,
                phys_addr, &vir_addr,
                am_code, map_functions);

    /*
     * Setup the register pointers (virtual)
     */
    register1_address = vir_addr + REGISTER_BASE + REGISTER1_OFFSET ;
    register2_address = vir_addr + REGISTER_BASE + REGISTER2_OFFSET ;
}
```

```
/* (we are now able to access the device's registers)
*
*   SET bit-0 in first register.
*   =====
*/
KAV$BUS_BITSET( &status,
                KAV$K_BYTE,
                register1_address,
                bit0 ) ;

/*
*   CLEAR bit-0 in second register.
*   =====
*/
KAV$BUS_BITCLR( &status,
                KAV$K_BYTE,
                register2_address,
                bit7 ) ;

} /*   end   -program-   */
```

---

## KAV\$BUS\_BITSET

Sets the bits at a specified VMEbus or VSB address according to a specified bit mask.

---

### Note

---

This service performs read-modify-write cycles on the VMEbus or VSB.

---

The VMEbus or VSB address that you specify must be mapped to the KAV30 system virtual address (S0) space via the incoming SGM.

You can set the bits in a byte, a word, or a longword. However, the VMEbus or VSB device containing the bits that you want to set must allow the type of access that you specify. Also, when you want to set the bits in a word or a longword, ensure that the address you specify is aligned to a word or longword.

The bit mask indicates the bits that you want to clear. For each bit that is set in the bit mask, the service sets the corresponding bit in the VMEbus or VSB address.

The calls to this service can result in VMEbus or VSB errors such as arbitration failures and bus timeouts. When the KAV\$M\_NO\_RETRY flag in the KAV\$OUT\_MAP service is not set, the KAV30 retries 29 times to set the bits. However, the KAV30 kernel can direct the module to perform additional retries — you can specify this when you build the system. See Section 5.4 for more information.

---

## Ada Call Format

```
WITH KAVDEF;

KAV_BUS_BITSET ([STATUS => status,]
                 DATA_TYPE => data_type,
                 VIRTUAL_ADDRESS => virtual_address,
                 MASK => mask);
```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>data_type</i> :	in	INTEGER;
<i>virtual_address</i> :	in	SYSTEM.ADDRESS;
<i>mask</i> :	in	INTEGER;

---

## C Call Format

```
#include $vaxelnc

#include "eln$:kavdef.h"

int kav$bus_bitset ([status],
                   data_type,
                   virtual_address,
                   mask)
```

## argument information

int	<i>*status</i> ;
int	<i>data_type</i> ;
void	<i>*virtual_address</i> ;
int	<i>mask</i> ;



# KAV\$BUS\_BITSET

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$BUS_BITSET ([status],  
                     %VAL(data_type),  
                     virtual_address,  
                     %VAL(mask))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>data_type</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>mask</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$BUS_BITSET ([STATUS := status,]  
               data_type,  
               virtual_address,  
               mask)
```

### argument information

<i>status</i> :	INTEGER;
<i>data_type</i> :	INTEGER;
<i>virtual_address</i> :	^ANYTYPE;
<i>mask</i> :	INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

### ***data\_type***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies whether this service sets the bits in a byte, a word, or a longword. Specify one of the following values:

KAV\$K_BYTE	Sets the bits in a byte
KAV\$K_LONGWORD	Sets the bits in a longword
KAV\$K_WORD	Sets the bits in a word

The VMEbus or VSB device containing the bits that you want to set must allow the type of access that you specify.

### ***virtual\_address***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the virtual address, in S0 address space, of the bits that you want to set.

To calculate the VMEbus or VSB address, follow these steps:

1. Calculate the base address of the VMEbus or VSB device containing the bits that you want to set. The KAV\$OUT\_MAP service returns the base VMEbus or VSB address of a device. See the description of the KAV\$OUT\_MAP service for more information.
2. Calculate the offset into the device's address space of the bits that you want to clear.

# KAV\$BUS\_BITSET

3. Add the base address to the offset.

**mask**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the bit mask that indicates the bits to set. The bit mask is of the type that you specify in the *data\_type* argument. For example, if you specify KAV\$K\_WORD for the *data\_type* argument, the bit mask is in the low-order 16 bits of the *mask* address.

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_BUS_ARB_ERROR	A VMEbus or VSB arbitration failure occurred.
KAV30\$_BUS_RD_ERROR	A VMEbus or VSB read error occurred.
KAV30\$_BUS_WRT_ERROR	A VMEbus or a VSB write error occurred.
KAV30\$_INVALID_SG_ENTRY	You specified an invalid SGM entry.
KAV30\$_NO_BUS_RD_RESP	There was no read response from the device.
KAV30\$_NO_BUS_WRT_RESP	There was no write response from the device.
KAV30\$_WRPROT_SG_ENTRY	You tried to write to a write-protected SGM entry.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service could not access an item.
KER\$_SUCCESS	The operation was successful.

---

**Related Services**

KAV\$BUS\_BITCLR  
KAV\$BUS\_READ

KAV\$BUS\_WRITE

---

**Examples**

See the examples in the description of the KAV\$BUS\_BITCLR service.

---

## KAV\$BUS\_READ

Reads the contents of a VMEbus or VSB address. The VMEbus or VSB address you read must be mapped to the KAV30 system virtual address (S0) space via the outgoing SGM.

You can read data in byte, word, or longword format. However, ensure that the VMEbus or VSB device from which you read data allows the type of access you specify.

When you call this service to read data from a FIFO buffer on another VMEbus or VSB device, specify the KAV\$M\_FIFO\_ACCESS and KAV\$K\_LONGWORD values in the *data\_type* argument. This causes the KAV\$BUS\_READ service to read data from the same VMEbus or VSB address each time. If you do not specify the value KAV\$M\_FIFO\_ACCESS, the KAV\$BUS\_READ service increments the address after each read operation, as follows:

- When the service reads data in byte format, it increments the address by 1 byte.
- When the service reads data in word format, it increments by 2 bytes.
- When it reads data in longword format, it increments by 4 bytes.

---

### Note

---

Digital recommends that you exchange data by directly accessing the VMEbus and VSB, rather than by calling the KAV\$BUS\_READ service. See Section 5.1.1 for more information.

---

---

## Ada Call Format

```
WITH KAVDEF;  
KAV_BUS_READ ([STATUS => status,  
              DATA_TYPE => data_type,  
              VIRTUAL_ADDRESS => virtual_address,  
              BUFFER => buffer,  
              COUNT => count);
```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>data_type</i> :	in	INTEGER;
<i>virtual_address</i> :	in	SYSTEM.ADDRESS;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>count</i> :	in	INTEGER;

---

## C Call Format

```
#include $vaxelnc  
#include "eln$:kavdef.h"  
  
int kav$bus_read ([status],  
                 data_type,  
                 virtual_address,  
                 buffer,  
                 count)
```

## KAV\$BUS\_READ

### argument information

int	<i>*status;</i>
int	<i>data_type;</i>
void	<i>*virtual_address;</i>
int	<i>*buffer;</i>
int	<i>count;</i>

---

### FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
  
CALL KAV$BUS_READ ([status],  
                   %VAL(data_type),  
                   virtual_address,  
                   buffer,  
                   %VAL(count))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>data_type</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>count</i>

---

### Pascal Call Format

```
INCLUDE $KAVDEF;  
  
KAV$BUS_READ ([STATUS := status,]  
              data_type,  
              virtual_address,  
              buffer,  
              count)
```

## argument information

*status* : INTEGER;  
*data\_type* : INTEGER;  
*virtual\_address* : ^ANYTYPE;  
*buffer* : ^ANYTYPE;  
*count* : INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
 VAX Type: cond\_value  
 Access: Write only  
 Mechanism: Reference

Receives the completion status.

### ***data\_type***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the format of the data you want to read.

Specify one of the following values:

KAV\$K_BYTE	Reads data in byte format
KAV\$K_LONGWORD	Reads data in longword format
KAV\$K_WORD	Reads data in word format
KAV\$M_FIFO_ACCESS	Reads data from a FIFO buffer

The VMEbus or VSB device that you want to read from must allow the type of access you specify.



KAV\$BUS\_READ

Note

When you want to read data from a FIFO buffer on another VMEbus or VSB device, specify the KAV\$M\_FIFO\_ACCESS and KAV\$K\_LONGWORD modifiers.

*virtual\_address*

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the virtual address, in S0 space, where you want to begin reading data. To calculate the address, follow these steps:

1. Calculate the base address of the VMEbus or VSB device containing the bits that you want to read from. The KAV\$OUT\_MAP service returns the base VMEbus or VSB address of a device. See the description of the KAV\$OUT\_MAP service for more information.
2. Calculate the offset, into the device's address space, that you want to read from.
3. Add the base address to the offset.

When you read data from a FIFO buffer on another KAV30, use the following offsets when calculating the address:

FIFO Buffer	Offset
0	4000 <sub>16</sub>
1	4010 <sub>16</sub>
2	4020 <sub>16</sub>
3	4030 <sub>16</sub>

*buffer*

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Specifies the address of a buffer into which the KAV\$BUS\_READ service places the data that it reads from the VMEbus or VSB device.

**count**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of data items (of the type specified by the *data\_type* argument) that the KAV\$BUS\_READ service reads from the VMEbus or VSB device.

---

**Status Values**

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_BUS_RD_ERROR	A VMEbus or VSB read error occurred.
KAV30\$_BUS_ARB_ERROR	A VMEbus or VSB arbitration failure occurred.
KAV30\$_INVALID_SG_ENTRY	You specified an invalid SGM entry.
KAV30\$_NO_BUS_RD_RESP	There was no read response from the device.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The services could not access an item.
KER\$_SUCCESS	The operation was successful.

---

**Related Services**

KAV\$BUS_BITCLR	KAV\$OUT_MAP
KAV\$BUS_BITSET	KAV\$BUS_WRITE

### Examples

- The following code is an example KAV\$BUS\_READ call:

```
vir_address = virtual_address + '3603'X
3020 CALL KAV$BUS_READ (status,
1 %VAL(data_type),
2 %VAL(vir_address),
3 buffer,
4 %VAL(count))
IF ( .NOT. status ) TYPE 3030, status
3030 FORMAT(1H, 'KAV$BUS_READ, status is :', Z4.4 )
```

- The file SYS\$COMMON:[SYSHLP.EXAMPLES.KAV]KAV\_MVME.FOR contains a program that calls the KAV\$BUS\_READ service

---

## KAV\$BUS\_WRITE

Writes data to a VMEbus or VSB address. The VMEbus or VSB address you write to must be mapped to the KAV30 system virtual address (S0) space via the outgoing SGM.

You can write data in byte, word, or longword format. However, ensure that the VMEbus or VSB device to which you write data allows the type of access you specify.

When you call this service to write data to a FIFO or LIFO buffer on another VMEbus or VSB device, specify the KAV\$M\_FIFO\_ACCESS and KAV\$K\_LONGWORD values in the *data\_type* argument. This causes the KAV\$BUS\_WRITE service to write data to the same VMEbus or VSB address each time. If you do not specify the KAV\$M\_FIFO\_ACCESS value, the KAV\$BUS\_WRITE service increments the VMEbus or VSB address after each write operation, as follows:

- When it writes data in byte format, it increments the address by 1 byte
- When it writes data in word format, it increments the address by 2 bytes
- When it writes data in longword format, it increments the address by 4 bytes

---

### Note

---

Digital recommends that you exchange data by directly accessing the VMEbus and VSB, rather than by calling the KAV\$BUS\_WRITE kernel service. See Section 5.1.1 for more information.

---

### Ada Call Format

```
WITH KAVDEF;  
KAV_BUS_WRITE ([STATUS => status,  
                DATA_TYPE => data_type,  
                VIRTUAL_ADDRESS => virtual_address,  
                BUFFER => buffer,  
                COUNT => count);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>data_type</i> :	in	INTEGER;
<i>virtual_address</i> :	in	SYSTEM.ADDRESS;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>count</i> :	in	INTEGER;

---

### C Call Format

```
#include $vaxelnc  
#include "eln$.kavdef.h"  
  
int kav$bus_write ([status,  
                  data_type,  
                  virtual_address,  
                  buffer,  
                  count])
```

argument information

int	<i>*status;</i>
int	<i>data_type;</i>
void	<i>*virtual_address;</i>
void	<i>*buffer;</i>
int	<i>count;</i>

---

**FORTRAN Call Format**

```
INCLUDE 'ELN$:KAVDEF.FOR'  
  
CALL KAV$BUS_WRITE ([status],  
                    %VAL(data_type),  
                    virtual_address,  
                    buffer,  
                    %VAL(count))
```

argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>data_type</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>count</i>

---

**Pascal Call Format**

```
INCLUDE $KAVDEF;  
  
KAV$BUS_WRITE ([STATUS := status,]  
              data_type,  
              virtual_address,  
              buffer,  
              count)
```

## KAV\$BUS\_WRITE

### argument information

<i>status</i> :	INTEGER;
<i>data_type</i> :	INTEGER;
<i>virtual_address</i> :	^ANYTYPE;
<i>buffer</i> :	^ANYTYPE;
<i>count</i> :	INTEGER;

---

### Arguments

#### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

#### ***data\_type***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the format of the data you want to write.

Specify one of the following values:

KAV\$K_BYTE	Writes data in byte format
KAV\$K_LONGWORD	Writes data in longword format
KAV\$K_WORD	Writes data in word format
KAV\$M_FIFO_ACCESS	Writes data to a FIFO or LIFO buffer

The VMEbus or VSB device that you want to write to must allow the type of access you specify.

---

**Note**

---

When you want to write data to a FIFO or LIFO buffer on another VMEbus or VSB device, specify the KAV\$M\_FIFO\_ACCESS and KAV\$K\_LONGWORD values.

---

***virtual\_address***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the virtual address, in S0 space, where you want to begin writing data. To calculate the address, follow these steps:

- Calculate the base address of the VMEbus or VSB device that you want to write to. The KAV\$OUT\_MAP service returns the base VMEbus or VSB address of a device. See the description of the KAV\$OUT\_MAP service for more information.
- Calculate the offset, into the device’s address space, that you want to write to.
- Add the address to the offset.

When you write data to a FIFO buffer on another KAV30, use the following offsets when calculating the address:

FIFO Buffer	Offset
0	4000 <sub>16</sub>
1	4010 <sub>16</sub>
2	4020 <sub>16</sub>
3	4030 <sub>16</sub>



# KAV\$BUS\_WRITE

When you write data to a LIFO buffer on another KAV30, use the following offsets when calculating the address:

LIFO Buffer	Offset
0	4040 <sub>16</sub>
1	4050 <sub>16</sub>
2	4060 <sub>16</sub>
3	4070 <sub>16</sub>

**buffer**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Reference

Specifies the address of the buffer of data that the KAV\$BUS\_WRITE service writes to the VMEbus or VSB.

**count**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of data items (of the type specified by the *data\_type* argument) that the KAV\$BUS\_WRITE service writes to the VMEbus or VSB.

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_BUS_WRT_ERROR	A VMEbus or a VSB write error occurred.
KAV30\$_BUS_ARB_ERROR	A VMEbus or VSB arbitration failure occurred.
KAV30\$_INVALID_SG_ENTRY	You specified an invalid SGM entry.
KAV30\$_NO_BUS_WRT_RESP	There was no write response from the device.

KAV30\$_WRPROT_SG_ENTRY	You tried to write to a write-protected SGM entry.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The services could not access an item.
KER\$_SUCCESS	The operation was successful.

---

Related Services

KAV\$BUS_BITCLR	KAV\$OUT_MAP
KAV\$BUS_BITSET	KAV\$BUS_READ

---

Examples

- The following code is an example KAV\$BUS\_WRITE call:

```
vir_address      = virtual_address + '3601'X
buffer           = '13'X
CALL    KAV$BUS_WRITE    (status,
1              %VAL(data_type),
2              %VAL(vir_address),
3              buffer,
4              %VAL(count))
IF ( .NOT. status )      TYPE 1040, status
1040  FORMAT(1H, 'KAV$BUS_WRITE, status is :', Z4.4 )
```
- The file SYS\$COMMON:[SYSHLP.EXAMPLES.KAV]KAV\_MVME.FOR contains a program that calls the KAV\$BUS\_WRITE service

# KAV\$CHECK\_BATTERY

---

## KAV\$CHECK\_BATTERY

Checks the power supply to the battery backed-up RAM and the calendar/clock.

The power supply to the battery backed-up RAM and calendar/clock can be one of the following:

- Sufficient  
When the power supply to the relevant devices is sufficient, the devices have enough power to operate normally.
- Dead  
When the power supply to the relevant devices is dead, the devices do not have enough power to operate normally. The contents of the relevant devices are unpredictable.

---

### Ada Call Format

```
WITH KAVDEF;  
KAV_CHECK_BATTERY (STATUS => status);
```

### argument information

```
status : out CONDITION_HANDLING.COND_VALUE_  
TYPE;
```

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"
int kav$check_battery (status)
```

### argument information

```
int                                *status;
```

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'
CALL KAV$CHECK_BATTERY (status)
```

### argument information

```
INTEGER*4                        status
```

---

## Pascal Call Format

```
INCLUDE $KAVDEF;
KAV$CHECK_BATTERY (STATUS := status)
```

### argument information

```
status :                        INTEGER;
```

# KAV\$CHECK\_BATTERY

---

## Arguments

*status*  
Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference  
Receives the completion status.

---

## Status Values

KAV30\$_BAD_BATTERY	The KAV30 battery is dead.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_SUCCESS	The operation was successful.

---

## Examples

- The file SYS\$COMMON:[SYSHLP.EXAMPLES.KAV]KAV\_MVME.FOR contains a program that calls the KAV\$CHECK\_BATTERY service

- The following code is an example KAV\$CHECK\_BATTERY call:

```
CALL      KAV$CHECK_BATTERY (status)
IF ( .NOT. status )      TYPE 100, status
100  FORMAT(1H, 'KAV$CHECK BATTERY status is :',Z4.4 )
      buffer              = '00000000'X
      setup_function      = KAV$K_ALLOW_VME_IRQ
      CALL      KAV$VME_SETUP      (status,
1                                %VAL(setup_function),
2                                buffer)
      IF ( .NOT. status )      TYPE 1000, status
1000  FORMAT(1H, 'KAV$VME_SETUP status is :',Z4.4 )
```

# KAV\$CLR\_AST

---

## KAV\$CLR\_AST

Clears a device's AST queue. The service clears all the ASB data structures and removes any pending ASTs.

This service uses the device code returned by the KAV\$DEF\_AST service to identify the AST queue to clear.

You can also use this service to remove any ASTs that are pending as a result of a call to the KAV\$SET\_AST service that specified a repeating AST. See the description of the KAV\$SET\_AST service for more information.

---

### Ada Call Format

```
WITH KAVDEF;  
KAV_CLR_AST ([STATUS => status,]  
             DEVICE_CODE => device_code);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_ TYPE;
<i>device_code</i> :	in	INTEGER;

---

### C Call Format

```
#include $vxelnc  
#include "eln$:kavdef.h"  
int kav$clr_ast ([status],  
                device_code)
```

### argument information

int	<i>*status</i> ;
int	<i>device_code</i> ;

---

## FORTTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$CLR_AST ([status],  
                  %VAL(device_code))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>device_code</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$CLR_AST ([STATUS := status,]  
             device_code)
```

### argument information

<i>status</i> :	INTEGER;
<i>device_code</i> :	INTEGER;

---

## Arguments

### ***status***

Usage:	Longword (unsigned)
VAX Type:	cond_value
Access:	Write only
Mechanism:	Reference

Receives the completion status.



# KAV\$CLR\_AST

***device\_code***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies a code that identifies the AST queue to clear. Use the code that the KAV\$DEF\_AST service returned when you defined the AST queue.

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$DEF_AST	KAV\$QUE_AST
KAV\$SET_AST	

---

## KAV\$DEF\_AST

Creates an AST queue for an event associated with a VMEbus or VSB device. It allocates the first available AST queue to the device event and returns a device code.

When you call the KAV\$CLR\_AST, KAV\$QUE\_AST, or KAV\$SET\_AST services in relation to the device event, use the device code to identify the AST queue. Call this service only once for each device event.

The KAV30 kernel sets up 256 AST queues, 37 of which are reserved for use by the KAV30. This leaves a total of 219 queues available for VMEbus or VSB devices.

See Section 3.1 for more information on ASTs.

---

### Ada Call Format

```
WITH KAVDEF;
KAV_DEF_AST ([STATUS => status,]
             DEVICE_CODE => device_code);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>device_code</i> :	in	INTEGER;

---

### C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"
int kav$def_ast ([status],
                 device_code)
```

# KAV\$DEF\_AST

## argument information

int	<i>*status;</i>
int	<i>*device_code;</i>

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$DEF_AST ([status],  
                  device_code)
```

## argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>device_code</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$DEF_AST ([STATUS := status,]  
             device_code)
```

## argument information

<i>status</i> :	INTEGER;
<i>device_code</i> :	INTEGER;

---

Arguments

*status*

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Receives the completion status.

*device\_code*

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Returns a code that identifies the AST queue. Use this code to identify the queue when calling the KAV\$CLR\_AST, KAV\$QUE\_AST, and KAV\$SET\_AST services.

---

Status Values

KAV30\$_ASBQUOTA	You have reached the maximum number of ASBs for this device code.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_SUCCESS	The operation is successful.

## KAV\$DEF\_AST

---

### Related Services

KAV\$CLR\_AST  
KAV\$QUE\_AST

KAV\$SET\_AST

---

### Examples

See the programs listed in Appendix C for examples of KAV\$DEF\_AST service calls.

---

# KAV\$FIFO\_READ

Reads a specified number of aligned longwords from one of the KAV30 FIFO buffers.

When you read data from a FIFO buffer on another KAV30, use the following offsets when calculating the address to read from:

FIFO Buffer	Offset
0	4000 <sub>16</sub>
1	4010 <sub>16</sub>
2	4020 <sub>16</sub>
3	4030 <sub>16</sub>

See Section 3.4 for information about the KAV30 FIFO buffers.

---

## Ada Call Format

```
WITH KAVDEF;  
KAV_FIFO_READ ([STATUS => status,]  
               FIFO_NUMBER => fifo_number,  
               BUFFER => buffer,  
               COUNT => count);
```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_ TYPE;
<i>fifo_number</i> :	in	INTEGER;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>count</i> :	in	INTEGER;

# KAV\$FIFO\_READ

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$fifo_read ([status],
                  fifo_number,
                  buffer,
                  count)
```

### argument information

int	<i>*status;</i>
int	<i>fifo_number;</i>
int	<i>*buffer;</i>
int	<i>count;</i>

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$FIFO_READ ([status],
                   %VAL(fifo_number),
                   buffer,
                   %VAL(count))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>fifo_number</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>count</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;

KAV$FIFO_READ ([STATUS := status,]
               fifo_number,
               buffer,
               count)
```

## argument information

<i>status</i> :	INTEGER;
<i>fifo_number</i> :	INTEGER;
<i>buffer</i> :	^ANYTYPE;
<i>count</i> :	INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
 VAX Type: cond\_value  
 Access: Write only  
 Mechanism: Reference

Receives the completion status.

### ***fifo\_number***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the FIFO buffer to read from. Specify one of the following values:

KAV\$K_FIFO_0	Reads data from FIFO number 0
KAV\$K_FIFO_1	Reads data from FIFO number 1
KAV\$K_FIFO_2	Reads data from FIFO number 2
KAV\$K_FIFO_3	Reads data from FIFO number 3



# KAV\$FIFO\_READ

**buffer**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Specifies the buffer into which this service places the data (in aligned longwords) it reads from the FIFO buffers.

**count**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of aligned longwords to read from the FIFO buffer.

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_COUNT_OVERFLOW	There is a FIFO counter overflow.
KER\$_NO_ACCESS	The services cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$FIFO_WRITE	KAV\$NOTIFY_FIFO
KAV\$LIFO_WRITE	

---

## **Examples**

See the programs listed in Appendix B for examples of KAV\$FIFO\_READ service calls.

---

KAV\$FIFO\_WRITE

Writes a specified number of aligned longwords to one of the KAV30 FIFO buffers in FIFO mode.

When you write data to a FIFO buffer on another KAV30, use the following offsets when calculating the address to write to:

FIFO Buffer	Offset
0	4000 <sub>16</sub>
1	4010 <sub>16</sub>
2	4020 <sub>16</sub>
3	4030 <sub>16</sub>

See Section 3.4 for information about the KAV30 FIFO buffers.

---

Ada Call Format

```
WITH KAVDEF;  
KAV_FIFO_WRITE ([STATUS => status,  
                FIFO_NUMBER => fifo_number,  
                BUFFER => buffer,  
                COUNT => count);
```

argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_ TYPE;
<i>fifo_number</i> :	in	INTEGER;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>count</i> :	in	INTEGER;

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$fifo_write ([status],
                   fifo_number,
                   buffer,
                   count)
```

### argument information

int	<i>*status</i> ;
int	<i>fifo_number</i> ;
void	<i>*buffer</i> ;
int	<i>count</i> ;

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$FIFO_WRITE ([status],
                    %VAL(fifo_number),
                    buffer,
                    %VAL(count))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>fifo_number</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>count</i>

### Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$FIFO_WRITE ([STATUS := status,]  
                fifo_number,  
                buffer,  
                count)
```

### argument information

<i>status</i> :	INTEGER;
<i>fifo_number</i> :	INTEGER;
<i>buffer</i> :	^ANYTYPE;
<i>count</i> :	INTEGER;

---

### Arguments

#### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

#### ***fifo\_number***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the FIFO buffer that you want to write to. Specify one of the following values:

KAV\$K_FIFO_0	Writes data to FIFO number 0
KAV\$K_FIFO_1	Writes data to FIFO number 1
KAV\$K_FIFO_2	Writes data to FIFO number 2
KAV\$K_FIFO_3	Writes data to FIFO number 3

***buffer***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Reference

Specifies the buffer of data that this service writes (in aligned longwords) into the FIFO buffer.

***count***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the number of aligned longwords that this service writes into the FIFO buffer.

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_COUNT_OVERFLOW	There is a FIFO counter overflow.
KER\$_NO_ACCESS	The services cannot access an item.
KER\$_SUCCESS	The operation is successful.

## KAV\$FIFO\_WRITE

---

### Related Services

KAV\$FIFO\_READ

KAV\$NOTIFY\_FIFO

KAV\$LIFO\_WRITE

---

### Examples

See the programs listed in Appendix B for examples of KAV\$FIFO\_WRITE service calls.

---

## KAV\$GATHER\_KAV\_ERRORLOG

Reads the error log entries from the error-log area of the KAV30 battery backed-up RAM.

When certain error conditions occur in devices on the KAV30, VMEbus, or VSB, the KAV30 kernel writes an error code to its battery backed-up RAM. See Section 3.8 for more information.

---

### Ada Call Format

```
WITH KAVDEF;

KAV_GATHER_KAV_ERRORLOG ([STATUS => status,]
                          ERRORLOG_FUNCTIONS => errorlog_functions,
                          BUFFER => buffer);
                          ENTRY_COUNT => entry_count);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>errorlog_functions</i> :	in	INTEGER;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>entry_count</i> :	in	INTEGER;

---

### C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$gather_kav_errorlog ([status],
                             errorlog_functions,
                             buffer,
                             entry_count)
```



# KAV\$GATHER\_KAV\_ERRORLOG

## argument information

int	<i>*status;</i>
int	<i>errorlog_functions;</i>
void	<i>*buffer;</i>
int	<i>entry_count;</i>

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$GATHER_KAV_ERRORLOG ([status],
                               %VAL(errorlog_functions),
                               buffer,
                               %VAL(entry_count))
```

## argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>errorlog_functions</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>entry_count</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;

KAV$GATHER_KAV_ERRORLOG ([STATUS := status,]
                          errorlog_functions,
                          buffer,
                          entry_count)
```

## argument information

*status* : INTEGER;  
*errorlog\_functions* : INTEGER;  
*buffer* : ^ANYTYPE;  
*entry\_count* : INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Write only  
 Mechanism: Reference

Receives the completion status.

### ***errorlog\_functions***

Usage: Longword (unsigned)  
 VAX Type: Longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the operation that you want to perform. Specify one of the following values:

KAV\$K_CLEAR_ERR	Initializes all the error log data and pointers
KAV\$K_INIT_RD_POINTER	Sets the error log read entry pointer to the value of the error log write entry pointer
KAV\$K_MASTER_ERR	Gathers the error log entries that were caused by the master VMEbus and VSB accesses
KAV\$K_SLAVE_ERR	Gathers the error log entries that were caused by the slave VMEbus accesses
KAV\$K_ALL_ERR	Gathers the error log entries that were caused by slave and master accesses

# KAV\$GATHER\_KAV\_ERRORLOG

**buffer**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Specifies the address of a buffer to which the service returns the error log entries. The service returns one 28-byte buffer segment for each error log entry it returns. The buffer you specify must be long enough to return 28-byte segments for each error log entry you want to read. The number of error log entries you want to read is specified in the *entry\_count* parameter. When the buffer area is not long enough, the kernel will return an error status.

The contents of the buffer segment detailing an error log entry depend on whether you read a master error or a slave error. Master errors have the following layout:

Message Status Code			0
AM	EC	Retry	4
VMEbus/VSB Address			8
PC			12
PSL			16
Absolute System Time			20
			24

Slave errors have the following layout:

Message Status Code	0
Error Status	4
Error Count	8
Reserved for Digital	12
Reserved for Digital	16
Absolute System Time	20 24

### ***entry\_count***

Usage: Longword (unsigned)  
 VAX Type: Longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the number of error log entries that you want to read from the battery backed-up RAM.

---

## **Status Values**

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_END_OF_ERRORLOG	You have reached the end of the KAV30 error log area.
KAV30\$_ERRORLOG_EMPTY	There is no error of the type that you specified in the error log area.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.

## KAV\$GATHER\_KAV\_ERRORLOG

KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The services cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

### Examples

The following code is an example program that calls the KAV\$GATHER\_KAV\_ERRORLOG service:

```
/*
 * Facility:      KAV30 VAXELN System Services programming example.
 *
 * Description:   This is an example program demonstrating the calling
 *               procedures for the following KAV System Services:
 *               1. KAV$OUT_MAP          (Map onto VMEbus address)
 *               2. KAV$BUS_READ        (Read from VMEbus address)
 *               3. KAV$GATHER_KAV_ERRORLOG (Read KAV error log)
 *
 * Abstract:      This program induces a KAV error condition, then requests
 *               the error log.
 *
 * Language:      Vax C; Version 3.1
 *
 * Notes: (1)     In the interests of program clarity, no error checking has
 *               been included.
 */
#include <stdio>
#include <$vaxelnc>
#include <eln$:kavdef.h> /* KAV30 definitions file. */
#define INVALID_ADDRESS 0xddeeff /* Non-existent address for error. */

main()
{
    unsigned long buff[4]; /* ...to receive the error report. */
    unsigned long entry,
        am_code,
        vir_addr,
        device_code,
        phys_addr;
    unsigned long *bad_address;
    int status,
        page_count,
        map_functions;
```

```

/*
 *   Map into the device register region.
 *   =====
 */
page_count      = 1 ;
phys_addr       = INVALID_ADDRESS ;
am_code         = KAV$K_USER 24 ;
map_functions    = KAV$M_VME+KAV$M_MODE_0_SWAP;

KAV$OUT_MAP(  &status, &entry, page_count,
              phys_addr, &vir_addr,
              am_code, map_functions);

/*
 *   READ from the VMEbus (this should induce an error).
 *   =====
 */
KAV$BUS_READ(  &status,
              KAV$K_BYTE,
              vir_addr,
              buff,
              1 ) ;

/*
 *   Request error report
 *   =====
 */
device_code = KAV$K_ALL_ERR ;
KAV$GATHER_KAV_ERRORLOG( &status,
                        device_code,
                        buff ) ;

printf("\n\nKAV Error report: /n");
printf("  Error Count          = %d (%x hex)/n", buff[0], buff[0] );
printf("  Address of last error = %d (%x hex)/n", buff[1], buff[1] );
printf("  VME/VSB error code    = %d (%x hex)/n", buff[2], buff[2] );
printf("  KAV error code        = %d (%x hex)/n", buff[3], buff[3] );
} /*   end   -program-   */

```

---

### KAV\$IN\_MAP

Maps one or more 64K byte pages (aligned on a 64K byte boundary) of the VMEbus address space into KAV30 process (P0) space or into the FIFO buffers on the KAV30.

This service uses the incoming SGM to perform the mapping. See Section 3.6 for more information.

The programs calling the KAV\$IN\_MAP service specify the SGM entry number for the first page of the VMEbus address space mapped into the KAV30 P0 space. These programs subsequently use this number in a call to the KAV\$UNMAP service to free pages of KAV30 P0 space when mapping is no longer required. The SGM entry must be in the range 0 to 15 for A24 mode VMEbus addresses and in the range 0 to 255 for A32 mode VMEbus addresses.

A calling program can set a modifier that forces the KAV30 to interrupt the kernel if any VMEbus device accesses the part of the VMEbus address space mapped by the incoming SGM map to KAV30 P0 space. The kernel then queues an AST to the process that called the KAV\$IN\_MAP service. This function is called a location monitor. The location monitor also specifies the interrupt priority level at which the KAV30 kernel delivers the interrupt.

The KAV30 kernel can also set write protection on the pages of P0 space to which the VMEbus address space is mapped. This prevents the VMEbus devices from accidentally writing the part of the VMEbus address space mapped to the KAV30 P0 space.

The KAV30 is a little-endian device, so to exchange data with a big-endian device, you must translate the data from big-endian format to little-endian format. This service can map the address space directly to KAV30 P0 space, or it can specify byte-swapping or word-swapping as part of the mapping. When a VMEbus device subsequently reads or writes the VMEbus address space, the address space mapped into the KAV30 P0 space is transformed according to the swapping operations specified by this service. See Section 3.6.3 for more information about data mapping.

---

## Ada Call Format

```

WITH KAVDEF;

KAV_IN_MAP ([STATUS => status,]
             SGM_ENTRY => sgm_entry,
             PAGE_COUNT => page_count,
             VIRTUAL_ADDRESS => virtual_address,
             [AST_ADDR => ast_addr,]
             [AST_PARAM => ast_param,]
             MAP_FUNCTIONS => map_functions);

```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>sgm_entry</i> :	in	INTEGER;
<i>page_count</i> :	in	INTEGER;
<i>virtual_address</i> :	out	SYSTEM.ADDRESS;
<i>ast_addr</i> :	in	SYSTEM.ADDRESS;
<i>ast_param</i> :	in	INTEGER;
<i>map_functions</i> :	in	INTEGER;

---

## C Call Format

```

#include $vaxelnc

#include "eln$:kavdef.h"

int kav$in_map ([status],
                entry,
                page_count,
                virtual_address,
                [ast_addr],
                [ast_param],
                map_functions)

```



## KAV\$IN\_MAP

### argument information

int	<i>*status;</i>
int	<i>entry;</i>
int	<i>page_count;</i>
void	<i>**virtual_address;</i>
void	<i>*ast_addr( );</i>
int	<i>ast_param;</i>
int	<i>map_functions;</i>

---

### FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$IN_MAP ([status,  
                 %VAL(entry),  
                 %VAL(page_count),  
                 virtual_address,  
                 [ast_addr],  
                 [%VAL(ast_param)],  
                 %VAL(map_functions))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>entry</i>
INTEGER*4	<i>page_count</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>ast_addr</i>
INTEGER*4	<i>ast_param</i>
INTEGER*4	<i>map_functions</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;
KAV$IN_MAP ([STATUS := status,]
            entry,
            page_count,
            virtual_address,
            [AST_ADDR := ast_addr,]
            [AST_PARAM := ast_param,]
            map_functions)
```

## argument information

<i>status</i> :	INTEGER;
<i>entry</i> :	INTEGER;
<i>page_count</i> :	INTEGER;
<i>virtual_address</i> :	^ANYTYPE;
<i>ast_addr</i> :	^ANYTYPE;
<i>ast_param</i> :	INTEGER;
<i>map_functions</i> :	INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
 VAX Type: cond\_value  
 Access: Write only  
 Mechanism: Reference

Receives the completion status.

### ***entry***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

## KAV\$IN\_MAP

Specifies one of the following:

- The SGM entry number for the first page of VMEbus address space that you want to map into KAV30 P0 space
- The SGM entry number for the first page of VMEbus address space that you want to map into the KAV30 FIFO buffers

### ***page\_count***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of successive 64K byte pages of data that you want to map.

### ***virtual\_address***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Returns the virtual address in KAV30 P0 space that corresponds to the starting address of the 64K byte page of VMEbus address space. This service does not return a virtual address if the *map\_functions* argument has the value KAV\$M\_CSR.

### ***ast\_addr***

Usage: Procedure entry mask  
VAX Type: procedure  
Access: Read only  
Mechanism: Reference

When you enable the location monitor in the *map\_functions* argument, this argument specifies the address of the AST routine that the KAV30 kernel executes whenever a device reads to or writes from the pages that you want to map. See Section 3.1 for more information about ASTs.

### ***ast\_param***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the parameter that you want to pass to the AST routine.

***map\_functions***

Usage: Longword (unsigned)  
 VAX Type: mask\_longword  
 Access: Read  
 Mechanism: Value

Specifies the following information that controls the mapping operation:

- The mapping direction—whether you want to map pages of VMEbus address space into the KAV30 P0 space or into the KAV30 FIFO buffers
- The location monitor
- The byte swapping mode
- The write-protection of a mapped area

Specify one or more of the following modifiers:

KAV\$M_LOCMON_IPL15	Interrupts at IPL 15 <sub>16</sub> .
KAV\$M_LOCMON_IPL16	Interrupts at IPL 16 <sub>16</sub> .
KAV\$M_LOCMON_IPL17	Interrupts at IPL 17 <sub>16</sub> .
KAV\$M_CSR	Maps data from the VMEbus address space into one of the KAV30 FIFO buffers. If you do not supply this modifier, the KAV30 kernel maps VMEbus address space into the KAV30 P0 space.
KAV\$M_MEMORY	Maps the VMEbus address space into the KAV30 P0 space.
KAV\$M_MODE_0_SWAP	Performs mode 0 swapping.
KAV\$M_MODE_2_SWAP	Performs mode 2 swapping.
KAV\$M_MODE_3_SWAP	Performs mode 3 swapping.
KAV\$M_WRT_PROT	Sets write-protection on the page of the system RAM that you want to map to the VMEbus.

Ensure that the values you specify do not conflict with each other. For example, do not specify the KAV\$M\_MODE\_3\_SWAP and KAV\$M\_MODE\_0\_SWAP values together.

# KAV\$IN\_MAP

---

## Status Values

KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KAV\$_NO_MEMORY	There is no physical memory available.
KER\$_NO_PORT	There are no free SGM entry ports. Unmap one or more SGM entries and retry the call.
KAV\$_NO_VIRTUAL	There is no virtual address space available.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$UNMAP

## Examples

The following code is an example program that calls the KAV\$IN\_MAP service:

```
C
C Description: This is an example program demonstrating the calling
C               procedures for the following KAV System Services:
C               1. KAV$IN_MAP (Map VMEbus address space into P0 space)
C               2. KAV$UNMAP (Free Scatter Gather Map [SGM])
C
C Abstract:      Maps an area of P0 space, then relinquishes it.
C
C Language:      Vax Fortran; Version 5.5
C
C Notes: (1)      In the interests of program clarity, no error checking
C               is performed.
C
C
```

```
PROGRAM EX MAPPING
IMPLICIT NONE

INCLUDE 'ELN$:KAVDEF.FOR'
```

```
INTEGER*4    status
INTEGER*4    entry
INTEGER*4    pagecnt
INTEGER*4    vir_addr
INTEGER*4    phys_addr
INTEGER*4    am_code
INTEGER*4    map_functions

INTEGER*4    index
```

```
C      =====
C      Example of *** IN_MAP *** (without AST parameters)
C      =====
      pagecnt      = 1
      index = 1
      entry  = 1
      map_functions = KAV$M_CSR + KAV$M_LOCMON_IPL17 + KAV$M_WRT_PROT

      CALL KAV$IN_MAP (status,
1          %VAL(entry),
2          %VAL(pagecnt),
3          vir_addr,
4          ,
5          ,
6          %VAL(map_functions))
```

## KAV\$IN\_MAP

```
C      =====
C      Example of *** UNMAP ***
C      =====
      map_functions = KAV$M_IN
      CALL KAV$UNMAP (status,
1         %VAL(entry),
2         %VAL(pagecnt),
3         %VAL(vir_addr),
4         %VAL(map_functions))

9999  STOP
END
```

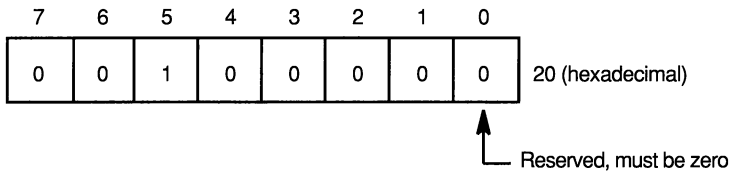
---

## KAV\$INT\_VME

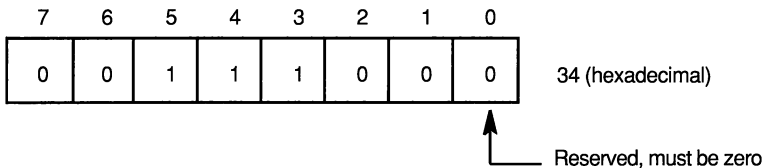
Delivers an IRQ to the VMEbus, reads pending IRQs on the KAV30, or clears any pending VMEbus interrupts.

The *int\_functions* argument specifies whether this service delivers, reads, or clears interrupts. The *irq\_level* argument specifies the IRQ level, and the *int\_vector* argument specifies the VMEbus interrupt vector.

The *irq\_level* argument is a bit mask (only the low-order byte is used). The bit mask specifies the IRQ level at which you want to generate the IRQ. For example, the following diagram shows a bit mask that specifies a level 5 IRQ:



The *int\_vector* argument is a value that uses only the low-order byte of the argument. This value specifies the interrupt vector that the KAV30 uses to interrupt the VMEbus. For example, the following diagram shows an interrupt vector that has the value 34 (hexadecimal):



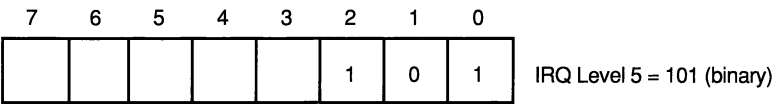
The KAV30 constructs the vector with which it interrupts the VMEbus module



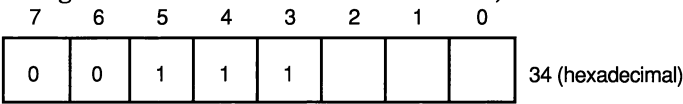
KAV\$INT\_VME

in the following steps:

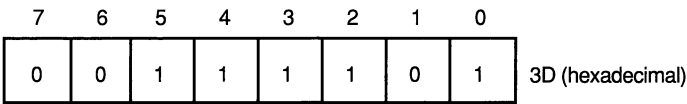
- 1. The KAV30 decodes the IRQ level in the *irq\_level* argument into its binary value and places this value in the 3 low-order bits of the VMEbus vector, as follows:



- 2. The KAV30 places the value specified by the *int\_vector* argument in the 5 high-order bits of the VMEbus vector, as follows:



Steps 1 and 2 result in the following VMEbus vector:



To place a particular vector on the VMEbus, follow these steps:

- 1. Place the binary representation of the IRQ level in the 3 low-order bits
- 2. Fill in the other 5 bits with the values required to give the specific interrupt vector
- 3. Place these values in the *int\_lvl* and *int\_vector* arguments respectively.

If the *int\_functions* argument specifies the value KAV\$K\_RD, this service uses this method to return the IRQ level in the *irq\_level* argument and the interrupt vector value in the *int\_vector* argument.

The programs that use this service to generate interrupt requests at a particular level must configure the KAV30 so that incoming VMEbus interrupts at that level are disabled. See Section 5.4 and the description of the KAV\$VME\_SETUP service for information about configuring the VMEbus.

---

## Ada Call Format

```
WITH KAVDEF;

KAV_INT_VME ([STATUS => status,]
             INT_FUNCTIONS => int_functions,
             IRQ_LEVEL => irq_level,
             INT_VECTOR => int_vector);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_ TYPE;
<i>int_functions</i> :	in	INTEGER;
<i>irq_level</i> :	in out	INTEGER;
<i>int_vector</i> :	in out	INTEGER;

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$int_vme ([status,]
                int_functions,
                irq_level,
                int_vector)
```

### argument information

int	<i>*status;</i>
int	<i>int_functions;</i>
int	<i>*irq_level;</i>
int	<i>*int_vector;</i>

---

**FORTRAN Call Format**

```
INCLUDE 'ELN$:KAVDEF.FOR'  
  
CALL KAV$INT_VME ([status],  
                  %VAL(int_functions),  
                  irq_level,  
                  int_vector)
```

**argument information**

INTEGER*4	<i>status</i>
INTEGER*4	<i>int_functions</i>
INTEGER*4	<i>irq_level</i>
INTEGER*4	<i>int_vector</i>

---

**Pascal Call Format**

```
INCLUDE $KAVDEF;  
  
KAV$INT_VME ([STATUS := status,]  
             int_functions,  
             irq_level,  
             int_vector)
```

**argument information**

<i>status</i> :	INTEGER;
<i>int_functions</i> :	INTEGER;
<i>irq_level</i> :	INTEGER;
<i>int_vector</i> :	INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

### ***int\_functions***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the function that you want to perform. Specify one of the following values:

KAV\$K_RD	Reads the IRQ currently pending on the KAV30 and returns the interrupt vector (in the <i>int_vector</i> argument) and the interrupt level (in the <i>irq_level</i> argument).
KAV\$K_VME_INT_CLR	Clears the interrupts that are currently pending on the KAV30.
KAV\$K_VME_REQ_INT	Requests an interrupt at the IRQ level specified by the <i>irq_level</i> argument.

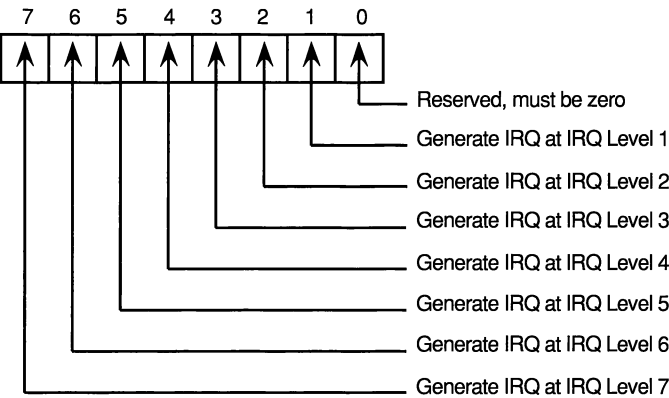
### ***irq\_level***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Modify  
Mechanism: Reference

If the *int\_functions* argument specifies the value KAV\$K\_VME\_REQ\_INT, this argument specifies the VMEbus IRQ level at which you want to generate an IRQ on the VMEbus.

KAV\$INT\_VME

The *irq\_level* argument is a bit mask. However, this service uses only the low-order byte of the bit mask. The bit mask specifies the IRQ level at which you want to generate the IRQ, as follows:



If the *int\_functions* argument specifies the KAV\$K\_RD value, the KAV\$INT\_VME service returns the IRQ level in this argument.

It is not necessary to specify an IRQ level when clearing an interrupt. Therefore, if the *int\_functions* argument has the value KAV\$K\_VME\_INT\_CLR, you can omit the *irq\_level* argument.

**int\_vector**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Modify  
Mechanism: Reference

Specifies the VMEbus vector that the KAV30 writes to the VMEbus when the KAV30 receives an IACK signal from the VMEbus. The *int\_vector* argument is a value in which the 3 low-order bits must be 0, and the 5 high-order bits contain the 5 high-order bits of the VMEbus interrupt vector.

If the *int\_functions* argument has the value KAV\$K\_RD, the KAV\$INT\_VME service returns the interrupt vector, for the IRQ currently pending on the KAV30, in the low-order byte of the *int\_vector* argument. However, it is not possible to decode the 3 low-order bits.

---

## Status Values

KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_VME_INT_PEND	An outgoing VMEbus interrupt is pending.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The services cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$VME\_SETUP

## KAV\$LIFO\_WRITE

---

## KAV\$LIFO\_WRITE

Writes a specified number of aligned longwords to one of the KAV30 FIFO buffers in LIFO mode.

See Section 3.4 for information about the KAV30 FIFO buffers.

---

### Ada Call Format

```
WITH KAVDEF;  
  
KAV_LIFO_WRITE ([STATUS => status,]  
                FIFO_NUMBER => fifo_number,  
                BUFFER => buffer,  
                COUNT => count);
```

### argument information

<i>status</i> :	out	SYSTEM.ADDRESS;
<i>fifo_number</i> :	in	INTEGER;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>count</i> :	in	INTEGER;

---

### C Call Format

```
#include $vaxelnc  
  
#include "eln$.kavdef.h"  
  
int kav$lifo_write ([status],  
                   fifo_number,  
                   buffer,  
                   count)
```

argument information

int	<i>*status;</i>
int	<i>fifo_number;</i>
void	<i>*buffer;</i>
int	<i>count;</i>

---

**FORTRAN Call Format**

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$LIFO_WRITE ([status],
                     %VAL(fifo_number),
                     buffer,
                     %VAL(count))
```

argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>fifo_number</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>count</i>

---

**Pascal Call Format**

```
INCLUDE $KAVDEF;

KAV$LIFO_WRITE ([STATUS := status,
                fifo_number,
                buffer,
                count])
```



# KAV\$LIFO\_WRITE

## argument information

<i>status</i> :	INTEGER;
<i>fifo_number</i> :	INTEGER;
<i>buffer</i> :	^ANYTYPE;
<i>count</i> :	INTEGER;

---

## Arguments

### *status*

Usage:	Longword (unsigned)
VAX Type:	cond_value
Access:	Write only
Mechanism:	Reference

Receives the completion status.

### *fifo\_number*

Usage:	Longword (unsigned)
VAX Type:	longword_unsigned
Access:	Read only
Mechanism:	Value

Specifies the FIFO buffer that you want to write to. Specify one of the following values:

KAV\$K_FIFO_0	Writes data to FIFO number 0
KAV\$K_FIFO_1	Writes data to FIFO number 1
KAV\$K_FIFO_2	Writes data to FIFO number 2
KAV\$K_FIFO_3	Writes data to FIFO number 3

### *buffer*

Usage:	Longword (unsigned)
VAX Type:	longword_unsigned
Access:	Read only
Mechanism:	Reference

Specifies the buffer of data that this service writes (in aligned longwords) into the FIFO buffer.

**count**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of aligned longwords that this service writes into the FIFO buffer.

---

**Status Values**

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_COUNT_OVERFLOW	There is a FIFO counter overflow.
KER\$_NO_ACCESS	The services cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

**Related Services**

KAV\$FIFO_READ	KAV\$NOTIFY_FIFO
KAV\$FIFO_WRITE	

---

## KAV\$NOTIFY\_FIFO

Delivers an AST when one of the KAV30 FIFO buffers make one of the following transitions:

- From the state *not full* to the state *full*
- From the state *not empty* to the state *empty*
- From the state *empty* to the state *not empty*

See Section 3.4 for information about KAV30 FIFO buffers.

---

### Ada Call Format

```
WITH KAVDEF;  
KAV_NOTIFY_FIFO ([STATUS => status,]  
                  FIFO_FUNCTIONS => fifo_functions,  
                  FIFO_NUMBER => fifo_number,  
                  AST_ADDR => ast_addr,  
                  [AST_PARAM => ast_param]);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_ TYPE;
<i>fifo_functions</i> :	in	INTEGER;
<i>fifo_number</i> :	in	INTEGER;
<i>ast_addr</i> :	in	SYSTEM.ADDRESS;
<i>ast_param</i> :	in	INTEGER;

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$notify_fifo ([status],
                     fifo_functions,
                     fifo_number,
                     ast_addr,
                     [ast_param])
```

## argument information

int	<i>*status</i> ;
int	<i>fifo_functions</i> ;
int	<i>fifo_number</i> ;
void	<i>*ast_addr</i> ( );
int	<i>ast_param</i> ;

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$NOTIFY_FIFO ([status],
                      %VAL(fifo_functions),
                      %VAL(fifo_number),
                      ast_addr,
                      [%VAL(ast_param)])
```

# KAV\$NOTIFY\_FIFO

## argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>fifo_functions</i>
INTEGER*4	<i>fifo_number</i>
INTEGER*4	<i>ast_addr</i>
INTEGER*4	<i>ast_param</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;

KAV$NOTIFY_FIFO ([STATUS := status,]
                  fifo_functions,
                  fifo_number,
                  ast_addr,
                  [AST_PARAM := ast_param])
```

## argument information

<i>status</i> :	INTEGER;
<i>fifo_functions</i> :	INTEGER;
<i>fifo_number</i> :	INTEGER;
<i>ast_addr</i> :	^ANYTYPE;
<i>ast_param</i> :	INTEGER;

---

## Arguments

<b><i>status</i></b>	
Usage:	Longword (unsigned)
VAX Type:	cond_value
Access:	Write only
Mechanism:	Reference
Receives the completion status.	

***fifo\_functions***

Usage: Longword (unsigned)  
 VAX Type: Longword  
 Access: Read only  
 Mechanism: Value

Specifies the conditions that determine when the KAV30 delivers the AST.  
 Specify one or more of the following modifiers:

KAV\$m_FIFO_EMPTY	Delivers the AST when the FIFO buffer makes the transition from the state <i>not empty</i> to the state <i>empty</i> . If the FIFO buffer is empty when you call this service, the KAV30 kernel delivers the AST immediately.
KAV\$m_FIFO_FULL	Delivers the AST when the FIFO buffer makes the transition from the state <i>not full</i> to the state <i>full</i> . If the FIFO buffer is full when you call this service, the KAV30 kernel delivers the AST immediately.
KAV\$m_FIFO_NOT_EMPTY	Delivers the AST when the FIFO buffer makes the transition from the state <i>empty</i> to the state <i>not empty</i> . If the FIFO buffer is not empty when you call this service, the KAV30 kernel delivers the AST immediately.
KAV\$m_RESET_FIFO	Resets the FIFO buffer.  The service clears the FIFO buffer memory, pending AST delivery, and FIFO condition interrupt.

You can specify the following combinations of values:

- KAV\$m\_FIFO\_FULL and KAV\$m\_FIFO\_EMPTY
- KAV\$m\_FIFO\_FULL and KAV\$m\_FIFO\_NOT\_EMPTY

***fifo\_number***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the FIFO buffer that you want to operate on.

## KAV\$NOTIFY\_FIFO

Specify one of the following values:

KAV\$K_FIFO_0	Operates on FIFO buffer 0
KAV\$K_FIFO_1	Operates on FIFO buffer 1
KAV\$K_FIFO_2	Operates on FIFO buffer 2
KAV\$K_FIFO_3	Operates on FIFO buffer 3

### ***ast\_addr***

Usage: Procedure entry mask  
VAX Type: procedure  
Access: Read only  
Mechanism: Reference

Specifies the address of the AST routine that you want to execute when the FIFO buffer meets the conditions specified by the *fifo\_functions* argument.

This argument is optional when the *fifo\_functions* argument specifies the value KAV\$M\_RESET\_FIFO.

### ***ast\_param***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the address of a parameter, which this service passes to the AST routine. See Section 3.1 for more information about ASTs.

This argument is optional when the *fifo\_functions* argument specifies the value KAV\$M\_RESET\_FIFO.

---

## Status Values

KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_FIFO_BUSY	The FIFO buffer you want to operate on is busy.

KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

## **Related Services**

KAV\$FIFO_READ	KAV\$LIFO_WRITE
KAV\$FIFO_WRITE	

---

## **Examples**

See the programs listed in Appendix B for examples of KAV\$NOTIFY\_FIFO service calls.



---

## KAV\$OUT\_MAP

Maps one or more 64K byte pages (aligned on a 64K byte boundary) of the KAV30 system virtual address (S0) space to the VMEbus or VSB address space.

After you map the address space, use one of the following methods to access it:

- Call the KAV\$BUS\_BITCLR, KAV\$BUS\_BITSET, KAV\$BUS\_READ, and KAV\$BUS\_WRITE services
- Write directly to the address space. See Section 5.1 for more information.

Before you access the VMEbus and VSB address space, configure the VMEbus and VSB. See Section 5.4.1 for more information.

This service uses the outgoing SGM to perform the mapping. See Section 3.6 for more information.

The KAV30 is a little-endian device, so to exchange data with a big-endian device, you must translate the data from the little-endian format to big-endian format. This service can map KAV30 S0 address space directly to the VMEbus or VSB address space, or it can specify byte-swapping or word-swapping as part of the mapping. When you call the KAV\$BUS\_READ or KAV\$BUS\_WRITE service, these services read or write the data according to the swapping operations that you specify using this service. See Section 3.6.3 for more information about data mapping.

This service returns a virtual address, in KAV30 S0 space, that corresponds to the base VMEbus or VSB address of the address space of the device. To read or write data at an offset into the VMEbus or VSB address space, add the offset to the virtual address and read or write that virtual address.

This service also returns the SGM entry number. Programs use this number in a call to the KAV\$UNMAP service to free pages of KAV30 S0 space when they are no longer required to be mapped to the VMEbus or VSB.

---

## Ada Call Format

```

WITH KAVDEF;

KAV_OUT_MAP ([STATUS => status,]
              SGM_ENTRY => sgm_entry,
              PAGE_COUNT => page_count,
              BUS_ADDRESS => bus_address,
              VIRTUAL_ADDRESS => virtual_address,
              AM_CODE => am_code,
              MAP_FUNCTIONS => map_functions);

```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>sgm_entry</i> :	out	INTEGER;
<i>page_count</i> :	in	INTEGER;
<i>bus_address</i> :	in	SYSTEM.ADDRESS;
<i>virtual_address</i> :	out	SYSTEM.ADDRESS;
<i>am_code</i> :	in	INTEGER;
<i>map_functions</i> :	in	INTEGER;

---

## C Call Format

```

#include $vaxelnc

#include "eln$:kavdef.h"

int kav$out_map ([status],
                 entry,
                 page_count,
                 bus_address,
                 virtual_address,
                 am_code,
                 map_functions)

```

## KAV\$OUT\_MAP

### argument information

int	<i>*status;</i>
int	<i>*entry;</i>
int	<i>page_count;</i>
int	<i>bus_address;</i>
void	<i>**virtual_address;</i>
int	<i>am_code;</i>
int	<i>map_functions;</i>

---

### FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$OUT_MAP ([status],  
                  entry,  
                  %VAL(page_count),  
                  %VAL(bus_address),  
                  virtual_address,  
                  %VAL(am_code),  
                  %VAL(map_functions))
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>entry</i>
INTEGER*4	<i>page_count</i>
INTEGER*4	<i>bus_address</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>am_code</i>
INTEGER*4	<i>map_functions</i>

---

## Pascal Call Format

```

INCLUDE $KAVDEF;
KAV$OUT_MAP ([STATUS := status,]
             entry,
             page_count,
             bus_address,
             virtual_address,
             am_code,
             map_functions)

```

## argument information

<i>status</i> :	INTEGER;
<i>entry</i> :	INTEGER;
<i>page_count</i> :	INTEGER;
<i>bus_address</i> :	INTEGER;
<i>virtual_address</i> :	^ANYTYPE;
<i>am_code</i> :	INTEGER;
<i>map_functions</i> :	INTEGER;

---

## Arguments

### ***status***

Usage:	Longword (unsigned)
VAX Type:	cond_value
Access:	Write only
Mechanism:	Reference

Receives the completion status.

## KAV\$OUT\_MAP

### ***entry***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Returns the SGM entry that corresponds to the first page of the KAV30 S0 space that you want to map to the VMEbus or VSB.

### ***page\_count***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of successive 64K byte pages of KAV30 S0 space that you want to map to the VMEbus or VSB.

### ***bus\_address***

Usage: Longword  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the base physical address of the VMEbus or VSB address space. This base physical address is the start of the first 64K byte page that this service maps to the VMEbus or VSB address space.

### ***virtual\_address***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Returns the KAV30 S0 space virtual address that corresponds to the base physical address (on the VMEbus or VSB) of the 64K byte page of memory that this service maps to the VMEbus or VSB.

***am\_code***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

This code specifies the VMEbus or VSB addressing mode. If you are mapping to the VMEbus, specify one of the following values:

Constant	Value	Explanation
KAV\$K_USER_16	29 <sub>16</sub>	Uses short addressing (16 address lines) in VMEbus user mode
KAV\$K_USER_24	39 <sub>16</sub>	Uses standard addressing (24 address lines) in VMEbus user mode
KAV\$K_USER_32	09 <sub>16</sub>	Uses extended addressing (32 address lines) in VMEbus user mode
KAV\$K_SUPER_16	2D <sub>16</sub>	Uses short addressing (16 address lines) in VMEbus supervisor mode
KAV\$K_SUPER_24	3D <sub>16</sub>	Uses standard addressing (24 address lines) in VMEbus supervisor mode
KAV\$K_SUPER_32	0D <sub>16</sub>	Uses extended addressing (32 address lines) in VMEbus supervisor mode

If you are mapping to the VSB, specify one of the following values:

Constant	Value	Explanation
KAV\$K_SYS	3	Uses the SYSTEM address space
KAV\$K_IO	2	Uses the I/O address space
KAV\$K_ALT	1	Uses the ALTERNATE address space
KAV\$K_VSB_IACK	0	VSB IACK

If you want to specify an address modifier code with a value other than one of these values, pass the value directly in the *am\_code* argument. See the *VMEbus Specification* and *The VME Subsystem Bus (VSB) Specification* for other values that you can pass in the *am\_code* argument.

The address modifier code that you specify in this argument must be the same as the address modifier code of the VMEbus or VSB device to which this 64K byte page of KAV30 S0 space is mapped.

# KAV\$OUT\_MAP

## map\_functions

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the following information that controls the mapping operation:

- Mapping destination—whether you want to map pages of KAV30 S0 space to the VMEbus or VSB address space
- Byte swapping mode

Specify one or more of the following modifiers:

KAV\$M_NO_RETRY	When you specify this modifier, the KAV30 performs one retry. When you do not specify this modifier, the KAV30 performs 29 successive retries. If the access does not succeed after 29 retries, the KAV30 signals that an access failure occurred. The bus arbitration failures and bus timeouts cause accesses to fail.
KAV\$M_MODE_0_SWAP	Performs mode 0 swapping.
KAV\$M_MODE_2_SWAP	Performs mode 2 swapping.
KAV\$M_MODE_3_SWAP	Performs mode 3 operations.
KAV\$M_VME	Maps KAV30 S0 space to the VMEbus.
KAV\$M_VSB	Maps KAV30 S0 space to the VSB.
KAV\$M_WRT_PROT	Sets write-protection on the page of system RAM that you want to map to the VMEbus.

You must ensure that the modifiers you specify do not conflict with each other. For example, do not specify the KAV\$M\_MODE\_3\_SWAP and KAV\$M\_MODE\_0\_SWAP modifiers together.

---

## Status Values

KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KAV\$_NO_MEMORY	There is no physical memory available.
KER\$_NO_PORT	There are no free SGM entry ports. Unmap one or more SGM entries and retry the call.
KAV\$_NO_VIRTUAL	There is no virtual address space available.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$BUS_BITCLR	KAV\$BUS_WRITE
KAV\$BUS_BITSET	KAV\$UNMAP
KAV\$BUS_READ	



### Examples

- See the examples in the descriptions of the KAV\$BUS\_READ and KAV\$BUS\_BITCLR services
- The following code is an example program that calls the KAV\$OUT\_MAP service:

```
MODULE ex_INT_VME ;
{++}
{
{ Facility:      KAV30 VAXELN System Services programming example.
{
{ Description:   This is an example program demonstrating the calling
{                procedures for the following KAV System Services:
{                1. KAV$OUT_MAP   (Map KAV addr space to VMEbus)
{                2. KAV$UNMAP     (Un-map ..... )
{                3. KAV$BUS_READ  (Read VMEbus address)
{
{ Abstract:     This program can be used to test the handling of
{                VMEbus interrupts. It does this by faking an IACK
{                cycle on the VMEbus.
{
{ Language:     Epascal; Version 4.2
{
{ Notes: (1)    In the interests of program clarity, no error
{                checking has been included.
{
{--}

INCLUDE $KAVDEF ;      { (KAV30 definitions)      }

TYPE
    comm_region = RECORD
        int_count      : INTEGER; { Interrupt Service Rtn}
        signal_count   : INTEGER;
        bus_page_ptr   : ^ANYTYPE;
    END;
    byte = [byte] 0..255;

VAR
    wait_count : INTEGER;
    device_obj : DEVICE;
```

```

{          *****          }
{          *** Interrupt Service Routine ***      }
{          *****          }

INTERRUPT_SERVICE vme_int_isr( register_ptr : ^anytype;
                               region_ptr   : ^comm_region );

VAR
  rd_int_count      : INTEGER;
  rd_signal_count   : INTEGER;
  value             : byte;
  temp_value        : INTEGER;
  page_ptr          : ^ANYTYPE;
  status            : INTEGER;

BEGIN
  { fake an IACK cycle to prevent handling of a vectorized
    interrupt }

  page_ptr      := READ_REGISTER(region_ptr^.bus_page_ptr);
  temp_value    := page_ptr :: INTEGER;
  temp_value    := temp_value + %xc;
  page_ptr      := temp_value :: ^ANYTYPE;

  REPEAT
    {          =====          }
    {          === BUS_READ ===  }
    {          =====          }

    KAV$BUS_READ( STATUS      := status,
                  DATA_TYPE  := KAV$K_BYTE,
                  VIRTUAL_ADDRESS := page_ptr,
                  BUFFER      := ADDRESS(value),
                  COUNT        := 1 );

  UNTIL ODD(status);

  SIGNAL_DEVICE( device_number := 0);

END;    {          *** End of Interrupt Service Routine ***      }

{ This process is activated by the ISR (see above) when it }
{ services an interrupt.                                   }

PROCESS_BLOCK server_process(region_ptr : ^comm_region);

VAR
  rd_int_count : INTEGER;
  text_string  : VARYING_STRING(80);

BEGIN
  text_string := 'KAV30 example program interrupted' ;

```

## KAV\$OUT\_MAP

```

        REPEAT
            WAIT_ANY( device_obj );
            rd_int count := READ_REGISTER( region_ptr^.int_count);
            WRITELN( text_string)
        UNTIL FALSE

END;

{
            *****
            ***  Main process  ***
            *****
}

PROGRAM INT_VME (INPUT, OUTPUT);

VAR
    status           : INTEGER;
    p_id             : PROCESS;
    i                : INTEGER;
    rd_int_count      : INTEGER;
    rd_signal_count   : INTEGER;
    irq_lvl           : INTEGER;
    int_vec           : INTEGER;
    vme_int_bitmask   : INTEGER;
    region_ptr        : ^comm_region;
    device_name       : VARYING STRING(31);
    temp_page_ptr     : ^ANYTYPE;
    sgm_entry         : INTEGER;

BEGIN

    device_name := PROGRAM_ARGUMENT(4) ; { Get the device name from
                                         { the EBUILD '.DAT' file.

    {
                                         ===== }
    {
                                         ===  IN_MAP  === }
    {
                                         ===== }

    KAV$OUT_MAP( STATUS           := status,
                  ENTRY           := sgm_entry,
                  PAGE_COUNT      := 1,
                  BUS_ADDRESS     := 0,
                  VIRTUAL_ADDRESS := temp_page_ptr,
                  AM_CODE         := %X80,
                  MAP_FUNCTIONS   := KAV$M_MODE_3_SWAP + KAV$M_VME );

    { Create Device object }

    CREATE_DEVICE( device_name,
                  device_obj,
                  SERVICE_ROUTINE := vme_int_isr,
                  REGION          := region_ptr,
                  STATUS           := status ) ;

```

```

WRITE_REGISTER( region_ptr^.int_count,    0);
WRITE_REGISTER( region_ptr^.signal_count, 0);
WRITE_REGISTER( region_ptr^.bus_page_ptr :: INTEGER,
                temp_page_ptr :: INTEGER );

{ Create the server process }

CREATE_PROCESS( p_id,
               server_process,
               region_ptr,
               STATUS := status);

WAIT_ANY( p_id, STATUS := status );

{                                     ===== }
{                                     ===  UN-MAP  === }
{                                     ===== }
KAV$UNMAP( STATUS      := status,
          ENTRY        := sgm_entry,
          PAGE_COUNT   := 1,
          VIRTUAL_ADDRESS := temp_page_ptr,
          UNMAP_FUNCTIONS := KAV$M_OUT );

END; {          *** End of main process ***          }
END;  {          *** End of INT_VME example program ***          }

```

---

## KAV\$QUE\_AST

Queues an AST for delivery to a process.

This service removes an ASB from the AST pending queue<sup>1</sup> and places it on the AST process<sup>2</sup> queue.

Before you call this service for a particular device code, call the KAV\$DEF\_AST service (to allocate an AST queue for the device code) and the KAV\$SET\_AST service to place an ASB for the device code in the AST pending queue.

This service uses the device code that the KAV\$DEF\_AST service returns to ensure that it delivers the correct AST for a device code.

See Section 3.1 for more information on ASTs.

---

### Ada Call Format

```
WITH KAVDEF;  
KAV_QUE_AST ([STATUS => status,]  
             DEVICE_CODE => device_code);
```

### argument information

<i>status</i> :	out	UNSIGNED_LONGWORD;
<i>device_code</i> :	in	UNSIGNED_LONGWORD;

---

### C Call Format

```
#include $vaxelnc  
#include "eln$:kavdef.h"  
int KAV$QUE_AST ([status],  
                 device_code)
```

---

<sup>1</sup> The pending queue is the queue that contains the ASBs that are waiting for an event that will cause an AST to be delivered.

<sup>2</sup> The process queue is the queue of ASTs for which an AST has been delivered, but an AST routine has not been executed.

**argument information**

int	<i>*status;</i>
int	<i>device_code;</i>

---

**FORTRAN Call Format**

```

INCLUDE 'ELN$:KAVDEF.FOR'
CALL KAV$QUE_AST ([status],
                  %VAL(device_code))

```

**argument information**

INTEGER*4	<i>status</i>
INTEGER*4	<i>device_code</i>

---

**Pascal Call Format**

```

INCLUDE $KAVDEF;
KAV$QUE_AST ([STATUS := status,]
             device_code)

```

**argument information**

<i>status</i> :	INTEGER;
<i>device_code</i> :	INTEGER;

---

Arguments

*status*

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Receives the completion status.

*device\_code*

Usage: Longword (unsigned)  
VAX Type: Read only  
Access: Value  
Mechanism: None

Specifies the device code that identifies the AST you want to queue. The KAV\$DEF\_AST service returns the device code when you define the AST.

---

Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_SUCCESS	The operation is successful.

---

**Related Services****KAV\$CLR\_AST****KAV\$SET\_AST****KAV\$DEF\_AST**

---

**Examples**

See the programs listed in Appendix C for examples of KAV\$QUE\_AST service calls.



---

## KAV\$RTC

Performs all the real-time clock functions, using the KAV30 calendar/clock.

This service allows you to configure the following real-time clock functions:

- Alarm
- Periodic alarm
- Read and write alarm
- Read and write calendar
- Read and write real-time clock RAM
- Read and write timesave RAM
- 16-bit timer functions

For more information about these functions, see Section 3.3.

The calendar/clock can operate in either 12-hour mode or 24-hour mode. You specify the mode when you write calendar information into the calendar/clock. Use the KAV\$M\_RTC\_12\_HOUR modifier to specify 12-hour mode, or use the KAV\$M\_RTC\_24\_HOUR modifier to specify 24-hour mode. You must initialize the calendar/clock to either 12-hour mode or 24-hour mode when you initialize the system. In 12-hour mode, the most significant bit in the *hours* byte indicates whether the time is A.M. or P.M. When you read or write the calendar, a 0 in this bit indicates an A.M. time while a 1 indicates a P.M. time.

The programs calling this service pass a modifier that indicates the function to be performed by the service. The programs also pass a buffer that contains the information required to perform the function. The buffer is a byte-oriented buffer. Figure 4–1 gives an example of a buffer that passes the date Sunday, March 17, 1991 and time 10:53:25.39 P.M. to the real-time clock.

The year value in the example is the offset from the base year. The base year value is 1990. Therefore, a 00 year value corresponds to the year 1990, a 01 year value corresponds to the year 1991, and so on. The following table explains the day of week value in the example.

Value	Day
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

You can specify an AST routine that executes when the current time and date are equal to the alarm time and date, or when a timer interval expires.

**Figure 4–1 Programming the Real-Time Clock**

7 ..... 4 3 ..... 0

3	9
2	5
5	3
1	0
1	7
0	3
0	1
7	6
0	0
0	7

Hundreds = 39

Seconds = 25

Minutes = 53

Hours = 10 (but MSB =1  
for PM Times)

Date = 17

Month = 03

Year = 1991

Julian Date = 76

Julian Date (Hundreds) = 0

Day of Week = 7 (Sunday)

Ada Call Format

```
WITH KAVDEF;
KAV_RTC ([STATUS => status,]
         RTC_FUNCTIONS => rtc_functions,
         BUFFER => buffer,
         LENGTH => length,
         [AST_ADDR => ast_addr,]
         [AST_PARAM => ast_param]);
```

argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_ TYPE;
<i>rtc_functions</i> :	in	INTEGER;
<i>buffer</i> :	in	SYSTEM.ADDRESS;
<i>length</i> :	in	INTEGER;
<i>ast_addr</i> :	in	INTEGER;
<i>ast_param</i> :	in	INTEGER;

---

C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"
int kav$rtc ([status],
            rtc_functions,
            buffer,
            length,
            [ast_addr],
            [ast_param])
```

**argument information**

int	<i>*status;</i>
int	<i>rtc_functions;</i>
void	<i>*buffer;</i>
int	<i>length;</i>
void	<i>*ast_addr( );</i>
int	<i>ast_param;</i>

---

**FORTRAN Call Format**

```

INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$RTC  ([status],
               %VAL(rtc_functions),
               buffer,
               %VAL(length),
               [ast_addr],
               [%VAL(ast_param)])

```

**argument information**

INTEGER*4	<i>status</i>
INTEGER*4	<i>rtc_functions</i>
INTEGER*4	<i>buffer</i>
INTEGER*4	<i>length</i>
INTEGER*4	<i>ast_addr</i>
INTEGER*4	<i>ast_param</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$RTC ([STATUS := status,]  
         rtc_functions,  
         buffer,  
         length,  
         [,ast_addr]  
         [,ast_param])
```

## argument information

<i>status</i> :	INTEGER;
<i>rtc_functions</i> :	INTEGER;
<i>buffer</i> :	^ANYTYPE;
<i>length</i> :	INTEGER;
<i>ast_addr</i> :	^ANYTYPE;
<i>ast_param</i> :	INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

### ***rtc\_functions***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the function that you want to perform.

You must specify one or more of the following modifiers:

KAV\$M_RTC_TMR_0	Performs a function on timer 0.
KAV\$M_RTC_TMR_1	Performs a function on timer 1.

---

### Warning

---

Timer 1 is reserved for the VMEbus timeout timer. It is set up by the KAV30 kernel when you boot the KAV30. Digital strongly recommends that you do not change or modify this timer.

---

KAV\$M_LOAD_TMR_CNT	<p>Loads a value into one of the timers. If you want to load the value into timer 0, also specify the KAV\$M_RTC_TMR_0 modifier. If you want to load a value into timer 1, also specify the KAV\$M_RTC_TMR_1 modifier. You must specify the AST routine address and parameters when you load the timer. The <i>buffer</i> argument specifies the value that you want to load into the timer register, along with the timer resolution.</p> <p>If you specify the KAV\$M_LOAD_TMR_CNT modifier, you cannot specify the KAV\$M_READ_TMR_CNT modifier.</p>
KAV\$M_START_TMR	<p>Starts the timer. The timer starts decrementing the value in the timer register. When the value in the register reaches zero, the KAV30 software issues an AST.</p> <p>If you want to start timer 0, also specify the KAV\$M_RTC_TMR_0 modifier. If you want to start timer 1, also specify the KAV\$M_RTC_TMR_1 modifier.</p> <p>If you specify this modifier, you cannot also specify the KAV\$M_STOP_TMR modifier.</p>

## KAV\$RTC

KAV\$M\_STOP\_TMR

Stops the timer. This service does not issue an AST when the timer stops—it issues an AST only when the number in the timer register reaches 0.

If you want to stop timer 0, also specify the KAV\$M\_RTC\_TMR\_0 modifier. If you want to stop timer 1, also specify the KAV\$M\_RTC\_TMR\_1 modifier.

If you specify this modifier, you cannot also specify the KAV\$M\_START\_TMR modifier.

KAV\$M\_READ\_TMR\_CNT

Reads the value stored in the timer register. Read the value in the timer register only when you also specify the KAV\$M\_STOP\_TMR modifier. That is, stop the timer before reading the value in the register.

To read the value in timer 0, also specify the KAV\$M\_RTC\_TMR\_0 modifier. To read the value in timer 1, also specify the KAV\$M\_RTC\_TMR\_1 modifier.

If you specify this modifier, you cannot also specify the KAV\$M\_LOAD\_TMR\_CNT modifier.

KAV\$M\_RESET\_TMR

Resets the calendar/clock.

To reset timer 0, also specify the KAV\$M\_RTC\_TMR\_0 modifier. To reset timer 1, also specify the KAV\$M\_RTC\_TMR\_1 modifier.

KAV\$M\_PERIODIC

Queues an AST repeatedly at the interval specified by the *buffer* argument.

KAV\$M\_ALARM

Delivers an AST at the time specified by an *rtc\_functions* argument specifying the KAV\$M\_WRITE\_ALARM modifier.

KAV\$M\_READ\_ALARM

Reads the alarm setting and returns the value in the *buffer* argument.

KAV\$M_WRITE_ALARM	Sets the alarm time to the value specified in the <i>buffer</i> argument. When the calendar/clock time becomes equal to the alarm time, the KAV30 kernel queues an AST to the AST pending queue when you are writing the alarm time for the first time. When you are not writing for the first time, call this service again with the KAV\$M_ALARM modifier.
KAV\$M_READ_CALENDAR	Reads the current calendar date and returns the value in the <i>buffer</i> argument.
KAV\$M_WRITE_CALENDAR	Sets the calendar date to the date specified in the <i>buffer</i> argument.
KAV\$M_RTC_12_HOUR	Sets the calendar/clock to operate in 12-hour mode. You can specify the KAV\$M_RTC_12_HOUR modifier only when you also specify the KAV\$M_WRITE_CALENDAR modifier. If you do not specify either the KAV\$M_RTC_12_HOUR modifier or the KAV\$M_RTC_24_HOUR modifier, the clock mode remains unchanged.
KAV\$M_RTC_24_HOUR	Sets the calendar/clock to operate in 24-hour mode. You can specify the KAV\$M_RTC_24_HOUR modifier only when you also specify the KAV\$M_WRITE_CALENDAR modifier. If you do not specify either the KAV\$M_RTC_12_HOUR modifier or the KAV\$M_RTC_24_HOUR modifier, the clock mode remains unchanged.
KAV\$M_READ_TIMESAVE	Reads the value stored in the timesave RAM and returns it in the <i>buffer</i> argument.
KAV\$M_WRITE_TIMESAVE	Writes the data specified in the <i>buffer</i> argument into timesave RAM.
KAV\$M_READ_RTCRAM	Reads up to 31 bytes of data from the calendar/clock battery backed-up RAM and returns the data in the <i>buffer</i> argument. The low-order word of this modifier specifies the number of bytes to be read. The high-order word specifies the base address in the battery backed-up RAM of the data to read.



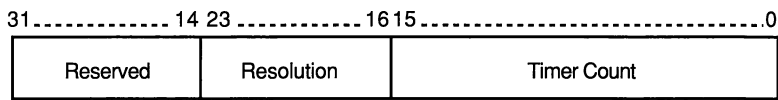
KAV\$M_WRITE_RTCRAM	Writes up to 31 bytes of data from the <i>buffer</i> argument to the calendar/clock battery backed-up RAM. The low-order word of this modifier specifies the number of bytes to be read. The high-order word specifies the base address in the battery backed-up RAM of the data to be read.
KAV\$M_RTC_HOLD_TMR	<p>Puts the timer on hold. The timer stops decrementing, and the value remains in the timer register.</p> <p>If you want to hold timer 0, also specify the KAV\$M_RTC_TMR_0 modifier. If you want to hold timer 1, also specify the KAV\$M_RTC_TMR_1 modifier.</p>
KAV\$M_RTC_RESTART_TMR	<p>Restarts the timer after a previous call to this service had put the timer on hold.</p> <p>If you want to restart timer 0, also specify the KAV\$M_RTC_TMR_0 modifier. If you want to restart timer 1, also specify the KAV\$M_RTC_TMR_1 modifier.</p>

**buffer**

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Modify  
Mechanism: Reference

Specifies the address of a buffer. The address contains the value that you want to write to or read from the calendar/clock. The value of the *buffer* argument depends on the function that you specify in the *rtc\_functions* argument, as follows:

- If the value of the *rtc\_functions* argument is KAV\$M\_LOAD\_TMR\_COUNT, the buffer is a longword whose layout is shown in the following diagram:



The service uses only the first 24 bits, the high-order byte must be zero.  
Bits <15..0> contain the value that the service loads into the timer register.  
Bits <23..16> specify the timer resolution.

Specify one of the following values for bits <23..16>:

KAV\$K_RTC_100NS	Specifies that the timer register decrements every 100 ns
KAV\$K_RTC_400NS	Specifies that the timer register decrements every 400 ns
KAV\$K_RTC_93US	Specifies that the timer register decrements every 93.5 $\mu$ s
KAV\$K_RTC_1MS	Specifies that the timer register decrements every 1 ms
KAV\$K_RTC_10MS	Specifies that the timer register decrements every 10 ms
KAV\$K_RTC_100MS	Specifies that the timer register decrements every 100 ms
KAV\$K_RTC_1000MS	Specifies that the timer register decrements every 1000 ms

- If the value of the *rtc\_functions* argument is KAV\$M\_READ\_TMR\_COUNT, the *buffer* argument returns the 16-bit value that the timer register contains.
- If the value of the *rtc\_functions* argument is KAV\$M\_PERIODIC, specify one of the following values for the *buffer* argument:

KAV\$K_PER_1MS	Queues an AST every 1 ms
KAV\$K_PER_10MS	Queues an AST every 10 ms
KAV\$K_PER_100MSEC	Queues an AST every 100 ms
KAV\$K_PER_1SEC	Queues an AST every 1 s
KAV\$K_PER_10SEC	Queues an AST every 10 s
KAV\$K_PER_60SEC	Queues an AST every 60 s

To reset the periodic queuing of ASTs, specify 0 in the *buffer* argument.

- When the value of the *rtc\_functions* argument is KAV\$M\_ALARM, specify one of the following values for the *buffer* argument:

KAV\$K_ALR_SECOND	Performs an alarm check every 1 second
KAV\$K_ALR_MINUTE	Performs an alarm check every 1 min
KAV\$K_ALR_HOUR	Performs an alarm check every 1 hour
KAV\$K_ALR_DOM	Performs an alarm check on one day every month

KAV\$RTC

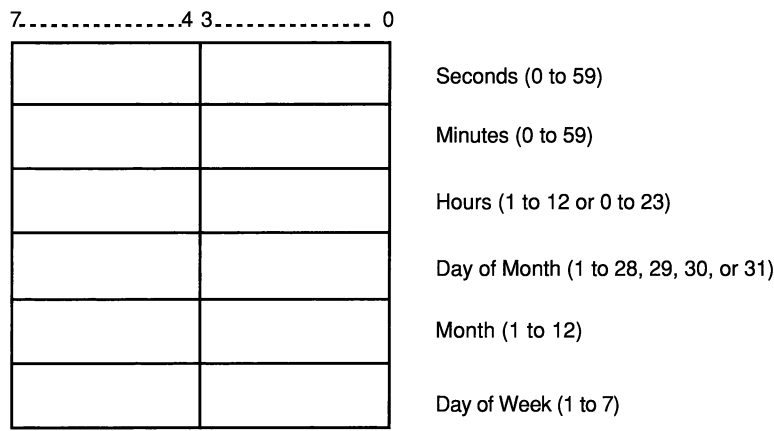
- KAV\$K\_ALR\_MONTH

Performs an alarm check every month
- KAV\$K\_ALR\_DOM

Performs an alarm check on one day every week

Before you execute this service you must first use the KAV\$M\_WRITE\_ALARM service to set the alarm date and time.

- If the value of the *rtc\_functions* argument is KAV\$M\_READ\_ALARM or KAV\$M\_WRITE\_ALARM, the *buffer* argument is 6 bytes long and contains the alarm information in BCD format, as shown in the following diagram.



When the calendar/clock time becomes equal to the time that the *buffer* argument specifies, the KAV30 kernel queues an AST to the AST pending queue.

To set up an alarm, follow these steps:

1. Call this service to write the alarm time.
2. Call this service with KAV\$M\_ALARM specified as an argument.

You cannot combine both actions in one service call. To reset the alarm, specify 0 in the *buffer* argument.

- If the value of the *rtc\_functions* argument is KAV\$M\_READ\_CALENDAR or KAV\$M\_WRITE\_CALENDAR, the *buffer* argument is ten bytes long and it contains the calendar information in BCD format, as shown in the

following diagram:

7..... 4 3..... 0


Hundreds (0 to 99)

Seconds (0 to 59)

Minutes (0 to 59)

Hours (1 to 12 or 0 to 23)

Day of Month (1 to 28, 29, 30, or 31)

Month (1 to 12)

Years (0 to 99)

Julian Date (1 to 99)

Julian Date (0 to 3)

Day of Week (1 to 7)

- If the value of the *rtc\_functions* argument is KAV\$M\_READ\_TIMESAVE or KAV\$M\_WRITE\_TIMESAVE, the *buffer* argument is 5 bytes long and it contains the timesave information in BCD format, as shown in the following diagram:

7..... 4 3..... 0


Seconds (0 to 59)

Minutes (0 to 59)

Hours (1 to 12 or 0 to 23)

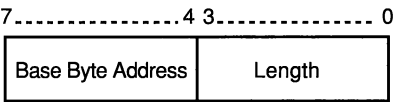
Day of Month (1 to 28, 29, 30, or 31)

Month (1 to 12)

KAV\$RTC

The date and time is automatically written when there is a power failure. You can read this date and time when you boot the system after a power failure to detect when the power failure occurred.

- If the value of the *rtc\_functions* argument is KAV\$M\_READ\_RTCRAM or KAV\$M\_WRITE\_RTCRAM, the *buffer* argument contains the data that you want to read from or write to the calendar/clock battery backed-up RAM. The *length* argument specifies the amount of data to read or write and the base address in the battery backed-up RAM, as follows:



The base byte address is an offset into the battery backed-up RAM. The length is the number of bytes.

**Note**

If the *rtc\_functions* argument specifies the value KAV\$M\_RTC\_12\_HOUR, the high-order bit in the *hours* byte is the A.M./P.M. bit—a zero indicates A.M. and a one indicates P.M.

***length***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the length (in bytes) of the buffer whose address you specify in the *buffer* argument.

***ast\_addr***

Usage: Procedure entry mask  
VAX Type: procedure  
Access: Read only  
Mechanism: Reference

Specifies the address of the AST routine, which the service calls when one or more of the following occur:

- A timeout
- An alarm

- A periodic alarm

See Section 3.1 for more information about ASTs.

### ***ast\_param***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Reference

Specifies a parameter that this service passes to the AST routine.

See Section 3.1 for more information about ASTs.

## **Status Values**

KAV30\$_ALR_ACTIVE	The alarm interrupts are active.
KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_PER_ACTIVE	The periodic interrupts are active.
KAV30\$_TMR_BUSY	The timer is busy.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KER\$_SUCCESS	The operation is successful.
KER\$_TIME_NOT_SET	The timer is not set.

## Examples

The following code is an example program that calls the KAV\$RTC service:

```
/*
 * Facility:      KAV30 VAXELN System Services programming example.
 *
 * Description:   This is an example program demonstrating the calling
 *                procedure for the following KAV System Service:
 *                KAV$RTC      (RealTime Clock functions)
 *
 * Abstract:      This program reads the RTC calendar. Simple as that.
 *
 * Language:      Vax C; Version 3.1
 *
 * Notes: (1)     In the interests of program clarity, no error checking has
 *                been included.
 */

#include      stdio
#include      $vaxelnc
#include      <eln$:kavdef.h>          /* KAV30 definitions file.      */

#define      BUFFER_LENGTH  100

/* =====      Main Program      ===== */
main()
{
    int          status, i ;
    unsigned long   rtc functions ;
    unsigned char   buffer[BUFFER_LENGTH] ;
    void   ast_routine() ;

    printf("\n\nKAV30 Test program for RTC System Service call\n\n") ;

    /*
     *          Read the CALENDAR from RTC
     *          =====
     */
    rtc_functions = KAV$M_READ_CALENDAR ;
    KAV$RTC      (  &status,
                   rtc_functions,
                   &buffer[0],
                   10,
                   &ast_routine, 0 ) ;

    printf("\n\nEND OF KAV30 Test program for RTC. \n\n") ;
} /*      end      -program-      */
```

```
void ast_routine() /* Dummy AST routine (NOT USED) */
{
  int i ;
  i = 1234 ;
  return ;
}
```



---

# KAV\$RW\_BBAM

Writes data to or reads data from the KAV30 battery backed-up RAM.

You can write data into the battery backed-up RAM by passing a buffer to this service. You can read data from the battery backed-up RAM by reading data from the buffer that this service returns. You specify a modifier when you call this service, which indicates whether you want to read from or write to the battery backed-up RAM.

See Section 3.5 for information about the KAV30 battery backed-up RAM.

---

## Ada Call Format

```
WITH KAVDEF;

KAV_RW_BBAM ([STATUS => status,]
              BUFFER_ADDRESS => buffer_address,
              BUFFER_LENGTH => buffer_length,
              BBAM_OFFSET => bbam_offset,
              BBAM_FUNCTIONS => bbam_functions;
```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>buffer_address</i> :	in	SYSTEM.ADDRESS;
<i>buffer_length</i> :	in	INTEGER;
<i>bbam_offset</i> :	in	INTEGER;
<i>bbam_functions</i> :	in	INTEGER;

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$rw_bbam ([status],
                 buffer_address,
                 buffer_length,
                 bbam_offset,
                 bbam_functions)
```

### argument information

int	<i>*status;</i>
void	<i>*buffer_address;</i>
int	<i>buffer_length;</i>
int	<i>bbam_offset;</i>
int	<i>bbam_functions;</i>

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$RW_BBAM ([status],
                  buffer_address,
                  %VAL(buffer_length),
                  %VAL(bbam_offset),
                  %VAL(bbam_functions))
```

## KAV\$RW\_BBRAM

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>buffer_address</i>
INTEGER*4	<i>buffer_length</i>
INTEGER*4	<i>bbram_offset</i>
INTEGER*4	<i>bbram_functions</i>

---

### Pascal Call Format

```
INCLUDE $KAVDEF;  
  
KAV$RW_BBRAM ([STATUS := STATUS,  
               BUFFER_ADDRESS := buffer_address,  
               BUFFER_LENGTH := buffer_length,  
               BBRAM_OFFSET := bbram_offset,  
               FUNCTION := bbram_functions)
```

### argument information

<i>status</i> :	INTEGER;
<i>buffer_address</i> :	^ANYTYPE;
<i>buffer_length</i> :	INTEGER;
<i>bbram_offset</i> :	INTEGER;
<i>bbram_functions</i> :	INTEGER;

---

### Arguments

#### ***status***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Write only  
Mechanism: Reference

Receives the completion status.

***buffer\_address***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Modify  
 Mechanism: Reference

Supplies the address of the buffer that this service uses. If the value of the *function* argument is KAV\$K\_BBR\_READ, this service reads data from the battery backed-up RAM and writes it to the buffer. If the value of the *function* argument is KAV\$K\_BBR\_WRITE, this service writes the data in the buffer to the battery backed-up RAM.

***buffer\_length***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Supplies the length of the buffer whose address is supplied by the *buffer\_address* argument. The maximum buffer length is 22K bytes.

***bbam\_offset***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies an offset into the 22K byte battery backed-up RAM area.

***bbam\_functions***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Supplies a code that determines whether this service writes the data in the buffer to the battery backed-up RAM or reads the data from the battery backed-up RAM and writes it to the buffer. Specify one of the following values:

KAV\$K_BBR_READ	Reads data from the battery backed-up RAM.
KAV\$K_BBR_WRITE	Writes data to the battery backed-up RAM.

### Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

### Examples

The following code is an example program that calls the KAV\$RW\_BBRAM service:

```
/*
 * Facility:      KAV30 VAXELN System Services programming example.
 *
 * Description:   This is an example program demonstrating the calling
 *                procedure for the following KAV System Service:
 *                KAV$RW_BBRAM      (Read/Write BB-Ram)
 *
 * Abstract:      This program reads and writes to the KAV30's Battery-
 *                backed-up RAM.
 *                Firstly; it initializes a buffer with known data which
 *                it writes to the BB-Ram.
 *                Secondly, it reads the BB-Ram and checks the data read
 *                against the data written.
 *
 * Language:      Vax C; Version 3.1
 *
 * Notes: (1)     In the interests of program clarity, no error checking has
 *                been included.
 */

#include  stdio
#include  $vaxelnc
#include  <eln$:kavdef.h>      /* KAV30 definitions file.      */

#define   BUFFER_SIZE      32      /* Test buffer size.      */
#define   BUFFER_OFFSET    0      /* Test buffer offset.    */
```

```

/* ===== Main Program ===== */
main()
{
    int          status, i ;
    unsigned char buffer_in[BUFFER_SIZE],
                buffer_out[BUFFER_SIZE] ;

    printf("\n\nKAV30 Test program for RW_BBRAM System Service call\n\n") ;

    /*
     *   Initialize the test buffer with a simple incrementing sequence,
     *   and clear out the output buffer.
     */
    for (i = 0; i < BUFFER_SIZE; i++ )
    {   buffer_in[i] = i+1 ;   buffer_out[i] = 0 ;   } ;

    /*
     *           WRITE to BB-RAM
     *   =====
     */
    KAV$RW_BBRAM(  &status,
                   &buffer_in[0],
                   BUFFER_SIZE,
                   BUFFER_OFFSET,
                   KAV$K_BBR_WRITE ) ;

    /*
     *           READ from BB-RAM
     *   =====
     */
    KAV$RW_BBRAM(  &status,
                   &buffer_out[0],
                   BUFFER_SIZE,
                   BUFFER_OFFSET,
                   KAV$K_BBR_READ ) ;

    /*
     *   Lastly, compare the two buffers - they should be identical.
     */
    for (i = 0, status = 0; i < BUFFER_SIZE; i++ )
    {   if (buffer_in[i] != buffer_out[i]) status++ ;   }   /* Flag an error */

    if (status != 0)
    {   printf("Total of %d errors found.\n", status) ;   } else
    {   printf("No errors found in data read back from BBRAM.\n") ;   } ;
    if (status != 0) { printf("Total of %d errors found.\n", status) ; }
    else             { printf("No errors found in data read back.\n") ; } ;

    printf("\n\nEND OF KAV30 Test program for RW_BBRAM \n\n") ;
} /*   end   -program-   */

```

---

## KAV\$SET\_AST

Places an ASB in the AST pending<sup>1</sup> queue.

You must call the KAV\$DEF\_AST service before you call this service. The KAV\$DEF\_AST service returns a device code that associates an AST queue with a particular device event. This service uses this device code to ensure that it places the ASB in the correct AST queue.

The KAV30 kernel deletes entries from the queue once it has queued the entries to a process, unless the *ast\_functions* argument specifies the value KAV\$M\_REPEAT. In that case, the KAV30 kernel requeues the AST to the pending queue, immediately after it has delivered the AST. You can call the KAV\$CLR\_AST service to cancel the repeating ASTs.

See Section 3.1 for more information on ASTs.

---

### Ada Call Format

```
WITH KAVDEF;  
  
KAV_SET_AST ([STATUS => status,]  
             AST_ADDR => ast_addr,  
             [AST_PARAM => ast_param,]  
             AST_FUNCTIONS => ast_functions,  
             DEVICE_CODE => device_code);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>ast_addr</i> :	in	SYSTEM.ADDRESS;
<i>ast_param</i> :	in	INTEGER;
<i>ast_functions</i> :	in	INTEGER;
<i>device_code</i> :	in	INTEGER;

---

<sup>1</sup> The pending queue is the queue of ASBs that is waiting for an event that will cause an AST to be delivered. The process queue is the queue of ASTs for which an AST has been delivered, but the AST routine has not yet been executed.

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$set_ast ([status],
                 ast_addr,
                 [ast_param],
                 ast_functions,
                 device_code)
```

### argument information

int	<i>*status</i> ;
void	<i>*ast_addr</i> ( );
int	<i>ast_param</i> ;
int	<i>ast_functions</i> ;
int	<i>device_code</i> ;

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$SET_AST ([status],
                  ast_addr,
                  [%VAL(ast_param)],
                  %VAL(ast_functions),
                  %VAL(device_code))
```



# KAV\$SET\_AST

## argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>ast_addr</i>
INTEGER*4	<i>ast_param</i>
INTEGER*4	<i>ast_functions</i>
INTEGER*4	<i>device_code</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;

KAV$SET_AST ([STATUS := status,]
             ast_addr,
             [AST_PARAM := ast_param,]
             ast_functions,
             device_code)
```

## argument information

<i>status</i> :	INTEGER;
<i>ast_addr</i> :	^ANYTYPE;
<i>ast_param</i> :	INTEGER;
<i>ast_functions</i> :	INTEGER;
<i>device_code</i> :	INTEGER;

---

## Arguments

<b><i>status</i></b>	
Usage:	Longword (unsigned)
VAX Type:	cond_value
Access:	Write only
Mechanism:	Reference
Receives the completion status.	

***ast\_addr***

Usage: Procedure entry mask  
 VAX Type: procedure  
 Access: Read only  
 Mechanism: Reference

Specifies the address of the AST routine. The KAV30 software calls the AST routine at this address when the device code that you specify in the *device\_code* argument causes an AST to be issued to the process.

***ast\_param***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies a parameter that this service passes to the AST routine.

***ast\_functions***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specify the KAV\$M\_REPEAT value for this argument when you want to queue the AST to the AST pending queue for the device code immediately after the KAV30 kernel delivers the AST.

***device\_code***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the device code that identifies the AST that you want to set. The KAV\$DEF\_AST service returns the device code when you define the AST.

# KAV\$SET\_AST

---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$CLR_AST	KAV\$QUE_AST
KAV\$DEF_AST	

---

## Examples

See the programs listed in Appendix C for examples of KAV\$SET\_AST service calls.

---

## KAV\$SET\_CLOCK

Allows you to perform the following actions:

- Read the value of the KAV30 real-time clock and place it in the VAXELN system time.
  - Read the value of the VAXELN system time, and place it in the KAV30 real-time clock. You can place the value in the KAV30 real-time clock in either 12- or 24-hour mode.
- 

### Ada Call Format

```
WITH KAVDEF;
KAV_SET_CLOCK ([STATUS => status,]
               CLOCK_FUNCTIONS => clock_functions);
```

### argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_
		TYPE;
<i>clock_functions</i> :	in	INTEGER;

---

### C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"
int kav$set_clock ([status,]
                  clock_functions)
```

# KAV\$SET\_CLOCK

## argument information

int	<i>*status;</i>
int	<i>clock_functions;</i>

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$SET_CLOCK ([status,]  
                    %VAL(clock_functions))
```

## argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>clock_functions</i>

---

## Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$SET_CLOCK ([STATUS := status,]  
              CLOCK_FUNCTIONS := clock_functions)
```

## argument information

<i>status</i> :	INTEGER;
<i>clock_functions</i> :	INTEGER;

---

Arguments

*status*

Usage: Longword (unsigned)  
VAX Type: Longword  
Access: Write only  
Mechanism: Reference

Receives the completion status.

*clock\_functions*

Usage: Longword (unsigned)  
VAX Type: Longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the function that you want to perform. Specify one of the following values:

KAV\$K_SET_VAX_TIME	Reads the value of the KAV30 real-time clock and places the value in the VAXELN system time.
KAV\$K_SET_RTC_TIME	Reads the value of the VAXELN system time and places the value in the KAV30 real-time clock.

When you specify the KAV\$K\_SET\_RTC\_TIME value, also specify one of the following modifiers:

KAV\$M_RTC_12_HOUR	Sets the real-time clock value in 12-hour mode.
KAV\$M_RTC_24_HOUR	Sets the real-time clock value in 24-hour mode.

---

Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_INVALID_TIME	The time that the service reads is invalid.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.

**KAV\$SET\_CLOCK**

KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_SUCCESS	The operation is successful.
KER\$_TIME_NOT_SET	The clock value that the service reads is not set.

---

**Related Services**

KAV\$RTC

---

## KAV\$TIMERS

Performs various timer functions on the timer you specify.

There are five 32-bit timers available for general use. This service allows you to load, start, stop, and reset these timers. You can also use this service to read the value in the timer register and to set the timer to repeat mode.

The service arguments specify the timer number and the function that you want to perform on the timer. If the function is to start the timer or to set the timer to repeat mode, the KAV30 kernel delivers an AST when the timer interval expires.

---

### Note

The KAV30 kernel delivers an AST only when the timer interval expires. It does not deliver an AST if this service stops the timer.

---

In addition to the five 32-bit timers, there are also two 16-bit timers. One of these is the watchdog timer, the other is the local bus timeout timer. If the watchdog timer expires, a KAV30 hardware reset occurs.

---

### Note

The local bus timeout timer specifies the maximum interval for local VAX bus accesses. Digital strongly recommends that you do not alter this value.

---

The KAV30 kernel does not deliver an AST when the local watchdog timer or the local bus timeout timer expires.

See Section 3.2 for more information about the KAV30 timers. See Section 3.1 for more information about ASTs.



Ada Call Format

```
WITH KAVDEF;  
KAV_TIMER ([STATUS => status,  
            TIMER_FUNCTIONS => timer_functions,  
            TIMER_NUMBER => timer_number,  
            TIMER_COUNT => timer_count,  
            AST_ADDR => ast_addr,  
            [AST_PARAM => ast_param,]);
```

argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_
		TYPE;
<i>timer_functions</i> :	in	INTEGER;
<i>timer_number</i> :	in	INTEGER;
<i>timer_count</i> :	in out	INTEGER;
<i>ast_addr</i> :	in	SYSTEM.ADDRESS;
<i>ast_param</i> :	in	INTEGER;

---

C Call Format

```
#include $vaxInc  
#include "eln$:kavdef.h"  
int kav$timers ([status,  
                timer_functions,  
                timer_number,  
                timer_count,  
                ast_addr,  
                [ast_param,])
```

**argument information**

int	<i>*status;</i>
int	<i>timer_functions;</i>
int	<i>timer_number;</i>
int	<i>*timer_count;</i>
void	<i>*ast_addr( );</i>
int	<i>ast_param;</i>

---

**FORTRAN Call Format**

```

INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$TIMERS ([status],
                 %VAL(timer_functions),
                 %VAL(timer_number),
                 timer_count,
                 %VAL(ast_addr),
                 [%VAL(ast_param)])

```

**argument information**

INTEGER*4	<i>status</i>
INTEGER*4	<i>timer_functions</i>
INTEGER*4	<i>timer_number</i>
INTEGER*4	<i>timer_count</i>
INTEGER*4	<i>ast_addr</i>
INTEGER*4	<i>ast_param</i>

### Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$TIMERS ([STATUS := status,]  
            timer_functions,  
            timer_number,  
            timer_count,  
            ast_addr,  
            [AST_PARAM := ast_param,])
```

### argument information

<i>status</i> :	INTEGER;
<i>timer_functions</i> :	INTEGER;
<i>timer_number</i> :	INTEGER;
<i>timer_count</i> :	INTEGER;
<i>ast_addr</i> :	^ANYTYPE;
<i>ast_param</i> :	INTEGER;

---

### Arguments

#### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

#### ***timer\_functions***

Usage: Longword (unsigned)  
VAX Type: mask\_longword  
Access: Read only  
Mechanism: Value

Specifies the function that you want to perform on the timer.

Specify one or more of the following modifiers:

KAV\$M_LOAD_TMR_CNT	<p>Loads a value into the timer register.</p> <p>Specify the value in the <i>timer_count</i> argument. Also, specify the AST routine address and parameters when loading the timer.</p> <p>If you specify the KAV\$M_LOAD_TMR_CNT modifier, you cannot also specify the KAV\$M_READ_TMR_CNT modifier.</p>
KAV\$M_START_TMR	<p>Starts the timer. The timer starts decrementing the value in the timer register. When the value in the register reaches zero, the KAV30 software issues an AST.</p> <p>If you specify the KAV\$M_START_TMR modifier, you cannot also specify the KAV\$M_STOP_TMR modifier.</p>
KAV\$M_STOP_TMR	<p>Stops the timer. This service does not issue an AST—it issues an AST only if the number in the timer register reaches 0.</p> <p>If you specify the KAV\$M_STOP_TMR modifier, you cannot also specify the KAV\$M_START_TMR modifier.</p>
KAV\$M_READ_TMR_CNT	<p>Reads the value stored in the timer register. Read the value in the timer register only when you also specify the KAV\$M_STOP_TMR modifier. That is, stop the timer before you read the value in its register.</p> <p>If you specify the KAV\$M_READ_TMR_CNT modifier, you cannot also specify the KAV\$M_LOAD_TMR_CNT modifier.</p>

KAV\$TIMERS

KAV\$M_REPEAT_TMR	Sets the timer to repeat mode. In this mode the KAV30 software requeues the AST after it is delivered, and then reloads and restarts the timer. You can specify the KAV\$M_REPEAT_TMR modifier only when you also specify the KAV\$M_LOAD_TMR modifier.
KAV\$M_RESET_TMR	Resets the specified timer and deletes the pending ASTs. If you specify the KAV\$M_RESET_TMR modifier, you cannot specify other modifiers.

*timer\_number*

Usage:	Longword (unsigned)
VAX Type:	longword_unsigned
Access:	Read only
Mechanism:	Value

Specifies the timer on which the KAV30 software performs the functions specified by the *timer\_functions* argument. Specify one of the following values:

16-bit Timers		
KAV\$K_LCL_TO	KAV\$K_WDOG	
32-bit Timers		
KAV\$K_CTMR0	KAV\$K_CTMR1	KAV\$K_CTMR2
KAV\$K_CTMR3	KAV\$K_CTMR4	

*timer\_count*

Usage:	Longword (unsigned)
VAX Type:	longword_unsigned
Access:	Modify
Mechanism:	Reference

Specifies the value that the KAV30 software loads into the timer register (the timer is the one specified by the *timer\_number* argument). You can use the KAV\$M\_READ\_TMR\_CNT modifier to read the timer register. The value of the *timer\_count* argument, when multiplied by the clock period for the timer (400 ns), specifies the time that elapses before the timer issues an AST.

For the 32-bit timers, the value of the *timer\_count* argument must not exceed  $(2^{32} - 1)$ . For the watchdog timer and the local bus timeout timer, which are 16-bit timers, the value of the *timer\_count* argument must not exceed  $(2^{16} - 1)$ . If the *timer\_count* argument specifies a value greater than the maximum allowed, the service truncates the value to the maximum value.

The minimum prescaler value is two. This gives a minimum time of 800 ns.

See Section 3.2 for more information about the KAV30 timers.

### ***ast\_addr***

Usage: Procedure entry mask  
 VAX Type: procedure  
 Access: Read only  
 Mechanism: Reference

Specifies the address of the AST routine that this service calls when the timer interval expires. See Section 3.1 for more information about ASTs.

Specify the *ast\_addr* argument only when the *timer\_functions* argument specifies the KAV\$M\_LOAD\_TMR\_CNT modifier.

### ***ast\_param***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies a parameter that this service passes to the AST routine that this service calls when the timer interval expires. See Section 3.1 for more information about ASTs.

Specify the *ast\_param* argument only when the *timer\_functions* argument specifies the KAV\$M\_LOAD\_TMR\_CNT modifier.

## KAV\$TIMERS

---

### Status Values

KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_TMR_BUSY	The timer is busy.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

### Examples

The following code is an example program that calls the KAV\$TIMERS service:

```
#module kav_timer
```

```

/*****
*
*  COPYRIGHT (C) 1991
*  BY DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
*
*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED*
*  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE*
*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER*
*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY*
*  OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY*
*  TRANSFERRED.
*
*  THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE*
*  AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
*  CORPORATION.
*
*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
*
*****/
/*
    FACILITY:  KAV system services DTM suite

    PURPOSE:  This routine uses one of the 32-bit timers to measure the
              time between to system services or to generate an AST when
              the timer times out.

*/

#include $vaxelnc
#include stdio
#include <eln$:kavdef.h>
#include $get_message_text

#define CONST    0x10000

void    error_text ();
void    ast_routine();

int     ast_count;

main    ()
{
    void    error_text ();
    int     status, ipl;                /* status for any call */
    int     i, j, value, tick;

    unsigned int    timer_count, timer_value;    /*params for KAV$TIMERS*/
    unsigned int    high_value, low_value;
    int             min_value, max_value;

```



## KAV\$TIMERS

```
    ipl                = 22;
    ast_count          = 0;
    tick               = 2;
    value              = 0;
    low_value          = 5;                      /* 5 * 400ns = 2us => tick */
    high_value         = 0xFFF;
    timer_count        = high_value * CONST + low_value;

    printf("start of timer test .....\\n");

/*
 *      raise IPL to prevent timer IRQ - Kernel mode only !!!
 */

/*****
    ELN$DISABLE INTERRUPT(ipl);
*****/

    for (i=0; i<100; i++)
    {

/*
 *      Load and start timer (from here timer can only be read in
 *      conjunction with STOP modifier)
 */

        KAV$TIMERS (&status,
                    KAV$M_START_TMR + KAV$M_LOAD_TMR_CNT,
                    KAV$K_CTMRI,
                    &timer_count,
                    ast_routine,
                    NULL);

/*
 *      Time is measured between these routines - any code put in
 *      here will be 'measured' ...
 *      if the time value (high_value * tick) is less then 90us,
 *      the high_counter will time out and an AST is generated.
 */

/*****
    *** put your code to 'measure' in here ...and see whats happening... ***
*****/

        KAV$TIMERS (&status,
                    KAV$M_STOP_TMR + KAV$M_READ_TMR_CNT,
                    KAV$K_CTMRI,
                    &timer_value,
                    NULL,
                    NULL);

        if (! (status & 1))
            error_text (status);
```

```

/*
 *   The timer_value high order word (high_counter) is shifted
 *   to the low order word (disregarding low_counter value) and
 *   then subtracted from high value (high counter start value).
 *   timer_value is then the elapsed time in tick's. timer_value
 *   is added to value, this cumulates the timer_values for all runs.
 */

timer_value = high_value - (timer_value / CONST);
value       = value + timer_value;

if (i > 0)
{
    if (timer_value < min_value) min_value = timer_value;
    if (timer_value > max_value) max_value = timer_value;
}
else
{
    min_value = timer_value;
    max_value = timer_value;
}

/*
 *   In order to run the timers again, they have to be reset
 */

KAV$TIMERS (&status,
            KAV$M_RESET_TMR,
            KAV$K_CTMR1,
            &timer_count,
            NULL,
            NULL);
if (! (status & 1))
    error_text (status);
} /* continue with loop */

/*
 *   The cumulated value is divided by i (number of runs) and
 *   multiplied with tick (low counter time). The first low counter
 *   timeout will load the high_counter value and the second (and
 *   every following) low_counter timeout will decrement the
 *   high_counter, therefore one tick is added to value.
 */

if (ast_count > 0)
    printf("number of AST's occurred: %d \n", ast_count);

```

## KAV\$TIMERS

```
else
{
    value      = ((value / i) * tick) + tick;
    min_value   = (min_value * tick) + tick;
    max_value   = (max_value * tick) + tick;
    printf("mean time for %d runs : %d microseconds \n", i, value);
    printf("best case: %d uS - worst case: %d uS\n", min_value,
    max_value);
}

printf("Test KAV_TIMER completed successfully");
exit (1);
}

/*****
*
*      Name:          AST_ROUTINE()
*
*      Abstrcat:      Control is transferred to this routine whenever
*                    the started timer counts to zero
*
*      Input:         ast_param (if defined in KAV$TIMERS)
*
*      Output:        none
*
*      Comment:       If the high_value is short enough, high counter
*                    will timeout before the STOP+READ service has
*                    executed therefore an AST will occur and control
*                    is transferred to this routine. Since the timeout
*                    will stop the high counter (low_counter will
*                    continue decrementing), any read of timer_value
*                    will show the original contents of the
*                    load registers. If timers are set to repeated
*                    mode, counters will be reloaded and started again.
*                    Any READ in the AST_ROUTINE without the STOP
*                    modifier will then return the TIMER_BUSY error.
*
*****/

void    ast_routine()
{
    void    error_text ();
    int     status;                      /* status for any call */

    int     timer_functions;
    int     timer_number;
    int     timer_value;

    ast_count++;
    return;
}
```

```

/*****
*
*      Name:          ERROR_TEXT()
*
*      Abstrcat:      Routine converts kernel error number's to text
*                      and print's it
*
*      Input:         status
*
*      Output:        none
*
*      Comment:       none
*
*****/
void      error_text (status)
          int status;
{
    int          text_map_functions; /* parameters for $get_message */
    char          text_buffer[255];
    VARYING_STRING(255) result_string;

    text_map_functions = STATUS$ALL;
    eln$get_status_text ( status,
                          text_map_functions,
                          &result_string);
    VARYING TO CSTRING ( result_string,text_buffer);
    printf("%s\n", text_buffer);
    printf("KAV$XXX Error : %d \n", status);

    return;
}

```

---

# KAV\$UNMAP

Frees the SGM entries that the KAV\$IN\_MAP and KAV\$OUT\_MAP services allocate.

If the SGM entries were allocated by calls to the KAV\$IN\_MAP service that specified a location monitor, the KAV\$UNMAP service clears any ASTs that are pending as a result of attempts by VMEbus devices to access the KAV30 P0 space.

See Section 3.6 for more information about the SGM.

---

## Ada Call Format

```
WITH KAVDEF;

KAV_UNMAP ([STATUS => status,]
           SGM_ENTRY => sgm_entry,
           PAGE_COUNT => page_count,
           VIRTUAL_ADDRESS => virtual_address,
           UNMAP_FUNCTIONS => unmap_functions);
```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>sgm_entry</i> :	in	INTEGER;
<i>page_count</i> :	in	INTEGER;
<i>virtual_address</i> :	in	SYSTEM.ADDRESS;
<i>unmap_functions</i> :	in	INTEGER;

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"

int kav$unmap ([status,
               entry,
               page_count,
               virtual_address,
               unmap_functions)
```

## argument information

int	<i>*status</i> ;
int	<i>entry</i> ;
int	<i>page_count</i> ;
void	<i>*virtual_address</i> ;
int	<i>unmap_functions</i> ;

---

## FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'

CALL KAV$UNMAP ([status,
                 %VAL(entry),
                 %VAL(page_count),
                 virtual_address,
                 %VAL(unmap_functions)])
```

## KAV\$UNMAP

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>entry</i>
INTEGER*4	<i>page_count</i>
INTEGER*4	<i>virtual_address</i>
INTEGER*4	<i>unmap_functions</i>

---

### Pascal Call Format

```
INCLUDE $KAVDEF;  
  
KAV$UNMAP ([STATUS := status,]  
           entry,  
           page_count,  
           virtual_address,  
           unmap_functions)
```

### argument information

<i>status</i> :	INTEGER;
<i>entry</i> :	INTEGER;
<i>page_count</i> :	INTEGER;
<i>virtual_address</i> :	^ANYTYPE;
<i>unmap_functions</i> :	INTEGER;

---

## Arguments

### ***status***

Usage: Longword (unsigned)  
VAX Type: cond\_value  
Access: Write only  
Mechanism: Reference

Receives the completion status.

### ***entry***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the first SGM entry that you want to unmap.

### ***page\_count***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the number of successive 64K byte pages that you want to unmap.

### ***virtual\_address***

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Read only  
Mechanism: Value

Specifies the virtual address at which the KAV30 software starts unmapping pages.



# KAV\$UNMAP

## *unmap\_functions*

Usage: Longword (unsigned)  
VAX Type: mask\_longword  
Access: Read only  
Mechanism: Value

Specifies whether the SGM entries that you want to unmap are incoming (mapping VMEbus address space into KAV30 I/O space) or outgoing (mapping KAV30 I/O space to VMEbus or VSB address space). Specify one of the following modifiers:

KAV\$M_IN	Unmaps the incoming SGM entries
KAV\$M_OUT	Unmaps the outgoing SGM entries
KAV\$M_CSR	Unmaps the SGM entries that map the KAV30 FIFO buffers to the VMEbus address space
KAV\$M_MEMORY	Unmaps the SGM entries that map the VMEbus or VSB address space

---

## Status Values

KAV30\$_BAD_MODIFIER	You did not specify a modifier in the correct format.
KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KAV30\$_SGMSETCLR	The SGM entry is valid but must be invalid.
KAV30\$_SGM_INCONS	The SGM entries are inconsistent.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

**Related Services****KAV\$IN\_MAP****KAV\$OUT\_MAP**

---

**Examples**

See the examples in the description of the KAV\$BUS\_READ, KAV\$IN\_MAP, and KAV\$OUT\_MAP services.

---

## KAV\$VME\_SETUP

Allows you to perform the following VMEbus and VSB configuration:

- Enable or disable the VMEbus IRQs
- Enable or disable the VSB IRQs
- Read the VMEbus A24 base address
- Set the VMEbus A32 base address
- Read the VSB slot number

This service specifies a subset of the configuration information that you specify when you use the VAXELN System Builder utility to build the system. However, the values that you specify in a call to this service override the values that you specify when you use the VAXELN System Builder. See Section 5.4 for more information about using the VAXELN System Builder.

This service allows you to perform the following VMEbus and VSB configuration:

- VMEbus IRQs

The VMEbus can send the IRQs at seven different levels. You can specify the IRQ levels at which the KAV30 can receive the IRQs. You pass a bit mask to the service to enable or disable the KAV30 to receive the IRQs at each IRQ level.

When you enable an IRQ line, you can pass a bit mask to this service to switch between autovectorized IRQs and vectored IRQs on that line. See Section 5.4.1 for more information.

- VSB IRQs

The VSB has one IRQ line, on which the KAV30 can receive autovectorized IRQs from the VSB. You can call this service to enable or disable an IRQ from the VSB.

- VMEbus A24 base address

You can use this service to read the setting of the KAV30 rotary switch. This switch determines the value of the KAV30 VMEbus A24 base address.

- VMEbus A32 base address  
You can use this service to specify the KAV30 VMEbus A32 base address.
- VSB slot number  
The VSB has up to six slots that accommodate from zero to six VSB modules. You can use this service to read the VSB slot number for the KAV30. When you do not have a VSB backplane, this service returns the slot number seven.

---

## Ada Call Format

```
WITH KAVDEF;
KAV_VME_SETUP ([STATUS => status,]
                SETUP_FUNCTIONS => setup_functions,
                BUFFER => buffer);
```

## argument information

<i>status</i> :	out	CONDITION_HANDLING.COND_VALUE_TYPE;
<i>setup_functions</i> :	in	INTEGER;
<i>buffer</i> :	in	SYSTEM.ADDRESS;

---

## C Call Format

```
#include $vaxelnc
#include "eln$:kavdef.h"
int kav$vme_setup ([status],
                  setup_functions,
                  buffer)
```

## KAV\$VME\_SETUP

### argument information

int	<i>*status;</i>
int	<i>setup_functions;</i>
int	<i>*buffer;</i>

---

### FORTRAN Call Format

```
INCLUDE 'ELN$:KAVDEF.FOR'  
CALL KAV$VME_SETUP ([status],  
                    %VAL(setup_functions),  
                    buffer)
```

### argument information

INTEGER*4	<i>status</i>
INTEGER*4	<i>setup_functions</i>
INTEGER*4	<i>buffer</i>

---

### Pascal Call Format

```
INCLUDE $KAVDEF;  
KAV$VME_SETUP ([STATUS := status,]  
               setup_functions,  
               buffer)
```

### argument information

<i>status</i> :	INTEGER;
<i>setup_functions</i> :	INTEGER;
<i>buffer</i> :	INTEGER;

## Arguments

### ***status***

Usage: Longword (unsigned)  
 VAX Type: cond\_value  
 Access: Write only  
 Mechanism: Reference

Receives the completion status.

### ***setup\_functions***

Usage: Longword (unsigned)  
 VAX Type: longword\_unsigned  
 Access: Read only  
 Mechanism: Value

Specifies the function that you want this service to perform. Specify one or more of the following values:

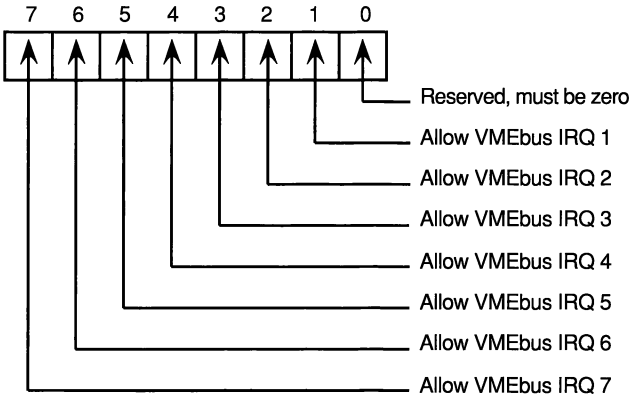
KAV\$K_ALLOW_VME_IRQ	Enables or disables the VMEbus interrupts according to the bit mask specified by the <i>buffer</i> argument.
KAV\$K_AUTO_VME_IRQ	Switches from vectored to autovectored IRQs when an IRQ line is enabled. If a bit is clear, the corresponding IRQ line handles vectored interrupts. If a bit is set, the corresponding IRQ line handles autovectored interrupts.
KAV\$K_DISABLE_VSB_IRQ	Disables the VSB interrupts.
KAV\$K_ENABLE_VSB_IRQ	Enables the VSB interrupts.
KAV\$K_RD_A24_ROTARY	Reads the value of the VMEbus A24 mode base address from the KAV30 rotary switch. This service returns the value of the switch in the low-order four bits of the <i>buffer</i> argument.
KAV\$K_RD_VSB_SLOT	Reads the VSB slot number into the low-order 3 bits of the <i>buffer</i> argument.
KAV\$K_SET_A32_BASE	Sets the high-order byte of the VMEbus A32 mode base address to the value specified in the <i>buffer</i> argument.

KAV\$VME\_SETUP

buffer

Usage: Longword (unsigned)  
VAX Type: longword\_unsigned  
Access: Modify  
Mechanism: Reference

Contains data that this service uses to carry out the function specified by the *setup\_functions* argument. The following table explains the contents of the *buffer* argument for each *setup\_functions* value.

setup_functions Value	Contents of buffer argument
KAV\$K_ALLOW_VME_IRQ	<p>The low-order byte contains a bit mask that controls whether the KAV30 enables or disables the VMEbus IRQs at each IRQ level. The KAV30 enables the VMEbus IRQs at each IRQ level for which a bit is set. The following diagram shows the bit mask:</p> 

KAV\$K_AUTO_VME_IRQ	<p>The low-order byte contains a bit mask that controls whether the KAV30 switches an enabled IRQ line from vectored to autovectored. The KAV30 switches an enabled IRQ line from vectored to autovectored at each IRQ level for which a bit is set. The bit mask has the same layout as the vectored IRQ bit mask.</p>
---------------------	---

---

<i>setup_functions</i> Value	Contents of <i>buffer</i> argument
KAV\$K_RD_A24_ROTARY	Returns the value of the VMEbus A24 mode base address from the rotary switch on the KAV30 in the low-order four bits of the <i>buffer</i> argument.
KAV\$K_RD_VSB_SLOT	Returns the VSB slot number in the low-order three bits of the <i>buffer</i> argument.
KAV\$K_SET_A32_BASE	The low-order byte contains the VMEbus A32 mode base address.

---



---

## Status Values

KAV30\$_BAD_PARAM	You did not specify a parameter in the correct format.
KER\$_BAD_COUNT	You did not specify the correct number of arguments.
KER\$_BAD_VALUE	You did not specify a value in the correct format.
KER\$_NO_ACCESS	The service cannot access an item.
KER\$_SUCCESS	The operation is successful.

---

## Related Services

KAV\$INT\_VME

---

## Examples

See the examples in the description of the KAV\$BUS\_BITCLR service.





---

# Developing KAV30 Applications

This chapter describes how to develop applications for the KAV30. It gives information on the following:

- Design guidelines
- Coding guidelines
- Compiling and linking KAV30 applications
- Building KAV30 system images
- Loading and running KAV30 system images
- Debugging KAV30 applications
- Developing SCSI class drivers
- Building a SCSI class driver into an application

## 5.1 Design Guidelines

This section gives guidelines for designing KAV30 applications. It gives guidelines for the following actions:

- Accessing the VMEbus and VSB address space
- Writing AST routines

### 5.1.1 Accessing the VMEbus and VSB Address Space

This section gives guidelines for accessing the VMEbus and VSB address space. There are two methods of accessing the VMEbus and VSB address space:

- Directly accessing the address space
- Using the KAV\$BUS\_READ and KAV\$BUS\_WRITE services

## Developing KAV30 Applications

### 5.1.1.1 Directly Accessing the VMEbus and VSB Address Space

Digital recommends that you directly access the VMEbus and VSB address space under the following circumstances:

- When the KAV30 is the only bus master and there are no slow devices on the bus
- When you want to migrate existing VMEbus or VSB applications to the KAV30

Use the virtual addresses that the KAV\$OUT\_MAP service returns to access the address space.

When errors occur during a direct access, the KAV30 kernel generates an exception condition. If you want your application to handle the exception, the application must include a condition handler for the exception. In the process context, when the application does not include a condition handler for the exception, the system invokes the last chance handler. Invoking the last chance handler usually deletes the process that causes the exception to occur.

---

#### Note

---

The system fails if an error occurs while you are directly accessing the VMEbus or VSB from an ISR.

---

If the system fails because an error occurs while you are directly accessing the VMEbus or VSB from an ISR, the stack contains the following data:

- The number of arguments
- The VAXELN status code
- The VMEbus or VSB address that the KAV30 tried to access
- The value of the Program Counter (PC) when the error occurred
- The value of the processor status longword when the error occurred

A sample stack dump follows:

```
      4
00007E3C
00F00000
8000B123
00C80009
```

### 5.1.1.2 Using the KAV\$BUS\_READ and KAV\$BUS\_WRITE Services

Digital recommends that you use the KAV\$BUS\_READ and KAV\$BUS\_WRITE services only under the following circumstances:

- When there is more than one bus master
- When there are slow devices on the bus
- When you want to communicate with another KAV30
- When you want to ensure that the errors due to bus timeouts and arbitration problems do not interfere with the data transfer
- When you are testing systems that are under development
- When you are writing an ISR routine (elevated IPL)

### 5.1.2 Writing Asynchronous System Trap Routines

Use the following guidelines for writing AST routines:

- Do not include mutexes
- Do not include I/O routines
- Do not include signal calls to your own process

When an AST routine contains a mutex or an I/O routine, unpredictable behavior can result, especially in cases where your application code is not reentrant. When an AST routine contains a signal call to your own process, the system can hang.

## 5.2 Coding Guidelines

This section provides guidelines for coding KAV30 applications in each of the supported languages. The supported languages are as follows:

- VAX Ada
- VAX C
- VAX FORTRAN
- VAXELN Pascal

## Developing KAV30 Applications

### 5.2.1 VAX Ada

When you write VAX Ada programs for the KAV30, follow these guidelines:

- Include the VAX Ada package for the KAV30 kernel. The name of this package is ELN\$:KAVDEF.ADA.
- When you call the KAV30 system services, specify their inclusion at the start of the code. For example:

```
WITH KAVDEF;  
USE KAVDEF;  
.  
.  
.  
KAV_DEF_AST(status, device_code);
```

If you omit the USE statement, call the system services as follows:

```
WITH KAVDEF;  
.  
.  
.  
KAVDEF.KAV_DEF_AST(status, device_code);
```

- When you call VAXELN kernel routines, include the following line in the code:

```
WITH VAXELN_SERVICES;  
.  
.  
.
```

See the *VAXELN Ada User's Manual* and the *VAXELN Ada Runtime Library Reference Manual* for more information about writing programs in VAX Ada.

#### 5.2.1.1 Coding Asynchronous System Trap Routines in VAX Ada

When you write AST routines in VAX Ada, declare the routines in a separate package. For example:

```
with VAXELN_SERVICES;  
package AST_ROUTINES is  
  TRIGGERED_EVENT : VAXELN_SERVICES.EVENT_TYPE;  
  procedure AST_ROUTINE;  
  pragma EXPORT_PROCEDURE (AST_ROUTINE);  
end AST_ROUTINES;  
package body AST_ROUTINES is  
  procedure AST_ROUTINE is
```

```
begin
    VAXELN_SERVICES.SIGNAL_EVENT( EVENT => TRIGGERED_EVENT);
end AST_ROUTINE;
end AST_ROUTINES;
```

When you specify an AST parameter in a call to a KAV30 service, specify the value of the AST parameter. However, because VAX Ada always passes arguments by reference, you must use the `SYSTEM.TO_INTEGER` function to convert the address of the AST parameter into an integer and pass the integer to the service. Because VAX Ada passes arguments by reference, it uses the integer as an address. However, because the address contains the required AST parameter value, the AST routine receives the correct value. The following example shows how to implement this mechanism in a call to the `KAV$IN_MAP` system service.

```
with AST_ROUTINES, VAXELN_SERVICES;

package signaller_task is
    SIGNALLER_EVENT : VAXELN_SERVICES.EVENT_TYPE;

    task type signaller is
    end signaller;
end signaller_task;

package body signaller_task is
    task body signaller is
    begin
        VAXELN_SERVICES.WAIT_ANY( VALUE1 => AST_ROUTINES.TRIGGERED_EVENT);
        VAXELN_SERVICES.CLEAR_EVENT( EVENT => AST_ROUTINES.TRIGGERED_EVENT);
        VAXELN_SERVICES.SIGNAL_EVENT( EVENT => AST_ROUTINES.TRIGGERED_EVENT);
    end signaller;
end signaller_task;

with AST_ROUTINES, TEXT_IO, KAVDEF, CONDITION_HANDLING, SYSTEM,
    VAXELN_SERVICES, SIGNALLER_TASK;

procedure AST_TEST is
```

## Developing KAV30 Applications

```
STATUS : CONDITION HANDLING.COND_VALUE_TYPE;
SGM_ENTRY : INTEGER;
BUS_PAGE : SYSTEM.ADDRESS;
IN_PAGE : SYSTEM.ADDRESS;
MAP_FUNCTIONS : INTEGER;
UNMAP_FUNCTIONS : INTEGER;
KAV_SERVICE_ERROR : exception;
package STATUS_IO is new
TEXT_IO.INTEGER_IO(CONDITION HANDLING.COND_VALUE_TYPE);
signaller : SIGNALLER_TASK.SIGNALLER;

begin

    MAP_FUNCTIONS := KAVDEF.KAV_M_MEMORY + KAVDEF.KAV_M_LOCMON_IPL17;

    VAXELN_SERVICES.CREATE_EVENT( EVENT          =>
                                   AST_ROUTINES.TRIGGERED_EVENT,
                                   INITIAL_STATE => VAXELN_SERVICES.CLEARED);

    VAXELN_SERVICES.CREATE_EVENT( EVENT          =>
                                   SIGNALLER_TASK.SIGNALLER_EVENT,
                                   INITIAL_STATE => VAXELN_SERVICES.CLEARED);

    SGM_ENTRY := 0;

    KAVDEF.KAV_IN_MAP( STATUS          => STATUS,
                       SGM_ENTRY       => SGM_ENTRY,
                       PAGE_COUNT      => 1,
                       VIRTUAL_ADDRESS => BUS_PAGE,
                       AST_ADDR        => AST_ROUTINES.AST_ROUTINE'ADDRESS,
                       MAP_FUNCTIONS   => MAP_FUNCTIONS);

    if not CONDITION_HANDLING.SUCCESS(STATUS) then
        raise KAV_SERVICE_ERROR;
    end if;

    VAXELN_SERVICES.WAIT_ANY( VALUE1 => SIGNALLER_TASK.SIGNALLER_EVENT);

    TEXT_IO.PUT_LINE("After call to WAIT_ANY");

    KAVDEF.KAV_UNMAP( STATUS          => STATUS,
                     SGM_ENTRY       => SGM_ENTRY,
                     PAGE_COUNT      => 1,
                     VIRTUAL_ADDRESS => BUS_PAGE,
                     UNMAP_FUNCTIONS => KAVDEF.KAV_M_IN);

    if not CONDITION_HANDLING.SUCCESS(STATUS) then
        raise KAV_SERVICE_ERROR;
    end if;

exception
```

```

when KAV_SERVICE_ERROR =>
  TEXT_IO.PUT("Error detected: ");
  STATUS_IO.PUT( ITEM => STATUS, WIDTH => 8, BASE => 16);
  TEXT_IO.NEW_LINE;

when others
  raise;

end AST_TEST;

```

See Section 3.1 for more information about ASTs and AST parameters. See the *VAX Ada Run-Time Reference Manual* for more information about writing VAXELN Ada programs that involve ASTs.

### 5.2.2 VAX C

When you write VAX C programs for the KAV30, follow these guidelines:

- Use the `#include` compiler preprocessor directive to include the following text libraries:
  - `$vaxelnc`, which defines the necessary VAXELN constants, data types, and procedures
  - `eln$:kavdef.h`, which defines the KAV30-specific constants, data types, and procedures
- Specify all the arguments, required and optional, in system service calls. Specify optional arguments—that is, arguments for which you want to use default values—as 0 or NULL. NULL is a constant (defined in the `$vaxelnc` library) that allows you to supply the value 0 for an argument, yet maintain readability.
- The default argument passing mechanism in VAX C is by value. Use one of the following methods to pass an argument by reference in VAX C:
  - Prefix the argument with the address-of operator (`&`). For example:
 

```

define_ASB()
{
  int *status;

  kav$def_ast(&status,
              device_code);
}
          
```
  - Create a pointer to the argument, then pass the pointer itself. For example:



## Developing KAV30 Applications

```
define_ASB()  
{  
  int *status_address, status;  
  
  status_address = &status;  
  
  kav$def_ast(status_address,  
              device_code);  
}
```

Digital recommends that you use the address-of operator method, because using the pointer method increases the number of variables that you use and the amount of work the compiler must perform.

The KAV30 system service descriptions prefix arguments that must be passed by reference with an asterisk (\*).

- Use a bitwise AND operation (&) on the status value and 1 to check the status values, or if you are testing for failure, negate the result, as follows:

```
if (status & 1)  
    success_statement; /* success */  
  
if (! (status & 1))  
    error_statement; /* failure */
```

See the *VAXELN C Reference Manual* and the *VAXELN C Runtime Library Reference Manual* for information about writing VAXELN programs in VAX C.

### 5.2.3 VAX FORTRAN

When you write VAX FORTRAN programs for the KAV30, follow these guidelines:

- Include the file ELN\$:KAVDEF.FOR. This file includes the KAV30-specific constants, data types, and procedures.
- Unlike VAX C, where you must provide all the arguments, you can omit optional arguments in calls to VAX FORTRAN run-time library routines. For example:

```
CALL KAV$DEF_AST(status, device_code) ! optional argument included  
CALL KAV$DEF_AST(,device_code)        ! optional argument omitted
```

- The default argument passing mechanism in VAX FORTRAN is by reference. Use the %VAL function to pass arguments by value.

The KAV30 system service descriptions use the %VAL function to pass arguments by value.

See the *VAX FORTRAN Language Reference Manual* and the *VAXELN FORTRAN Runtime Library Reference Manual* for more information about writing programs in VAX FORTRAN.

## 5.2.4 VAXELN Pascal

When you write VAXELN Pascal programs for the KAV30, include the file `ELN$:KAVDEF.PAS`. This file includes the KAV30-specific constants, data types, and procedures.

See the *VAXELN Pascal Language Reference Manual* and the *VAXELN Pascal Runtime Library Reference Manual* for more information about writing programs in VAXELN Pascal.

### 5.2.4.1 Coding AST Routines in VAXELN Pascal

When you write the AST routines in VAXELN Pascal, perform the following actions:

- Declare the routines in a separate compilation module.
- In programs that call system services with an AST routine as an argument, declare the routines to be external with data type `KAV$AST_ROUTINE_TYPE`. For example:

```
MODULE ast_routine;

VAR
    ast_triggered_event : [EXTERNAL] event;

PROCEDURE ast_routine;

BEGIN
    SIGNAL(ast_triggered_event);

END;

END;
```

The following example shows a program called `TEST`, which includes a call to the AST routine declared in the module described:

```
MODULE ast_test;

INCLUDE $KAVDEF;

TYPE
    byte = [byte] 0..255;
    vmebus_page = packed array [0..65535] of byte;

VAR
    p_id : PROCESS;
    ast_triggered_event : EVENT;
    signaller_event : EVENT;
    ast_routine : [EXTERNAL] kav$ast_routine_type;

PROCESS_BLOCK signaller;
```

## Developing KAV30 Applications

```
BEGIN
    WAIT_ANY( ast_triggered_event);
    CLEAR_EVENT( ast_triggered_event);
    SIGNAL( signaller_event)
END;

PROGRAM test(input, output);
VAR
    status : INTEGER;
    entry : INTEGER;
    bus_page : ^vmebus_page;
    in_page : ^vmebus_page;
    map_functions : INTEGER;
BEGIN
    CREATE_EVENT( ast_triggered_event, EVENT$CLEARED, STATUS := status);
    IF NOT ODD(status) THEN
        BEGIN
            WRITELN('CREATE_EVENT error : ', HEX(status));
            EXIT
        END;

    CREATE_EVENT( signaller_event, EVENT$CLEARED, STATUS := status);
    IF NOT ODD(status) THEN
        BEGIN
            WRITELN('CREATE_EVENT error : ', HEX(status));
            EXIT
        END;

    CREATE_PROCESS( p_id, signaller, STATUS := status);
    IF NOT ODD(status) THEN
        BEGIN
            WRITELN('CREATE_PROCESS error : ', HEX(status));
            EXIT
        END;

    map_functions := KAV$M_MEMORY + KAV$M_LOCMON_IPL17;
    entry := 0;

    KAV$IN_MAP( STATUS := status,
                ENTRY := entry,
                PAGE COUNT := 1,
                VIRTUAL ADDRESS := bus_page,
                AST_ADDR := ADDRESS(ast_routine),
                MAP_FUNCTIONS := map_functions);
```

```

IF NOT ODD(status) THEN
  BEGIN
    WRITELN('KAV$IN_MAP error : ', HEX(status));
    EXIT
  END;

WAIT_ANY( signaller_event,
          STATUS := status);

IF NOT ODD(status) THEN
  BEGIN
    WRITELN('WAIT_ANY error : ', HEX(status));
    EXIT
  END;
CLEAR_EVENT( signaller_event);
WRITELN('After call to WAIT_ANY');

KAV$UNMAP( STATUS := status,
           ENTRY := entry,
           PAGE_COUNT := 1,
           VIRTUAL_ADDRESS := bus_page,
           UNMAP_FUNCTIONS := KAV$M_IN);

IF NOT ODD(status) THEN
  BEGIN
    WRITELN('KAV$UNMAP error : ', HEX(status));
    EXIT
  END;

END.
END;
```

Because the AST routine is an external reference, you can resolve it at link time. For example, if the program containing the AST routine is called `AST_ROUTINES.PAS`, build it into the system as follows (where `TEST.PAS` is the name of the main program):

```
$ LINK TEST + AST_ROUTINE + ELN$:KAV$RTL_OBJLIB/LIB + ELN$:RTLSHARE/LIB + -
_$ + RTL/LIB
```

When the AST routines include calls to the KAV30 system services, follow these steps:

1. Include the file `ELN$:KAVDEF.PAS` in the program.
2. When you compile the program, include the KAV30 object library in the command line. For example:

```
$ EPASCAL AST_ROUTINE + ELN$:KAV$RTL_OBJLIB/LIB
```

For more information on the ASTs and AST parameters, see Section 3.1. For more information on writing VAXELN Pascal programs that involve ASTs, see the *VAXELN Pascal Run-Time Language Reference Manual*.

5.3 Compiling and Linking KAV30 Applications

For each language supported by the VAXELN Toolkit, there is an optimizing compiler that generates position-independent object code from the source code.

Use the VMS Linker to link the object modules with the appropriate run-time libraries for the language in which your source code is written. For example, you must link C object modules with the VAXELN C Run-Time Library. You must link all object modules with the VAXELN Kernel general purpose run-time library (RTL.OLB), and with the VAXELN Pascal run-time shareable library (RTLSHARE.OLB).

Linking the object modules with one or more run-time libraries results in a single copy of each run-time library being built into the application image, where it can be shared by all the programs that make up the image.

Table 5–1 lists the commands that compile and link programs written for the KAV30 in each of the languages supported by VAXELN.

You can add qualifiers to the compiler command line to control the actions of the compiler. For example, the /DEBUG qualifier instructs the compiler to build symbolic debugging information into the application.

For more information about compiling VAXELN programs, see the appropriate language reference manual. For more information about linking VAXELN programs with run-time libraries, see the *VAXELN Development Utilities Guide*.

Table 5–1 Compiling and Linking Commands

Language	Compile Command
VAXELN Ada	\$ ADA FILENAME.ADA
VAX C	\$ CC FILENAME.C + ELN\$:VAXELNC/LIB
VAX FORTRAN	\$ FORTRAN FILENAME.FOR
VAXELN Pascal	\$ EPASCAL FILENAME.PAS + ELN\$:KAV\$RTL_OBJS/LIB - + ELN\$:RTLSHARE/LIB

Language	Link Command
VAXELN Ada	\$ ACS LINK UNITNAME + ELN\$:KAV\$RTL_OBJS/LIB + - + ...
VAX C	\$ LINK FILENAME + ELN\$:KAV\$RTL_OBJS/LIB + - + ELN\$:CRTLSHARE/LIB + RTLSHARE/LIB + - + ELN\$:RTL/LIB
VAX FORTRAN	\$ LINK FILENAME + ELN\$:KAV\$RTL_OBJS/LIB + - + ELN\$:FRTLOBJECT/LIB + ELN\$:RTLSHARE/LIB + - + ELN\$:RTL/LIB
VAXELN Pascal	\$ LINK FILENAME + ELN\$:KAV\$RTL_OBJS/LIB + - + ELN\$:RTLSHARE/LIB + ELN\$:RTL/LIB

## 5.4 Building KAV30 System Images

The VAXELN System Builder component of the VAXELN Toolkit combines your application image with the VAXELN-supplied software components to create a VAXELN system image, which you can load on the KAV30.

The System Builder provides a menu interface through which you enter information about the system you are building. For example, you can enter the names of the files that make up your application image on the Program Description menu, and you can specify the external device information on the Device Characteristics menu.

For more information about the System Builder, see the *VAXELN Development Utilities Guide*.

To build a system image that runs as a target system on the KAV30, follow these steps:

1. Invoke the VAXELN System Builder with the following command:

```
$ EBUILD/MAP mydatafile
```

(The /MAP qualifier generates a system map file called MYDATAFILE.MAP, which contains a listing of the images in the system, the devices and terminals you specify, and the system characteristics.)

2. On the Target Processor menu, choose *rtVAX 300* for the Target Processor entry.
3. Return to the Main Menu and add your program descriptions, device descriptions, and so on as described in the *VAXELN Development Utilities Guide*.

# Developing KAV30 Applications

The Ethernet adapter on the rtVAX 300 is a Second Generation Ethernet Controller (SGEC), type EZA. Therefore, specify *EZA* for the Network device entry on the Network Node Characteristics Menu.

---

**Note**

---

For each 64K byte page of system RAM space mapped to the VMEbus or VSB bus address space by the KAV\$OUT\_MAP service, you must increase the System region size entry in the System Characteristics menu by 128 pages. This is in addition to the 128 pages of system space required to support KAV30 internal data structures.

For each 64K byte page of VMEbus address space mapped into KAV30 process address space by the KAV\$IN\_MAP service, you must increase the P0 virtual size entry in the System Characteristics menu by 128 pages. You do not have to do this for calls to the KAV\$IN\_MAP service in which the service maps data from the KAV30 CSR pages to the VMEbus address space.

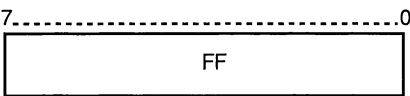
If you are running in kernel mode, increase the kernel stack by at least two pages.

## 5.4.1 Configuring the VMEbus and VSB

When you build the KAV30 system with the VAXELN System Builder, you can specify information that controls how the KAV30 interacts with other devices on the VMEbus and VSB. You specify this information by setting the contents of the System Parameter 1 and System Parameter 2 options in the EBUILD System Characteristics Menu (the System Parameter 3 and System Parameter 4 options are available for use by the customers' applications).

The following list describes how to set the System Parameter 1 and System Parameter 2 options.

- System Parameter 1
  - Enable System Parameter 1 and System Parameter 2 byte

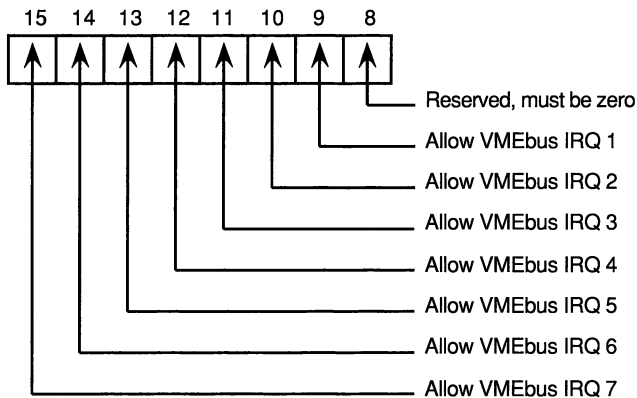


This byte controls whether the System Builder uses System Parameter 1 and System Parameter 2 when building the system. When you do not use System Parameter 1 and System Parameter 2, you can specify

parts of the information in programs by calling the KAV\$VME\_SETUP system service. See the description of the KAV\$VME\_SETUP service for more information.

By default this byte contains the value 0. When you want to use the default configurations, set this byte to  $FF_{16}$  and make sure that System Parameter 1 and System Parameter 2 contain valid settings. When you do not want to use the default configurations, set this byte to any value other than  $FF_{16}$ .

– VMEbus vectored interrupt mask byte



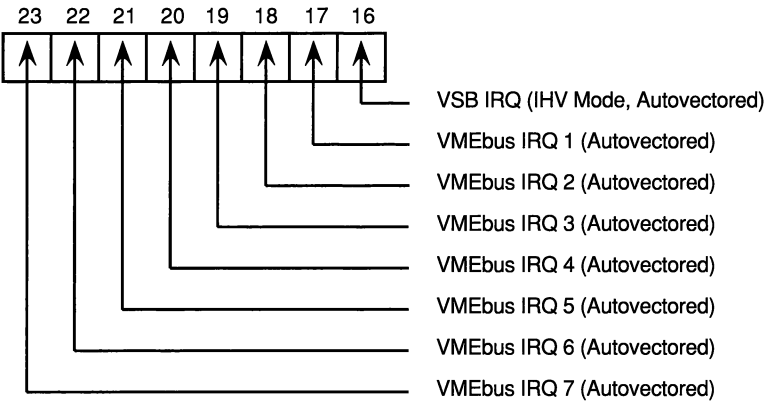
This byte controls whether the KAV30 enables or disables the VMEbus IRQs at each IRQ level. The module allows the IRQs at each IRQ level for which a bit is set (1).

This byte has the following default settings:

Bit	Value	Bit	Value
8	0	12	0
9	0	13	0
10	0	14	0
11	0	15	0



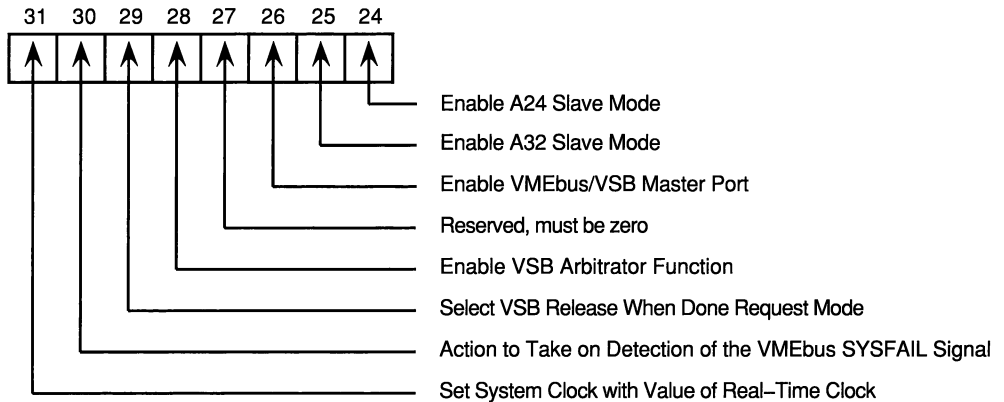
– VMEbus and VSB autovectored interrupt mask byte



This byte controls whether the KAV30 enables or disables the autovectored IRQs at each IRQ level. The module allows the autovectored IRQs at each IRQ level for which a bit is set (1). The module allows vectored IRQs at each IRQ level for which a bit is clear (0). This byte has the following default settings:

Bit	Value	Bit	Value
16	0	20	0
17	0	21	0
18	0	22	0
19	0	23	0

## – VMEbus and VSB control byte



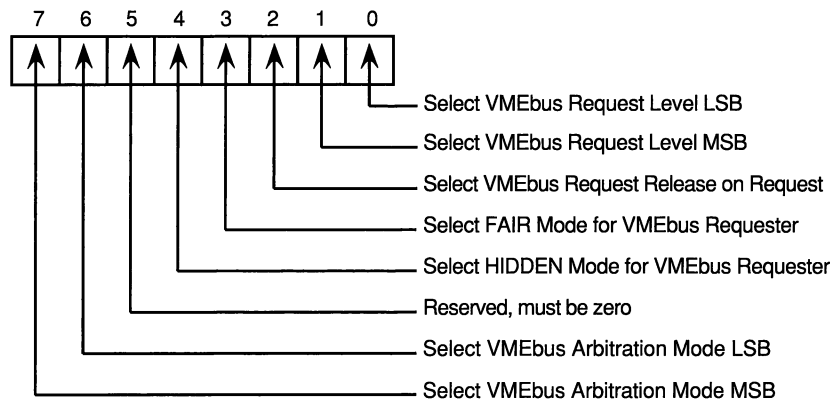
This byte controls the interaction of the KAV30 with the VMEbus and VSB. For example, to set the KAV30 to operate in A32 slave mode, set bit 25 to 1. This byte has the following default settings:

Bit	Value	Bit	Value
24	1	28	0
25	1	29	0
26	1	30	0
27	0	31	0

The value of bit 30 determines what action the KAV30 takes when it detects the assertion of the VMEbus SYSFAIL signal. The KAV30 delivers an asynchronous system trap when bit 30 has the value 0. The KAV30 calls an interrupt service routine (ISR) at vector 540<sub>16</sub> when bit 30 has the value 1.

# Developing KAV30 Applications

- System Parameter 2
  - VMEbus arbiter selection byte



This byte specifies the VMEbus arbiter information. The following table explains the VMEbus request level’s most significant bit (MSB) and least significant bit (LSB):

MSB	LSB	VMEbus Request Level
0	0	BR0
0	1	BR1
1	0	BR2
1	1	BR3

The following table explains the VMEbus arbitration mode’s MSB and LSB:

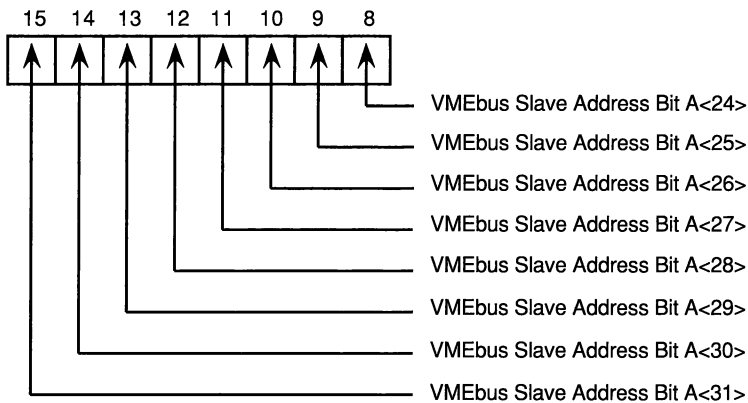
MSB	LSB	VMEbus Arbitration Mode
0	0	Priority encoded
0	1	Round-robin
1	0	Reserved for Digital
1	1	Reserved for Digital

This byte has the following default settings:

Bit	Value	Bit	Value
0	1	4	0
1	1	5	0
2	1	6	0
3	0	7	0

See Section 2.4 for more information about the KAV30 VMEbus arbiter functionality.

- VMEbus slave mode A32 base address byte



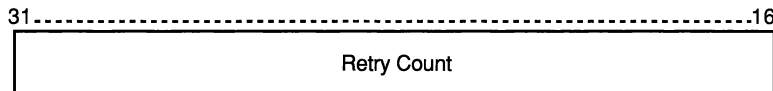
This byte specifies the VMEbus base address for the KAV30 acting as a slave and using A32 addressing mode.

This byte has the following default settings:

Bit	Value	Bit	Value
8	0	12	0
9	0	13	0
10	0	14	0
11	0	15	0

## Developing KAV30 Applications

- Bus access software retry count word



The KAV30 kernel performs software retries of a bus access in addition to the 29 retries performed by the hardware (you can disable hardware retries, using the KAV\$OUT\_MAP kernel service). This word contains the maximum number of times that a bus access retries before the KAV30 kernel returns an error to the application. By default this word contains the value 10 (decimal). Software retries can be necessary because of bus arbitration contention or bus timeouts.

The number of software retries must be between 0 and 65 535.

### 5.5 Loading and Running KAV30 System Images

Once you have built the KAV30 system image, use one of the following procedures to load it onto the KAV30:

- Down-line loading the system image over the Ethernet from the VMS host system or from another VAXELN target system
- Boot the system image from the KAV30 ROM, a tape, or a disk
- Load the system image from an ULTRIX™ system (Digital does not currently supply a VAXELN Toolkit for an ULTRIX host system)
- Boot the system image from a DEC™ SCSI floppy disk or hard disk

For more information about down-line loading VAXELN system images, see the *VAXELN Development Utilities Guide*.

### 5.6 Debugging KAV30 Applications

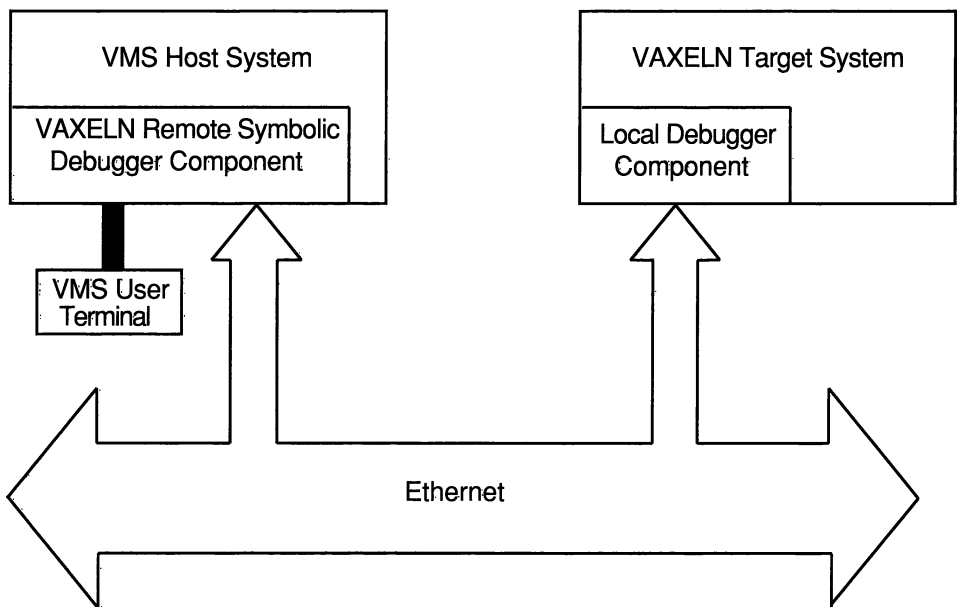
The VAXELN Debugger enables you to debug your application while it runs on the target computer (in this case, the KAV30). The Debugger allows you to set breakpoints, examine variables and addresses, deposit values, and control the execution of your application.

You can run the Debugger remotely (from a terminal connected to the host computer) or locally (from the console terminal connected to the KAV30).

In remote mode, you can view source code and refer to program variables by their symbolic names. In local mode, those operations that require source-file or other host information, for example, operations that refer to variables by name, are unavailable. You choose local or remote mode when you build the system image.

Choosing remote debugging places only a portion of the Debugger in the system image; the remainder resides on the host system, as shown in Figure 5–1.

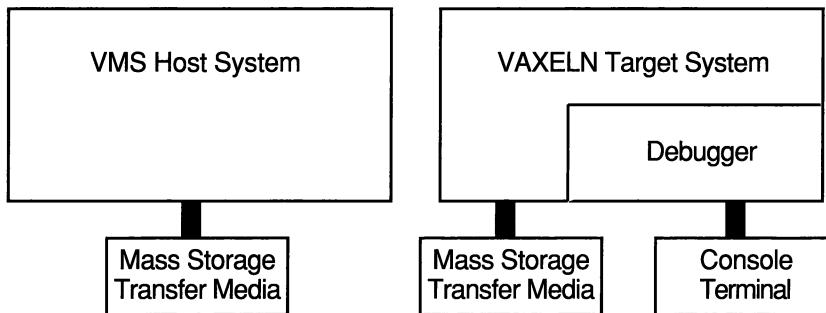
**Figure 5–1 A Remote Debugging Configuration**



Choosing local debugging places the entire Debugger in the system image, as shown in Figure 5–2.

See the *VAXELN Development Utilities Guide* for more information about using the VAXELN Debugger.

**Figure 5–2 A Local Debugging Configuration**



## 5.7 Developing SCSI Class Drivers

To develop a SCSI class driver, follow these steps:

1. Write the SCSI class driver in one of the supported languages.
2. Compile the SCSI class driver. For example, to compile the SCSIUSER class driver in VAX C, enter the following command:  

```
$ CC SCSIUSER + ELN$:VAXELNC/LIBRARY
```
3. Modify the SCSI driver startup module (SCDRIVER.C) for the new SCSI driver.
4. Compile the SCSI driver startup module. For example, to compile the startup module in VAX C, enter the following command:  

```
$ CC SCDRIVER + ELN$:VAXELNC/LIBRARY
```
5. Link the SCSI class driver and the SCSI driver startup module with the VAXELN SCSI driver components to produce a new VAXELN SCSI driver image. For example:

```
$ LINK/EXE=KRDRIVER SCDRIVER + SCISISNIF + SCSIDISK + SCSIGNRC + -  
_ $ SCSIUSER + SCSI53C700 + SCSI53C700 SCRIPT + SCSI53C700/OPT + -  
_ $ ELN$:CRTLSHARE/LIB + RTLSHARE/LIB + RTL/LIB
```

This LINK command links the SCSI class driver with the startup module, the sniffer module, the supplied disk and generic class drivers, and the port driver. If you modified the startup module so that it does not include the supplied class drivers, omit those driver modules when linking the driver image as follows:

```
$ LINK/EXE=KRDRIVER SCDDriver + SCISISNIF + SCSIUSER + SCSI53C700 + -
_$ SCSI53C700_SCRIPT + SCSI53C700/OPT + ELN$:CRTLSHARE/LIB + -
_$ RTLSHARE/LIB + RTL/LIB
```

6. Build the image into the VAXELN system. See Section 5.8 for information about building the SCSI driver into a user application.

See the *VAXELN Runtime Facilities Guide* for more information about developing user-defined SCSI class drivers.

## 5.8 Building a SCSI Class Driver into an Application

Before you use SCSI devices in an application, follow these steps:

1. Set the KAV30 SCSI ID

See the *KAV30 Hardware Installation and User's Information* for more information.

2. Connect the SCSI devices to the KAV30

See the *KAV30 Hardware Installation and User's Information* for more information.

3. Include the KAV30 SCSI class driver in the system image

Enter the following information at the Add Device Description menu in the VAXELN System Builder:

VAXELN System Builder Prompt	Information to Enter
Name	DUA
Vector address	%O2520
Interrupt priority	6
Default file spec	ELN\$:KRDRIVER.EXE
Device-dependent parameter	%X00000? <sup>1</sup> FF

<sup>1</sup>Enter the KAV30 SCSI ID instead of the question mark (?).

Do not enter information at the other prompts, use the default selections for these prompts. Enter the KAV30 SCSI ID instead of the question mark (?) in the information that you must enter as a response to the Device-dependent parameter prompt. The Device-dependent parameter determines the KAV30 SCSI ID when there is no valid SCSI ID in the KAV30 battery backed-up RAM.



Figure 5–3 shows the Add Device Description menu in the VAXELN System Builder when you enter the information and specify seven as the SCSI ID.

**Figure 5–3 Sample Add Device Description Menu**

System KAU30A.DAT - Editing Device			
Name	DUA		
Register address	%00000000		
Vector address	%02520		
Interrupt priority	6		
BI number	0		
Adapter number	0		
Autoload driver	Yes	No	
Default file spec	ELN\$: KRDRIVER.EXE		
Network device	Yes	No	Default
Device-dependent parameter	%X000007FF		
DO      HELP      BACK			

#### 4. Specify the devices for automatic mounting at boot time

List the devices that you want to specify at the Disk/volume names prompt of the Edit System Characteristics menu in the VAXELN System Builder.

See the *VAXELN Development Utilities Guide* for information about the VAXELN System Builder.

While you use SCSI devices in applications, you can perform the following actions:

- Access devices on the SCSI bus  
Use the device's unique SCSI device name to access the device. The device name consists of the characters DUA followed by the SCSI ID of the device. For example DUA2.

- Manipulate disks and files

Use the VAXELN Command Language Utility (ECL) to manipulate local disks and files. See the *VAXELN Development Utilities Guide* for more information.

When the system image is correctly configured, use DECnet to manipulate remote disks and files. See the following documents for more information about manipulating remote disks and files in each supported language:

- *VAXELN C Reference Manual*
- *VAXELN FORTRAN Runtime Library Reference Manual*
- *VAXELN Ada User's Guide*
- *VAXELN Pascal Language Reference Manual*

- Use local error logging to disk

See the *VAXELN Development Utilities Guide* for more information.



---

## Initial KAV30 Configuration

This appendix describes the initial KAV30 hardware and software configuration.

### A.1 Hardware Configuration

This section describes the initial KAV30 hardware configuration.

- The KAV30 does not supply power to the SCSI bus TERMPWR signal.
- The KAV30 responds to the VMEbus RESET signal.
- The KAV30 has 256K bytes of user ROM.
- The rtVAX 300 Ethernet controller can assert the KAV30 VAX HALT signal.
- The break key assertions on the devices connected to the auxiliary port do not assert the KAV30 VAX HALT signal.
- There is no power supply to the battery backed-up RAM and the calendar/clock.
- The KAV30 VMEbus arbiter functionality is disabled.
- The break key assertions on the devices connected to the serial line ports assert the VAX HALT signal.
- The VMEbus ACFAIL signal asserts the KAV30 VAX POWER\_FAIL signal.
- The KAV30 VMEbus A24 base slave address is set to zero.

### A.2 Software Settings

This section describes the initial KAV30 software configuration.

- The VSB arbiter functionality is disabled.
- The VMEbus A32 base slave address is set to zero.
- The VMEbus master functionality is enabled.
- The VSB master functionality is enabled.
- All the VMEbus IRQs are disabled.
- All the VMEbus autovectorized IRQs are disabled.
- The VMEbus A24 slave functionality is enabled.
- The VMEbus A32 slave functionality is enabled.
- The VSB bus requester operates in ROR mode.
- The VMEbus bus requester operates in ROR mode.
- The VMEbus bus requester uses the VMEbus BR3 line.
- The VMEbus arbiter operates in priority mode.
- The VMEbus arbiter operates in not fair mode.
- The VMEbus arbiter operates in not hidden mode.
- The FIFO buffers are clear.
- All SGM entries are invalid.
- The local bus timeout is 25  $\mu$ s.
- The VMEbus timeout is 125  $\mu$ s.
- All counter/timers are reset and clear.
- The VAXELN system time contains the value of the KAV30 calendar/clock.
- The default SCSI ID is seven.

---

## Example Programs—Interprocessor Communication

This appendix lists a pair of VAX Ada programs that demonstrate interprocessor communication between two KAV30s. The first program implements a FIFO producer. The second program implements a FIFO consumer.

### B.1 FIFO Producer

```
-- FIFO_PRODUCER.ADA
--
-- This program is one of a pair that demonstrates how the KAV30 FIFOs may be
-- used for inter-processor communication
--
with KAVDEF, CONDITION_HANDLING, SYSTEM, TEXT_IO, ERROR_HANDLING,
    VAXELN_SERVICES, AST_ROUTINES, SIGNALLER_TASK;

-- The ERROR_HANDLING package ships with VAXELN ADA to utilize this package
-- from your programs you must enter this package into your ADA program
-- manager library

procedure FIFO_PRODUCER is
    STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
    BUS_PAGE_PTR : SYSTEM.ADDRESS;
    IN_PAGE_PTR : SYSTEM.ADDRESS;
    OUTGOING_SGM_ENTRY : INTEGER;
    INCOMING_SGM_ENTRY : INTEGER;
    MAP_FUNCTIONS : INTEGER;
    UNMAP_FUNCTIONS : INTEGER;
    KAV_SERVICE_ERROR : exception;
    VAXELN_SERVICE_ERROR : exception;
    BUFFER : INTEGER;
    AST_PARAM : INTEGER;
    SIGNALLER : SIGNALLER_TASK.SIGNALLER;

package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);

begin
```

## Example Programs—Interprocessor Communication

```
-- create events for synchronization

VAXELN_SERVICES.CREATE_EVENT( EVENT      => AST_ROUTINES.TRIGGERED_EVENT,
                              INITIAL_STATE => VAXELN_SERVICES.CLEARED,
                              STATUS      => STATUS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise VAXELN_SERVICE_ERROR;
end if;

VAXELN_SERVICES.CREATE_EVENT( EVENT      =>
                              SIGNALLER_TASK.SIGNALLER_EVENT,
                              INITIAL_STATE => VAXELN_SERVICES.CLEARED,
                              STATUS      => STATUS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise VAXELN_SERVICE_ERROR;
end if;

-- start signaller task

SIGNALLER.START;

-- map CSR page (FIFOs) on other KAV30

MAP_FUNCTIONS := KAVDEF.KAV_M_VME + KAVDEF.KAV_M_MODE_0_SWAP;

KAVDEF.KAV_OUT_MAP( STATUS      => STATUS,
                   SGM_ENTRY    => OUTGOING_SGM_ENTRY,
                   PAGE_COUNT   => 1,
                   BUS_ADDRESS  => 16#F00000#,
                   VIRTUAL_ADDRESS => BUS_PAGE_PTR,
                   AM_CODE       => KAVDEF.KAV_K_USER_24,
                   MAP_FUNCTIONS => MAP_FUNCTIONS );

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- make our CSR page (FIFOs) visible to another KAV30

MAP_FUNCTIONS := KAVDEF.KAV_M_CSR + KAVDEF.KAV_M_MODE_0_SWAP;
INCOMING_SGM_ENTRY := 0;

KAVDEF.KAV_IN_MAP( STATUS      => STATUS,
                  SGM_ENTRY    => INCOMING_SGM_ENTRY,
                  PAGE_COUNT   => 1,
                  VIRTUAL_ADDRESS => IN_PAGE_PTR,
                  MAP_FUNCTIONS => MAP_FUNCTIONS );

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- set virtual address to point to FIFO 0

BUS_PAGE_PTR := SYSTEM."> (BUS_PAGE_PTR, 16#4000#);
```

## Example Programs—Interprocessor Communication

```
declare VALUE : INTEGER;
for VALUE use at BUS_PAGE_PTR;

begin
-- set up fifo notification on empty to not-empty state
    KAVDEF.KAV_NOTIFY_FIFO( STATUS      => STATUS,
                           FIFO_NUMBER => KAVDEF.KAV_K_FIFO_0,
                           FIFO_FUNCTIONS =>
                               KAVDEF.KAV_M_FIFO_NOT_EMPTY,
                           AST_ADDR      =>
                               AST_ROUTINES.AST_ROUTINE'ADDRESS);

    if not CONDITION_HANDLING.SUCCESS(STATUS) then
        raise KAV_SERVICE_ERROR;
    end if;

-- write a value to the remote KAV30 to start things off
    VALUE := 1;

    loop

-- wait for FIFO transition from empty to not empty
        VAXELN_SERVICES.WAIT_ANY( VALUE1 => SIGNALLER_TASK.SIGNALLER_EVENT,
                                STATUS => STATUS);

        if not CONDITION_HANDLING.SUCCESS(STATUS) then
            raise VAXELN_SERVICE_ERROR;
        end if;

-- clear signaller event
        VAXELN_SERVICES.CLEAR_EVENT( EVENT  =>
                                    SIGNALLER_TASK.SIGNALLER_EVENT,
                                    STATUS => STATUS);

        if not CONDITION_HANDLING.SUCCESS(STATUS) then
            raise VAXELN_SERVICE_ERROR;
        end if;

-- read 1 longword from FIFO 0
        KAVDEF.KAV_FIFO_READ( STATUS      => STATUS,
                             FIFO_NUMBER => KAVDEF.KAV_K_FIFO_0,
                             BUFFER      => BUFFER'ADDRESS,
                             COUNT       => 1);

        if not CONDITION_HANDLING.SUCCESS(STATUS) then
            raise KAV_SERVICE_ERROR;
        end if;

-- re-establish FIFO notification on empty to not-empty transition
```



## Example Programs—Interprocessor Communication

```
KAVDEF.KAV_NOTIFY_FIFO( STATUS          => STATUS,
                        FIFO_NUMBER      => KAVDEF.KAV_K_FIFO_0,
                        FIFO_FUNCTIONS   =>
                        KAVDEF.KAV_M_FIFO_NOT_EMPTY,
                        AST_ADDR         =>
                        AST_ROUTINES.AST_ROUTINE'ADDRESS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- display message on console

TEXT_IO.PUT("Read ");
INT_IO.PUT(BUFFER);
TEXT_IO.PUT(" writing ");
INT_IO.PUT(BUFFER+1);
TEXT_IO.NEW_LINE;

-- increment value read from FIFO 0 and write it to the other KAV30

VALUE := BUFFER + 1;

end loop;

end;

exception

when VAXELN_SERVICE_ERROR =>
    ERROR_HANDLING.DISPLAY_ERROR_MESSAGE( STATUS);

when KAV_SERVICE_ERROR =>
    ERROR_HANDLING.DISPLAY_ERROR_MESSAGE( STATUS);

when others
    =>
    raise;

end FIFO_PRODUCER;
```

## B.2 FIFO Consumer

```
-- FIFO_CONSUMER.ADA
--
-- This module forms one half of a pair of programs that demonstrate how the
-- KAV30 FIFOs can be used for inter-processor communication over VMEbus
--

with KAVDEF, CONDITION_HANDLING, SYSTEM, TEXT_IO, ERROR_HANDLING,
     VAXELN_SERVICES, AST_ROUTINES, SIGNALLER_TASK;

-- The ERROR_HANDLING package ships with VAXELN ADA to utilize this package
-- from your programs you must enter the ERROR_HANDLING package into your ADA
-- program manager library

procedure FIFO_CONSUMER is
STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
BUS PAGE_PTR : SYSTEM.ADDRESS;
IN PAGE_PTR : SYSTEM.ADDRESS;
OUTGOING_SGM_ENTRY : INTEGER;
INCOMING_SGM_ENTRY : INTEGER;
MAP_FUNCTIONS : INTEGER;
UNMAP_FUNCTIONS : INTEGER;
KAV_SERVICE_ERROR : exception;
VAXELN_SERVICE_ERROR : exception;
BUFFER : INTEGER;
AST_PARAM : INTEGER;
SIGNALLER : SIGNALLER_TASK.SIGNALLER;

package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);

begin
-- create the event objects that will be used for synchronization
VAXELN_SERVICES.CREATE_EVENT( EVENT      => AST_ROUTINES.TRIGGERED_EVENT,
                              INITIAL_STATE => VAXELN_SERVICES.CLEARED,
                              STATUS      => STATUS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise VAXELN_SERVICE_ERROR;
end if;

VAXELN_SERVICES.CREATE_EVENT( EVENT      =>
                              SIGNALLER_TASK.SIGNALLER_EVENT,
                              INITIAL_STATE => VAXELN_SERVICES.CLEARED,
                              STATUS      => STATUS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise VAXELN_SERVICE_ERROR;
end if;

-- start signaller task
```

## Example Programs—Interprocessor Communication

```
SIGNALLER.START;

-- map CSR page (FIFOs) on second KAV30
MAP_FUNCTIONS := KAVDEF.KAV_M_VME + KAVDEF.KAV_M_MODE_0_SWAP;

KAVDEF.KAV_OUT_MAP( STATUS      => STATUS,
                    SGM_ENTRY   => OUTGOING_SGM_ENTRY,
                    PAGE_COUNT  => 1,
                    BUS_ADDRESS => 16#E00000#,
                    VIRTUAL_ADDRESS => BUS_PAGE_PTR,
                    AM_CODE     => KAVDEF.KAV_K_USER_24,
                    MAP_FUNCTIONS => MAP_FUNCTIONS );

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- map CSR page (FIFOs) on this module so that another KAV30 can access it
MAP_FUNCTIONS := KAVDEF.KAV_M_CSR + KAVDEF.KAV_M_MODE_0_SWAP;
INCOMING_SGM_ENTRY := 0;

KAVDEF.KAV_IN_MAP( STATUS      => STATUS,
                  SGM_ENTRY   => INCOMING_SGM_ENTRY,
                  PAGE_COUNT  => 1,
                  VIRTUAL_ADDRESS => IN_PAGE_PTR,
                  MAP_FUNCTIONS => MAP_FUNCTIONS );

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- set virtual address to access FIFO 0 on the other KAV30
BUS_PAGE_PTR := SYSTEM."+"(BUS_PAGE_PTR,16#4000#);

declare VALUE : INTEGER;
for VALUE use at BUS_PAGE_PTR;

begin

-- set up FIFO notification on empty to not empty transition
KAVDEF.KAV_NOTIFY_FIFO( STATUS      => STATUS,
                      FIFO_NUMBER  => KAVDEF.KAV_K_FIFO_0,
                      FIFO_FUNCTIONS =>
                        KAVDEF.KAV_M_FIFO_NOT_EMPTY,
                      AST_ADDR     =>
                        AST_ROUTINES.AST_ROUTINE' ADDRESS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

loop

-- wait for signaller to signal us
```

## Example Programs—Interprocessor Communication

```
VAXELN_SERVICES.WAIT_ANY( VALUE1 => SIGNALLER_TASK.SIGNALLER_EVENT,
                          STATUS => STATUS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise VAXELN_SERVICE_ERROR;
end if;

-- clear signaller_event
VAXELN_SERVICES.CLEAR_EVENT( EVENT =>
                              SIGNALLER_TASK.SIGNALLER_EVENT,
                              STATUS => STATUS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise VAXELN_SERVICE_ERROR;
end if;

-- read 1 longword from FIFO
KAVDEF.KAV_FIFO_READ( STATUS      => STATUS,
                     FIFO_NUMBER => KAVDEF.KAV_K_FIFO_0,
                     BUFFER      => BUFFER'ADDRESS,
                     COUNT       => 1);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- re-establish fifo state transition notification
KAVDEF.KAV_NOTIFY_FIFO( STATUS      => STATUS,
                      FIFO_NUMBER  => KAVDEF.KAV_K_FIFO_0,
                      FIFO_FUNCTIONS =>
                      KAVDEF.KAV_M_FIFO_NOT_EMPTY,
                      AST_ADDR     =>
                      AST_ROUTINES.AST_ROUTINE'ADDRESS);

if not CONDITION_HANDLING.SUCCESS(STATUS) then
    raise KAV_SERVICE_ERROR;
end if;

-- write message to console
TEXT_IO.PUT("Read ");
INT_IO.PUT(BUFFER);
TEXT_IO.PUT(" writing ");
INT_IO.PUT(BUFFER);
TEXT_IO.NEW_LINE;

-- write value read from FIFO back to the other KAV30
VALUE := BUFFER;

end loop;

end;

exception
```

## Example Programs—Interprocessor Communication

```
when VAXELN_SERVICE_ERROR =>
    ERROR_HANDLING.DISPLAY_ERROR_MESSAGE( STATUS);

when KAV_SERVICE_ERROR =>
    ERROR_HANDLING.DISPLAY_ERROR_MESSAGE( STATUS);

when others
    =>
    raise;

end FIFO_CONSUMER;

-- AST_ROUTINES.ADA
--
-- AST routine for use with FIFO_PRODUCER / FIFO_CONSUMER example programs
--

with VAXELN_SERVICES;
package AST_ROUTINES is
    TRIGGERED_EVENT : VAXELN_SERVICES.EVENT_TYPE;
    procedure AST_ROUTINE;
    pragma EXPORT_PROCEDURE(AST_ROUTINE);
end AST_ROUTINES;

package body AST_ROUTINES is
    procedure AST_ROUTINE is
    begin
        VAXELN_SERVICES.SIGNAL_EVENT( EVENT => TRIGGERED_EVENT);
    end AST_ROUTINE;
end AST_ROUTINES;

-- SIGNALLER_TASK.ADA
--
-- This package contains the signaller process used by the FIFO_PRODUCER /
-- FIFO_CONSUMER example

with SYSTEM, VAXELN_SERVICES, AST_ROUTINES;
package SIGNALLER_TASK is
    SIGNALLER_EVENT : VAXELN_SERVICES.EVENT_TYPE;
    task type SIGNALLER is
        entry START;
    end SIGNALLER;
end SIGNALLER_TASK;

package body SIGNALLER_TASK is
```

## Example Programs—Interprocessor Communication

```
task body SIGNALLER is
STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    accept START;
    loop
-- wait for event to be signalled by the ast routine
        VAXELN_SERVICES.WAIT_ANY( STATUS => STATUS,
                                VALUE1 => AST_ROUTINES.TRIGGERED_EVENT);
-- clear the event
        VAXELN_SERVICES.CLEAR_EVENT( STATUS => STATUS,
                                    EVENT => AST_ROUTINES.TRIGGERED_EVENT);
-- signal signaller event
        VAXELN_SERVICES.SIGNAL_EVENT( STATUS => STATUS,
                                    EVENT => SIGNALLER_EVENT);
    end loop;
end SIGNALLER;
end SIGNALLER_TASK;

CHARACTERISTIC /SHARED STATUS /NOFILE /NET_DEVICE=EZA /NOSERVER /OBJECTS=512 -
/EMULATOR=BOTH /DEBUG=BOTH /P0_VIRTUAL_SIZE=4096 /P1_VIRTUAL_SIZE=512 -
/IO_REGION=2048 /TARGET=24
PROGRAM FIFO CONSUMER /WARM DEBUG
DEVICE EZA /VECTOR=%X130 /NET_DEF

CHARACTERISTIC /REMOTE_CLI /REMOTE_TERM /SHARED STATUS /NOFILE /NET_DEVICE=EZA -
/NOSERVER /OBJECTS=512 /EMULATOR=BOTH /DEBUG=BOTH /P0_VIRTUAL_SIZE=4096 -
/P1_VIRTUAL_SIZE=512 /IO_REGION=2048 /TARGET=24
PROGRAM FIFO PRODUCER /WARM DEBUG
DEVICE EZA /VECTOR=%X130 /NET_DEF

$!
$! INTERPROCESSOR_BUILD.COM
$
$ ACS SET LIBRARY USER:[USER.ADALIB]
$ ADA ERROR HANDLING, AST_ROUTINE, SIGNALLER_TASK
$ ADA/DEBUG FIFO_PRODUCER
$ ADA/DEBUG FIFO_CONSUMER
$ ACS LINK/DEBUG/SYSTEM=VAXELN_FIFO_PRODUCER ELN$:KAV$RTL_OBJS/LIB
$ ACS LINK/DEBUG/SYSTEM=VAXELN_FIFO_CONSUMER ELN$:KAV$RTL_OBJS/LIB
$ EBUILD/NOEDIT FIFO_CONSUMER
$ EBUILD/NOEDIT FIFO_PRODUCER
```



## Example Programs—MVME335 Device Driver

This appendix lists a pair of VAX C programs that implement a device driver for the MVME335 serial line module. The programs use ASTs to allow the ISR and the driver to communicate with each other. The first program is the device driver main body. The second program is the driver ISR.

Both programs are part of the KAV30 software kit. See the *KAV30 Software Installation and System Testing Information* for more information.

### C.1 Device Driver

```
#module mvme_driver_ast
/*****
 *
 * COPYRIGHT (C) 1991
 * BY DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
 *
 * THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
 * ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
 * INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
 * COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
 * OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
 * TRANSFERRED.
 *
 * THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
 * AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
 * CORPORATION.
 *
 *****/
```



## Example Programs—MVME335 Device Driver

```
*
*   DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
*   SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
*
*****/
/*
    FACILITY:  Device driver example for KAV30 using mvme335 serial line module
    DESCRIPTION:  driver body including initialisation and isr
                  this version uses 'KAV$QUE_AST' instead of
                  'ker$signal_device' to communicate from ISR
                  to driver body (this program).
*/

#include $vaxelnc
#include <eln$:kavdef.h>
#include "mvmedef.h"                                /* definition of module reg. */

#include stdio
#include $get_message_text
#include $kerneldef
#include $kernelmsg
#include descrip

/*
 *   Definitions and global variables for the driver
 */

#define mvme_base 0x03600                          /* mvme modul base address */
#define MVME_IRQLEV 0x08                          /* irq level on board = bit3 */
#define MVME_PHYS_ADDR 0x00

/*
 *   Define the COPY_BYTES macro -
 *   This macro copies the specified number of
 *   bytes from one string to another without
 *   any character interpretation.
 */

#define COPY_BYTES(src,dst,cnt)                    \
{                                                    \
    char *s = (src);                                \
    char *d = (dst);                                \
    int  c;                                          \
    for(c=(cnt);c;c--)                              \
        *d++ = *s++;                                \
}

struct mvme$dul_region *mvme$dul_reg_ptr;
```

## Example Programs—MVME335 Device Driver

```
struct mvme$packet
{
    int    function;
    int    error;
    int    length;
    char   buffer[];
};

/*
 *      'outside' definitions for global use
 */

DEVICE mvme$device;
PORT   mvme$driver_port;
MESSAGE mvme$message;
NAME   mvme$name;

struct mvme$packet *mvme$request;
int    request_size;

/*
 *      forward references for functions
 */

void    mvme$tx_ast      ();
void    mvme$rx_ast      ();

void    mvme$error_text  (int    status);
BOOLEAN mvme$cond_handler ();

/*****
 *
 *      Name:          MVME_DRIVER()
 *
 *      Abstract:      This is the driver main body
 *
 *      Inputs:        none
 *
 *      Outputs:       none
 *
 *      Comment:       has to run run in kernel mode
 *
 *****/

mvme_driver      ()
{
    extern void    mvme$duart1_isr      ();
    int    mvme$setup_module      ();
    int    status;

    int    mvme$status;
    int    mvme$ipl;
    int    mvme$vector;
```

## Example Programs—MVME335 Device Driver

```
BOOLEAN done;

VARYING_STRING(32) mvme$device_name;

static $DESCRIPTOR(mvme$port_name,"MVME$DRIVER_PORT");
static $DESCRIPTOR(device_name,"");

/*
 *      get the device name from the program argument list
 */

eln$program_argument(&mvme$device_name,1);
device_name.dsc$w_length = mvme$device_name.count;
device_name.dsc$a_pointer = mvme$device_name.data;

/*
 *      Create the device object and connect to ISR and communication region
 */

ker$create_device(&status,          /* status */
                  &device_name,    /* device name */
                  1,                /* relative vector */
                  mvme$duart1_isr,  /* interrupt service routine */
                  sizeof(struct mvme$dul_region), /* size of communications region */
                  &mvme$dul_reg_ptr, /* address of communications region */
                  NULL,            /* register pointer */
                  NULL,            /* adapter pointer */
                  &mvme$vector,    /* pointer to vector */
                  &mvme$ipl,       /* interrupt priority */
                  &mvme$device,    /* pointer to device variable */
                  MAX_CHANNELS,    /* number of devices to create */
                  NULL);           /* power fail isr (not needed) */
if (status != KER$SUCCESS)
    mvme$error_text (status);      /* return ker$create_device status */

/*
 *      establish condition handler
 */

VAXC$ESTABLISH      (mvme$cond_handler);

status = mvme$setup_module();      /* setup MVME335 module */
if (! (status & 1))
    mvme$error_text (status);

ker$job_port(&status,
             &mvme$driver_port);
if (! (status & 1))
    mvme$error_text (status);
```

## Example Programs—MVME335 Device Driver

```
ker$create_name(&status,
                &mvme$name,
                &mvme$port_name,
                &mvme$driver_port,
                NAME$LOCAL);
if (! (status & 1))
    mvme$error_text (status);
/*
 *   Set up the AST queue's for Send and Receive
 */

KAV$DEF_AST(&status,
            &mvme$dul_reg_ptr->tx_dev);
if (! (status & 1))
    mvme$error_text (status);

KAV$DEF_AST(&status,
            &mvme$dul_reg_ptr->rx_dev);
if (! (status & 1))
    mvme$error_text (status);
/*
 *   Initialization complete - inform the kernel.
 */

ker$initialization_done(NULL);

/*****
 *
 *   Driver main routine - we stay here forever -
 *
 *   the main tasks are:
 *   - wait for any caller and connect on request
 *   - wait for a request package, perform the request
 *   - disconnect on request
 *
 *   This version of the driver uses AST's for communication
 *   between ISR and driver instead of SIGNAL_DEVICE,
 *   the ISR to be used is MVMEDRIVER_ISR_TIME_AST.C
 *
 *****/
```

## Example Programs—MVME335 Device Driver

```
    for (;;)
    {
/*
 *      Connect on request
 */
        ker$accept_circuit(&status,
                           &mvme$driver_port,
                           NULL,
                           TRUE,
                           NULL,
                           NULL);

        if (! (status & 1))
            mvme$error_text (status);
        for (done = FALSE; !done;)
        {
/*
 *      Wait for request package and receive it
 */
            ker$wait_any(&status,
                        NULL,
                        NULL,
                        &mvme$driver_port);
            if (! (status & 1))
                mvme$error_text (status);
            ker$receive(&status,
                       &mvme$message,
                       &mvme$request,
                       &request_size,
                       &mvme$driver_port,
                       NULL,
                       NULL);
            if (! (status & 1))
                mvme$error_text (status);
/*
 *      Case on requested operation.
 */
            switch(mvme$request->function)
            {
                case RD_BLOCK_FUNC:
                    /* perform the READ_BLOCK_FUNCTION */
/*
 *      set up AST in pending queue (rx_device)
 */
```

## Example Programs—MVME335 Device Driver

```
KAV$SET_AST(&status,
            mvme$rx_ast,
            NULL,
            NULL,
            mvme$dul_reg_ptr->rx_dev);
if (! (status & 1))
    mvme$error_text (status);

/*
 * Initialize com. region at device_ipl (this disables irq's)
 */

ELN$DISABLE_INTERRUPT(mvme$ipl);
mvme$dul_reg_ptr->read_count      =
                                mvme$request->length;
mvme$dul_reg_ptr->rxbuf_ptr      = 0;
mvme$dul_reg_ptr->error          = FALSE;
mvme$dul_reg_ptr->read_in_progr  = TRUE;
mvme$dul_reg_ptr->dul_irqmask.rxa_ready = ENAB;
ELN$ENABLE_INTERRUPT();

/*
 * Enable receiver irq - since rx is enabled, we will see it immediately!
 */

write_register(mvme$dul_reg_ptr->dul_irqmask,
               mvme$dul_reg_ptr->a_dul_w_imr);

/*
 * Error checking and informing the caller about completion
 * is done in the AST routine (mvme$tx_ast) - so nothing to do here..
 */

break;

case WR_BLOCK_FUNC:
/* perform the WRITE_BLOCK_FUNCTION */

COPY_BYTES(mvme$request->buffer,
            mvme$dul_reg_ptr->write_buffer,
            mvme$request->length);

mvme$dul_reg_ptr->write_count= mvme$request->length;
mvme$dul_reg_ptr->txbuf_ptr  = 0;

/*
 * set up AST in pending queue (tx_device)
 */
```

## Example Programs—MVME335 Device Driver

```
        KAV$SET_AST(&status,
                    mvme$tx_ast,
                    NULL,
                    NULL,
                    mvme$dul_reg_ptr->tx_dev);
        if (! (status & 1))
            mvme$error_text (status);

/*
 *   Initialize com. region at device_ipl (this disables irq's)
 */

        ELN$DISABLE_INTERRUPT(mvme$ipl);
        mvme$dul_reg_ptr->write_in_progr      = TRUE;
        mvme$dul_reg_ptr->dul_irqmask.txa_ready = ENAB;
        mvme$dul_reg_ptr->dul_irqmask.rxa_ready = ENAB;
        ELN$ENABLE_INTERRUPT();

/*
 *   Enable transmitter irq - since tx is enabled, we will see it immediately!
 *   Enable receiver in order to get control char's
 */

        write_register(mvme$dul_reg_ptr->dul_irqmask,
                       mvme$dul_reg_ptr->a_dul_w_imr);

/*
 *   Error checking and informing the caller about completion
 *   is done in the AST routine (mvme$tx_ast) - so nothing to do here..
 */

        break;

    case DONE_FUNC:

        mvme$request->error = 0;
        ker$send( &status,
                  mvme$message,
                  request_size,
                  &mvme$driver_port,
                  NULL,
                  FALSE);
        if (! (status & 1))
            mvme$error_text (status);
        done      = TRUE;

    } /* end of switch block */

} /* end of for_loop (done = FALSE) */

/*
 *   Wait for disconnect message and disconnect
 */
```

## Example Programs—MVME335 Device Driver

```

        ker$wait_any(&status,
                    NULL,
                    NULL,
                    &mvme$driver_port);
    if (! (status & 1))
        mvme$error_text (status);

    ker$disconnect_circuit(&status,
                          &mvme$driver_port);
    if (! (status & 1))
        mvme$error_text (status);

    } /* and of for'ever' loop */
} /* end of driver main body */

/*****
*
*      Name:          MVME$SETUP_MODULE ()
*
*      Abstract:      This function will setup the initial register
*                    values needed by the module (on VMEbus). First
*                    the VME register addresses are mapped to S0.
*
*      Inputs:        None
*
*      Outputs:       Status
*
*      Comment:       On any exception the condition handler
*                    'mvme$cond_handler' is called
*
*****/

int mvme$setup_module ()
{
    int      status;

    struct  mvme_status      *a_dul_r_sra;
    struct  mvme_mode_one    *a_dul_mrla;
    struct  mvme_command     *a_dul_w_cra;
    struct  mvme_aux_control *a_dul_w_acr;
    struct  mvme_clock       *a_dul_w_csra;
    struct  mvme_irq_mask    *a_dul_w_imr;
    struct  mvme_irq_status  *a_dul_r_isr;
    struct  mvme_irq_vector  *a_dul_irv;
    struct  mvme_txbuf       *a_dul_w_txa;
    struct  mvme_rxbuf       *a_dul_r_rxa;

    int setup_function, buffer; /* parameters for $VME_SETUP */
    int entry, pagecnt;        /* parameters for $OUT_MAP */
    int physical_addr, vir_addr;
    int am_code, map_functions;

```



## Example Programs—MVME335 Device Driver

```
/*
 *      Enable VMEirq level 3
 */

    buffer          = 0x08;                /* VME irq 3 !! */
    setup_function  = KAV$K_ALLOW_VME_IRQ;
    KAV$VME_SETUP   (&status,
                     setup_function,
                     &buffer);
    if (! (status & 1))
        mvme$error_text(status);

/*
 * map mvme335 register space (on vme-bus) to S0 space
 */
    pagecnt        = 1;
    physical_addr   = 0x00000000;          /* phys address at page bound.*/
    am_code         = KAV$K_USER 16;       /* 'short user mode 0x29' */
    map_functions   = KAV$M_VME+KAV$M_MODE_0_SWAP;
    KAV$OUT_MAP     (&status,
                     &entry,
                     pagecnt,
                     physical_addr,
                     &vir_addr,
                     am_code,
                     map_functions);
    if (! (status & 1))
        mvme$error_text(status);

/*
 * set up mvme register address (virtual) using <vir_addr> as base
 */

    a_dul_r_sra = vir_addr + mvme_base + dul_r_sra; /* status register a */
    a_dul_w_imr = vir_addr + mvme_base + dul_w_imr; /* irq mask register a&b*/
    a_dul_w_cra = vir_addr + mvme_base + dul_w_cra; /* command register a */
    a_dul_mrla = vir_addr + mvme_base + dul_mrla; /* mode register a one */
    a_dul_w_acr = vir_addr + mvme_base + dul_w_acr; /* aux. command reg. a */
    a_dul_w_csra = vir_addr + mvme_base + dul_w_csra; /* clock select reg. a */
    a_dul_w_txa = vir_addr + mvme_base + dul_w_txa; /* transmitter buffer a */
    a_dul_r_rxa = vir_addr + mvme_base + dul_r_rxba; /* receiver buffer a */
    a_dul_r_isr = vir_addr + mvme_base + dul_r_isr; /* irq status duart1 */
    a_dul_irv = vir_addr + mvme_base + dul_irv; /* irq vector duart1 */

/*
 * copy register address to com.region
 */
```

## Example Programs—MVME335 Device Driver

```
mvme$dul_reg_ptr->a_dul_r_sra = a_dul_r_sra;
mvme$dul_reg_ptr->a_dul_w_imr = a_dul_w_imr;
mvme$dul_reg_ptr->a_dul_w_cra = a_dul_w_cra;
mvme$dul_reg_ptr->a_dul_mrla = a_dul_mrla;
mvme$dul_reg_ptr->a_dul_w_acr = a_dul_w_acr;
mvme$dul_reg_ptr->a_dul_w_csra = a_dul_w_csra;
mvme$dul_reg_ptr->a_dul_w_txa = a_dul_w_txa;
mvme$dul_reg_ptr->a_dul_r_rxa = a_dul_r_rxa;
mvme$dul_reg_ptr->a_dul_r_isr = a_dul_r_isr;
mvme$dul_reg_ptr->a_dul_irv = a_dul_irv;

/*
 * initialize registers on mvme335 duart1 channel a only with the following
 * parameters: 9.6 kBaud, 8bits, no parity
 * all default parameters can be found in mvmedef.h
 */

/* initialize duart1 irq mask register [dul_w_imr]: no irq */
write_register      (dul_irqmask, a_dul_w_imr);

/* perform bit set on channel a command register [dul_w_cra] reset mrla ptr */
write_register      (dula_command, a_dul_w_cra);

/* write channel a mode register one [dul_mrla] ptr -> mode register two */
write_register      (dula_mode_one, a_dul_mrla);

/* write channel a mode register two [dul_mrla] */
write_register      (dula_mode_two, a_dul_mrla);

/* write channel a command register [dul_w_cra]: reset rx, flush FIFO
dula_command.misc = RESET_RX;
write_register      (dula_command, a_dul_w_cra);

/* write channel a command register [dul_w_cra]: reset tx */
dula_command.misc = RESET_TX;
write_register      (dula_command, a_dul_w_cra);

/* write channel a command register [dul_w_cra]: reset error status */
dula_command.misc = RESET_ERR;
write_register      (dula_command, a_dul_w_cra);

/* write channel a auxiliary control register [dul_w_acr] */
write_register      (dul_acr, a_dul_w_acr);

/* write channel a clock register [dul_w_csra]: 9600 baud both */
write_register      (dula_clock, a_dul_w_csra);

/* initialize duart1 vector register [dul_irv] (default = 0x00) */
dul_irqvec.irq_vector= 0x02;
write_register      (dul_irqvec, a_dul_irv);

/* enable transmitter and receiver [dula_command]*/
mvme$dul_reg_ptr->dula_comm.ena_tx = ENAB;
mvme$dul_reg_ptr->dula_comm.ena_rx = ENAB;
write_register      (mvme$dul_reg_ptr->dula_comm, a_dul_w_cra);
```

## Example Programs—MVME335 Device Driver

```
    return (status);
    /* end of SETUP_MODULE */
}

/*****
 *
 *      Name:          RX_AST
 *
 *      Abstract:      ast routine for rx events
 *
 *      Inputs:        None
 *
 *      Outputs:       None
 *
 *      Comment:       This routine is called every time the ISR delivers
 *                    an AST for the rx_device.
 *
 *****/

void    mvme$rx_ast    ()
{
    int    status;

    /*
     *      check for error and correct length
     */

    if (mvme$dul_reg_ptr->error)
    {
        mvme$request->error = -1;
        mvme$request->length = mvme$dul_reg_ptr->rxbuf_ptr;
    }
    else
        mvme$request->error = 0;

    /*
     *      copy buffer
     */

    COPY_BYTES(mvme$dul_reg_ptr->read_buffer,
                mvme$request->buffer,
                mvme$request->length);

    /*
     *      send message to caller and return
     */
}
```

## Example Programs—MVME335 Device Driver

```
ker$send( &status,
          mvme$message,
          request_size,
          &mvme$driver_port,
          NULL,
          FALSE);
if (! (status & 1))
    mvme$error_text (status);
return;
}
/*****
*
*      Name:          TX_AST
*
*      Abstract:      ast routine for tx events
*
*      Inputs:        None
*
*      Outputs:       None
*
*      Comment:       this routine is called every time the ISR delivers
*                     an AST for the tx_device
*
*****/
void mvme$tx_ast      ()
{
/*
*      reset error and send message to caller
*/
    mvme$request->error = 0;
    ker$send( NULL,
              mvme$message,
              request_size,
              &mvme$driver_port,
              NULL,
              FALSE);
```

## Example Programs—MVME335 Device Driver

```
    return;
}
/*****
*
*      Name:          MVME$COND_HANDLER (signal_ptr,mechanism_ptr)
*
*      Abstract:      condition handler invoked by any exception
*
*      Inputs:        None
*
*      Outputs:       status = 1
*
*      Comment:       handler will not terminate the program,
*                    the condition is not really handled here...!
*
*****/
BOOLEAN mvme$cond_handler (signal_ptr,mechanism_ptr)
struct  chf$signal_array
{
    int chf$l_sig_args;
    int chf$l_sig_name;
    int chf$l_sig_arg1;
};

struct  chf$signal_array *signal_ptr;
struct  chf$mech_array *mechanism_ptr;

{
    void    mvme$error_text (int status);          /* show error text    */
    static int status, address;

    printf("Condition handler:");
    status = signal_ptr->chf$l_sig_name;
    mvme$error_text (status);

    address = signal_ptr->chf$l_sig_arg1;
    printf("at VME address %x \n", address);

    exit (1);
}
```

## Example Programs—MVME335 Device Driver

```
/******
*
*      Name:          MVME$ERROR_TEXT()
*
*      Abstract:      Routine converts kernel error number's to text
*                    and print's it
*
*      Inputs:        status
*
*      Outputs:       none
*
*      Comment:       none
*
******/
void    mvme$error_text (int status)          /* send error message to console*/
{
    int          text_flags;                /* parameters for $get_message */
    char         text_buffer[255];
    VARYING_STRING(255) result_string;
    text_flags = STATUS$ALL;
    eln$get_status_text ( status,
                          text_flags,
                          &result_string);
    VARYING_TO_CSTRING ( result_string, text_buffer);
    printf("%s\n", text_buffer);
    printf("KAV$XXX Error : %d \n", status);
    return;
}
```

## Example Programs—MVME335 Device Driver

### C.2 Interrupt Service Routine

```
#module mvme_isr_ast

/*****
 *
 *   COPYRIGHT (C) 1991
 *   BY DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
 *
 *   THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
 *   ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
 *   INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
 *   COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
 *   OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
 *   TRANSFERRED.
 *   THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
 *   AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
 *   CORPORATION.
 *
 *   DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
 *   SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
 *
 *****/

#include $vaxelnc
#include <eln$:kavdef.h>
#include "mvmedef.h"                /* definition of module reg. */

#include stdio
#include $kerneldef

/*****
 *
 *   Name:          MVME$DUART1_ISR()
 *
 *   Abstract:      This is the ISR for duart1
 *
 *   Inputs:        *register_ptr    = ptr to register (not used)
 *                  *mvme$dul_reg_ptr= ptr to comm. region
 *
 *   Outputs:       none
 *
 *   Comment:       this routine is invoked by an irq on the VMEbus,
 *                  it executes in kernel mode.
 *                  Every acces to the VMEbus has to use the
 *                  KAV$BUS_READ or KAV$BUS_WRITE system service.
 *****/

void    mvme$duart1_isr (register_ptr, mvme$dul_ptr)

struct  register_def    *register_ptr;    /* has to be there! */
struct  mvme$dul_region *mvme$dul_ptr;
```

## Example Programs—MVME335 Device Driver

```
{
int     status, data_type, count, char_num, i;
char    control_buf[3];

count   = 1;
char_num = 0;
i       = 0;
data_type = KAV$K_BYTE;

/*
 *      Determine source of irq - get irq_reg and check bit's
 */

    KAV$BUS_READ(&status,
                 data_type,
                 mvme$dul_ptr->a_dul_r_isr,
                 &mvme$dul_ptr->dul_irqstat,
                 count);

/*****
 *      If irq is for tx channel_a - process it
 *****/

    if (mvme$dul_ptr->dul_irqstat.txa_ready)
    {

/*
 *      If the driver is waiting for output, output
 *      characters to the mvme-module until done.
 */

        if (mvme$dul_ptr->write_in_progr)
            if (mvme$dul_ptr->write_count > mvme$dul_ptr->txbuf_ptr)
                KAV$BUS_WRITE(&status,
                              data_type,
                              mvme$dul_ptr->a_dul_w_txa,
                              &mvme$dul_ptr->write_buffer
                              [mvme$dul_ptr->txbuf_ptr++],
                              count);

            else

/*
 *      All done, reset tx & rx irq_mask bit
 */

                {
                    mvme$dul_ptr->write_in_progr      = FALSE;
                    mvme$dul_ptr->dul_irqmask.rxa_ready = 0;
                    mvme$dul_ptr->dul_irqmask.txa_ready = 0;
```



## Example Programs—MVME335 Device Driver

```
        KAV$BUS_WRITE(&status,
                      data_type,
                      mvme$dul_ptr->a_dul_w_imr,
                      &mvme$dul_ptr->dul_irqmask,
                      count);
/*
 *      Copy error status to comm.reg and signal device
 */

        mvme$dul_ptr->status          = status;
        KAV$QUE_AST (&status,
                     mvme$dul_ptr->tx_dev);
    }
}

/*****
 *      If irq is for rx channel a - process it
 *****/
    if (mvme$dul_ptr->dul_irqstat.rxa_ready)
    {
/*
 *      First check status reg. for any error (not implemented)
 */

        KAV$BUS_READ(&status,
                     data_type,
                     mvme$dul_ptr->a_dul_r_sra,
                     &mvme$dul_ptr->dula_stat,
                     count);

/*
 *      If the driver is waiting for input, read
 *      characters from the mvme-module until done.
 */
```

## Example Programs—MVME335 Device Driver

```
if (mvme$dul_ptr->read_in_progr)
{
    if (mvme$dul_ptr->read_count > mvme$dul_ptr->rxbuf_ptr)
    {
        KAV$BUS_READ(&status,
            data_type,
            mvme$dul_ptr->a_dul_r_rxa,
            &mvme$dul_ptr->read_buffer
            [mvme$dul_ptr->rxbuf_ptr],
            count);
/*
 *   check for carriage control and stop if ...
 */
        if (mvme$dul_ptr->read_buffer[mvme$dul_ptr->rxbuf_ptr] == CR)
        {
/*
 *   CR received -> all done, reset rx irq_mask bit
 */
            mvme$dul_ptr->error          = TRUE;
            mvme$dul_ptr->read_in_progr  = FALSE;
            mvme$dul_ptr->dul_irqmask.rxa_ready = 0;

            KAV$BUS_WRITE(&status,
                data_type,
                mvme$dul_ptr->a_dul_w_imr,
                &mvme$dul_ptr->dul_irqmask,
                count);
/*
 *   Copy error status to comm.reg and signal device
 */
            mvme$dul_ptr->status          = status;
            KAV$QUEUE_AST (&status,
                mvme$dul_ptr->rx_dev);
        }
        mvme$dul_ptr->rxbuf_ptr++;
    }
    else
/*
 *   Buffer full, all done, reset rx irq_mask bit
 */
    {
        mvme$dul_ptr->read_in_progr      = FALSE;
        mvme$dul_ptr->dul_irqmask.rxa_ready = 0;

        KAV$BUS_WRITE(&status,
            data_type,
            mvme$dul_ptr->a_dul_w_imr,
            &mvme$dul_ptr->dul_irqmask,
            count);
/*
 *   Copy error status to comm.reg and signal device
 */
    }
```

## Example Programs—MVME335 Device Driver

```
        mvme$dul_ptr->status          = status;
        KAV$QUE_AST (&status,
                    mvme$dul_ptr->rx_dev);
    }
}

/*
 *   No read_in_progr, is it a control char for write_in_progr ?
 */

else    if (mvme$dul_ptr->write_in_progr)
{
    while      (mvme$dul_ptr->dula_stat.rx_ready)
    {
        KAV$BUS_READ(&status,
                    data_type,
                    mvme$dul_ptr->a_dul_r_rxa,
                    &control_buf[char_num],
                    count);

/*
 *   check status again if there is more than one char
 */
        ,

        KAV$BUS_READ(&status,
                    data_type,
                    mvme$dul_ptr->a_dul_r_sra,
                    &mvme$dul_ptr->dula_stat,
                    count);

        char_num++;
    }

    while      (i < char_num)
    {

/*
 *   If XOFF, disable transmitter
 */

        if (control_buf[i] == XOFF)
        {
            mvme$dul_ptr->dula_comm.ena_tx  = DISAB;
            KAV$BUS_WRITE(&status,
                        data_type,
                        mvme$dul_ptr->a_dul_w_cra,
                        &mvme$dul_ptr->dula_comm,
                        count);

        }

/*
 *   If XON, enable transmitter
 */
    }
```

## Example Programs—MVME335 Device Driver

```
else    if  (control_buf[i] == XON)
        {
            mvme$dul_ptr->dula_comm.ena_tx  = ENAB;
            KAV$BUS_WRITE(&status,
                          data_type,
                          mvme$dul_ptr->a_dul_w_cra,
                          &mvme$dul_ptr->dula_comm,
                          count);
        }

        i++;
    }

/*
 *   If it's an unexpected char, disregard it
 */
    }

/*
 *   If irq is not expected here, disregard
 */
    }

}
```



---

## Example Programs—VDAD Device Driver

This appendix lists the following files:

- A VAX C program that implements a device driver for the VDAD (I/O) module
- A definitions file for the VDAD I/O module device driver
- A VAX C program that tests the VDAD I/O module device driver
- A build file for the VDAD I/O module device driver and test program
- A VAXELN System Builder data file for the VDAD I/O module device driver test program

All the files are part of the KAV30 software kit. See the *KAV30 Software Installation and System Testing Information* for more information.

### D.1 Device Driver

```
#module VDADdriver
/*
 * FACILITY:
 * VAXELN Run Time System
 *
 * ABSTRACT:
 *     This module contains an ELN Driver for a VMEbus device.
 *
 * VERSION:
 *     V1.04 13-Mar-1991 Field Test Release.
```

## Example Programs—VDAD Device Driver

```
*
* NOTES:
*   This is an example of a driver for a VMEbus device - the PEP VDAD
*   module. This device offers ADC, DAC, Timers and Digital I/O, although
*   only Analaoq-to-Digital conversion is provided here.
*
*   The program illustrates the most important Driver functions, namely:
*       ELN Driver interface,
*       interrupt handling,
*       VMEbus device access,
*       user calling interface
*
*   It does NOT provide access to the full functionality of this module
*   - it merely demonstrates the methods required in order to access the
*   board under ELN driver philosophy.
*       *** It provides basic ADC sampling only ***
*
*   Similarly, error handling and parameter checking are incomplete.
*
*/

#include stdio
#include $get_message_text
#include types
#include chfdef
#include in
#include descrip          /* descriptor definitions */
#include $mutex           /* mutex */
#include $kernelmsg       /* kernel messages */
#include $vaxelnc
#include <eln$:kavdef.h>   /* KAV300      "      */
#include "vdaddriver.h"    /* VDAD definitions */
/*
*   ****
*   *** Global variables for Driver routines ***
*   ****
*/

int          vdad$status ;          /* Global status variable.      */
unsigned long vdad$kav_setup ;      /* for $VME_SETUP              */
unsigned long vdad$kav_data ;
unsigned long vdad$entry ;
unsigned long vdad$vir_addr_M0 ;    /* addr mode: Byte/Word swapping */
unsigned long vdad$vir_addr_M3 ;    /* addr mode: NO swapping.      */
VARYING_STRING(32) vdad$controller_name; /* device name for controller */
int          vdad$ipl ;             /* device priority level        */
```

## Example Programs—VDAD Device Driver

```
vdad_region_type *vdad$region ;           /* ptr to region for ISR comms. */
EVENT            vdad$event_init;         /* initialization sync event */
DEVICE           vdad$device[MAX_CHANNELS] ; /* device objects for signalling */
MUTEX            vdad$mutex[MAX_CHANNELS] ; /* controller ownership mutex's */
int              vdad$channels[MAX_CHANNELS]; /* Array of process (channel) id's */
BOOLEAN         vdad$interrupts_enabled ; /* Polling or Interrupt ops. */
int             vdad$vector ;             /* Interrupt vector. */
BOOLEAN         vdad$monitoring ;         /* Statistics-gathering off/on. */
unsigned long    vdad$conditions,         /* ...for statistics-gathering. */
                vdad$setups,              /* (see 'monitoring'). */
                vdad$reads ;              /* (see 'monitoring'). */

/*
 * Forward References for functions
 */
void          vdad$error_text();
BOOLEAN      vdad$cond_handler();

/*
 * Pointers to VDAD registers.
 */
unsigned short *p_adc_read_conv_addr ;    /* (WORD-addressable register) */
unsigned char  *p_portA_data_dir_reg,     /* (BYTE-addressable registers) */
                *p_portA_control_reg,
                *p_portA_data_reg,
                *p_port_gen_control_reg,
                *p_port_service_req_reg,
                *p_port_int_vec_reg,
                *p_port_status_reg,
                *p_int_level_reg ;
```



## Example Programs—VDAD Device Driver

```
/*
 * Useful macros.
 */
#define ERROR_DETECTED (!(status & 1)) /* ...for error-testing... */
/*
 * *****
 * *** VDAD-DRIVER main routine code ***
 * *****
 *
 * This is the main routine for the PEP VDAD driver.
 * It first initialises the ELN Device control structures, then
 * the VMEbus-specific structures.
 *
 * This is a ONCE-ONLY routine - it must NOT be called more than once.
 *
 * Note that we have to drop into Kernel mode in order to execute the
 * KER$CREATE_DEVICE call. Otherwise, we remain in User mode (or whatever
 * mode we were in).
 */
int vdad$init()
{
    int status ;
    int initialise() ;
    struct { int arg_count, *status ; } arg_block = { 1, 0 } ;

    /*
     * Initialise the VDAD Device into ELN.
     */
    ker$enter_kernel_context ( &status,
                               initialise,
                               &arg_block ) ;
    if ERROR_DETECTED vdad$error_text(status);

    /*
     * Initialise the VME-bus mapping, etc..
     */
    status = VMEbus_init();
    if ERROR_DETECTED vdad$error_text(status);

    return( status ) ;
}
```

## Example Programs—VDAD Device Driver

```

/*
 *
 * *****
 * ***   I N I T I A L I S E   ***
 * *****
 *
 *      This routine must execute in KERNEL MODE !!!
 *      -----
 */
int initialise()
{
    void    vdad_isr();
    void    channel_process();
    int     status, channel ;
    static  $DESCRIPTOR(device_name, "");

    vdad$setups    = 0 ;    /* Number of times SETUP has been called. */

    /*
     * Get the device name from the program argument list
     * (NB: Program Argument number FOUR ! [params 1,2 & 3 are required
     *      for Vax-C I/O])
     */

    eln$program_argument(&vdad$controller_name, 4);
    device_name.dsc$a_pointer  = vdad$controller_name.data;
    device_name.dsc$w_length   = vdad$controller_name.count;
    /*
     * Create the device object
     */
    ker$create_device(
        &status,                /* Status */
        &device_name,          /* Device name */
        RELATIVE_VECTOR,       /* Relative vector (NEVER ZERO !) */
        vdad_isr,              /* Interrupt service routine */
        sizeof(vdad$region),    /* Size of communications region */
        &vdad$region,          /* Address of communications region */
        NULL,                  /* Register pointer */
        NULL,                  /* Adapter pointer */
        &vdad$vector,          /* Pointer to vector */
        &vdad$iopl,            /* Interrupt priority */
        &vdad$device[0],       /* ptr to receive device variable */
        MAX_CHANNELS,          /* Number of devices to create */
        NULL);                 /* Power fail isr (not needed) */
    if ERROR_DETECTED vdad$error_text(status);

    /*
     * Create the controller protection mutex's (one per channel).
     */
    for (channel = 0; channel < MAX_CHANNELS; channel++)
    {
        ELN$CREATE_MUTEX( vdad$mutex[channel] , &status);
        if ERROR_DETECTED    vdad$error_text(status);
    }
}

```

## Example Programs—VDAD Device Driver

```
return( status ) ;
}
/* *****
*      ***   I N I T I A L I Z E   V M E - b u s   ***
*      *****
*
* This routine is called to initialise the VMEbus mapping, and also
* internal pointers into the VDAD's registers.
*
* Returns: status
*
* Inputs:  none
*/

int  VMEbus_init()
{
unsigned long  phy_addr, addr_mode;    /* address ptrs for quick ref      */
unsigned long  pagecnt;
int            status, KAV_flags;

vdad$kav_data      = 0x00000000;        /* No IRQ allowed - */
vdad$kav_setup     = KAV$K_ALLOW_VME_IRQ; /* (initially, at least) */
KAV$VME_SETUP( &status,
               vdad$kav_setup,
               &vdad$kav_data );
if ERROR_DETECTED  vdad$error_text(status);
```

## Example Programs—VDAD Device Driver

```

pagecnt    = 1;                                /* No. of 64K pages */
phy_addr   = VDAD_PHYS_ADDR ;
addr_mode  = KAV$K_USER 24 ;                    /* Standard User Mode */
KAV_flags  = KAV$M_VME+KAV$M_MODE_0_SWAP;      /* Byte Swapping */
/*
 * Note regarding byte/word swapping:
 * =====
 *
 *      Since most of the VDAD registers are byte accessed,
 *      and the VDAD itself is a "big endian", then we MUST
 *      set up for byte-swapping.
 *      HOWEVER, one register - where the 12-bit sampled data
 *      is read - is put onto the bus UN-BYTE-SWAPPED. If we
 *      attempt to access it in byte/word swapping mode, then
 *      the two bytes (containing the 12-bit sample) will be
 *      erroneously swapped, necessitating byte-swapping here
 *      in the Driver. Since this introduces an unacceptable
 *      overhead (ie. byte-swapping EACH sample), we must use
 *      another method.
 *      The solution is to create TWO 'OUT_MAP's - one with
 *      byte-swapping, the other without.
 */
KAV$OUT_MAP ( &status,
              &vdad$entry,
              pagecnt,
              phy_addr,
              &vdad$vir_addr_M0,
              addr_mode,
              KAV_flags);
if ERROR_DETECTED vdad$error_text(status);

KAV_flags = KAV$M_VME+KAV$M_MODE_3_SWAP;        /* NO Byte Swapping */
/*
 * Now map with NO byte-swapping. We use this when accessing
 * WORD registers on VDAD.
 */
KAV$OUT_MAP ( &status,
              &vdad$entry,
              pagecnt,
              phy_addr,
              &vdad$vir_addr_M3,
              addr_mode,
              KAV_flags);
if ERROR_DETECTED vdad$error_text(status);

```

## Example Programs—VDAD Device Driver

```
/*
 *      Setup the VDAD register pointers (virtual)
 */
p_portA_data_dir_reg = vdad$vir_addr_M0 + VDAD_BASE + -
+ OFFSET_portA_data_dir_reg;
p_portA_control_reg = vdad$vir_addr_M0+VDAD_BASE+OFFSET_portA_control_reg;
p_portA_data_reg = vdad$vir_addr_M0 + VDAD_BASE + OFFSET_portA_data_reg;
p_port_gen_control_reg = vdad$vir_addr_M0 + VDAD_BASE + -
+ OFFSET_port_gen_control_reg;
p_port_service_req_reg = vdad$vir_addr_M0 + VDAD_BASE + -
+ OFFSET_port_service_req_reg;
p_port_int_vec_reg = vdad$vir_addr_M0 + VDAD_BASE + OFFSET_port_int_vec_reg;
p_port_status_reg = vdad$vir_addr_M0 + VDAD_BASE + OFFSET_port_status_reg;
p_int_level_reg = vdad$vir_addr_M0 + VDAD_BASE + OFFSET_int_level_reg;
p_adc_read_conv_addr = vdad$vir_addr_M3 + VDAD_BASE + -
+ OFFSET_adc_read_conv_addr;

/*
 *      Copy the device registers into the comms region, so that the
 *      Interrupt Service rtne (vdad_isr) can access the VDAD registers.
 */
vdad$region->p_portA_data_dir_reg    = p_portA_data_dir_reg ;
vdad$region->p_portA_control_reg     = p_portA_control_reg ;
vdad$region->p_portA_data_reg        = p_portA_data_reg ;
vdad$region->p_port_gen_control_reg  = p_port_gen_control_reg ;
vdad$region->p_port_service_req_reg  = p_port_service_req_reg ;
vdad$region->p_port_int_vec_reg      = p_port_int_vec_reg ;
vdad$region->p_port_status_reg       = p_port_status_reg ;
vdad$region->p_int_level_reg         = p_int_level_reg ;
vdad$region->p_adc_read_conv_addr    = p_adc_read_conv_addr ;

return( status ) ;
} /*      end    initialise_controller      */
```

## Example Programs—VDAD Device Driver

```

/*-----*/
/*
 *  s e t u p _ c o n t r o l l e r
 *
 *  This routine is called to perform the controller setup.
 *
 *  Returns: int      status  (see .H file for definitions)
 *
 *  Inputs:  int      gain,
 *
 *              input_config,
 *              channel_mode,
 *              trigger_type,
 *              interrupt_handling,
 *              condition_handling,
 *              monitor_switch
 *
 *  Note: Any of the above params may be NULled, in which case the 'current'
 *        default is used. This means that it is possible to, say, change
 *        ONLY the GAIN whilst the device is operating.
 *
 */
int  vdad$setup( gain, input_config, channel_mode, trigger_type,
                interrupt_handling, condition_handling, monitor_switch )
int
    gain,
    input_config,
    channel_mode,
    trigger_type,
    interrupt_handling,
    condition_handling,
    monitor_switch ;

{
    int          status ;
    unsigned char temp ;
    union {
        portA_data_reg  bits ;      /* (for access to bits)          */
        unsigned char   byte ;      /* (BYTE access to entire register) */
    } setup_reg ;
    union {
        portA_control_reg bits ;      /* (for access to bits)          */
        unsigned char     byte ;      /* (BYTE access to entire register) */
    } control_reg ;

    vdad$conditions = 0 ;           /* Zero this out (a driver global). */
    vdad$reads      = 0 ;           /* .....                          */
    vdad$setups++ ;                 /* Log this SETUP call (a driver global). */

    if ( condition_handling == VDAD$SETUP_COND_HANDLING_DRIVER )
    {
        VAXC$ESTABLISH(vdad$cond_handler); /* Establish the DRIVER's Handler */
    } ;
}

```

## Example Programs—VDAD Device Driver

```
/*
 *
 *          Device Initialization
 *          -----
 *   We first have to read the register contents to access the 'current'
 *   defaults, in the case where the user has NULL'ed one or more of
 *   the input parameters.
 */
kav$bus_read(&status, KAV$K_BYTE, p_portA_data_reg, &setup_reg, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

kav$bus_read(&status, KAV$K_BYTE, p_portA_control_reg, &control_reg, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

switch ( gain )
{
    /* <<< GAIN >>> */
    /* ----- */
    case VDAD$SETUP_GAIN_1 :      setup_reg.bits.gain = 0 ; break ;
    case VDAD$SETUP_GAIN_10 :     setup_reg.bits.gain = 1 ; break ;
    case VDAD$SETUP_GAIN_100 :    setup_reg.bits.gain = 2 ; break ;
    default: break ;
} ; /* end CASE */

switch ( input_config )
{
    /* <<< INPUT CONFIGURATION >>> */
    /* ----- */
    case VDAD$SETUP_INP_CONFIG_1 : setup_reg.bits.input_config = 0 ;
    break ;
    case VDAD$SETUP_INP_CONFIG_2 : setup_reg.bits.input_config = 1 ;
    break ;
    case VDAD$SETUP_INP_CONFIG_3 : setup_reg.bits.input_config = 2 ;
    break ;
    case VDAD$SETUP_INP_CONFIG_4 : setup_reg.bits.input_config = 3 ;
    break ;
    default: break ;
} ; /* end CASE */

switch ( channel_mode )
{
    /* <<< SINGLE/MULTI-CHANNEL MODE >>> */
    /* ----- */
    case VDAD$SETUP_CHANNEL_MODE_SINGLE: setup_reg.bits.channel_mode = 0 ;
    break ;
    case VDAD$SETUP_CHANNEL_MODE_MULTIPLE: setup_reg.bits.channel_mode = 1 ;
    break ;
    default: break ;
} ; /* end CASE */
```

## Example Programs—VDAD Device Driver

```

switch ( trigger_type )          /* <<< TRIGGER >>> */
{                                /* ----- */
    case VDAD$SETUP_TRIGGER_SOFTWARE :      setup_reg.bits.trigger_type = 0;
    break ;
    case VDAD$SETUP_TRIGGER_EXTERN   :      setup_reg.bits.trigger_type = 1;
    break ;
    case VDAD$SETUP_TRIGGER_TIMER    :      setup_reg.bits.trigger_type = 2;
    break ;
    case VDAD$SETUP_TRIGGER_EXTERN_AND_TIMER: setup_reg.bits.trigger_type = 3;
    break ;
    default: break ;
}; /* end CASE */

switch ( interrupt_handling )     /* <<< INTERRUPTS or POLLING >>> */
{                                /* ----- */
    case VDAD$SETUP_INT_DISABLE : control_reg.bits.EOC_ien = 0 ;
                                vdad$interrupts_enabled = FALSE ;
                                break ;
    case VDAD$SETUP_INT_ENABLE  : control_reg.bits.EOC_ien = 1 ;
                                vdad$interrupts_enabled = TRUE  ;
                                break ;

    default: break ;
}; /* end CASE */

switch ( monitor_switch )        /* <<< Statistics-gathering >>> */
{                                /* ----- */
    case VDAD$SETUP_MONITORING_OFF : vdad$monitoring = FALSE ; break ;
    case VDAD$SETUP_MONITORING_ON  : vdad$monitoring = TRUE  ; break ;
    default: break ;
}; /* end CASE */

/*
 * Register bits set up OK. Now initialise the device, depending on
 * whether interrupts or polling has been selected.
 */
if (vdad$interrupts_enabled)

    /* ===== */
    {                                /* <<< Setup for INTERRUPTS >>> */
                                    /* ===== */
        temp = VDAD_IRQL;           /* (Interrupt Request Level) */
        kav$bus_write(&status, KAV$K_BYTE, p_int_level_reg, &temp, 1);
        if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

        temp = VDAD_IRQV;           /* (Interrupt Vector Number) */
        kav$bus_write(&status, KAV$K_BYTE, p_port_int_vec_reg, &temp, 1);
        if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

        temp = VDAD_PGCR_INIT;      /* (Port Gen Control reg) */
        kav$bus_write(&status, KAV$K_BYTE, p_port_gen_control_reg, &temp, 1);
        if ERROR_DETECTED { vdad$error_text(status); exit(-1); }
    }

```



## Example Programs—VDAD Device Driver

```
temp = VDAD_PSRR_INIT ;          /* (Port Service Request reg) */
kav$bus_write(&status, KAV$K_BYTE, p_port_service_req_reg, &temp, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

/* (Port Control reg) */
control_reg.bits.fixed = 1 ; /* Set submode for A/D register. */
control_reg.bits.EOC_ien = 1 ; /* Enable interrupts. */
kav$bus_write(&status, KAV$K_BYTE, p_portA_control_reg, &control_reg, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

temp = VDAD_PADDR_INIT ;          /* (Port Data Direction reg) */
/* (Set up for A/D mode) */
kav$bus_write(&status, KAV$K_BYTE, p_portA_data_dir_reg, &temp, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

kav$bus_write(&status, KAV$K_BYTE, p_portA_data_reg, &setup_reg, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }
/*
 * Finally, we must set up the KAV to allow interrupts.
 */
vdad$kav_data = 0x00000002; /* IRQ 1 allowed. */
vdad$kav_setup = KAV$K_ALLOW_VME_IRQ;
KAV$VME_SETUP( &status, vdad$kav_setup, &vdad$kav_data );
if ERROR_DETECTED vdad$error_text(status);

} else

/* ===== */
{ /* <<< Setup for POLLING >>> */
/* ===== */
temp = VDAD_PADDR_INIT ; /* Port-A Data Dir Reg init value. */
/* (Set up for A/D mode) */
kav$bus_write(&status, KAV$K_BYTE, p_portA_data_dir_reg, &temp, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

control_reg.bits.fixed = 1 ; /* Set submode for A/D register. */
kav$bus_write(&status, KAV$K_BYTE, p_portA_control_reg, &control_reg, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

kav$bus_write(&status, KAV$K_BYTE, p_portA_data_reg, &setup_reg, 1);
if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

} ;

return( status ) ;
} /* end -VDAD$SETUP- */
```

## Example Programs—VDAD Device Driver

```

/*-----*/
/*
 * *****
 * ***   R E A D   C H A N N E L   ***
 * *****
 *
 * This routine is called to read a block of data from a selected channel.
 *
 * Returns:  int      status   (see .H file for definitions)
 *
 * Inputs:   int      channel,      (A/D channel: 0..15)
 *           timeout,
 *           *buffer,      (buffer to receive the data)
 *           num_samples,  (number of samples requested)
 *           *num_samples_read (actual number ....received)
 *
 * Notes:
 * (1) On return to the caller, "num_samples" should always equal
 *     "num_samples_read". In the case of timeout on a single conversion,
 *     the latter will reflect the number of successful samples up to the
 *     point of the timeout.
 * (2) The "ADC Read/Convert" register is DANGEROUS !!
 *     ie. do NOT access it unless the EOC bit has signalled a successful
 *     conversion.
 */

int  vdad$read( channel, timeout, buffer, num_samples, num_samples_read )
int      channel ;
long     timeout ;
unsigned short *buffer ;
int      num_samples, *num_samples_read ;
{
int      status, temp ;
register int  i;
unsigned short *p_sample ; /* This is purely for quick reference */
DEVICE      dev ; /* ...ditto... */

    *num_samples_read = 0 ; /* Zero this out (user's data). */
    status            = SUCCESS ;
    p_sample          = &(amp; vdad$region->value[channel] ) ;
    dev               = vdad$device[channel] ;

    if (vdad$monitoring) { vdad$reads++ ; } /* Count calls to this rtne. */

```

## Example Programs—VDAD Device Driver

```
switch (vdad$interrupts_enabled) /* Poll or use interrupts ? */
{
    /* ----- */
    case TRUE :
        /* ***** */
        /* *** INTERRUPTS *** */
        /* ***** */
        for (i = 0; i < num_samples; i++, buffer++)
        {
            /*
             * Write in the requested channel, starting the conversion.
             */
            *p_adc_read_conv_addr = channel ;

            /*
             * Now wait for our ISR to do the read under interrupt control.
             */
            ker$wait_any( &status, NULL, NULL, dev ) ;
            if ERROR_DETECTED { vdad$error_text(status); exit(-1); }

            /*
             * Copy data back to the user buffer.
             */
            *buffer = *p_sample ;
        } ;
        break ; /* end -for- */
    case FALSE :
        /* ***** */
        /* *** POLLING *** */
        /* ***** */
        for (i = 0; i < num_samples; i++, buffer++)
        {
            /*
             * Write in the requested channel, starting the conversion.
             */
            *p_adc_read_conv_addr = channel ;

            while (((*p_portA_data_reg) & 0x80) != 0x80);
            {
                /* ADC conversion complete. Copy the sampled data, */
                /* ignoring bits 12-15 (ie. the channel). */
                /* [we should really check the channel here] */
                *buffer = (*p_adc_read_conv_addr & 0x0FFF) ;
            }
        } ;
        break ; /* end -for- */
    default: break ;
} ; /* end -switch- */

*num_samples_read = i ; /* Return successful reads to user. */
```

```

return( status );
}      /*      ***  end  :-  vdad_read  ***      */

/*-----*/
/*
 *      *****
 *      *****  E X C E P T I O N  H A N D L E R  *****
 *      *****
 *
 * This routine is called AUTOMATICALLY when ELN raises an exception.
 *
 * Returns:   int    status          (see .H file for definitions)
 *
 * Notes:
 */
BOOLEAN vdad$cond_handler(signal_ptr, mechanism_ptr)
struct chf$signal_array *signal_ptr;
struct chf$mech_array   *mechanism_ptr;
{
    void vdad$error_text(int status);
    static int status;

    if (vdad$monitoring)
    {
        printf("\nVDAD-Driver: Condition Handled (number %d) - ",
            vdad$conditions++);
    } else
    {
        printf("\nVDAD-Driver: Condition Handled - " );
    } ;
    status = signal_ptr->chf$l_sig_name;
    vdad$error_text(status);
    exit(1);
}

void vdad$error_text( int error )
{
    int text_flags;
    char text_buffer[255];
    VARYING_STRING(255) result_string;

    text_flags = STATUS$ALL;
    eln$get_status_text(error, text_flags, &result_string);
    VARYING_TO_CSTRING(result_string, text_buffer);
    printf("\n%s\n", text_buffer) ;
}

```

## Example Programs—VDAD Device Driver

```
    return;
}
/*
 *  V D A D _ i s r
 *
 *  This is the device interrupt status routine. It is called by the kernel
 *  when a device interrupt occurs.
 *  It reads the sampled data, strips off the 4-bit channel id (bits 12..15),
 *  and stores the result into the Comms Region.
 *  It then signals the waiting process. The Channel number indicates which
 *  Device Object is signalled.
 *
 *  Returns:  The sampled data is copied into the Device Comms
 *            Region.
 *
 *  Inputs:
 *  register_ptr = pointer to device registers
 *  region_ptr = pointer to driver communications region
 *
 *  Notes:
 *
 */
void vdad_isr(register_ptr, vdad$region )
char      *register_ptr;
vdad_region_type *vdad$region ;
{
    unsigned short  reset_irq = VDAD_EOC_IRQ ;
    static short    channel ;
    int             status ;

    /*
     *  Read the data (a word) from VDAD.
     */
    kav$bus_read( &status,
                  KAV$K_WORD,
                  vdad$region->p_adc_read_conv_addr,
                  &(vdad$region->reg.word),
                  1 ) ;

    /*
     *  Extract the channel (top 4 bits) and sampled value (bottom 12 bits).
     */
    channel              = vdad$region->reg.ADC_register.channel ;
    vdad$region->value[channel] = vdad$region->reg.ADC_register.value ;

    /*
     *  Reset the device for further interrupts.
     */
    kav$bus_bitset( &status,
                   KAV$K_BYTE,
                   vdad$region->p_port_status_reg,
                   reset_irq ) ;
}
```

```

/*
 *   Signal the device (indexed by 'channel').
 */
ker$signal_device(NULL, channel );
} /* --- end of VDAD ISR --- */

```

## D.2 Definitions File

```

/*
 *   C Definitions file for:-
 *   =====
 *
 *   ****
 *   *** PEP VDAD Digital-Analog-Digital Converter ***
 *   ****
 *
 */
#define      VDAD_BASE      0x0e00      /*Check the rotary switch on the */
                                           /*KAV - they should agree.      */
#define      VDAD_PHYS_ADDR 0xfe0000    /*24-bit access.                  */
/* #define    VDAD_PHYS_ADDR 0x0000    */ /*16-bit access.                  */
#define      VDAD_EOC_IRQ_  0x01        /*Interrupt Reset value.          */
#define      VDAD_IRQL      0x01        /*Interrupt Request Level.         */
#define      VDAD_IRQV      0x04        /*Interrupt Vector number.         */
#define      VDAD_PADDR_INIT 0x7F       /*Port-A Data Dir Reg init value. */
#define      VDAD_PGCR_INIT  0x11       /*Port Gen Cntrl Reg init value.  */
#define      VDAD_PSRR_INIT  0x18       /*Port Service Req Reg init value.*/
#define      VDAD_PACR_INIT_INT 0x82    /*Port-A Cntrl Reg init val (INTER).*/
#define      VDAD_PACR_INIT_POLL 0x80   /*Port-A Cntrl Reg init val (POLL).*/

/*
 *   This next assignment is IMPORTANT !!! It should NEVER be zero, and it
 *   should NOT be changed. The "Create_device" call needs it.
 */
#define      RELATIVE_VECTOR 0x01        /* (NEVER ZERO !!!)              */

/*
 *   Maximum 16 single-ended, 8 differential channels per board.
 */
#define MAX_CHANNELS      16

```

## Example Programs—VDAD Device Driver

```

/*
 *   Status return values
 */
#define SUCCESS      1
#define ERROR        -1
#define DEVOFFLINE   0x84
#define ILLIOFUNC     0xF4
#define IVADDR        0x134
#define IVBUFLLEN     0x34C
#define NOSUCHDEV     0x908
#define TIMEOUT       0x22C
/*
 *   VDAD Setup options
 *   =====
 *
 *   Note: None of these can have the value zero, since the user may NULL
 *   one or more of the parameters in a VDAD call.
 *
 */
/*
 *   Interrupts or Polling.
 */
/*
 *   =====
 */
#define VDAD$SETUP_INT_DISABLE 1 /* Disable interrupts. */
#define VDAD$SETUP_INT_ENABLE 2 /* Enable interrupts. */

#define VDAD$SETUP_GAIN_1 1 /* Three options for GAIN value. */
#define VDAD$SETUP_GAIN_10 2
#define VDAD$SETUP_GAIN_100 3

/*
 *   Input channel configuration :-
 */
/*
 *   =====
 */
#define VDAD$SETUP_INP_CONFIG_1 1 /* 0..16 n/a */
#define VDAD$SETUP_INP_CONFIG_2 2 /* 0..5, 8..13 6,7 */
#define VDAD$SETUP_INP_CONFIG_3 3 /* 0..3, 8..11 4..7 */
#define VDAD$SETUP_INP_CONFIG_4 4 /* n/a 0..7 */

/*
 *   Sampling mode
 */
/*
 *   =====
 */
#define VDAD$SETUP_CHANNEL_MODE_SINGLE 1 /* Single-channel. */
#define VDAD$SETUP_CHANNEL_MODE_MULTIPLE 2 /* Multi-..... */

/*
 *   Trigger types.
 */
/*
 *   =====
 */
#define VDAD$SETUP_TRIGGER_SOFTWARE 1 /* ...by software. */
#define VDAD$SETUP_TRIGGER_EXTERN 2 /* ...by external stim. */
#define VDAD$SETUP_TRIGGER_TIMER 3 /* ...by Timer. */
#define VDAD$SETUP_TRIGGER_EXTERN_AND_TIMER 4 /* ...by extern & Timer. */

/* Use Driver's or User's Condition Handler.*/
#define VDAD$SETUP_COND_HANDLING_DRIVER 1
#define VDAD$SETUP_COND_HANDLING_USER 2

/* Turn 'monitoring' (statistics) on/off. */
#define VDAD$SETUP_MONITORING_ON 1
#define VDAD$SETUP_MONITORING_OFF 2

```

## Example Programs—VDAD Device Driver

```

/*
 *      VDAD Register offsets
 *      =====
 */
#define OFFSET_port_gen_control_reg      0x01
#define OFFSET_port_service_req_reg     0x03
#define OFFSET_portA_data_dir_reg       0x05
#define OFFSET_port_int_vec_reg         0x0b
#define OFFSET_portA_control_reg        0x0d
#define OFFSET_portA_data_reg           0x11
#define OFFSET_port_status_reg          0x1b
#define OFFSET_adc_read_conv_addr       0x40
#define OFFSET_int_level_reg            0xff

/*      There are more registers on this module, but we do not require
 *      access to them in this version.
 */

/*
 *      *****
 *      ***  VDAD Device register definitions  ***
 *      *****
 */
typedef struct          /* <<< Port-A Data Register >>> */
{
    /* ===== */
    unsigned gain       : 2 ;
    unsigned input_config : 2 ;
    unsigned channel_mode : 1 ;
    unsigned trigger_type : 2 ;
    unsigned unused      : 1 ;
} portA_data_reg ;

typedef struct          /* <<< Port General Control Register >>> */
{
    /* ===== */
    unsigned EOC_IRQ     : 1 ;
    unsigned edge        : 3 ;
    unsigned H1H2_enable : 1 ;
    unsigned H3H4_enable : 1 ;
    unsigned fixed       : 2 ; /* (always zero) */
} port_gen_control_reg ;

typedef struct          /* <<< Port-A Control Register >>> */
{
    /* ===== */
    unsigned unused1     : 1 ;
    unsigned EOC_iен     : 1 ;
    unsigned H2_iен      : 1 ;
    unsigned H2_transition : 3 ;
    unsigned unused2     : 1 ;
    unsigned fixed       : 2 ;
} portA_control_reg ;

```



## Example Programs—VDAD Device Driver

```
typedef struct                                /* <<< Port Status Register >>> */
{
    /* ===== */
    unsigned H1234_levels : 4 ; /* Current levels at H1-H4 pins */
    unsigned H432S_status : 3 ; /* H4S, H3S, H2S status pins */
    unsigned EOC_status   : 1 ; /* EOC status bit. */
} port_status_reg ;

/*
 *
 * Driver-specific structures.
 *
 */

typedef struct                                /* <<< A/D Read/Convert Address Register >>> */
{
    /* ===== */
    unsigned value : 12 ; /* 12-bit converted value (HI NIBBLE). */
    unsigned channel : 4 ; /* Channel number. */
} AD_RC_reg ;

/*
 * Note that when you WRITE to this register, you write the channel number
 * into bits 0-3. However, when you READ this, VDAD puts the channel into
 * bits 12-15.
 */

/*
 * This is the data structure which allows the Interrupt Service Routine
 * to pass data to/from the rest of the Driver.
 * This is the only method, since an ELN ISR executes in a different context
 * from the 'owning' process.
 */

typedef struct                                /* <<< Communications Region for Interrupt Service >>> */
{
    /* ===== */
    union {
        AD_RC_reg ADC_register ; /* 'bare' contents of register */
        unsigned short word ; /* (WORD access to register) */
    } reg ;

    unsigned short value[MAX_CHANNELS] ; /* ADC value only (per channel) */
    unsigned char *p_portA_data_dir_reg, /* VDAD register pointers. */
        *p_portA_control_reg,
        *p_portA_data_reg,
        *p_port_gen_control_reg,
        *p_port_service_req_reg,
        *p_port_int_vec_reg,
        *p_port_status_reg,
        *p_int_level_reg ;
    unsigned short *p_adc_read_conv_addr ;
} vdad_region_type ;
```

## D.3 Test Program

```

/*
 * FACILITY:
 * VAXELN Run Time System
 *
 * ABSTRACT:
 *     This program demonstrates how to access the VDAD Driver.
 *
 * VERSION:
 *     V1.00  13-Mar-1991  Field Test Release.
 *
 * NOTES:
 *     This is a simple example of how to access the VDAD Driver.
 */

/*
 * Include Files
 */

#include stdio
#include $vaxelnc
#include $dda_utility
#include $mutex
#include chfdef
#include descrip
#include <eln$:kavdef.h>
#include $get_message_text
#include types
#include "vdaddriver.h"                /* VDAD Driver definitions */

#define SIZE_OF_DATA_BUFFER 1024

unsigned short  sample_buffer[SIZE_OF_DATA_BUFFER] ;
unsigned short  *p_buf ;

main()
{
    void error_text();
    void display_buffer() ;
    BOOLEAN display_samples ;
    char beep[2] = " " ;
    int  status, temp, chan, use_interrupts ;
    long int num_buffers, block_count, num_samples_read ;

    p_buf = &sample_buffer[0] ;
    block_count = 0 ;
    printf("VDAD test process started\n");
    printf("Setting up the VDAD Device...\n");

```

## Example Programs—VDAD Device Driver

```
/*
 *      Initialise the VDAD Driver (this is a once-only call).
 */
status = vdad$init( ) ;
if (!(status & 1)) { error_text(status); exit(-1); }

/*
 *      Ask for number of iterations.
 */
printf("\nHow many iterations (1 Kbyte per iteration) ? : ");
scanf("%d", &num_buffers );

/*
 *      Ask for ADC Channel number.
 */
get_chan:
printf("\nWhich ADC channel [0-15] ? : ");
scanf("%d", &chan );
if ( (chan > 15) || (chan < 0) )
{
    printf("\nInvalid channel number. Re-enter correctly.");
    goto get_chan ;
} ;

/*
 *      Ask for "Interrupts or Polling".
 */
printf("\nDo you wish to use Interrupts [1=Yes, 0=No] ? : ");
scanf("%d", &temp );
if ( temp == 1 )
{
    use_interrupts = VDAD$SETUP_INT_ENABLE ;
    printf("\nInterrupts in use! \n");
} else
{
    use_interrupts = VDAD$SETUP_INT_DISABLE ;
    printf("\nPolling (ie. NO interrupts)\n");
} ;

/*
 *      Does the user want a "sample-dump" ?
 */
printf("\nDo you wish to see the sampled buffer [1=Yes, 0=No] ? : ");
scanf("%d", &temp );
if ( temp == 1 ) { display_samples = TRUE;} else { display_samples = FALSE;};
```

## Example Programs—VDAD Device Driver

```
/*
 *   Set up the VDAD Driver (this can be called at any time).
 */
status = vdad$setup(  VDAD$SETUP_GAIN 1,
                      VDAD$SETUP_INP_CONFIG 1,
                      VDAD$SETUP_CHANNEL_MODE_SINGLE,
                      VDAD$SETUP_TRIGGER_SOFTWARE ,
                      use_interrupts,
                      VDAD$SETUP_COND_HANDLING_DRIVER,
                      VDAD$SETUP_MONITORING_OFF ) ;
if (!(status & 1)) { error_text(status); exit(-1); }

printf("\nVDAD Set up OK. Commencing sampling with block size of %d.",
        SIZE_OF_DATA_BUFFER) ;
printf("\nNote that each '.' represents ONE BLOCK successfully sampled.");
printf("\nA total of %d blocks will be sampled.", num_buffers);
printf("\n[have you switched the console terminal into AUTO-WRAP ?]\n\n");
printf("\nHit a character, then <RETURN> to start sampling : ");
scanf("%s", &temp);

printf("\nSampling commencing...\n");
printf(beep);

block_count = 0 ;
while (block_count++ < num_buffers )
{
    /*
     *   Request the VDAD Driver to fill in the buffer with the requested
     *   number of samples.
     */
    status = vdad$read( chan,
                       NULL,
                       &(sample_buffer[0]),
                       SIZE_OF_DATA_BUFFER,
                       &num_samples_read ) ;

    if (!(status & 1)) { error_text(status); exit(-1); }
    printf(".") ;

    if (display_samples==TRUE) { display_buffer() ; } ;
} ; /* end -while- */

printf(beep);
printf("\n\nVDAD Test process finito !\n");
}

/*-----*/

void error_text(status)
int status;
{
    int text_flags;
    char text_buffer[255];
    VARYING_STRING(255) result_string;
```

## Example Programs—VDAD Device Driver

```
text_flags = STATUS$ALL;
eln$get_status_text(status, text_flags, &result_string);
VARYING TO CSTRING(result_string, text_buffer);
printf("%s\n", text_buffer);

return;
}

void display_buffer()
{
register int i, j ;

printf("\n      ****  Dump of sample buffer (16 samples per line)  ****\n") ;
for (i=0; i < SIZE_OF_DATA_BUFFER; )
{
    for (j=0; (j < 16) &&(j+i < SIZE_OF_DATA_BUFFER); j++ )
    {
        printf("%3x,", sample_buffer[i+j]) ;
/*
        sample_buffer[i+j] = SIZE_OF_DATA_BUFFER - (i+j) ;
*/
    }
    i = i + 16 ;
    printf("\n") ;
}

return;
}
/* ----- */
```

## D.4 Build File

```
$!
$!      Command Procedure to compile, link and EBUILD the VDAD test program.
$!
$ cc /NOOPTIMIZE test_vdad.c +eln$:vaxelnc /library
$ cc /NOOPTIMIZE vdaddriver.c +eln$:vaxelnc /library
$!
$ If "'F$Search("VDAD.OLB")'" .EQS. "" Then LIBRARY/Create VDAD.OLB
$ LIBRARY/REPLACE VDAD.OLB test_vdad.OBJ
$ LIBRARY/REPLACE VDAD.OLB VDADDRIIVER.OBJ
$!
$ DEFINE C$LIBRARY ELN$:VAXELNC.TLB
$ DEFINE LNK$LIBRARY ELN$:CRTLSHARE
$ DEFINE LNK$LIBRARY_1 ELN$:RTLSHARE
$ DEFINE LNK$LIBRARY_2 ELN$:RTL
$ DEFINE LNK$LIBRARY_3 ELN$:KAV$RTL_OBJLIB
$ LINK /EXE=test_vdad vdad.olb/library/include=(test_vdad) -
/nosyslib/nosysshr
$!
$ EBUILD /noedi test_vdad
$!
$exit
```

## D.5 Data File

```
characteristic /nofile /net_device=EZA /node_address=63.740 -
/noserver /objects=512 /debug=none /io_region=1024 -
/target=24 /image_list=(IPCSHR,AUXCSHR,ICSSHR)
program TEST VDAD.EXE /kernel_stack=64 /user_stack=10 /job_priority=10 -
/argument=("", "", "", """"VDAD""")
device EZA /vector=%X130 /net_def
device VDAD /vector=%X810 /noautoload
```



---

# Glossary

The glossary defines some of the important terms used in this guide.

## **application program**

A program that performs an end-user task.

## **ASB**

Asynchronous system block. The ASB contains information about the AST routine for a particular event.

## **AST**

Asynchronous system trap. A procedure that the operating system calls when a particular event occurs.

## **autovectored interrupt**

An interrupt for which the interrupt handler provides the interrupt vector address.

## **backup process**

The process of making copies of the data stored on the disk, so that you can recover that data after an accidental loss. You make backup copies on RX33 diskettes, TK50 tape cartridges, or over a network.

## **backup copy**

A copy of the data stored on the disk.

## **BCD**

Binary coded decimal. Pertaining to a number representation system in which each decimal digit is represented by a unique arrangement of binary digits.

## **big-endian device**

A device based on the 68000® family of processors.



**BR line**

Bus request line. A signal line on which a device issues a bus request signal.

**CPU**

Central processing unit. The main unit of a computer that contains the circuits that control the interpretation and execution of instructions. The CPU holds the main storage, arithmetic unit, and special registers.

**CSR bit**

Control and status register bit. The CSR bits consist of input bits and output bits. The CSR input bits report on the status of the KAV30 hardware, while the CSR output bits control the KAV30 hardware.

**DAL bus**

Data and address lines bus. A 32-bit multiplexed bus. The rtVAX 300 is the source of the DAL bus.

**DMA**

Direct memory access. A method of accessing a device's memory without interacting with the device's CPU.

**FIFO**

First-in/first-out. The order in which processing is performed. For example, a FIFO queue processes data on a first-come, first-served basis.

**FIFO buffer**

A hardware area in which devices can store and retrieve data.

**host system**

The primary or controlling computer in a multiple computer network.

**IACK**

Interrupt-acknowledge signal. A signal, issued by an interrupt handler device, which indicates that the device will handle an interrupt request.

**interrupt**

A break in the usual flow of a program to process an external request.

**interrupt handler**

A device that executes interrupt service routines for interrupt requesters. The device receives interrupt requests from the bus.

**IPL**

Interrupt priority level. The interrupt level at which an interrupt is generated. There are 31 possible interrupt priority levels: IPL 1 is the lowest, 31 is the highest. The levels arbitrate contention for processor service.

**interrupt requester**

A device that requests the execution of an interrupt service routine. The device sends an interrupt request on the bus, which an interrupt handler responds to.

**interrupt vector address**

An indirect address that points to the starting address of an interrupt service routine.

**IRQ**

Interrupt-request signal. A signal, issued by a device, to execute an interrupt service routine.

**ISR**

Interrupt service routine. The software that processes interrupt requests.

**LIFO**

Last-in/first-out. The order in which processing is performed. For example, a LIFO queue processes data on a last-come, first-served basis.

**little-endian device**

A device based on the Intel™ family of processors.

**RAM**

Random-access memory. A read/write memory device.

**ROM**

Read-only memory. A memory in which information is permanently stored at the time of production and is not alterable by computer instructions.

**ROR**

Release-on-request. When a VMEbus requester operates in ROR mode, it gives up the data transfer bus when another VMEbus module requests the bus.

**RWD**

Release-when-done. When a VMEbus requester operates in RWD mode, it gives up the data transfer bus only when it finishes using the bus.

**SCB**

System control block. The data structure in system space that contains all the interrupt and exception vectors known to the system.

**SCSI**

Small computer systems interface. An interface designed for connecting disks and other peripheral devices to computer systems. SCSI is defined by an American National Standards Institute (ANSI) standard.

**SGM**

Scatter-gather map. A means of allowing either of the following types of data transfer:

- From pages in memory that are not contiguous to contiguous blocks on a bus
- From contiguous blocks on a bus to pages in memory that are not contiguous

**target system**

A system in which a task executes.

**vectored interrupts**

An interrupt for which the interrupt requester provides the interrupt vector address.

**VSB**

VME subsystem bus.

---

# Index

## A

---

A16 addressing, 2-1, 2-2, 3-13  
A24 addressing, 2-1, 2-2, 3-13, 3-16,  
4-150, 5-17  
A32 addressing, 2-1, 2-2, 3-12, 3-15,  
4-150, 5-17, 5-19  
ACFAIL signal, 2-5, 2-14, A-1  
ALTERNATE, 2-3, 4-89  
Arbiter, 1-2  
    fair mode, A-2  
    hidden mode, A-2  
    priority mode, A-2  
    VMEbus, 2-3 to 2-4, 5-18  
    VSB, 2-4, 5-17  
Arbitration  
    prioritized, 2-3  
    round-robin, 2-3  
ASB, 1-5, 3-1, 3-2, 4-34, 4-96, 4-122  
ASB\$K\_ASBFREE, 3-3  
ASB\$K\_ASBPEND, 3-3  
AST, 1-5, 1-6, 2-5, 3-1 to 3-4, 4-34 to  
4-36, 4-58, 4-131  
    coding in VAX Ada, 5-4 to 5-7  
    coding in VAXELN Pascal, 5-9 to 5-12  
    data structures, 3-3  
    defining, 4-37 to 4-40  
    parameters, 3-2  
    queuing, 4-96 to 4-99  
    routines, 3-2  
    setting, 4-122 to 4-126  
    writing, 5-3  
AST queues  
    clearing, 4-34 to 4-36

Asynchronous context block

*See* ASB

Asynchronous System Trap

*See* AST

Autovectorred interrupts

    VMEbus, 5-16, A-2

    VSB, 5-16

Autovectorred Interrupts, 3-2

Auxiliary port, 1-2, A-1

## B

---

Battery, 1-5, 3-7

    checking, 4-30 to 4-33

Battery backed-up RAM, 1-2, 1-5, 1-6,  
3-10, 4-30, 4-51, 4-116 to 4-121, A-1

Battery backed-up random-access memory

*See* Battery backed-up RAM

Big-endian, 3-18, 4-58, 4-84

BR lines, 2-1

Break command, 2-14

Break key, A-1

Bus request, 1-2

    ROR mode, A-2

    VMEbus, 5-18

    VSB, 5-17

Bus Request lines

*See* BR lines

## C

---

Calendar/clock, 1-2, 1-5, 1-6, 3-6 to 3-8,  
4-30, 4-100 to 4-115, 5-17, A-1, A-2  
Clock period, 3-5  
Compiling  
    KAV30 applications, 5-12  
Condition handler, 5-2  
Configuration  
    KAV30, A-1 to A-2  
Console port, 1-2  
Consumer, 3-9  
Control and status register bits  
    *See* CSR bits  
Control and status register page  
    *See* CSR page  
Counter/timers, 1-2, 1-5, A-2  
CSR bits, 2-14  
CSR page, 5-14  
CVAX microprocessor, 1-1

## D

---

D08 transfers, 2-2  
D16 transfers, 2-2  
D32 transfers, 2-2  
DAL bus, 2-3, 2-4, 3-6  
    master, 2-5  
    timeout period, 2-5  
    timeouts, 2-5  
Data and address lines bus  
    *See* DAL bus  
Data and Address Lines bus  
    *See* DAL bus  
Debugging KAV30 applications, 5-20 to  
5-21  
Device drivers, 3-2  
Direct memory access  
    *See* DMA, 2-3  
DMA, 2-3

## E

---

ELN\$GET\_STATUS\_TEXT, 3-26  
ERR signal, 2-9  
Error logging, 3-24 to 3-29  
Ethernet, 2-14  
Example programs  
    FIFO consumer, B-5 to B-9  
    FIFO producer, B-1 to B-4  
    interprocessor communication, B-1 to  
    B-9  
    MVME335 device driver, C-1 to C-21  
    VDAD device driver, D-1 to D-25  
Exception handling, 5-2

## F

---

Fair mode, 2-2, A-2  
FIFO buffers, 1-1, 1-5, 1-6, 2-2, 3-9, 3-23,  
4-16, 4-19, 4-20, 4-23, 4-27, 4-41,  
4-46, 4-58, 4-74, 4-78, A-2  
    consumer, 3-9  
    errors, 2-14, 2-15  
    producer, 3-9  
FIFO modes, 4-46  
First-in/first-out buffers  
    *See* FIFO buffers

## H

---

HALT signal, A-1  
Hidden mode, 2-2, A-2  
Host system, 1-1, 1-3

## I

---

I/O, 2-3, 4-89  
    routines, 5-3  
IACK, 3-2  
    cycle, 2-6, 2-9  
Input/Output routines  
    *See* I/O routines  
Interrupt handler  
    VMEbus, 2-6 to 2-10

Interrupt handler (cont'd)

VSB, 2-12

Interrupt priority level

*See* IPL

Interrupt requester

VMEbus, 2-10 to 2-12

Interrupt service routines

*See* ISR

Interrupt-acknowledge cycle

*See* IACK cycle

Interrupts, 1-2, 1-6, 4-67 to 4-73

autovectored, 2-6, 2-8 to 2-10, 3-2

KAV30 source codes, 2-7

pins, 2-13

priority scheme, 2-12 to 2-15

vectored, 2-6 to 2-8

VMEbus, A-2

VMEbus autovectored, 5-16

VMEbus vectored, 5-15

VSB autovectored, 5-16

IPL, 2-12, 3-2

IRQ, 3-2, 4-67

ISR, 2-5, 2-8, 2-9, 3-2

accessing an, 5-2

## K

KAV\$BUS\_BITCLR, 1-5, 4-2 to 4-9, 4-15,  
4-21, 4-29, 4-84, 4-91

KAV\$BUS\_BITSET, 1-5, 4-7, 4-10 to 4-15,  
4-21, 4-29, 4-84, 4-91

KAV\$BUS\_READ, 1-5, 4-7, 4-15, 4-16 to  
4-22, 4-23 to 4-29, 4-84, 4-91, 5-3

KAV\$BUS\_WRITE, 1-5, 3-23, 4-7, 4-15,  
4-21, 4-23, 4-28, 4-84, 4-91, 5-3

KAV\$CHECK\_BATTERY, 1-5, 4-30 to 4-33

KAV\$CLR\_AST, 1-5, 3-2, 3-3, 4-34 to  
4-36, 4-37, 4-40, 4-99, 4-126

KAV\$DEF\_AST, 1-5, 3-2, 4-34, 4-36, 4-37  
to 4-40, 4-96, 4-99, 4-122, 4-126

KAV\$FIFO\_READ, 1-5, 3-9, 3-23, 4-41 to  
4-45, 4-50, 4-77, 4-83

KAV\$FIFO\_WRITE, 1-5, 3-9, 4-44, 4-46 to  
4-50, 4-77, 4-83

KAV\$GATHER\_KAV\_ERRORLOG, 1-5,  
4-51 to 4-57

KAV\$INT\_VME, 1-6, 4-67 to 4-73, 4-155

KAV\$IN\_MAP, 1-5, 3-14, 4-58 to 4-66,  
4-149, 5-14

KAV\$K\_ALLOW\_VME\_IRQ, 4-153, 4-154

KAV\$K\_ALL\_ERR, 4-53

KAV\$K\_ALR\_DOM, 4-109, 4-110

KAV\$K\_ALR\_HOUR, 4-109

KAV\$K\_ALR\_MINUTE, 4-109

KAV\$K\_ALR\_MONTH, 4-110

KAV\$K\_ALR\_SECOND, 4-109

KAV\$K\_AUTO\_VME\_IRQ, 4-153, 4-154

KAV\$K\_BBR\_READ, 4-119

KAV\$K\_BBR\_WRITE, 4-119

KAV\$K\_BYTE, 4-5, 4-13, 4-19, 4-26

KAV\$K\_CLEAR\_ERR, 4-53

KAV\$K\_CTMR0, 4-136

KAV\$K\_CTMR1, 4-136

KAV\$K\_CTMR2, 4-136

KAV\$K\_CTMR3, 4-136

KAV\$K\_CTMR4, 4-136

KAV\$K\_DISABLE\_VSB\_IRQ, 4-153

KAV\$K\_ENABLE\_VSB\_IRQ, 4-153

KAV\$K\_FIFO\_0, 4-49, 4-76, 4-82

KAV\$K\_FIFO\_1, 4-49, 4-76, 4-82

KAV\$K\_FIFO\_2, 4-49, 4-76, 4-82

KAV\$K\_FIFO\_3, 4-49, 4-76, 4-82

KAV\$K\_INIT\_RD\_POINTER, 4-53

KAV\$K\_LCL\_TO, 4-136

KAV\$K\_LONGWORD, 4-5, 4-13, 4-16,  
4-19, 4-20, 4-23, 4-26, 4-27

KAV\$K\_MASTER\_ERR, 4-53

KAV\$K\_PER\_100MS, 4-109

KAV\$K\_PER\_10MS, 4-109

KAV\$K\_PER\_10SEC, 4-109

KAV\$K\_PER\_1MS, 4-109

KAV\$K\_PER\_1SEC, 4-109

KAV\$K\_PER\_60SEC, 4-109

KAV\$K\_RD, 4-68, 4-71, 4-72

KAV\$K\_RD\_A24\_ROTARY, 4-153, 4-155

KAV\$K\_RD\_VSB\_SLOT, 4-153, 4-155

KAV\$K\_RTC\_1000MS, 4-109

KAV\$K\_RTC\_100MS, 4-109

KAV\$K\_RTC\_100NS, 4-109  
 KAV\$K\_RTC\_10MS, 4-109  
 KAV\$K\_RTC\_1MS, 4-109  
 KAV\$K\_RTC\_400NS, 4-109  
 KAV\$K\_RTC\_93US, 4-109  
 KAV\$K\_SET\_A32\_BASE, 4-153, 4-155  
 KAV\$K\_SET\_RTC\_TIME, 4-129  
 KAV\$K\_SET\_VAX\_TIME, 4-129  
 KAV\$K\_SLAVE\_ERR, 4-53  
 KAV\$K\_SUPER\_16, 4-89  
 KAV\$K\_SUPER\_24, 4-89  
 KAV\$K\_SUPER\_32, 4-89  
 KAV\$K\_USER\_16, 4-89  
 KAV\$K\_USER\_24, 4-89  
 KAV\$K\_USER\_32, 4-89  
 KAV\$K\_VME\_INT\_CLR, 4-71, 4-72  
 KAV\$K\_VME\_REQ\_INT, 4-71  
 KAV\$K\_VME\_SYSFAIL, 2-5  
 KAV\$K\_WDOG, 4-136  
 KAV\$K\_WORD, 4-5, 4-13, 4-19, 4-26  
 KAV\$LIFO\_WIRTE, 1-6  
 KAV\$LIFO\_WRITE, 3-9, 4-44, 4-50, 4-74  
 to 4-77, 4-83  
 KAV\$M\_ALARM, 4-106  
 KAV\$M\_CSR, 4-62, 4-63, 4-148  
 KAV\$M\_FIFO\_ACCESS, 3-23, 4-16, 4-19,  
 4-20, 4-23, 4-26, 4-27  
 KAV\$M\_FIFO\_EMPTY, 4-81  
 KAV\$M\_FIFO\_FULL, 4-81  
 KAV\$M\_FIFO\_NOT\_EMPTY, 4-81  
 KAV\$M\_IN, 4-148  
 KAV\$M\_LOAD\_TMR\_CNT, 4-105, 4-135  
 KAV\$M\_LOCMON\_IPL15, 4-63  
 KAV\$M\_LOCMON\_IPL16, 4-63  
 KAV\$M\_LOCMON\_IPL17, 4-63  
 KAV\$M\_MEMORY, 4-63, 4-148  
 KAV\$M\_MODE\_0\_SWAP, 4-63, 4-90  
 KAV\$M\_MODE\_2\_SWAP, 4-63, 4-90  
 KAV\$M\_MODE\_3\_SWAP, 4-63, 4-90  
 KAV\$M\_NO\_RETRY, 4-2, 4-10, 4-90  
 KAV\$M\_OUT, 4-148  
 KAV\$M\_PERIODIC, 4-106  
 KAV\$M\_READ\_ALARM, 4-106  
 KAV\$M\_READ\_CALENDAR, 4-107  
 KAV\$M\_READ\_RTCRAM, 4-107  
 KAV\$M\_READ\_TMR\_CNT, 4-106, 4-135  
 KAV\$M\_REPEAT, 4-122, 4-125  
 KAV\$M\_REPEAT\_TMR, 4-136  
 KAV\$M\_RESET\_FIFO, 4-81  
 KAV\$M\_RESET\_TMR, 4-106, 4-136  
 KAV\$M\_RTC\_12\_HOUR, 4-100, 4-107,  
 4-129  
 KAV\$M\_RTC\_24\_HOUR, 4-100, 4-107,  
 4-129  
 KAV\$M\_RTC\_HOLD\_TMR, 4-108  
 KAV\$M\_RTC\_READ\_TIMESAVE, 4-107  
 KAV\$M\_RTC\_RESTART\_TMR, 4-108  
 KAV\$M\_RTC\_TMR\_0, 4-105  
 KAV\$M\_RTC\_TMR\_1, 4-105  
 KAV\$M\_START\_TMR, 4-105, 4-135  
 KAV\$M\_STOP\_TMR, 4-106  
 KAV\$M\_VME, 4-90  
 KAV\$M\_VSB, 4-90  
 KAV\$M\_WRITE\_ALARM, 4-107  
 KAV\$M\_WRITE\_CALENDAR, 4-107  
 KAV\$M\_WRITE\_RTCRAM, 4-108  
 KAV\$M\_WRITE\_TIMESAVE, 4-107  
 KAV\$M\_WRT\_PROT, 4-63, 4-90  
 KAV\$NOTIFY\_FIFO, 1-6, 4-44, 4-50, 4-77,  
 4-78 to 4-83  
 KAV\$OUT\_MAP, 1-6, 3-11, 4-5, 4-13,  
 4-21, 4-29, 4-84 to 4-95, 4-149, 5-2,  
 5-14, 5-20  
 KAV\$QUEUE\_AST, 1-6, 3-2, 4-36, 4-37, 4-40,  
 4-96 to 4-99, 4-126  
 KAV\$RTC, 1-6, 3-7, 4-130  
 KAV\$RW\_BBRAM, 1-6, 3-10, 4-116 to  
 4-121  
 KAV\$SET\_AST, 1-6, 2-5, 3-2, 3-3, 4-34,  
 4-36, 4-37, 4-40, 4-96, 4-99, 4-122 to  
 4-126  
 KAV\$SET\_CLOCK, 1-6, 4-127 to 4-130  
 KAV\$TIMERS, 1-6, 2-5, 3-5, 3-6, 4-131 to  
 4-143  
 KAV\$UNMAP, 1-6, 4-58, 4-64, 4-84, 4-91,  
 4-144 to 4-149  
 KAV\$VME\_SETUP, 1-6, 3-16, 4-68, 4-73,  
 4-150 to 4-155, 5-15

KAV30  
    initial configuration, A-1 to A-2  
KAV30 applications  
    coding, 5-3 to 5-12  
    compiling, 5-12  
    debugging, 5-20 to 5-21  
    designing, 5-1 to 5-3  
    developing, 5-1 to 5-25  
    including SCSI devices, 5-23 to 5-25  
    linking, 5-12  
KAV30 hardware, 1-1 to 1-2  
    configuration, A-1  
KAV30 software, 1-1, 1-4 to 1-6  
    configuration, A-2  
KAV30 system image  
    building, 5-13 to 5-14  
    loading, 5-20  
    running, 5-20  
Kernel, 1-4  
Kernel mode, 3-2

## L

---

Last chance handler, 5-2  
Last-in/first-out buffers  
    *See* LIFO buffers  
LIFO buffers, 1-6, 3-9, 4-23, 4-27, 4-28  
LIFO mode, 4-74  
Linking  
    KAV30 applications, 5-12  
Little-endian, 3-18, 4-58, 4-84  
Local bus  
    timeout, A-2  
Local bus timer, 3-5, 3-6, 4-131  
Location monitor, 4-58, 4-62  
LOCK, 2-3

## M

---

Master, 2-5, 3-10  
    VMEbus, 2-1 to 2-2, 5-17  
    VSB, 2-3, 5-17  
Mode 0 swapping, 3-19  
Mode 1 swapping, 3-19

Mode 2 swapping, 3-21  
Mode 3 swapping, 3-21  
Modes  
    kernel, 3-2  
    user, 3-2  
Mutex, 5-3

## N

---

Not fair mode, 2-2  
Not hidden mode, 2-2

## P

---

Parity errors, 2-6  
PC, 5-2  
PCB, 3-4  
PCB\$A\_ASTBLK, 3-4  
PCB\$A\_ASTFLK, 3-4  
Ports  
    auxiliary, 1-2, A-1  
    console, 1-2  
    serial line, 1-2, 2-14, A-1  
POWER\_FAIL signal, A-1  
Prescaler, 3-5  
Prioritized arbitration, 2-3  
Priority mode, A-2  
Process Control Block  
    *See* PCB  
Producer, 3-9  
Program Counter  
    *See* PC

## R

---

Read-modify-write, 2-2, 2-3  
Read-modify-write cycles, 4-2, 4-10  
Real-time clock, 4-127 to 4-130  
Release-on-request  
    *See* ROR  
Release-on-request mode  
    ROR mode  
Release-when-done  
    *See* RWD



- RESET signal, A-1
- Reset/halt switch, 2-14
- Retry count, 5-20
- ROR, 2-2
- ROR mode, A-2
- Round-robin arbitration, 2-3
- RTC/begin, 4-100
- RTC/end, 4-115
- rtVAX 300, 1-1, 5-14
  - Ethernet controller, A-1
  - timer, 2-15
- RWD, 2-2

## S

---

- S0 space, 4-2, 4-5, 4-10, 4-13, 4-16, 4-23, 4-27
- Scatter-gather map
  - See* SGM
- SCB, 2-8, 2-9, 3-2
- SCSI bus, A-1
- SCSI class driver
  - building, 5-23 to 5-25
- SCSI class drivers
  - developing, 5-22 to 5-23
- SCSI controller, 1-2, 2-5
- SCSI ID, 5-23, A-2
- Second Generation Ethernet Controller
  - See* SGEC
- Serial line ports, 1-2, 2-14
- SGEC, 5-14
- SGM, 1-1, 3-10 to 3-21, 4-2, 4-10, 4-16, 4-23, 4-58, 4-84, 4-144 to 4-149, A-2
  - byte swapping, 3-18 to 3-21
  - incoming, 3-14 to 3-18
  - outgoing, 3-10 to 3-13
- Shared memory pages, 3-21
- Signal calls, 5-3
- Slave, 3-10
  - VMEbus, 2-2 to 2-3, 2-4, 5-19
- Small computer systems interface controller
  - See* SCSI controller
- Stack, 5-2

- SYSFAIL signal, 2-5
- SYSRESET signal, 2-4
- SYSTEM, 2-3, 4-89
- System clock, 5-17
- System Control Block
  - See* SCB
- System failure, 5-2
- System image, 5-13
- System Parameter 1, 5-14 to 5-17
- System Parameter 2, 5-14, 5-18 to 5-20
- System RAM, 1-1, 2-2
  - parity errors, 2-6
- System random-access memory
  - See* System RAM
- System services, 1-5 to 1-6
- System virtual address space
  - See* S0 space

## T

---

- Target system, 1-1, 1-3
- TERMPWR signal, A-1
- Timer
  - interval, 3-5
- Timers, 3-5 to 3-6
  - prescaler, 3-5
- Trigger boot, 2-14

## U

---

- User mode, 3-2
- User read-only memory
  - See* User ROM
- User ROM, 1-1, A-1

## V

---

- VAX
  - ERR signal, 2-9
  - HALT signal, A-1
  - POWER\_FAIL signal, A-1
- VAX Ada, 1-4
  - coding guidelines, 5-4 to 5-7

- VAX C, 1-2
  - coding guidelines, 5-7 to 5-8
- VAX FORTRAN, 1-2
  - coding guidelines, 5-8
- VAXELN
  - status code, 5-2
  - system time, A-2
- VAXELN Ada, 1-2
- VAXELN applications
  - building, 1-1
  - debugging, 1-1
  - developing, 1-1
  - down-line loading, 1-1
  - running, 1-1
- VAXELN Debugger, 1-2, 5-20
- VAXELN kernel, 1-2
- VAXELN Pascal, 1-2
  - coding guidelines, 5-9 to 5-12
- VAXELN System Builder, 1-2, 5-13, 5-14
  - invoking, 5-13
- VAXELN system time, 4-127 to 4-130
- VAXELN Toolkit, 1-1, 1-2 to 1-4
- VDAD device driver, D-1
- Vectored interrupts
  - VMEbus, 5-15
- VME subsystem bus
  - See* VSB
- VMEbus, 1-1, 1-5
  - A24 base slave address, A-1
  - A24 slave, 5-17, A-2
  - A32 base slave address, A-2
  - A32 slave, 5-17, 5-19, A-2
  - accessing, 5-1 to 5-3
  - ACFAIL signal, 2-5, 2-14, A-1
  - arbiter, 1-2, 2-3 to 2-4, 5-18, A-1, A-2
  - autovectored interrupts, 5-16, A-2
  - BR lines, 2-1
  - BR3 line, A-2
  - bus request, 1-2, A-2
  - Bus Request lines, 2-1
  - configuring, 4-150, 5-14 to 5-20
  - deadlock, 2-4
  - global reset register, 2-5
  - interrupt handler, 1-2, 2-6 to 2-10
  - interrupt request, 1-2
  - interrupt requester, 2-10 to 2-12
  - interrupts, 1-6, A-2
  - master, 2-1 to 2-2, 5-17, A-2
  - reading from, 4-16 to 4-22
  - reset register, 2-2
  - RESET signal, A-1
  - retry count, 5-20
  - slave, 2-2 to 2-3, 2-4, 5-19
  - standby power supply, 3-7
  - SYSFAIL signal, 2-5
  - SYSRESET signal, 2-4
  - system clock, 2-4
  - timeout, A-2
  - utility bus signals, 2-4 to 2-5
  - vectored interrupts, 5-15
  - writing to, 4-2 to 4-9, 4-10 to 4-15, 4-23 to 4-29
- VMEbus bus request, 5-18
- VMEbus interrupt requester, 4-150
- VMEbus interrupts, 1-2
- VMS linker, 5-12
- VSB, 1-1, 1-5
  - accessing, 5-1 to 5-3
  - address spaces, 2-3
  - ALTERNATE, 2-3, 4-89
  - arbiter, 1-2, 2-4, 5-17
  - autovectored interrupts, 5-16
  - bus request, 1-2, 5-17, A-2
  - configuring, 4-150, 5-14 to 5-20
  - I/O, 2-3, 4-89
  - interrupt handler, 1-2, 2-12
  - interrupt request, 1-2
  - interrupts, 1-6
  - LOCK signal, 2-3
  - master, 2-3, 5-17, A-2
  - reading from, 4-16 to 4-22
  - retry count, 5-20
  - SYSTEM, 2-3, 4-89
  - writing to, 4-2 to 4-9, 4-10 to 4-15, 4-23 to 4-29
- VSB interrupt requester, 4-150
- VSB interrupts, 1-2

## **W**

---

Watchdog timer, 1–2, 3–5, 3–6, 4–131

# Reader's Comments

## KAV30 Programmer's Reference Information

AA-PEYCA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less \_\_\_\_\_

What I like best about this manual is \_\_\_\_\_

What I like least about this manual is \_\_\_\_\_

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

I am using **Version** \_\_\_\_\_ of the software this manual describes.

Name/Title \_\_\_\_\_ Dept. \_\_\_\_\_

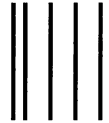
Company \_\_\_\_\_ Date \_\_\_\_\_

Mailing Address \_\_\_\_\_

Phone \_\_\_\_\_

----- Do Not Tear - Fold Here and Tape -----

**digital**™



ATTACH  
STAMP  
HERE

**DIGITAL EQUIPMENT CORPORATION**  
**Corporate User Information Products**  
**ZKO1-3/J35**  
**110 SPIT BROOK RD**  
**NASHUA, NH 03062-9987**



----- Do Not Tear - Fold Here -----

# Reader's Comments

## KAV30 Programmer's Reference Information

AA-PEYCA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less \_\_\_\_\_

What I like best about this manual is \_\_\_\_\_

What I like least about this manual is \_\_\_\_\_

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

I am using **Version** \_\_\_\_\_ of the software this manual describes.

Name/Title \_\_\_\_\_ Dept. \_\_\_\_\_  
Company \_\_\_\_\_ Date \_\_\_\_\_  
Mailing Address \_\_\_\_\_  
\_\_\_\_\_ Phone \_\_\_\_\_

----- Do Not Tear - Fold Here and Tape -----

**digital**™



AFFIX  
STAMP  
HERE

**DIGITAL EQUIPMENT CORPORATION**  
**Corporate User Information Products**  
**ZKO1-3/J35**  
**110 SPIT BROOK RD**  
**NASHUA, NH 03062-9987**



----- Do Not Tear - Fold Here -----

digital