OpenVMS AXP I/O Bus Support

Turbochannel
Futurebus+
EISA

OpenVMS AXP I/O Group

James P. Janetos

Scott W. Apgar

9/22/93

The information in this document is preliminary and is subject to change. The information in this document will be included in the OpenVMS AXP Device Support Developer's Guide, which will be available for review in November 1993.

Revision 0.1 JPJ 10/26/92
Initial Version. Introduction, overview, Turbochannel.
Revision 0.2 JPJ 1/22/93
Updates from ROW/SWA review. Add Futurebus+ chapter.
Revision 0.3 JPJ 5/04/93
More updates from ROW review. Explain longword oriented register view of the world. Fix Futurebus+ manual configuration. Describe Futurebus+ address space allocation.
Revision 0.4 JPJ 9/1/93
Add EISA chapter. Replace system names with model numbers.
Revision 0.5 JPJ 9/22/93
Edits.

ZK6321

1 Introduction

This document describes I/O bus support in the OpenVMS AXP operating system. The first sections describe parts of OpenVMS AXP I/O bus support that are common to all I/O buses. The following sections describe each of the I/O buses currently supported on various Alpha platforms.

Each of the bus-specific sections discusses the bus in general terms, describes the support provided by OpenVMS AXP for the bus, and describes any platform specific differences in the bus interface implementation.

The term "bus support" refers to code in the operating system that 1) probes the system I/O buses to discover I/O adapters, 2) performs appropriate hardware and software initialization on the adapters it finds, 3) constructs a software map of the I/O adapter and I/O bus hierarchy, 4) provides routines to allow access to I/O device registers, 5) provides counted resource (map register) management, 6) provides support for I/O interrupts, and 7) provides autoconfiguration support for the I/O adapters and devices.

In this document, "I/O bus" means the interconnect between hardware modules. Examples of I/O buses are XMI, Turbochannel, Futurebus+, and the EISA bus. The term "I/O adapter" is used to refer to a hardware module connected to an I/O bus.

This document refers to information that is available in other books, including the "Creating a Temporary Alpha VMS Device Driver from a VAX VMS Device Driver" book, and the "VMS for Alpha Platforms Internals and Data Structures" book. This document is not a complete guide to how to write a driver for a particular bus. It does relate information in other driver books to specific I/O bus implementations.

2 Alpha I/O Architecture

Alpha system implementors are allowed a great deal of freedom in designing interfaces to the I/O system. This section discusses the two basic ways of accessing I/O device registers -- directly or through I/O mailboxes. The register access support routines provided by OpenVMS AXP also are discussed in this section.

2.1 I/O Device Register Access

In the VAX architecture, an implementation usually defines a physical address boundary between memory space and I/O space. I/O space physical addresses are mapped into the processors' virtual address space and are accessed using normal VAX instructions (MOV, BIS, etc).

For Alpha, there are no strict rules about how to access I/O space. Some Alpha platforms allow VAX-style I/O space access -- that is, physical I/O space addresses are mapped into the processors' virtual address space and are accessed using load and store instructions. Other platforms provide access to I/O space through I/O mailboxes. I/O mailboxes are data structures (defined by the Alpha architecture) built in system memory and accessed by special I/O subsystem hardware. Some plaforms implement both styles of I/O register access.

One of the challenges presented by the Alpha architecture in I/O device register access is that the architecture defines no byte or word length load and store instructions. Some I/O buses and adapters require byte or word register access granularity for correct adapter operation. Alpha system hardware designers have invented two basic mechanisms for providing byte and word access granularity for I/O adapter register access. One way is termed "sparse" space addressing, where upper order bits in the processor physical address map to an I/O bus address, while lower order bits are used to implement I/O bus byte enable signals. The effect of this is that a large amount of processor physical address space is used to represent the I/O bus address space. The other mechanism for solving the byte/word access problem is the I/O mailbox, as mentioned above.

A significant part of I/O bus support in the OpenVMS AXP operating system is to provide a standard way to access I/O device registers. OpenVMS AXP provides a set of data structures and routines that can be used for register access on any system, regardless of the underlying I/O hardware. The basic paradigm is that the programmer sets up a data structure (the Controller Register Access Mailbox) that describes the register access (read, write, address, etc) and then calls a routine (IOC$CRAM_IO) to perform the actual access. In this way most platform and bus specific differences in the I/O subsystem are hidden from the device driver writer, and the driver can run on multiple platforms without modification.

2.1.1 Direct I/O Device Register Access

Registers that are mapped into the processors' virtual address space and accessed with load and store instructions are said to be accessed directly. This is similar to VAX-style I/O register access. In an Alpha system, registers that are implemented on hardware directly connected to the processor-memory interconnect are usually accessed in this manner. Registers that are accessed directly are also said to be in "local" I/O space. Sparse space register access is an example of direct I/O device register access.

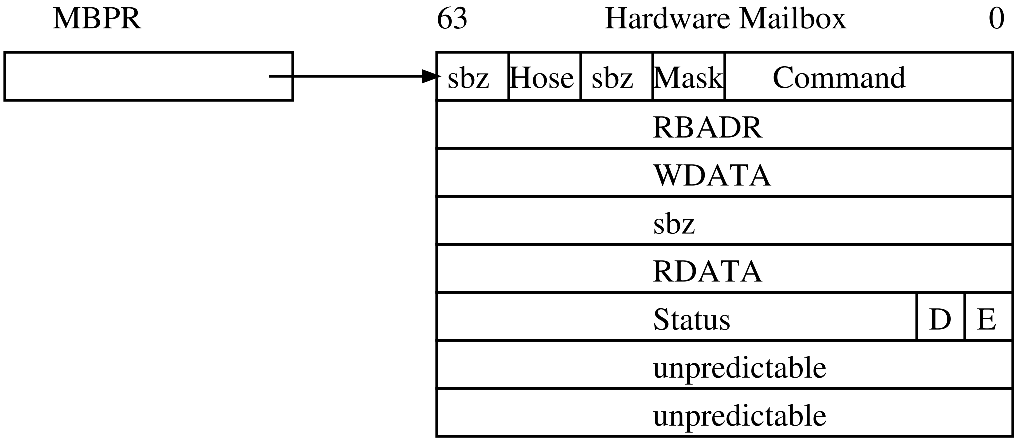
2.1.2 Mailbox I/O Device Register Access

For some Alpha platforms and I/O buses, direct register access is not practical. There may not be enough physical address bits in the system to allow mapping of all of the address space for a particular bus. This is the case for Futurebus+, which supports 64 bit bus addresses. There is the byte/word access problem, as mentioned previously. Also, long latencies in accessing I/O device registers require either stalling the processor/memory interconnect until the access is complete, or more complicated bus designs which may adversely affect performance in other ways.

To solve these problems, the Alpha architecture defines an I/O access mechanism called the I/O mailbox. The I/O mailbox consists of two parts -- a mailbox pointer register (called the MBPR) and the mailbox itself, which is a data structure built in system memory. Software initiates an I/O register access by filling in the necessary fields in the mailbox and then writing the physical address of the mailbox to the MBPR. Special I/O adapter hardware then reads the mailbox data structure from memory, performs the desired access on the I/O bus, and returns data and status in the mailbox.

I/O buses and device registers that are accessed through mailboxes are said to be in "remote" I/O space.

The hardware mailbox is a 64 byte physically contiguous 64 byte aligned data structure. A diagram of a hardware mailbox is shown below:



The fields of the I/O mailbox are defined in the Alpha SRM. A quick summary is given below.

Field	**Contents**
Command	The actual command issued to the remote I/O bus.
Mask	Byte mask -- enables/disables writing of specific bytes of register data.
Hose	Remote I/O bus number.
sbz	Should be Zero.
RBADR	Remote bus physical address of register to be accessed.
WDATA	Data to be written (write operations).
RDATA	Data returned (read operations).
D	Done bit. When set on read operations, indicates mailbox is not in use by the bridge hardware and that read operation is complete. On writes, indi cates mailbox is not in use by the bridge hardware and that mailbox may be reused.
E	Error bit. When set, indicates an error in performing the mailbox opera tion.
Status	Additional status if E bit is set.

There is at most a single MBPR per remote I/O bus. Thus, there must be at least one MBPR per system and no more than the number of remote I/O buses. Access to the MBPR (writing the mailbox physical address) is done with the STQ/C instruction. If the STQ/C fails, software retries up until a timeout limit. Use of the STQ/C instruction provides flow control on an MBPR resource. The MBPR is accessed in local I/O space (that is, the MBPR address is mapped into the processors’ virtual address space).

A mailbox operation is started by writing the physical address of the mailbox to the MBPR using a STQ/C instruction (this is called "queuing the mailbox"). This signals I/O bridge hardware to read the mailbox from memory and to perform the operation on the remote I/O bus. Software can spin waiting for the DONE bit to be set. When the DONE bit is eventually set (by the I/O bridge hardware) the mailbox operation has completed and the data (if the operation was a read) can be extracted from the mailbox. The setting of the DONE bit also indicates that the mailbox is not in use by the I/O bridge and is therefore free to be reused for another mailbox operation.

If there were any errors in the mailbox operation, then the ERROR bit will be set by the I/O bridge. The Status field of the mailbox may contain more information about the error.

On write operations, the setting of the DONE bit does not necessarily mean that the write has actually completed on the remote I/O bus. It does mean, however, that the mailbox can be used to queue another operation. Ordering of mailbox operations on a remote I/O bus is enforced by the I/O bridge hardware.

Note also that it is not necessary to spin waiting for the DONE bit. Software can queue the mailbox, do other work, and come back later to check the DONE bit. However, on reads, the mailbox RDATA field is not valid until the DONE bit is set.

The exact details of how to perform an I/O access using the I/O mailbox are defined in the Alpha SRM. A summary is given below.

- Allocate a mailbox, properly sized and aligned.
- Initiliaze the fields for the requested operation (command, mask, hose, rbadr, wdata).
- Execute an MB to insure that all memory writes to the mailbox have completed before starting the operation.
- Obtain the physical address of the mailbox.
- In the queuing loop:
  - Write the mailbox physical address to the MBPR register using STQ/C. On success, exit the loop.
  - On failure, check that the timeout for queuing has not expired. If no timeout, continue to loop, else return error status.
- Other work can be done at this point.
- In the waiting loop:
  - Check the DONE bit in the mailbox. If set, exit waiting loop.
  - If not set, check the wait-for-completion timeout. If no timeout continue looping, else return error status.
- Execute an MB.
- Check the ERROR bit

The above flow is completely implemented in the IOC$CRAM_IO routine, which is explained in a later section.

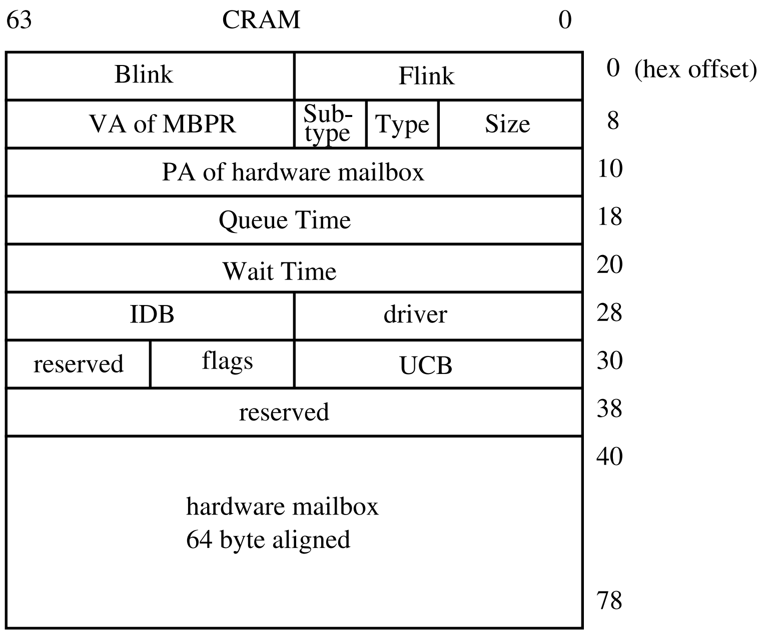
It should be noted that an I/O mailbox operation is simply a way of moving data to or from a device register. A mailbox operation does not, by itself, cause interrupts or IPL changes. It has the same effect as a VAX MOVL instruction--data is moved to or from a device register.

2.2 OpenVMS AXP Mailbox Data Structures and Routines

The previous section described I/O mailbox access as specified by the Alpha architecture. This section discusses the data structures and routines provided by OpenVMS AXP for allocating, managing, and using I/O mailboxes. This section also describes how these same data structures and routines are used to perform I/O device register accesses on all platforms, regardless of whether or not the I/O subsystem hardware actually supports hardware mailboxes or uses direct access techniques. Much of the information in this section can also be found in the book "Creating a Temporary Alpha VMS Device Driver from a VAX VMS Device Driver".

2.2.1 Controller Register Access Mailbox

OpenVMS AXP encloses the hardware mailbox in a data structure called the Controller Register Access Mailbox, or CRAM. The CRAM contains additional information (other than that supplied in the hardware mailbox) that allow bus support routines to efficiently manage hardware mailboxes, such as timeout values for queuing and waiting, the virtual address of the MBPR, the physical address of the hardware mailbox, and pointers to driver data structures. In this way all of the information required to perform an I/O device register access is contained in the CRAM. A diagram of the CRAM is shown below.



The CRAM format is defined in [LIB.LIS]CRAMDEF.SDL. A summary of the field definitions is given below:

Field	Contents
Flink	Forward link. Reserved for driver*
Blink	Backward link. Reserved for driver*
Size	Size of structure (bytes)
Type	Structure type (DYN$C_MISC)
Subtype	Structure subtype (DYN$C_CRAM)
MBPR	Virtual address of MBPR register
PA of HW MBX	Physical address of hardware mailbox
Queue Time	Timeout value for queuing mailbox (ns units)
Wait Time	Timeout value for waiting for DONE bit (ns units)
Driver	Field reserved for driver*
IDB	Pointer to IDB associated with this mailbox
UCB	Pointer to UCB associated with this mailbox
Flags	Software flags

* The fields that are marked as "reserved for driver" are available for driver use.

See the "Creating a Temporary OpenVMS AXP Device Driver from an OpenVMS VAX Device Driver" book for information on the CRAM software flags.

2.2.2 The Adapter Control Block and CRAM Initialization

In general, an Adapter Control Block (ADP) is a data structure used by OpenVMS AXP to represent the interface to an I/O bus. When CRAMs are allocated many of the CRAM fields are initialized from information stored in the ADP. This allows many of the values (such as queue time and wait time) to be tailored to the characteristics of a specific bus and platform.

The Command field of the hardware mailbox is another piece of bus-specific information that must be initialized. The intent of the Command field is that its contents can be directly driven onto target bus command wires in order to accomplish the desired operation. This simplifies I/O bridge hardware design. The software implication is that the command field is defined differently for every I/O bus -- for example, a read command on the XMI is represented by a different bit pattern than a read command on the Futurebus+ and so on. It is even possible that a bus interface on one platform may implement the command field differently than the interface for the same bus on a different platform.

This is a problem for drivers, as a programmer cannot be expected to know all of the possible commands for every platform and every I/O bus. To help solve this problem, OpenVMS AXP defines a set of "commands" that are the same for every bus and platform. These "commands" are actually indices into bus-specific tables (connected to the bus ADP) that contain the command field bit patterns for that bus and platform. Drivers call a system-specific routine with the desired command index, and the routine returns the command field for that bus and platform. In this way drivers are insulated to some degree from bus and platform differences in register access.

The command indices are defined in [LIB.LIS]CRAMDEF.SDL and are listed below.

Command Index	**Command**
CRAMCMD$K_RDQUAD32	Quadword read in 32 bit space
CRAMCMD$K_RDLONG32	Longword read in 32 bit space
CRAMCMD$K_RDTRIBYTE32	3-byte read in 32 bit space
CRAMCMD$K_RDWORD32	Word read in 32 bit space
CRAMCMD$K_RDBYTE32	Byte read in 32 bit space
CRAMCMD$K_WTQUAD32	Quadword write in 32 bit space
CRAMCMD$K_WTLONG32	Longword write in 32 bit space
CRAMCMD$K_WTTRIBYTE32	3-byte write in 32 bit space
CRAMCMD$K_WTWORD32	Word write in 32 bit space
CRAMCMD$K_WTBYTE32	Byte write in 32 bit space
CRAMCMD$K_RDQUAD64	Quadword read in 64 bit space
CRAMCMD$K_RDLONG64	Longword read in 64 bit space
CRAMCMD$K_RDTRIBYTE64	3-byte read in 64 bit space
CRAMCMD$K_RDWORD64	Word read in 64 bit space
CRAMCMD$K_RDBYTE64	Byte read in 64 bit space
CRAMCMD$K_WTQUAD64	Quadword write in 64 bit space
CRAMCMD$K_WTLONG64	Longword write in 64 bit space
CRAMCMD$K_WTTRIBYTE64	3-byte write in 64 bit space

CRAMCMD$K_WTWORD64	Word write in 64 bit space
CRAMCMD$K_WTBYTE64	Byte write in 64 bit space

Most buses do not implement all of the defined command types. The command indices were chosen to support the types of expected register accesses normally supported on I/O buses.

2.2.3 CRAM Routines

This section lists the routines provided by OpenVMS AXP to allocate and use mailboxes. A complete description of these routines can be found in the appendix of the book "Creating a Temporary Alpha/VMS Device Driver from a VAX VMS Device Driver".

Routine	Description
IOC$ALLOCATE_CRAM	Allocates a CRAM and does some initialization (size, type, subtype, MBPR address, queue and wait time, hose, hardware mailbox PA)
IOC$DEALLOCATE_CRAM	Deallocates a CRAM.
IOC$CRAM_CMD	Initializes Command field, Mask field, and RBADR field according to the characteristics of the target bus.
IOC$CRAM_IO	Performs an entire mailbox opera tion including queueing the mailbox and waiting for the DONE bit.
IOC$CRAM_QUEUE	Queues the CRAM for a mailbox operation and returns. It is assumed the caller will eventually call IOC$CRAM_WAIT to await com pletion of the mailbox operation.
IOC$CRAM_WAIT	Returns when DONE bit is set or when mailbox operation times out.

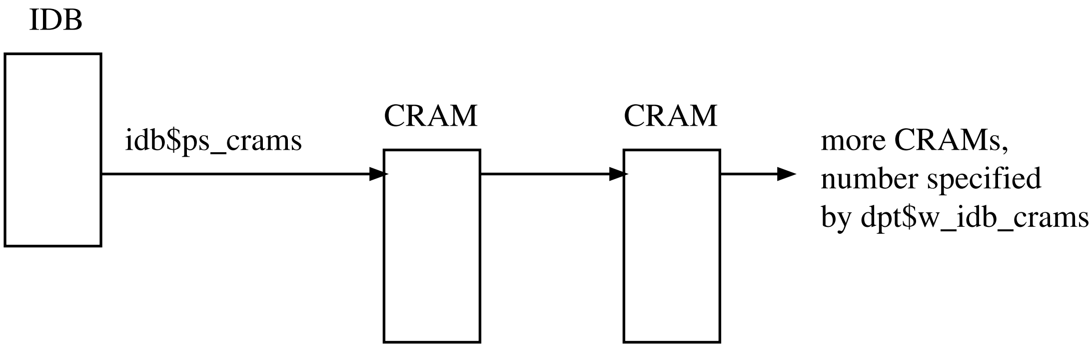
Note -- a call to IOC$CRAM_IO is equivalent to a call to IOC$CRAM_QUEUE followed by a call to IOC$CRAM_WAIT.

2.2.4 CRAM Usage Models

The purpose of a CRAM is to access a register. Drivers have flexibility in how they use CRAMs. One way, suitable for very rare register accesses, is to allocate a CRAM, initialize the CRAM, use the CRAM, and deallocate the CRAM. This might be done for accesses to error registers, for example.

Another way is to allocate and initialize a CRAM for each of the common register accesses the driver expects to perform, prior to doing any I/O. CRAM initialization might be done in a controller init routine, for example. Then, to issue the register access, the driver need only call the appropriate mailbox operation routine (IOC$CRAM_IO or IOC$CRAM_QUEUE followed by IOC$CRAM_WAIT). This technique might be used for register access in a performance path, such as an interrupt service routine.

OpenVMS AXP provides a convenient way for drivers to preallocate CRAMs. The OpenVMS AXP driver loading program examines 2 new fields in the Driver Prologue Table, DPT$W_IDB_CRAMS and DPT$W_UCB_CRAMS. These fields allow the programmer to specify the number of CRAMs to be automatically allocated for driver use, and to associate them with either an IDB or a UCB. The driver loading program calls IOC$ALLOCATE_CRAM for each requested CRAM, and inserts the CRAMs into one of two linked lists, headed by either IDB$PS_CRAM or UCB$PS_CRAM. An example of this is shown in the following diagram:



Each CRAM contains a backpointer to the IDB (cram$ps_idb)

The basic flow for any register access using CRAMs is to allocate a CRAM, initialize the CRAM command, mask, and address fields (and write data, if a write operation), and then issue the access. CRAM allocation can be done automatically by the driver loading program (as shown above) or by the IOC$ALLOCATE_CRAM routine. Initialization of the CRAM command, mask, and address fields is accomplished using the IOC$CRAM_CMD routine. IOC$CRAM_IO issues the actual access to the device register. The following example illustrates a typical use of IOC$CRAM_CMD.

```
/* Call ioc$cram_cmd to initialize the cram */

status = ioc$cram_cmd (cramcmd$k_rdlong32  /* Read/write and access length */
                       4,                  /* Register byte offset         */
                       adp_address,        /* Address of bus ADP           */
                       cram_address);      /* Address of CRAM              */
```

If the transaction is a register write, the WDATA field in the CRAM would be initialized by the driver with the write data at this point.

Note that it is useful to check the return status from IOC$CRAM_CMD. Not all command indices are legal on all buses. The return status will indicate this.

IOC$CRAM_CMD initializes the COMMAND, MASK, and RBADR fields of the CRAM based on the input parameters and the characteristics of the I/O bus. In general, the COMMAND and MASK are determined from the command_index parameter (which specifies read or write and access length) and the byte_offset parameter (which specifies the first byte involved in the transfer). The RBADR field in the CRAM is initialized using the byte_offset parameter and the IDB$Q_CSR field in the IDB. The byte_offset is combined with IDB$Q_CSR and the result is stored in the RBADR field in the CRAM. The RBADR value may also be long or quad aligned, depending on the addressing characteristics of the I/O bus.

To actually issue the register access:

```
status = ioc$cram_io (cram_address);
```

If the transaction was a read, the returned data is available in the RDATA field in the CRAM.

Using a routine to accomplish a register access raises the question of what to do with the return status. On VAX, there was no concept of return status from a register read, since registers were accessed with normal VAX instructions. If a fatal bus error occurred during a register access (a non-existent address error or a parity error, for example) the system would generally crash.

On AXP systems, many severe bus errors result in system crashes as well. In these cases the call to IOC$CRAM_IO will never return. However, for non-fatal bus errors, checking the return status from IOC$CRAM_IO gives the driver writer the opportunity to implement error handling for device register accesses that fail.

2.2.5 Register Data Byte Lane Alignment

The CRAM routines supplied by OpenVMS/AXP enforce a "longword oriented" view of I/O adapter register space. "Longword oriented" means that adapter register space is viewed as if register bytes occupy a 32 bit data path, as follows:

Adapter Register space

31 24	23 16	15 8	7 0	offset
byte 3	byte 2	byte 1	byte 0	0
byte 7	byte 6	byte 5	byte 4	4

etc

Write example: To write a byte to register byte 2, specify IOC$CRAM_CMD parameters as follows:

command_index = cramcmd$k_wtbyte32
byte_offset = 2
adp_address = adp address
cram_address = cram address

The data to be written must be positioned in bits 23:16 of the CRAM write data field (CRAM$Q_WDATA).

Read example: To read a byte from register byte 2, specify IOC$CRAM_CMD parameters as above except use cramcmd$k_rdbyte32 as the command_index.

The data from register byte 2 will be returned in bits 23:16 of the CRAM read data field (CRAM$Q_RDATA).

It is up to the programmer to perform the proper byte lane alignment of data for register writes. On register reads, the data is returned in its natural byte lane without any shifting. Note that this way of looking at adapter register space maps directly to the semantics of most I/O buses, but is distinctly unlike VAX behaviour.

2.2.6 Standard I/O Device Register Access

Previous sections explained the difference between direct register access and mailbox register access, and described the OpenVMS AXP routines and data structures that support register access. It should be noted again that the I/O mailbox data structures and routines exist for all platforms and buses, regardless of whether or not the I/O subsystem hardware actually supports hardware mailboxes. The CRAM should be viewed simply as a data structure that describes an I/O register reference. The use of CRAM data structures and routines for I/O register accesses contributes to driver portability, as most platform and bus implementation differences can be hidden from the driver writer.

3 OpenVMS AXP Counted Resource Management

In OpenVMS AXP, map register management has been generalized to "counted resource" management. The book "Creating a Temporary Alpha VMS Device Driver from a VAX VMS Device Driver" describes the counted resource management data structures and routines in detail The information presented here is a summary of the information in that book.

A counted resource is a resource with the following attributes:

- The resource consists of an ordered set of items.
- The requestor of the resource can request one or more of the items and expects to receive a contiguous set.
- Allocations and deallocations of the resource are common operations -- these operations must be efficient and quick.
- A deallocation of the resource may allow stalled allocation requests to proceed.

Map registers have all of these attributes and therefore can be managed as a generic counted resource.

OpenVMS AXP uses a data structure called the Counted Resource Allocation Block (CRAB) to describe a counted resource. For an I/O adapter that contains map registers, the CRAB that describes the adapter map registers (which are a counted resource) would be pointed to by the ADP$L_CRAB field in the ADP that represents the I/O adapter.

Another data structure, the Counted Resource Context Block (CRCTX), is used to describe a specific request for a counted resource. A driver allocates a CRCTX to describe the map reqister request, and then calls the counted resource allocation routine to actually request the map registers.

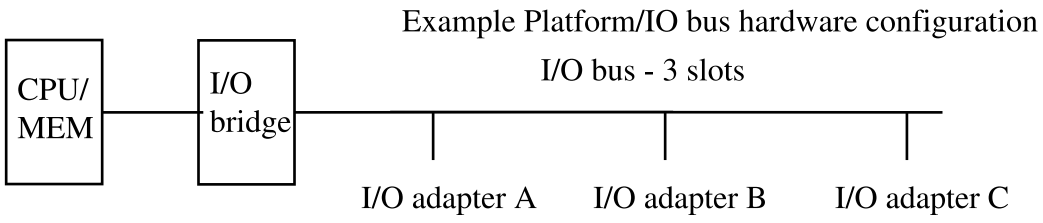
The basic flow a driver follows for setting up a device transfer using map registers is as follows:

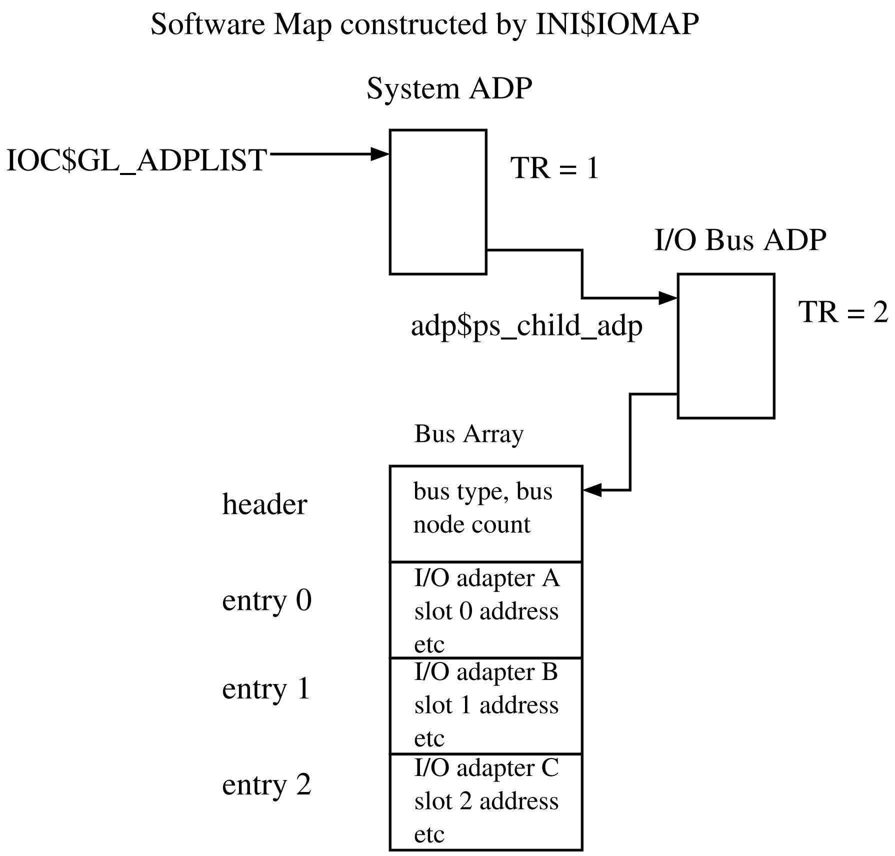
- Call IOC$ALLOC_CRCTX to get a CRCTX that describes a request for map registers.
- Call IOC$ALLOC_CNT_RES to request the map registers.
- Call IOC$LOAD_MAP to load the map registers.
- Set up the appropriate device registers to initiate device activity.
- Call IOC$DEALLOC_CNT_RES to free the map registers.
- Call IOC$DEALLOC_CRCTX to deallocate the CRCTX.

More information on platform specific map register implementations follows in the platform/bus specific chapters.

4 I/O Bus Mapping--The ADP List

INI$IOMAP is a platform specific routine called during booting that probes all of the I/O buses in the system. Its output is a software map that describes the I/O bus and I/O adapter hierarchy. This software map consists of a list of ADPs and Bus Arrays that describe the configuration of the system I/O buses. The ADP list is the set of data structures that is eventually used to drive autoconfiguration of the system I/O buses. The ADP list also provides the hooks for manual configuration of devices and drivers. A hypothetical hardware configuration and the associated ADP list is shown below:





Notes:
System ADP is the root of the ADP hierarchy. It represents the cpu/memory interconnect.
I/O Bus ADP represents I/O bridge.
Every ADP in the system is identified by a unique TR number.
All ADPs have an associated Bus Array.
Bus Array is a variable length structure. It has a header and an entry for every bus slot.
Bus Array entries contain I/O adapter hardware identifiers and I/O bus slot addresses.
[LIB.LIS]ADPDEF.SDL and [LIB.LIS]BUSARRAYDEF.SDL define the structures.

The Bus Array consists of a header and a number of entries (one entry per bus slot). The primary purpose of the ADP list and Bus Arrays is to record the hardware configuration of the I/O adapters and buses. The information in the Bus Arrays is used by autoconfiguration routines to select the correct set of drivers for the I/O adapters that are installed in the system. A picture of a Bus Array entry and an explanation of its fields is given below:

Bus Array entry

<table>
<tr><td>63</td><td align="right">0</td></tr>
<tr><td colspan="2" align="center">hardware ID</td></tr>
<tr><td colspan="2" align="center">base CSR</td></tr>
<tr><td>flags</td><td>Node number</td></tr>
<tr><td>ADP</td><td>CRB</td></tr>
<tr><td>cntrllr letter</td><td>autoconfigure</td></tr>
<tr><td colspan="2" align="center">Bus specific quadword</td></tr>
</table>

BUSARRAY$Q_HW_ID -- hardware identifier, usually a value read from a device type register. Different buses use different methods of identifying adapters, hence the interpretation of this field is bus specific. Initialized by INI$IOMAP.

BUSARRAY$Q_CSR -- bus slot address, initialized by INI$IOMAP. May be a physical address, virtual address, or remote I/O bus address. Interpretation is bus-specific.

BUSARRAY$L_NODE_NUMBER -- bus slot number or bus node number, initialized by INI$IOMAP.

BUSARRAY$L_FLAGS -- communicates information from INI$IOMAP to autoconfigure routine.

BUSARRAY$PS_CRB -- initialized by autoconfigure routine to point to device CRB.

BUSARRAY$PS_ADP -- may point to an ADP lower in the I/O bus hierarchy.

BUSARRAY$PS_AUTOCONFIG -- reserved for use by autoconfigure routine

BUSARRAY$L_CTRLLTR -- records controller letter of device. Initialized by auto configure routine.

BUSARRAY$Q_BUS_SPECIFIC -- communicates bus specific information from INI$IOMAP to autoconfigure routine.

5 Driver Loading and Autoconfiguration

Driver loading and autoconfiguration are covered in detail in the "Creating a Temporary OpenVMS AXP Device Driver from an OpenVMS VAX Device Driver" book. An overview is presented in this section.

5.1 SYSMAN

A driver is built as an executive image and can be loaded by the SYSMAN utility. For devices and drivers supported by Digital, devices are configured automatically by the SYSMAN IO AUTOCONFIGURE command. For 3d party devices and for devices under development for which there is no autoconfigure support, the device must be configured manually. This is accomplished by the SYSMAN IO CONNECT command, which is described below. SYSMAN calls the SYS$LOAD_DRIVER program to actually load the driver executive image. SYS$LOAD_DRIVER allocates a set of data structures describing the device and properly connects them to the rest of the I/O data base.

5.2 SYSMAN IO CONNECT

SYS$LOAD_DRIVER allocates a UCB, CRB, and IDB (and more) in response to a CONNECT command. Use the SYSMAN utility to issue the CONNECT command, as follows:

```
SYSMAN> io connect devname/adapter=x/csr=y/vec=z/driver_name=xxdriver.exe
```

The "devname" parameter is a standard device name consisting of a two letter device type, a controller letter, and a unit number, such as "XXA0"

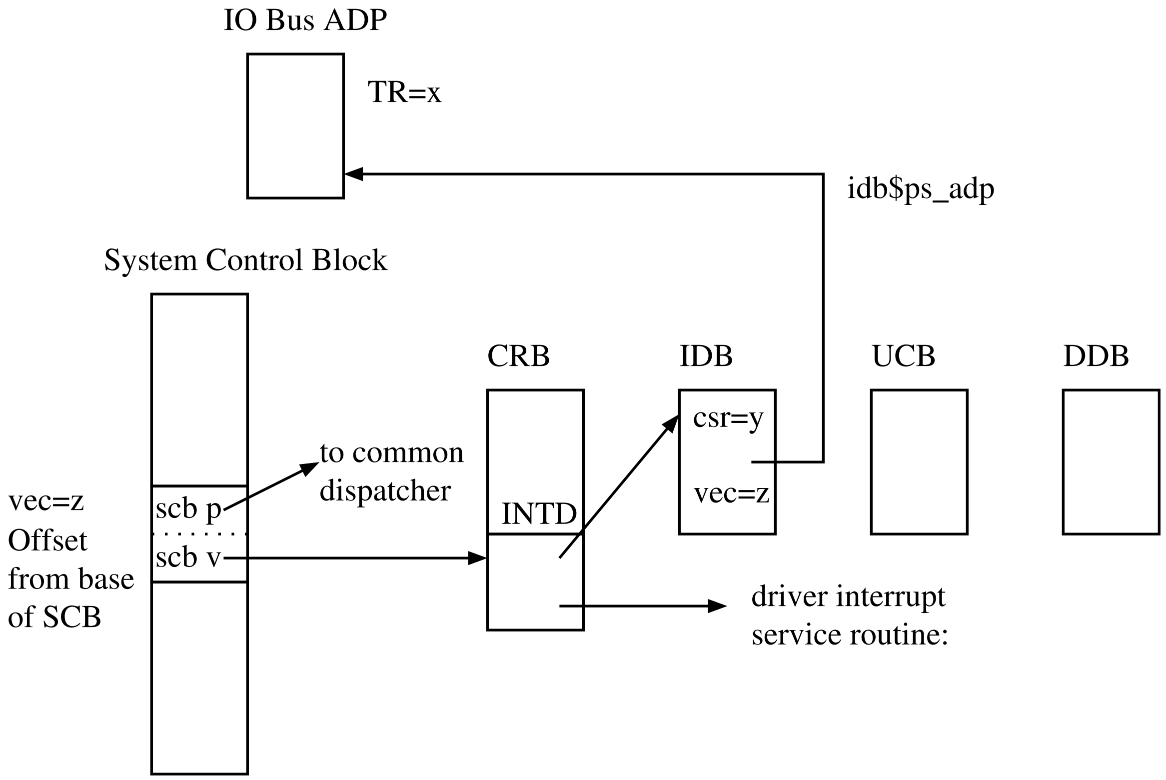
The "adapter" parameter specifies the TR number of the ADP in the system with which the device is associated.

The "csr" parameter specifies the base CSR of the device. This value is copied to the IDB$Q_CSR field of the IDB created by SYS$LOAD_DRIVER. Note that the "csr" parameter is often specified differently for different buses. For example, on Futurebus+, the csr parameter is a Futurebus+ physical address, while for EISA, the csr parameter is a virtual address that maps to the physical address of EISA slot-specific I/O space. See the chapters on each bus for the details of the csr on that bus.

The "vec" parameter specifies the interrupt vector(s) of the device. Interrupt vector assignment varies from bus to bus and from platform to platform. See the specific chapter on the platform/bus of interest to see how to specify the "vec" parameter. If a single vector is specified, it is copied to the IDB$L_VECTOR field in the IDB by SYS$LOAD_DRIVER. If a list of vectors are specified, they are contained in a VLE structure pointed to by the IDB$L_VECTOR field in the IDB.

The "driver_name" parameter specifies the driver name.

A example of the data structures created by SYS$LOAD_DRIVER in response to a CONNECT command is shown below:



Notes:
SYS$LOAD_DRIVER creates CRB, IDB, UCB, DDB.
"csr" parameter from CONNECT command is copied to IDB$Q_CSR
"adapter" parameter from CONNECT command is used to find ADP.
"vec" parameter from CONNECT command is used to connect interrupt dispatch table (which is either the System Control Block for directly dispatched interrupts, or a vector table attached to the I/O Bus ADP for indirectly dispatched interrupts) to INTD block (contained in CRB) and to driver interrupt service routine.
The interrupt vector(s) are also stored in the IDB or in a VLE pointed to by the IDB.
[LIB.LIS]VECDEF.SDL defines layout of INTD block.

The SYSMAN IO SHOW BUS command uses the ADP list and Bus Arrays to display the I/O configuration of the system. The IO SHOW BUS display is intended to provide the information needed to properly issue an IO CONNECT command to manually configure a device.

6 Turbochannel Support on OpenVMS AXP

This section discusses base Turbochannel support in the OpenVMS AXP operating system. First a general description of Turbochannel concepts is given. This is followed by a description of specific Turbochannel implementations on different Alpha platforms.

6.1 Turbochannel Overview

The Turbochannel is a synchronous I/O bus with a 32 bit multiplexed address and data path. It can operate at clock frequencies from 12.5 to 25 MHz with a peak DMA bandwidth of 100 Mbytes per second. The Turbochannel architecture distinguishes between a system module, containing the processor/memory system, and option modules, which are generally I/O controllers. The Turbochannel is asymmetric, in that the system module can read or write option modules and option modules can read or write the system module, but direct communication between option modules is not allowed.

Option modules occupy Turbochannel backplane slots. The backplane slot determines the base address of the option module. Each slot has from 4-512 Mbytes of address space. The actual address space allocated to each Turbochannel slot and the base address of each slot is system specific. The Turbochannel supports the notion of integral options, which are logical Turbochannel options, physically implemented on the system module, that do not occupy backplane slots.

I/O transactions on the Turbochannel are defined as loads and stores from the system module to option module addresses. I/O transaction addresses on the Turbochannel are 29 bits, with the two least significant bits implicitly zero.

DMA transactions on the Turbochannel are defined as option module reads and writes to system memory. The Turbochannel architecture defines a 34 bit byte address, with the two least significant bits implicitly zero, for DMA address space. The actual DMA address space is system specific. All Turbochannel DMA transfers are in units of 32 bit longwords, and can be of any length up to a system specific limit.

Interrupts on the Turbochannel are associated with the Turbochannel slots and are level sensitive. The interrupt priority of each slot is system specific. An option must maintain its interrupt signal until software explicitly clears the interrupt condition at the option.

All Turbochannel options must implement an Option ROM, which contains information about the option module.

6.2 Turbochannel on DEC 3000 Model 500

The following sections cover the DEC 3000 Model 500 Turbochannel address map, DEC 3000 Model 500 Turbochannel dense and sparse space addressing, register access, DMA and map register support, bus interface hardware registers, I/O space mapping, and loading a driver for a Turbochannel option.

6.2.1 DEC 3000 Model 500 Turbochannel Address Map

The Turbochannel bus on a DEC 3000 Model 500 system contains 6 option slots, labelled 0-5 by OpenVMS AXP. The Turbochannel address map for DEC 3000 Model 500 is shown below:

DEC 3000 Model 500 Turbochannel Address Map

Slot	Base Physical Address	Size	Space
0	1 0000 0000	128 MB	Slot 0 Dense space
	1 0800 0000	128 MB	Reserved
	1 1000 0000	256 MB	Slot 0 Sparse space
1	1 2000 0000	128 MB	Slot 1 Dense space
	1 2800 0000	128 MB	Reserved
	1 3000 0000	256 MB	Slot 1 Sparse space
2	1 4000 0000	128 MB	Slot 2 Dense space
	1 4800 0000	128 MB	Reserved
	1 5000 0000	256 MB	Slot 2 Sparse space
3	1 6000 0000	128 MB	Slot 3 Dense space
	1 6800 0000	128 MB	Reserved
	1 7000 0000	256 MB	Slot 3 Sparse space
4	1 8000 0000	128 MB	Slot 4 Dense space
	1 8800 0000	128 MB	Reserved
	1 9000 0000	256 MB	Slot 4 Sparse space
5	1 A000 0000	128 MB	Slot 5 Dense space
	1 A800 0000	128 MB	Reserved
	1 B000 0000	256 MB	Slot 5 Sparse space

DEC 3000 Model 500 also contains 3 integral options--the integrated SCSI adapter (Turbochannel slot 6), the Core I/O subsystem (Turbochannel slot 7), and an integrated graphics controller (Turbochannel slot 8).

Symbols defining the slot base physical addresses for the Turbochannel on DEC 3000 Model 500 can be found in file [LIB.LIS]IO0402DEF.SDL. These symbols are of the form IO0402$Q_SLOTx_DENSE_BASE or IO0402$Q_SLOTx_SPARSE_BASE.

6.2.2 Dense and Sparse Space Addressing

The DEC 3000 Model 500 Turbochannel bus interface hardware maps every byte of Turbochannel address space into two distinct address spaces. These address spaces are called dense and sparse address space. Dense space addresses correspond to "normal" addresses, where successive longword addresses are 4 bytes apart. In sparse space, the address space has been expanded by a factor of two, so that successive longword addresses are actually 8 bytes apart. This allows space for a byte mask to be supplied on I/O write transactions. This can be better illustrated by an example.

Suppose an option implements two longword registers as shown below:

Option Register Layout

31 ... 0	
register A	slot base + 4 0000
register B	slot base + 4 0004

If this option was installed in slot 0 in the DEC 3000 Model 500 Turbochannel, the option registers would appear at two different physical addresses, as shown:

Option Register Layout--Dense Space

31 0	physical address
register A	1 0004 0000
register B	1 0004 0004

Option Register Layout--Sparse Space

63 32	31 0	physical address
mask	register A	1 1008 0000
mask	register B	1 1008 0008

Notes:
Option space addresses are expanded by a factor of two in sparse space.
To convert from a dense space address to an equivalent sparse space address, set bit 28 of the physical address and shift 26:0 left by one bit.
In sparse space, bits 35:32 of the data longword are used as a byte mask on I/O write transactions. Byte mask bit set to a "1" causes corresponding data byte to be written.

The access characteristics of dense space and sparse space are different. In general, LDL/STL/LDQ/STQ are legal in either dense or sparse space. The main reason for having sparse space is that it allows the programmer to specify a byte mask for I/O write transactions, which allows byte write granularity.

The following general guidelines apply to sparse and dense space access

- To read a longword from an option register, use a LDL to the sparse space address.
- To write a longword to an option register, use a STL to either the dense or sparse space address.
- To write 1-3 bytes to an option register, use a STQ to the sparse space address with the appropriate mask bits set in the upper longword of the data quadword (mask bit equal to a 1 causes corresponding data byte to be written).
- LDL or LDQ to a dense space address always results in two Turbochannel I/O read transactions to consecutive longword addresses Use caution in dense space if the option has registers with read side effects.

- STQ to a dense space address causes two Turbochannel I/O write transactions to consecutive longword addresses. This may result in writing more data than intended.

See section 6.2.3.1 for examples of dense and sparse space register access.

6.2.3 DEC 3000 Model 500 Turbochannel Register Access

DEC 3000 Model 500 does not implement hardware mailboxes for I/O register access. To reference a Turbochannel option module register, the option slot address space must be mapped into the processors' virtual address space. The option registers can then be accessed using load or store instructions. This is an example of direct register access.

However, the OpenVMS AXP mailbox data structures and routines can still be used for I/O register access. The following sections give examples of how to access registers directly or through CRAMs.

6.2.3.1 Direct Register Access on DEC 3000 Model 500 Turbochannel

On DEC 3000 Model 500, the programmer has the option of accessing registers directly or through the OpenVMS AXP mailbox routines. For direct access, the programmer must decide in which space (sparse or dense) to map the registers. Then, registers can be accessed using load and store instructions.

Consider the previous example of the option with two registers. To access these registers in dense space, the programmer should map the option into virtual address space as follows:

Option Register Layout--Dense Space

virtual address	31 0	physical address
base VA	register A	1 0004 0000
base VA + 4	register B	1 0004 0004

Register A and B now can be accessed in a number of ways, with somewhat different results. In the following examples, assume the base VA has been loaded into register R0.

A. LDL R1, (R0)

Two Turbochannel I/O read transactions are issued. Register A and register B are both read. The data from register A is sign extended and returned in bits 31:0 of R1. The data from register B is discarded.

B. LDQ R1, (R0)

Two Turbochannel read transactions are issued. Register A and register B are both read. The data from register A is returned in bits 31:0 of R1, and the data from register B is returned in bits 63:32 of R1.

C. STL R1, (R0)

Two Turbochannel write transactions are issued. Bits 31:0 of R1 are written to both register A and register B.

D. STQ R1, (R0)

Two Turbochannel write transactions are issued. Bits 31:0 of R1 are written to register A, and bits 63:32 of R1 are written to register B.

Now suppose that the registers are mapped in sparse space, as shown below.

Option Register Layout--Sparse Space

virtual address	63 32	31 0	physical address
base VA		register A	1 1008 0000
base VA+8		register B	1 1008 0008

In sparse space, the register access behaviour is as follows. Assume the base VA is contained in R0.

A. LDL R1, (R0)

Register A is read. The data from register A is sign extended and returned in bits 31:0 of R1.

B. LDQ R1, (R0)

Register A is read. The data from register A is returned in bits 31:0 and in bits 63:32 of R1.

C. STL R1, (R0)

Bits 31:0 of R1 are stored in register A.

D. STQ R1, (R0)

1-4 bytes of the data in R1 are stored in register A, depending on the byte mask in bits 35:32 of R1. A "1" in the byte mask causes the corresponding byte to be written.

As can be seen from the above examples, the access characteristics of each space are different. These characteristics must be considered when deciding how to do option register access.

6.2.3.2 Mailbox Register Access on DEC 3000 Model 500 Turbochannel

DEC 3000 Model 500 does not implement hardware mailboxes as described in the Alpha SRM. However, OpenVMS AXP still provides the CRAM data structures and CRAM routines for I/O register access on DEC 3000 Model 500 Turbochannel. Basically, IOC$CRAM_CMD is used to specify the transaction type and length, and then the IOC$CRAM_IO routine issues the appropriate load or store instruction based on the command field in the CRAM. There is no MBPR, the mailbox is not actually queued to an I/O bridge, and there is no need to wait for the DONE bit. The actual access is accomplished by a load or a store directed to an address in the processors' virtual address space, exactly as in direct access.

On DEC 3000 Model 500, IOC$CRAM_QUEUE simply calls IOC$CRAM_IO, and IOC$CRAM_WAIT simply returns SS$_NORMAL.

For I/O register access, the OpenVMS/AXP mailbox routines expect the option registers to be mapped into sparse space. Sparse space was chosen as the design center for IOC$CRAM_CMD on DEC 3000 Model 500/Turbochannel because it allows byte write granularity and avoids dense space side effects.

Before performing any kind of register access on DEC 3000 Model 500 Turbochannel, the option registers must be mapped into the processors' virtual address space. Thus the driver must be told the base physical address of the Turbochannel slot into which its option is installed. When a driver is loaded using the SYSMAN IO CONNECT command, the value specified as the "csr" parameter is copied to the IDB$Q_CSR field in the IDB. For the Turbochannel, the csr parameter should be specified as the Turbochannel slot base physical address of the Turbochannel option. For mailbox register access on DEC 3000 Model 500 Turbochannel, the csr parameter should be the sparse space base slot physical address (which is displayed in the SYSMAN IO SHOW BUS command). Then in the driver controller init routine, the driver should map its option registers into sparse space and copy the base virtual address of the option registers back into the IDB$Q_CSR field. The IDB$Q_CSR field is used by IOC$CRAM_CMD as the base address of the option registers.

In OpenVMS AXP V2.0, we will be implementing a platform/bus independent mapping routine that a driver can call to do the mapping described above. More information on this can be found in the Device Support Reference Manual for OpenVMS AXP V2.0.

IOC$CRAM_CMD and IOC$CRAM_IO on DEC 3000 Model 500 have been designed together to accomplish register access in sparse space. IOC$CRAM_CMD initializes the COMMAND, MASK, and RBADR fields based on the command_index and byte_offset input parameters. IOC$CRAM_IO interprets the COMMAND field and generates the proper instruction (LDL, STL, or STQ) to the address in the RBADR field.

The RBADR field in the CRAM is initialized using the byte offset parameter and the IDB$Q_CSR field in the IDB. The byte offset is longword aligned, multiplied by 2 to account for sparse space mapping, and added to the IDB$Q_CSR field. The resulting address is stored in the RBADR field of the CRAM.

The COMMAND field in the CRAM is derived from the command index parameter. All quadword length command indices are rejected, as the Turbochannel does not support quadword I/O transactions. If the caller specifies a read transaction (one of cramcmd$k_rdlong32/64, cramcmd$k_rdword32/64, or cramcmd$k_rdbyte32/64), the command index is converted to cramcmd$k_rdlong32 and stored in the COMMAND field in the CRAM, so that IOC$CRAM_IO will issue a LDL instruction for all Turbochannel I/O reads.

If the caller specifies a byte or word length write (one of cramcmd$k_wtbyte32/64 or cramcmd$k_wtword32/64), IOC$CRAM_CMD must also generate the proper byte mask. The byte mask is generated using the command index to determine the transaction length (byte or word), and the byte offset parameter to determine the first byte involved in the transaction. IOC$CRAM_CMD forms a 4 bit mask and copies it to the MASK field in the CRAM and also to bits 35:32 of the WDATA field. The byte or word command index is converted to cramcmd$k_wtquad32 and stored in the COMMAND field of the CRAM, so that IOC$CRAM_IO will generate a STQ (with the proper mask bits set in bits 35:32 of the data quadword) for all byte and word length writes.

On byte or word length writes, the programmer must be careful not to overwrite the mask in bits 35:32 of the WDATA field when loading write data into the CRAM.

If the caller specifies a longword write (cramcmd$k_wtlong32), IOC$CRAM_CMD copies the command index directly to the COMMAND field. IOC$CRAM_IO will generate a STL for all longword writes, which causes all 4 bytes of the register data to be written.

The following section gives examples of register reads and writes using CRAMS.

Consider our previous example of the option with two registers, mapped into sparse space as shown.

Option Register Layout

31 0	physical address
register A	slot base + 4 0000
register B	slot base + 4 0004

Option Register Layout--Sparse space

	31 0	virtual address
mask	register A	base VA
mask	register B	base VA+8

To initialize the CRAM for a read to register A, call IOC$CRAM_CMD as follows:

```
status = ioc$cram_cmd (cramcmd$k_rdlong32,
                       0,
                       adp_address,
                       cram_address);
```

IOC$CRAM_CMD forms the RBADR field of the CRAM by 1) longword aligning the byte offset input parameter, and 2) multiplying the longword aligned byte offset by 2 (to account for sparse space) and 3) adding the longword aligned byte offset to the value from IDB$Q_CSR. In this example with a byte offset parameter of zero, the resulting address would be "base VA".

Then, to perform the actual read to register A, call IOC$CRAM_IO as follows:

```
status = ioc$cram_io (cram_address);
```

IOC$CRAM_IO will issue a LDL to the address in the RBADR field in the CRAM. In this example, this will be a LDL to "base VA". The data from register A will be returned in the RDATA field in the CRAM.

To set up a CRAM for a read to register B, call IOC$CRAM_CMD as follows:

```
status = ioc$cram_cmd (cramcmd$k_rdlong32,
                       4,
                       adp_address,
                       cram_address);
```

IOC$CRAM_CMD longword aligns the byte offset parameter, multiplies the aligned byte offset by 2 (to account for sparse space) and adds the result to the value from IDB$Q_CSR. In this example the resulting address would be "base VA+8". To perform the actual read, issue a call to IOC$CRAM_IO. The data from register B will be returned in the RDATA field of the CRAM.

To set up a CRAM for a longword write to register B, call IOC$CRAM_CMD as follows:

```
status = ioc$cram_cmd (cramcmd$k_wtlong32,
                       4,
                       adp_address,
                       cram_address);
/* Load the CRAM WDATA field with the data to be written. */
```

IOC$CRAM_CMD longword aligns the byte offset parameter, multiplies the aligned byte offset by 2 (to account for sparse space) and adds the result to the value from IDB$Q_CSR. In this example the resulting address would be "base VA+8". A subsequent call to IOC$CRAM_IO will cause a STL using the WDATA field of the CRAM as the write data, to the address in RBADR (which is "base VA+8").

Now suppose that the programmer wants to write byte 3 of register A without affecting bytes 0, 1, or 2. To set up a CRAM for this operation, call IOC$CRAM_CMD as follows:

```
status = ioc$cram_cmd (cramcmd$k_wtbyte32,
                       3,
                       adp_address,
                       cram_address);
/* Now load the CRAM WDATA field with the data to be written. */
```

As above, IOC$CRAM_CMD will form RBADR by longword aligning the byte offset and multiplying it by 2, then adding the result to the value from IDB$Q_CSR. In this example the resulting address will be "base VA". IOC$CRAM_CMD will use the command index (wtbyte32) and the byte offset to form a mask of 1000 (binary). This mask will be copied to bits 35:32 of the WDATA field. To perform the actual write, issue a call IOC$CRAM_IO.

```
status = ioc$cram_io (cram_address);
```

IOC$CRAM_IO will generate a STQ using the WDATA field as the write data, to the address in the RBADR field in the CRAM. Since bits 35:32 of the WDATA field contain a mask of 1000 (binary), only byte 3 of register A will actually be written.

6.2.4 DEC 3000 Model 500 Turbochannel DMA

This section summarizes and describes the OpenVMS AXP routines that support DEC 3000 Model 500 map register usage.

The Turbochannel architecture defines a 34 bit DMA address, however, DEC 3000 Model 500/Turbochannel implements a 30 bit DMA address. DEC 3000 Model 500 restricts DMA bursts on the Turbochannel to be 128 longwords or less. A DMA burst must not cross a 2 KB address boundary.

DEC 3000 Model 500 implements a set of map registers (also called a Scatter/Gather map) that can be used to direct DMA transfers to anywhere in DEC 3000 Model 500 system memory. Use of the Scatter/Gather map for DMA can be enabled on a per-slot basis (described later). DEC 3000 Model 500 refers to "physical DMA" as DMA that does not use the Scatter/Gather map, and "virtual DMA" as DMA that uses mapping information from the Scatter/Gather map.

6.2.4.1 Physical DMA

In physical DMA, the Turbochannel option emits a 30 bit DMA address on the Turbochannel (it may emit more bits, but DEC 3000 Model 500/Turbochannel only implements 30 bits). This address is passed directly to system memory. The pages of the system memory buffer to which the DMA is directed must be physically contiguous. As mentioned, an individual DMA burst may not exceed 128 longwords in length, and may not cross a 2 KB address boundary.

6.2.4.2 Virtual DMA

In virtual DMA, part of the DMA address from the Turbochannel option is used to access the Scatter/Gather map. The Scatter/Gather map entry provides the "physical page number", which is appended to the byte offset from the option DMA address to form the address that is sent to system memory. The format of a Scatter/Gather map entry and a diagram of the address translation is shown below:

Scatter/Gather Map entry

23	22	21	20 ... 4	3 ... 0
V	F	P	physical page number	unused

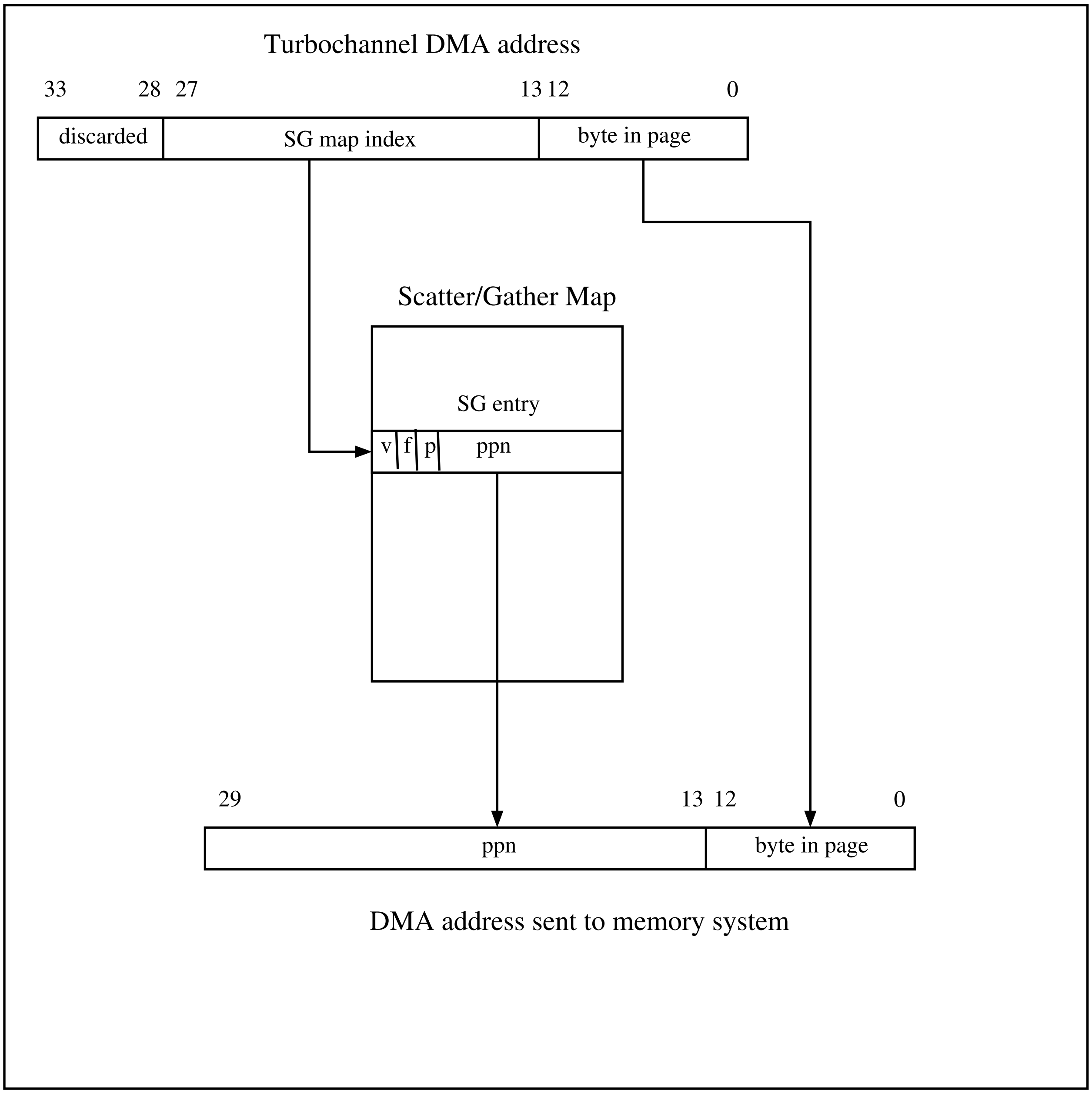
Bit 23 -- Valid bit, indicates entry is valid

Bit 22 -- Funny bit, used by diagnostics

Bit 21 -- Parity bit, protects entire entry

Bits 20:4 -- Physical Page Number

The Scatter/Gather Map is used to translate a Turbochannel DMA address as follows:



For virtual DMA, the programmer sets up the Scatter/Gather map to properly map the DMA buffer in system memory. The DMA buffer need not be physically contiguous in system memory. The Turbochannel DMA addresses emitted by the Turbochannel option are intercepted by the Scatter/Gather map and translated as shown above.

A "SG Enable" bit must also be set in the IOSLOT register. This is explained in sections 6.2.5.1 and 6.2.5.3.

The following sections explain how to allocate and initialize Scatter/Gather map entries.

6.2.4.3 Scatter/Gather Map Management

The Scatter/Gather map is an example of a "counted resource", described earlier. Counted resource management is accomplished using a Counted Resource Context Block (CRAB), which is pointed to by the ADP$L_CRAB field in the Turbochannel ADP. The CRAB contains a count of the number of Scatter/Gather entries, the allocation granularity, and an allocation array, which keeps track of the allocated entries.

6.2.4.4 Allocating Scatter/Gather Map Entries

Allocating Scatter/Gather entries for a DMA operation involves two routines. The programmer first must allocate a Counted Resource Context Block (CRCTX), which is used to describe the request for a counted resource. A call to IOC$ALLOC_CRCTX is shown below:

```
status = ioc$alloc_crctx (crab_address, /* Address of CRAB              */
                          crctx_ref);   /*Address of longword cell to   */
                                        /* receive address              */
                                        /* of CRCTX                     */
```

The programmer then initializes the CRCTX$L_ITEM_CNT field of the CRCTX with the requested number of Scatter/Gather entries. The caller must request two more entries than required to actually map the DMA buffer, since the second to last entry of a request is always initialized to point to the black hole page (from EXE$GL_BLAKHOLE), and the last entry of a request is always initialized to zero to protect against runaway transfers.

To allocate the Scatter/Gather entries, the programmer calls IOC$ALLOC_CNT_RES, as follows:

```
status = ioc$alloc_cnt_res (crab_address,     /* Address of CRAB   */
                            crctx_address);   /* Address of CRCTX  */
```

On DEC 3000 Model 500, the requested number of Scatter/Gather map entries (from CRCTX$L_ITEM_CNT) is rounded up to a multiple of 16, so that Scatter/Gather entries are are always allocated in multiples of 16. The reason for this is to lessen the management overhead in the CRAB. IOC$ALLOC_CNT_RES returns the item number of the first allocated resource in CRCTX$L_ITEM_NUM.

See the "Creating a Temporary OpenVMS AXP Device Driver from an OpenVMS VAX Device Driver" book for a complete description of IOC$ALLOC_CRCTX and IOC$ALLOC_CNT_RES.

6.2.4.5 Loading Scatter/Gather Map Entries

After the Scatter/Gather entries have been allocated, they must be loaded such that they properly map the DMA buffer in system memory. This is accomplished by a call to IOC$LOAD_MAP, as follows:

```
status = ioc$load_map (adp_address,        /* Address of Turbo ADP   */
                       crctx_address,      /* Address of CRCTX       */
                       svapte,             /* System virtual address of PTE */
                                           /* for first page of DMA buffer  */
                       boff,               /* Byte offset into first page of */
```

```
                                   /* DMA buffer   */
           dma_addr_ref);          /* Address of location to receive */
                                   /* Turbochannel DMA address   */
```

IOC$LOAD_MAP uses the CRCTX$L_ITEM_NUM field to find the first Scatter/Gather entry that has been allocated, and then initializes the Scatter/Gather entries based on the starting system virtual address of the DMA buffer. The DMA address is written to the location specified by dma_addr_ref. Note that the DMA address is a full byte address--the lower two bits must be cleared before the address is written to a Turbochannel option DMA address register.

6.2.5 DEC 3000 Model 500/Turbochannel Interface Registers

DEC 3000 Model 500 provides two registers that provide control over the characteristics of a Turbochannel slot. These registers are the IOSLOT register and the IMASK register. The IOSLOT register, on a per-slot basis, enables or disables Scatter/Gather map usage and Turbochannel parity generation and checking. The IMASK register, on a per-slot basis, enables or disables the delivery of Turbochannel interrupts to the processor.

OpenVMS AXP provides access to these registers via the IOC$NODE_FUNCTION routine.

6.2.5.1 IOSLOT Register

The IOSLOT register contains 3 bits per Turbochannel slot, as shown below:

IOSLOT Register

17 15	14 12	11 9	8 6	5 3	2 0
P B S	P B S	P B S	P B S	P B S	P B S
slot 5	slot 4	slot 3	slot 2	slot 1	slot 0

P -- Parity bit
B -- Block Mode bit
S -- Scatter/Gather bit

IOSLOT physical address: 1 C200 0000 (dense), 1 D400 0000 (sparse)

The parity bit, when set, means that the system will check parity on cycles during which the option in that slot is driving the bus (the system always generates parity when the system is driving the bus).

The Block Mode bit enables a specialized DEC 3000 Model 500 Turbochannel style of I/O write transactions. No option should ever set this bit.

The Scatter/Gather bit, when set, means that DMA transfers from that Turbochannel slot will use the Scatter/Gather map.

6.2.5.2 IMASK Register

The Interrupt Mask (IMASK) register enables or disables Turbochannel interrupts on a per-slot basis. When the console transfers control to the operating system, the IMASK is set to all ones, meaning that interrupts from all Turbochannel slots are masked (disabled). It is up to the Turbochannel option driver to enable interrupts for its slot. The IMASK register is shown below:

IMASK

5	4	3	2	1	0

Bits 5:0 enable/disable interrupts from corresponding Turbochannel slot.
When bit is set to a one, interrupts are masked (disabled).
When bit is set to a zero, interrupts are unmasked (enabled).

IMASK physical address: 1 C2400 0000 (dense space only).

The IMASK register should be set up during the driver's controller or unit init routine by using the IOC$NODE_FUNCTION routine as explained below.

6.2.5.3 IOC$NODE_FUNCTION

IOC$NODE_FUNCTION provides an easy way for drivers to manipulate the IOSLOT register and the IMASK register.

IOC$NODE_FUNCTION accepts the CRB address and a function code as input arguments. The CRB$L_NODE field must contain the Turbochannel slot number of the Turbochannel option. On DEC 3000 Model 500 Turbochannel, when a device is manually configured using the SYSMAN IO CONNECT command, the user must use the /NODE qualifier, specifying the Turbochannel slot number into which the option is installed. This causes SYS$LOAD_DRIVER to copy the Turbochannel slot number to the CRB$L_NODE field in the CRB.

The function codes are defined in [LIB.LIS]IOCDEF.SDL and are listed below:

ioc$k_enable_intr	Causes IMASK bit for TC slot to be cleared.
ioc$k_disable_intr	Causes IMASK bit for TC slot to be set.
ioc$k_enable_sg	Causes S bit in IOSLOT for TC slot to be set.
ioc$k_disable_sg	Causes S bit in IOSLOT for TC slot to be cleared.
ioc$k_enable_par	Causes P bit in IOSLOT for TC slot to be set.
ioc$k_disable_par	Causes P bit in IOSLOT for TC slot to be cleared.

ioc$k_enable_blkm	Causes B bit in IOSLOT for TC slot to be set.
ioc$k_disable_blkm	Causes B bit in IOSLOT for TC slot to be cleared.

An example call to IOC$NODE_FUNCTION is shown below:

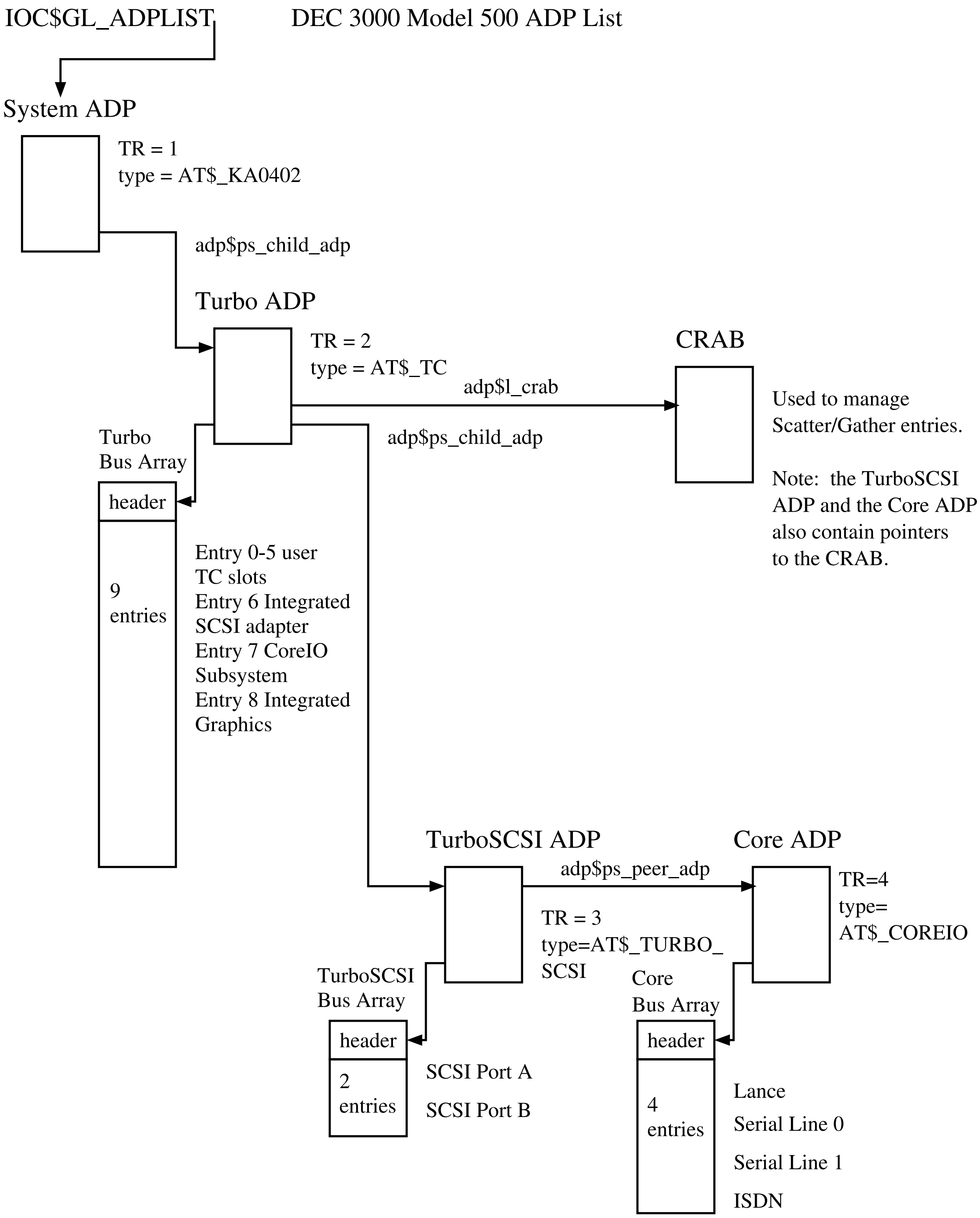
```
status = ioc$node_function (crb_address,
                            ioc$k_enable_intr);
```

IOC$NODE_FUNCTION finds the Turbochannel slot number from the CRB$L_NODE field, and sets or clears the appropriate bit in IOSLOT or IMASK, depending on the function code.

Note that IOC$NODE_FUNCTION acquires the MEGA spinlock while performing all of the above functions.

6.2.6 DEC 3000 Model 500 Turbochannel I/O Space Map

During booting, INI$IOMAP creates an ADP list for DEC 3000 Model 500 Turbochannel. The ADP list looks as follows:


IOC$GL_ADPLIST
DEC 3000 Model 500 ADP List
System ADP
TR = 1
type = AT$_KA0402
adp$ps_child_adp
Turbo ADP
TR = 2
type = AT$_TC
adp$l_crab
CRAB
Used to manage
Scatter/Gather entries.
Note: the TurboSCSI
ADP and the Core ADP
also contain pointers
to the CRAB.
Turbo
Bus Array
adp$ps_child_adp
header
9
entries
Entry 0-5 user
TC slots
Entry 6 Integrated
SCSI adapter
Entry 7 CoreIO
Subsystem
Entry 8 Integrated
Graphics
TurboSCSI ADP
Core ADP
adp$ps_peer_adp
TR=4
type=
AT$_COREIO
TR = 3
type=AT$_TURBO_
SCSI
TurboSCSI
Bus Array
Core
Bus Array
header
2
entries
SCSI Port A
SCSI Port B
header
4
entries
Lance
Serial Line 0
Serial Line 1
ISDN

After setting up the data structures, INI$IOMAP tests each Turbochannel slot for the presence of an option module. For each slot, INI$IOMAP tests the option ROM base address. If the option responds, INI$IOMAP reads the test pattern locations in the ROM to verify that the ROM is valid. If the ROM is valid, the corresponding bus array entry for the Turbochannel slot is initialized as follows:

- BUSARRAY$Q_HW_ID -- the Module Name string (8 bytes) is read from the ROM and stored in this field.
- BUSARRAY$Q_CSR -- the slot base address (sparse space) is stored in this field.
- BUSARRAY$L_NODE_NUMBER -- the Turbochannel slot number of the option is stored in this field.
- BUSARRAY$L_AUTOCONFIG -- the slot interrupt vector is stored in this field.
- BUSARRAY$Q_BUS_SPECIFIC -- the slot base address (dense space) is stored in this field.

6.2.7 Configuring a Device on DEC 3000 Model 500/Turbochannel

The SYSMAN utility is used to manually configure devices. In order to configure a Turbochannel option on DEC 3000 Model 500, the first step is to issue the SYSMAN IO SHOW BUS command. This command uses the ADP list (pictured above) to display information on all of the I/O options that are present in the system.

The information in the SYSMAN IO SHOW BUS display is most of the information that is necessary to configure a device.

The SYSMAN IO CONNECT command is used to actually configure the device. On DEC 3000 Model 500, this command should be issued as follows:

SYSMAN>IO CONNECT devname /adapter=x /csr=y /vector=z /node=w /driver=yourdriver.exe

The "devname" parameter should be specified in standard device naming format -- a 2 letter device code, controller letter, and unit number (such as XXA0).

The "adapter" parameter should be specified as the TR number of the Turbochannel ADP. This is part of the SYSMAN IO SHOW BUS display.

The "csr" parameter is the base physical address of the Turbochannel option slot. The SYSMAN IO SHOW BUS display shows the sparse space slot base address. The value specified in the "csr" parameter is copied directly to the IDB$Q_CSR field in the IDB by the SYS$LOAD_DRIVER program.

The "vector" parameter specifies the Turbochannel slot interrupt vector. This value is not part of the SYSMAN IO SHOW BUS display. For DEC 3000 Model 500 Turbochannel, the vector should be specified as:

(Turbochannel slot number * 4)

The "node" parameter specifies the Turbochannel slot number. The node number is part of the SYSMAN IO SHOW BUS display. The specified value is copied to the CRB$L_NODE field in the CRB by the SYS$LOAD_DRIVER program. This parameter must be specified in order for routines IOC$NODE_FUNCTION and IOC$NODE_DATA to work correctly.

The "drivername" parameter is the file name of the driver.

6.2.8 IOC$NODE_DATA

IOC$NODE_DATA is a system specific routine that returns bus-slot specific information to the caller. On DEC 3000 Model 500 Turbochannel, IOC$NODE_DATA can be used by a driver to obtain the Turbochannel slot sparse or dense space base physical address.

IOC$NODE_DATA requires a CRB address, a function code, and a pointer to a buffer as input arguments. The CRB$L_NODE field must contain the Turbochannel slot number of the Turbochannel option (the driver must be loaded using the /node qualifer). It is up to the caller to supply the address of a buffer large enough for the requested information. On DEC 3000 Model 500/Turbochannel, a quadword buffer is required to contain a Turbochannel slot physical address.

The function codes available for DEC 3000 Model 500/Turbochannel are defined in [LIB.LIS]IOCDEF.SDL and are listed below:

ioc$k_turbo_slot_sparse_pa	returns sparse space slot base address
ioc$k_turbo_slot_dense_pa	returns dense space slot base address

An example call to IOC$NODE_DATA is shown below:

```
status = ioc$node_data (crb_address,
                        ioc$k_turbo_slot_sparse_pa,
                        address_of_buffer);
```

IOC$NODE_DATA reads the Turbochannel slot number from the CRB$L_NODE field of the CRB and returns the sparse space slot physical address in the caller's buffer.

6.3 Turbochannel on DEC 3000 Model 400

The DEC 3000 Model 400 platform is based on the DEC 3000 Model 500 platform. The only differences are that DEC 3000 Model 400 implements 3 Turbochannel option slots and does not contain an integrated graphics controller.

The only difference in Turbochannel bus support on DEC 3000 Model 500 and DEC 3000 Model 400 is the Turbochannel address map. The DEC 3000 Model 400/Turbochannel address map is shown below.

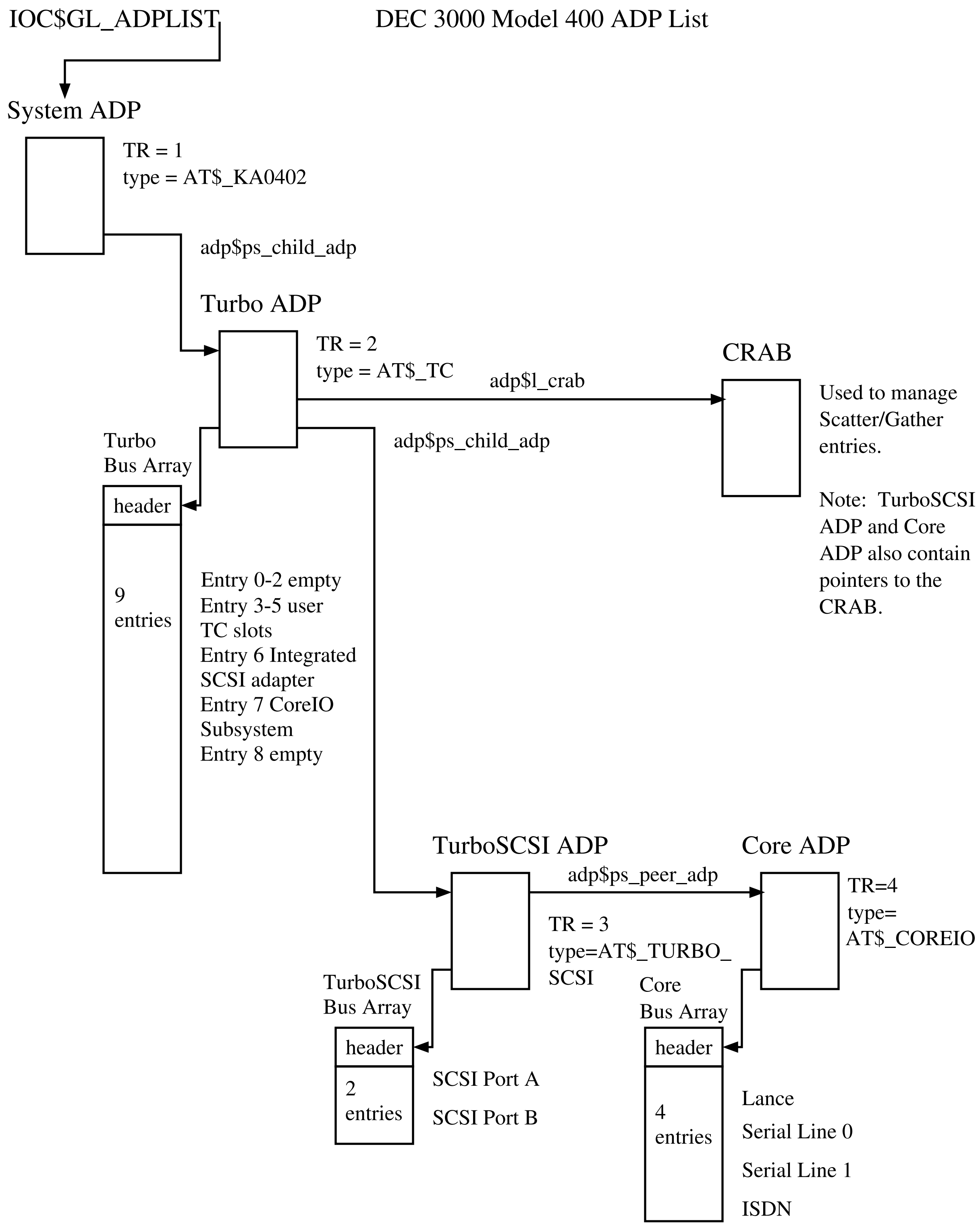
DEC 3000 Model 400 Turbochannel Address Map

Slot	Base Physical Address	Size	Space
3	1 6000 0000	128 MB	Slot 3 Dense space
	1 6800 0000	128 MB	Reserved
	1 7000 0000	256 MB	Slot 3 Sparse space
4	1 8000 0000	128 MB	Slot 4 Dense space
	1 8800 0000	128 MB	Reserved
	1 9000 0000	256 MB	Slot 4 Sparse space
5	1 A000 0000	128 MB	Slot 5 Dense space
	1 A800 0000	128 MB	Reserved
	1 B000 0000	256 MB	Slot 5 Sparse space

DEC 3000 Model 400 implements the same integral options as DEC 3000 Model 500, except that there is no integrated graphics controller. Thus, the address map also contains an integrated SCSI adapter (Turbochannel slot 6) and the Core I/O subsystem (Turbochannel slot 7).

OpenVMS AXP uses the same platform specific image to support both DEC 3000 Model 500 and DEC 3000 Model 400. Therefore, the 3 Turbochannel option slots available on DEC 3000 Model 400 are labeled 3, 4, and 5 by OpenVMS AXP, since their option slot addresses correspond exactly to DEC 3000 Model 500/Turbochannel slots 3, 4, and 5. [LIB.LIS}IO0402DEF.SDL defines symbols for DEC 3000 Model 500 and DEC 3000 Model 400 physical addresses. Note that the DEC 3000 Model 400 console uses a different slot labeling scheme.

The ADP list looks exactly the same on DEC 3000 Model 400 as on DEC 3000 Model 500, except that entries 0, 1, 2, and 8 will always be empty on DEC 3000 Model 400. The DEC 3000 Model 400 ADP list is shown below:


IOC$GL_ADPLIST
DEC 3000 Model 400 ADP List
System ADP
TR = 1
type = AT$_KA0402
adp$ps_child_adp
Turbo ADP
TR = 2
type = AT$_TC
adp$l_crab
CRAB
Used to manage Scatter/Gather entries.
Note: TurboSCSI ADP and Core ADP also contain pointers to the CRAB.
adp$ps_child_adp
Turbo Bus Array
header
9 entries
Entry 0-2 empty
Entry 3-5 user TC slots
Entry 6 Integrated SCSI adapter
Entry 7 CoreIO Subsystem
Entry 8 empty
TurboSCSI ADP
Core ADP
adp$ps_peer_adp
TR=4
type=
AT$_COREIO
TR = 3
type=AT$_TURBO_
SCSI
TurboSCSI Bus Array
header
2 entries
SCSI Port A
SCSI Port B
Core Bus Array
header
4 entries
Lance
Serial Line 0
Serial Line 1
ISDN

Dense and sparse space addressing, register access, DMA and Scatter/Gather map management, bus interface registers, and device configuration are all treated exactly the same on DEC 3000 Model 400 as on DEC 3000 Model 500. IOC$NODE_DATA and IOC$NODE_FUNCTION work identically on DEC 3000 Model 400 as on DEC 3000 Model 500.

6.4 Turbochannel on DEC 3000 Model 300

The Turbochannel on DEC 3000 Model 300 has some differences from the DEC 3000 Model 500/DEC 3000 Model 400 implementation. There are only two option slots, and the address map is different. The Integrated SCSI adapter offers only a single SCSI port. However, the most significant difference is the lack of a Scatter/Gather map on DEC 3000 Model 300. All Turbochannel DMA on DEC 3000 Model 300 must be physical DMA. The DMA burst size is limited to 64 longwords (compared to 128 longwords on DEC 3000 Model 500 and DEC 3000 Model 400). Also, DEC 3000 Model 300 does not support Turbochannel parity. Option modules that support parity checking should disable this feature on DEC 3000 Model 300.

The details of register access and driver loading are the same on DEC 3000 Model 300 as on DEC 3000 Model 500 and DEC 3000 Model 400.

DEC 3000 Model 300/Turbochannel differences are covered in the following sections.

6.4.1 DEC 3000 Model 300/Turbochannal Address Map

The DEC 3000 Model 300/Turbochannel address map is shown below:

DEC 3000 Model 300 Turbochannel Address Map

Slot	Base Physical Address	Size	Space
0	1 0000 0000	32 MB	Slot 0 Dense space
	1 0200 0000	224 MB	Reserved
	1 1000 0000	64 MB	Slot 0 Sparse space
1	1 2000 0000	32 MB	Slot 1 Dense space
	1 2200 0000	224 MB	Reserved
	1 3000 0000	256 MB	Slot 1 Sparse space

On DEC 3000 Model 300, each of the two option slots has 32 MB dense and 64 MB sparse space, as compared to 128 MB dense and 256 MB sparse on DEC 3000 Model 500 and DEC 3000 Model 400. [LIB.LIS]IO0702DEF.SDL defines symbols for DEC 3000 Model 300 physical addresses.

DEC 3000 Model 300 has the same integral options as DEC 3000 Model 500, but in different slots. On DEC 3000 Model 300, the integrated SCSI adapter is in slot 4, the Core I/O subsystem is in slot 5, and the integrated graphics controller is in slot 6.

6.4.2 Turbochannel Interrupts on DEC 3000 Model 300

DEC 3000 Model 300 does not implement the IMASK register for enabling and disabling interrupts from a Turbochannel option slot. The option slot interrupt enable bits are implemented in the SIR and SIMR registers, which are Core I/O subsystem interface registers.

6.4.3 IOC$NODE_FUNCTION on DEC 3000 Model 300

Routine IOC$NODE_FUNCTION should be used to enable option slot interrupts on DEC 3000 Model 300. See the DEC 3000 Model 500 description of IOC$NODE_FUNCTION for an explanation of how to call this routine.

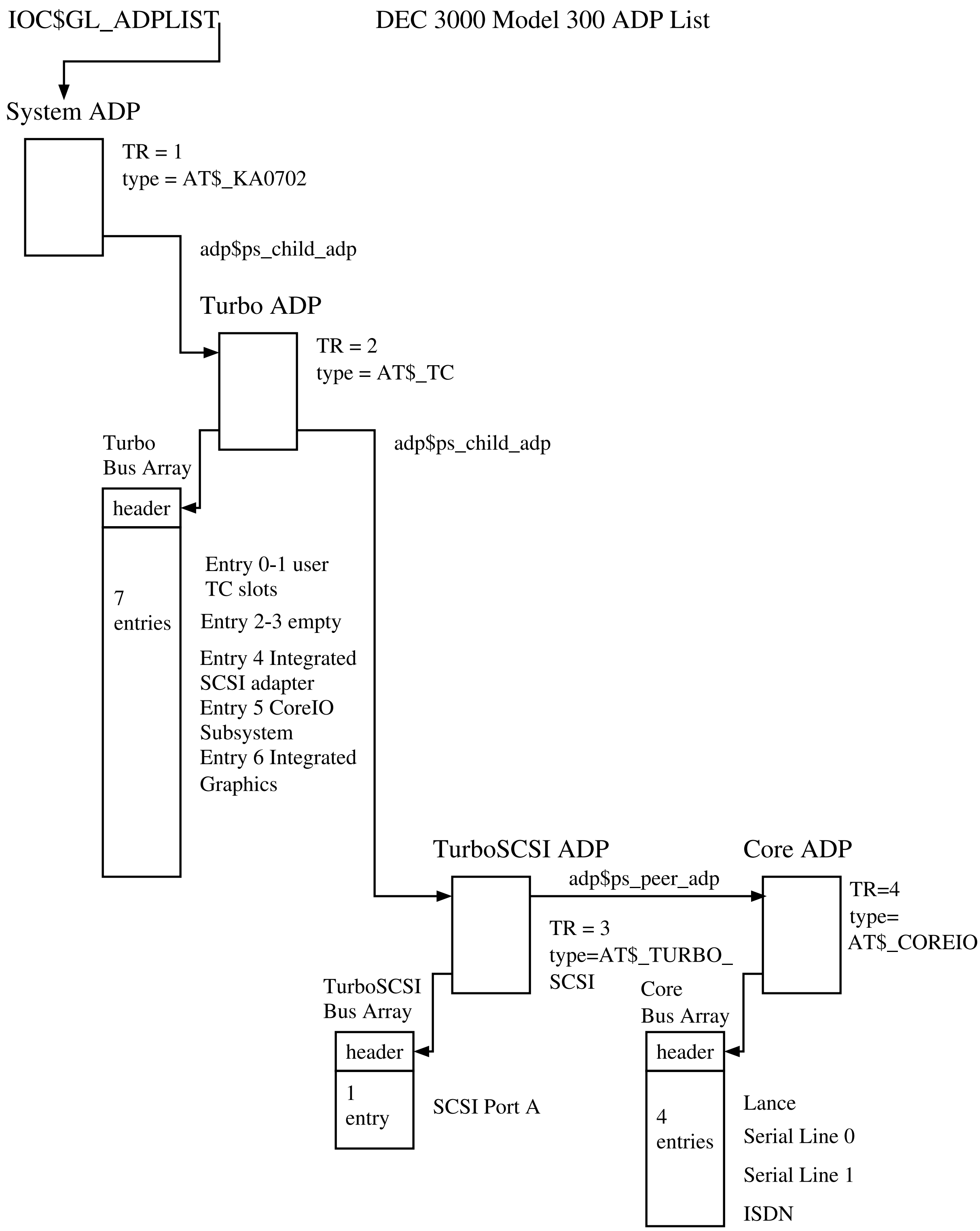
The only IOC$NODE_FUNCTION function codes supported on DEC 3000 Model 300 are ioc$k_enable_intr and ioc$k_disable_intr.

6.4.4 IOC$NODE_DATA on DEC 3000 Model 300

IOC$NODE_DATA on DEC 3000 Model 300 works exactly the same as IOC$NODE_DATA on DEC 3000 Model 500. It can be used to obtain the Turbochannel dense or sparse space base physical addresses for a Turbochannel slot.

6.4.5 DEC 3000 Model 300/Turbochannel I/O Map

INI$IOMAP creates an ADP list for DEC 3000 Model 300 as follows:


IOC$GL_ADPLIST
DEC 3000 Model 300 ADP List
System ADP
TR = 1
type = AT$_KA0702
adp$ps_child_adp
Turbo ADP
TR = 2
type = AT$_TC
Turbo
Bus Array
adp$ps_child_adp
header
7
entries
Entry 0-1 user
TC slots
Entry 2-3 empty
Entry 4 Integrated
SCSI adapter
Entry 5 CoreIO
Subsystem
Entry 6 Integrated
Graphics
TurboSCSI ADP
Core ADP
adp$ps_peer_adp
TR=4
type=
AT$_COREIO
TR = 3
type=AT$_TURBO_
SCSI
TurboSCSI
Bus Array
Core
Bus Array
header
1
entry
SCSI Port A
header
4
entries
Lance
Serial Line 0
Serial Line 1
ISDN

7 Futurebus+ Support in OpenVMS AXP

This section discusses base Futurebus+ support in the OpenVMS AXP operating system. The support described in this chapter will be available in an upcoming release of the operating system. First a general description of Futurebus+ concepts is given, followed by a description of Futurebus+ support in the operating system and a description of Futurebus+ support on the DEC 4000 and DEC 10000/7000 platforms.

7.1 Futurebus+ Overview

Futurebus+ is an industry standard bus defined by IEEE standards 896.1 (Logical Layer), 896.2 (Physical Layer and Profiles), 896.3 (Recommended Practices), and 1212 (CSR architecture). Digital AXP platforms implement Profile B, which is intended as a general purpose I/O bus.

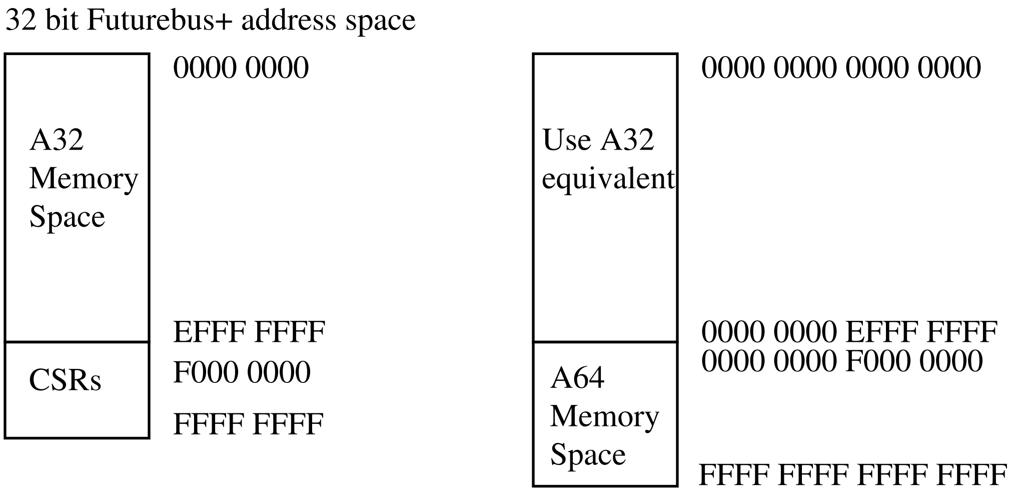
Futurebus+ Profile B is required to support 32 bit addressing (A32) and optionally supports 64 bit addressing (A64). Data widths from 32 to 256 bits are supported, though only 32 bit support (D32) is required. The bus uses an asynchronous protocol, so that the achievable bandwidth depends on the components used to build the modules. Current implementations (DEC 4000 and DEC 10000/7000) have demonstrated bandwidths on the order of 150 MB/second.

The Futurebus+ specifications describe a module as occupying a physical backplane slot. A module can implement one or two nodes. A maximum of 62 nodes per bus is supported. The physical backplane slot number of a module determines the base address of Futurebus+ node space for the nodes on the module.

Futurebus+ is currently available on the DEC 10000/7000 and DEC 4000 platforms.

7.2 Futurebus+ Address Space

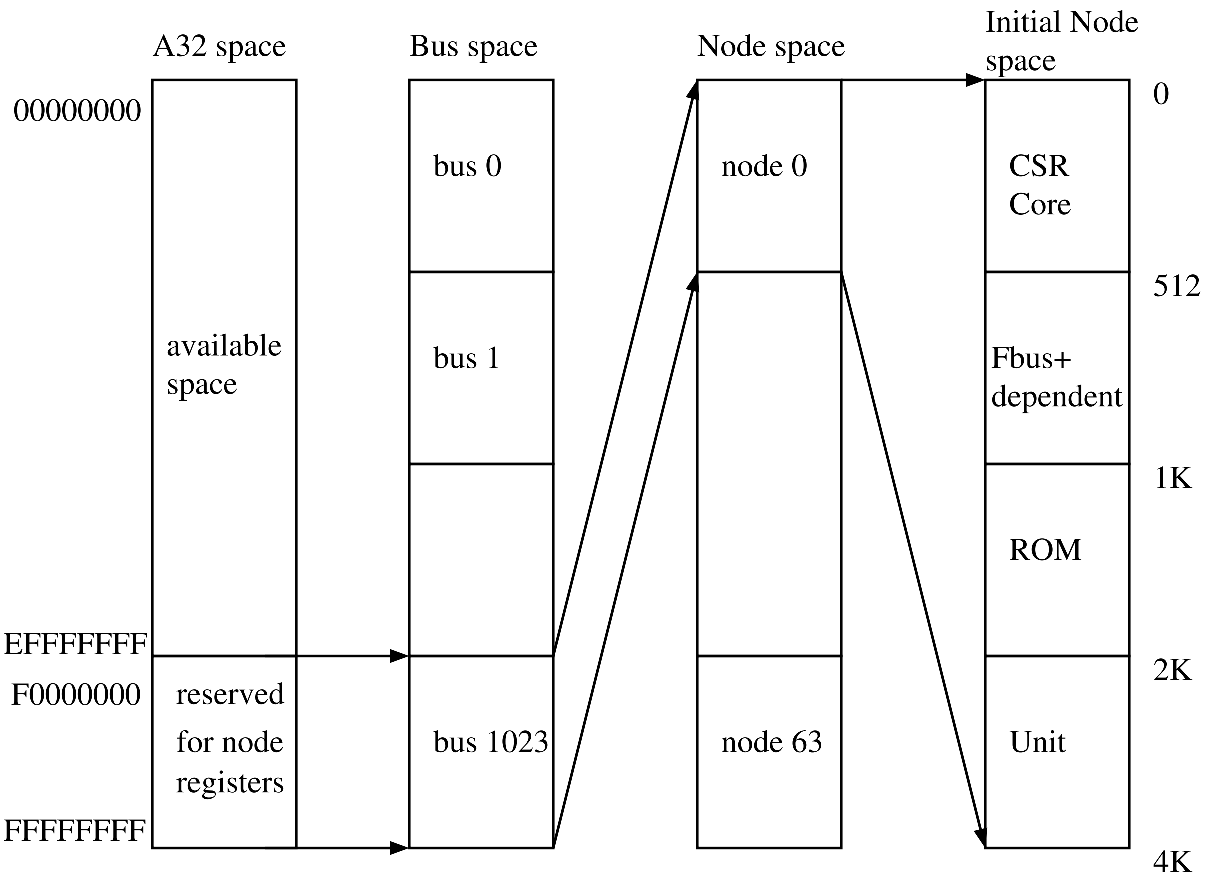
The Futurebus+ supports 2 distinct address spaces, 32 bit address space (A32) and 64 bit address space (A64). Since all nodes are required to support A32, all A64 transactions to addresses with 32 bit equivalents should use A32 to insure interoperability (this is explained in detail in 896.1). 32 bit and 64 bit Futurebus+ address space is pictured below:



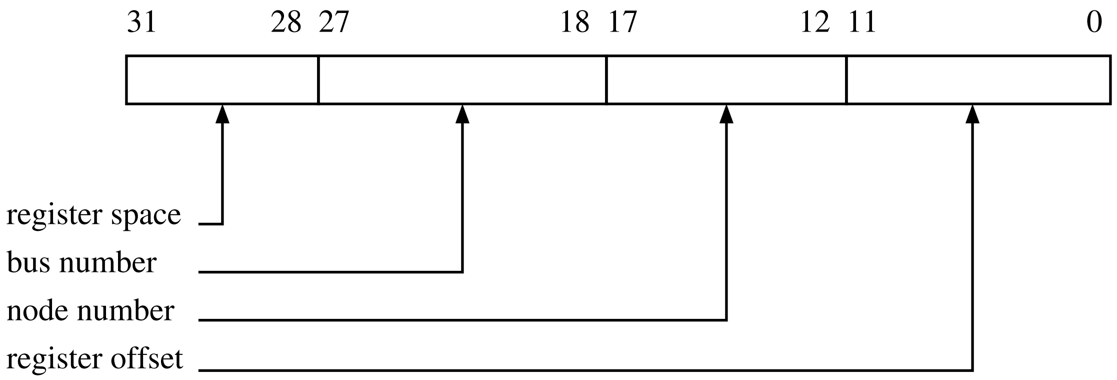
7.3 Futurebus+ CSR Addressing

Futurebus+ requires that all nodes support A32 addressing. While referencing CSRs only 32 bit addressing should be used. Futurebus+ A32 space and the format of an A32 CSR address are shown in the following diagram.

A32 CSR Addressing



A32 CSR address



Futurebus+ node 0 Initial Node Space base address = FFFC0000

Futurebus+ A32 register space is accessed when bits 31:28 of the address are 1111 (binary). This uses up 1/16 of the available 32 bit address space for register space. Futurebus+ A32 addressing allows for bus numbers, such that 1023 separate buses, each with 63 nodes, can exist in the same address space.

Futurebus+ reserves node 63 as a "broadcast space" address, so Futurebus+ systems are limited to 62 nodes on a bus.

Configuration software is expected to number all of the buses that may be present in a system. Nodes on the same Futurebus+ can access each other's register space using the actual bus number (from the NODE_IDS register) or bus number 1023, which is always interpreted as the local bus. OpenVMS AXP uses bus number 1023 for all accesses to Futurebus+ address space.

Modules get the 5 most significant bits of their 6 bit node address from the Geographical Address lines GA<4:0>, which are hardwired in each backplane slot. The least significant bit of the node address determines which node on a module is addressed (0 or 1). Modules which only implement a single node are supposed to use node 0 addresses.

As shown in the above diagram, each node is assigned a 4KB block of initial node space for CSRs. The 4KB space is broken up into 4 regions--CSR Core, Futurebus+ Dependent, ROM, and Initial Units Space.

The CSR Core space is defined by IEEE 1212 (CSR Architecture). Futurebus+ Dependent space is defined by IEEE 896.2 (Physical Layer and Profile). ROM space is defined by both IEEE 1212 and 896.2. Initial Units Space is vendor defined.

[LIB.LIS]FBUSDEF.SDL defines symbolic register offsets for the 4KB initial node space.

7.4 CSR Data Format

CSR register definitions for the CSR Core, Futurebus+ Dependent, and ROM area are given in IEEE 1212 and 896.2 in "big-endian" data format. Big-endian register data format is specified as follows:

Big-endian register data format

7 0	15 8	23 16	31 24
byte 0	byte 1	byte 2	byte 3
Most Significant Byte			Least Significant Byte

Little-endian register format is defined as follows:

Little endian register data format

31 24	23 16	15 8	7 0
byte 3	byte 2	byte 1	byte 0
Most Significant Byte			Least Significant Byte

When a big-endian register is driven onto the Futurebus+, byte 0 is driven on Futurebus+ AD<7:0>, byte 1 is driven on AD<15:8>, byte 2 is driven on AD<23:16>, and byte 3 is driven on AD<31:24>. FBUSDEF.SDL defines CSR Core, Futurebus+ Dependent, and ROM registers in little-endian format, thus, byte lane swapping must be performed after reading and before writing registers in the CSR Core, Futurebus+ Dependent, and ROM areas in order to make the register data match the FBUSDEF.SDL definitions. In general a driver has no need to access registers in the CSR Core, Futurebus+ Dependent, or ROM areas. These areas are primarily used for initial node setup and are only accessed during booting and system configuration.

Initial Unit Space is vendor-defined, so registers in Initial Unit Space may be either big or little endian. Digital adapters generally use little-endian register definitions for Initial Unit Space registers.

7.5 Futurebus+ Register Access

On both the DEC 10000/7000 and DEC 4000 platforms, access to Futurebus+ CSR space is accomplished through hardware I/O mailboxes. A driver uses the standard register access routines provided by OpenVMS/AXP to accomplish a register access.

7.5.1 Allocating CRAMs for Futurebus+ Register Access

There is nothing that is specific to Futurebus+ in the area of CRAM allocation. As mentioned previously, CRAM allocation can be done automatically by the driver loading program by specifying the desired number of CRAMs in DPT$W_IDB_CRAMS or DPT$W_UCB_CRAMS, or the driver can directly call IOC$ALLOCATE_CRAM.

7.5.2 Initializing CRAMS

As with all mailbox register accesses, IOC$CRAM_CMD is used to initialize the COMMAND, MASK, and RBADR fields of the CRAM. The initialization of these fields is done in a bus-specific manner. For the Futurebus+, the COMMAND field is initialized (based on the command index) with bit patterns that are directly driven onto the Futurebus+ CM<7:0> wires during a Futurebus+ command cycle. The MASK bits become Futurebus+ byte enable signals at the appropriate time during a Futurebus+ cycle. The RBADR field is initialized with the target address of the Futurebus+ CSR.

The following table shows which command indices are supported on Futurebus+, and the COMMAND field bit patterns that are generated for each command index.

Command Index	Mailbox Command Field Contents (bits 7:0)	Description
cramcmd$k_rdquad32	0 0 1 0 0 0 0 0	read, unlocked, aw=32, dw=64
cramcmd$k_rdlong32	0 0 0 0 0 0 0 0	read, unlocked, aw=32, dw=32
cramcmd$k_rdword32	0 0 0 0 0 0 1 0	read, partial, aw=32, dw=32
cramcmd$k_rdbyte32	0 0 0 0 0 0 1 0	read, partial, aw=32, dw=32
cramcmd$k_wtquad32	0 0 1 1 0 0 0 0	write, unlocked, aw=32, dw=64
cramcmd$k_wtlong32	0 0 0 1 0 0 0 0	write, unlocked, aw=32, dw=32
cramcmd$k_wtword32	0 0 0 1 0 0 1 0	write, partial, aw=32, dw=32
cramcmd$k_wtbyte32	0 0 0 1 0 0 1 0	write, partial, aw=32, dw=32
cramcmd$k_rdquad64	1 0 1 0 0 0 0 0	read, unlocked, aw=64, dw=64
cramcmd$k_rdlong64	1 0 0 0 0 0 0 0	read, unlocked, aw=64, dw=32
cramcmd$k_rdword64	1 0 0 0 0 0 1 0	read, partial, aw=64, dw=32
cramcmd$k_rdbyte64	1 0 0 0 0 0 1 0	read, partial, aw=64, dw=32
cramcmd$k_wtquad64	1 0 1 1 0 0 0 0	write, unlocked, aw=64, dw=64
cramcmd$k_wtlong64	1 0 0 1 0 0 0 0	write, unlocked, aw=64, dw=32
cramcmd$k_wtword64	1 0 0 1 0 0 1 0	write, partial, aw=64, dw=32
cramcmd$k_wtbyte64	1 0 0 1 0 0 1 0	write, partial, aw=64, dw=32

An example call to IOC$CRAM_CMD is shown below:

```
status = ioc$cram_cmd(cramcmd$k_rdlong32,  /* command index    */
                      4,                   /* byte offset      */
                      adp_address,
                      cram_address);
```

The COMMAND field bit patterns are stored in a table pointed to by field ADP$PS_COMMAND_TBL in the Fbus ADP. IOC$CRAM_CMD uses the command index to find the proper COMMAND field bit pattern, and copies the command field bits to CRAM$L_COMMAND.

RBADR is calculated from the byte offset input parameter and the IDB$Q_CSR field in the IDB. The byte offset is added to the value in IDB$Q_CSR, and the result is copied to CRAM$Q_RBADR. No further alignment is applied to RBADR.

The MASK field is calculated based on the command index and the byte offset. If the caller specifies a byte or word length read or write, the MASK bits are set based on the transfer size (byte or word) and the first byte involved in the transfer.

7.5.3 Issuing the Futurebus+ Register Access

There is nothing specific to the Futurebus+ in issuing the CRAM. A driver calls IOC$CRAM_IO, as follows:

status = ioc$cram_io (cram_address);

On Futurebus+ platforms, IOC$CRAM_IO performs the entire Alpha I/O mailbox operation, including queueing the mailbox and waiting for the DONE bit.

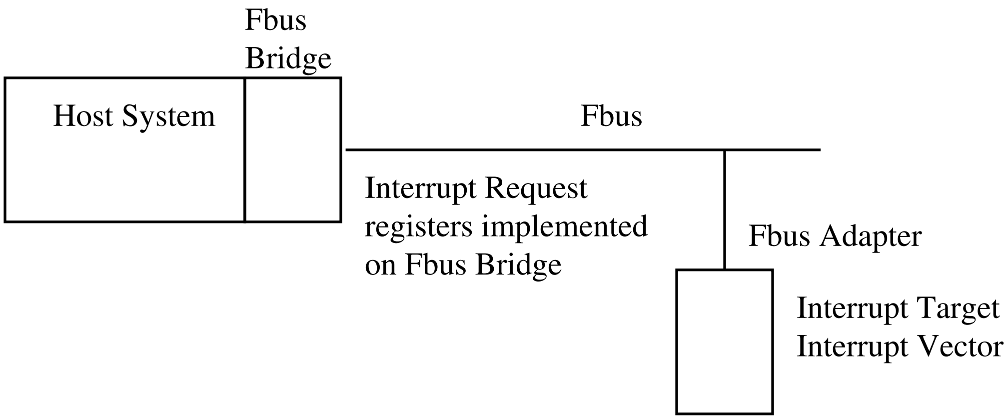
A driver may also call IOC$CRAM_QUEUE, which queues the hardware mailbox and returns to the caller without waiting for the DONE bit to be set. If a driver uses IOC$CRAM_QUEUE, the driver should call IOC$CRAM_WAIT to check that the DONE bit is set (indicating the previous operation has completed) before re-using the CRAM for another register access.

7.6 DMA

On the DEC 4000 and DEC 10000/7000 platforms, Futurebus+ adapters access system memory through the Futurebus bridge. The bridge accepts all non-register space Futurebus+ addresses, and passes them directly to the system memory. There are no map registers in either DEC 4000 or DEC 10000/7000, hence all DMA is "physical" DMA.

7.7 Futurebus+ Interrupts

A Futurebus+ adapter interrupts the host by writing a 32 bit quantity (an interrupt vector) to an Interrupt Request register address. The Interrupt Request register address is not specified by the Futurebus+ specifications--it is left up to system implementors to define the interrupt mechanisms for a particular system. Digital Futurebus+ bridges implement 4 Interrupt Request register addresses at offsets 800, 804, 808, and 80C (hex) in the bridge’s Initial Unit Space. Futurebus+ adapters generally implement an Interrupt Target and an Interrupt Vector register in the adapter Initial Unit Space. During driver initialization, the driver writes the adapter Interrupt Target register with the address of one of the bridge’s Interrupt Request registers, and writes the adapter Interrupt Vector register with the assigned interrupt vector (the adapter interrupt vector is assigned when the driver is loaded). The Futurebus+ adapter interrupts the host by writing the value in its Interrupt Vector register to the address stored in its Interrupt Target register. This is pictured in the following diagram:



Interrupt Target programmed with Interrupt Request register address during driver initialization.
Interrupt Vector programmed with vector during driver initialization.
Fbus Adapter interrupts host by writing Interrupt Vector contents to Interrupt Request register on bridge.

The Fbus Bridge supplies different Interrupt Request register addresses to allow Futurebus+ adapters to interrupt the host at different Interrupt Priority Levels. On the DEC 10000/7000 and DEC 4000 platforms, the Interrupt Request register addresses are used as follows:

Interrupt Register offset	DEC 10000/7000	DEC 4000
800	IPL 14 (hex) interrupt	IPL 14 (hex) interrupt
804	IPL 15 (hex) interrupt	IPL 14 (hex) interrupt
808	IPL 16 (hex) interrupt	IPL 14 (hex) interrupt
80C	IPL 17 (hex) interrupt	IPL 14 (hex) interrupt

The actual address of the bridge Interrupt Register depends on the Futurebus+ node number of the bridge. The bridge may reside in different Futurebus+ slots on different platforms, so OpenVMS AXP uses system routine IOC$NODE_DATA to return the base Interrupt Register address to a driver. An example call to IOC$NODE_DATA to get the base Interrupt Register address is shown below:

```
status = ioc$node_data (crb_address,
                        ioc$k_fbus_int_loc,
                        addr_of_longword);
```

IOC$NODE_DATA finds the bridge Futurebus+ slot number and returns

```
bridge_nodespace_base + 800
```

in the caller's buffer. If the bridge is installed in Futurebus+ slot 0, for example, IOC$NODE_DATA would return FFFC0800 to the caller. The driver should program this value (if interrupt priority level 14H is desired) to the Interrupt Target register on the driver's Futurebus+ adapter.

7.8 Futurebus+ System Routines

The Initial Node Space base address of a Futurebus+ adapter is determined by the backplane slot into which the adapter is installed. For most adapters, device control functions can be accomplished through registers located in the adapter Initial Unit Space. However, some adapters require additional Futurebus+ address space for their operation. An example of such an adapter may be a bus bridge, which may require a large chunk of Futurebus+ address space to serve as a window to a remote bus.

OpenVMS/AXP provides two routines to manage the allocation of Futurebus+ address space. These routines are IOC$RESERVE_FBUS_A32, which manages Futurebus+ A32 space, and IOC$RESERVE_FBUS_A64, which manages Futurebus+ A64 space. A driver which requires Futurebus+ address space in addition to its Initial Node Space may call one of these routines to reserve a region of Futurebus+ address space. The operation of these routines is explained below.

7.8.1 IOC$RESERVE_FBUS_A32

IOC$RESERVE_FBUS_A32 may be called by a driver that requires a region of Futurebus+ A32 space. A bus bridge, which requires address space for a window to a remote bus, is an example of an adapter that may require Futurebus+ address space in addition to its registers. The minimum granularity of A32 space address allocation is 16 MB (that is, the smallest possible request for A32 address space is for a 16 MB region). The reason for this granularity is to reduce the management overhead in keeping track of the allocated space. An example call is given below:

```
status = ioc$reserve_fbus_a32 (fbus_adp,
                               request_amt,
                               buffer_addr);
```

Inputs

fbus_adp : address of Futurebus+ ADP
request_amt : Amount of requested A32 space in 16 MB chunks. For example, a request of 1 is a request for a single 16 MB region. A request of 2 is a request for a contiguous 32 MB region.
buffer_addr : Address of user buffer. The user buffer must be at least a longword in size. On success, the starting address of the

reserved A32 space is written to the user buffer. The starting address is guaranteed to have 16 MB alignment.

Outputs

ss$_normal : Success. Starting A32 address written to caller's buffer.
ss$_badparam : Bad argument.
ss$_insfmem : Not enough Futurebus+ A32 space available to satisfy request.

IOC$RESERVE_FBUS_A32 acquires the Fbus ADP spinlock (IPL 23). Therefore callers of this routine must be executing at or below device IPL.

7.8.2 IOC$RESERVE_FBUS_A64

IOC$RESERVE_FBUS_A64 is similar in operation to IOC$RESERVE_FBUS_A32. The minimum granularity of A64 space address allocation is 4 GB (that is, the smallest possible request for A64 address space is for a 4 GB region). The reason for this granularity is to reduce the management overhead in keeping track of the allocated space. An example call is given below:

```
status = ioc$reserve_fbus_a64 (fbus_adp,
                               request_amt,
                               buffer_addr);
```

Inputs

fbus_adp : address of Futurebus+ ADP
request_amt : Amount of requested A64 space in 4 GB chunks. For example, a request of 1 is a request for a single 4 GB region. A request of 2 is a request for a contiguous 8 GB region.
buffer_addr : Address of user buffer. The user buffer must be at least a quadword in size. On success, the starting address of the reserved A64 space is written to the user buffer. The starting address is guaranteed to have 4 GB alignment.

Outputs

ss$_normal : Success. Starting A64 address written to caller's buffer.
ss$_badparam : Bad argument.
ss$_insfmem : Not enough Futurebus+ A64 space available to satisfy request.

IOC$RESERVE_FBUS_A64 acquires the Fbus ADP spinlock (IPL 23). Therefore callers of this routine must be executing at or below device IPL.

7.9 Configuring a Futurebus+ Adapter

Manual configuration of adapters is very similar across platforms and buses. There are some bus-specific differences, usually in the format of the base CSR address and in the specification of the interrupt vector(s).

The SYSMAN utility is used to manually configure adapters. In order to manually configure a Futurebus+ adapter, the first step is to issue the SYSMAN IO SHOW BUS command. This command uses the ADP list (pictured in the DEC 4000 and DEC 10000/7000 chapter) to display information on all of the I/O options that are present in the system.

The SYSMAN IO CONNECT command is used to actually configure an adapter. For a Futurebus+ adapter, this command should be issued as follows:

SYSMAN> IO CONNECT devname /adapter=x /csr=y /node=w /vec=z /driver=yourdriver.exe

The "devname" parameter should be specified in the standard device naming format--a 2 letter device code, controller letter, and unit number (such as XXA0).

The "adapter" parameter should be specified as the TR number of the Fbus ADP. This is part of the SYSMAN IO SHOW BUS display.

The "csr" parameter is the base of Futurebus+ Initial Node Space for the adapter. This is an A32 Futurebus+ address in Futurebus+ CSR space. The value specified in the "csr" parameter is copied directly to the IDB$Q_CSR field in the IDB by the SYS$LOAD_DRIVER program.

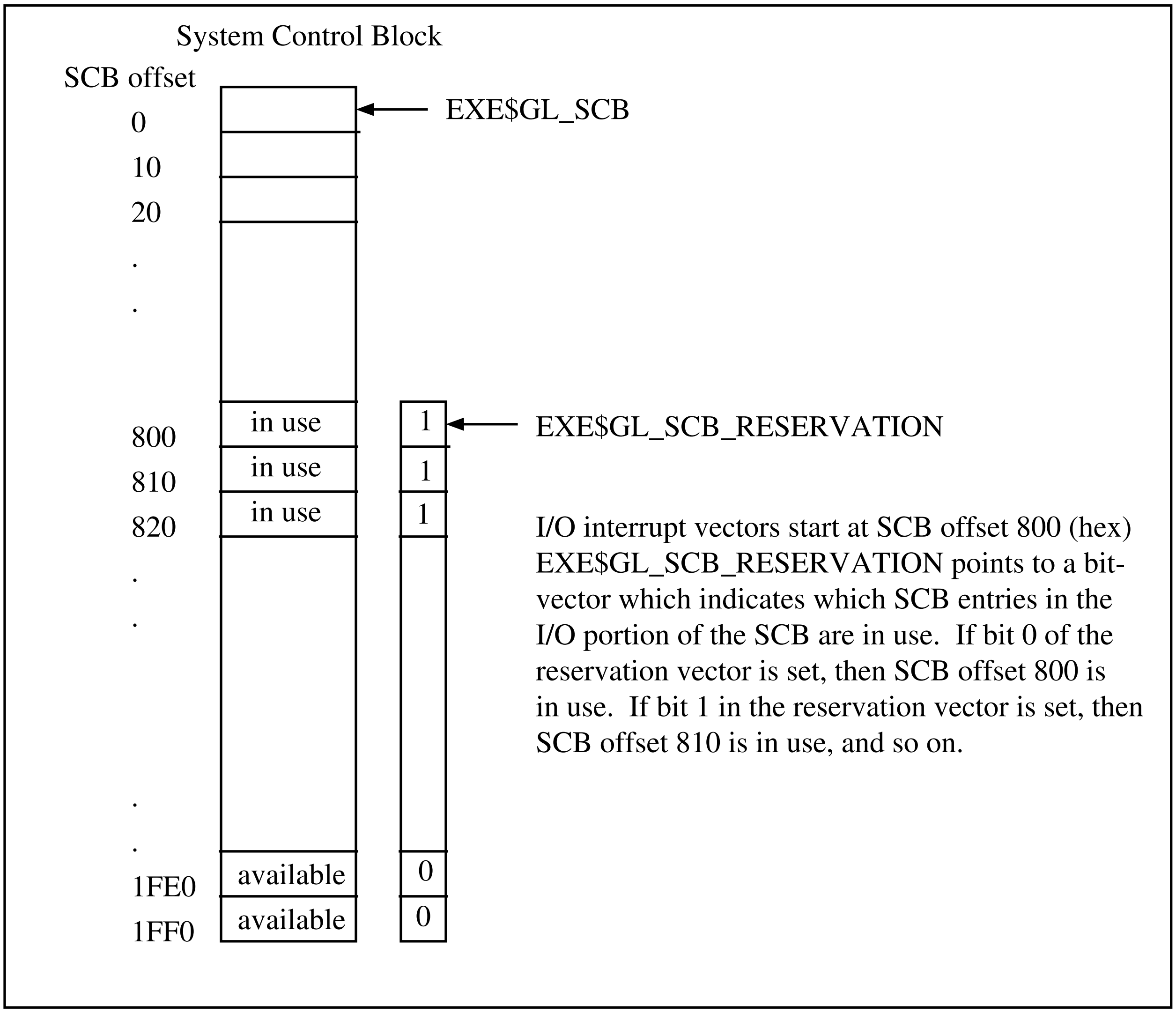
The "/node" parameter specifies the Futurebus+ node number of the adapter. The node number is part of the SYSMAN IO SHOW BUS display. The "/node" parameter is copied to the CRB$L_NODE field in the CRB by the SYS$LOAD_DRIVER program.

The "/driver" parameter is the file name of the driver.

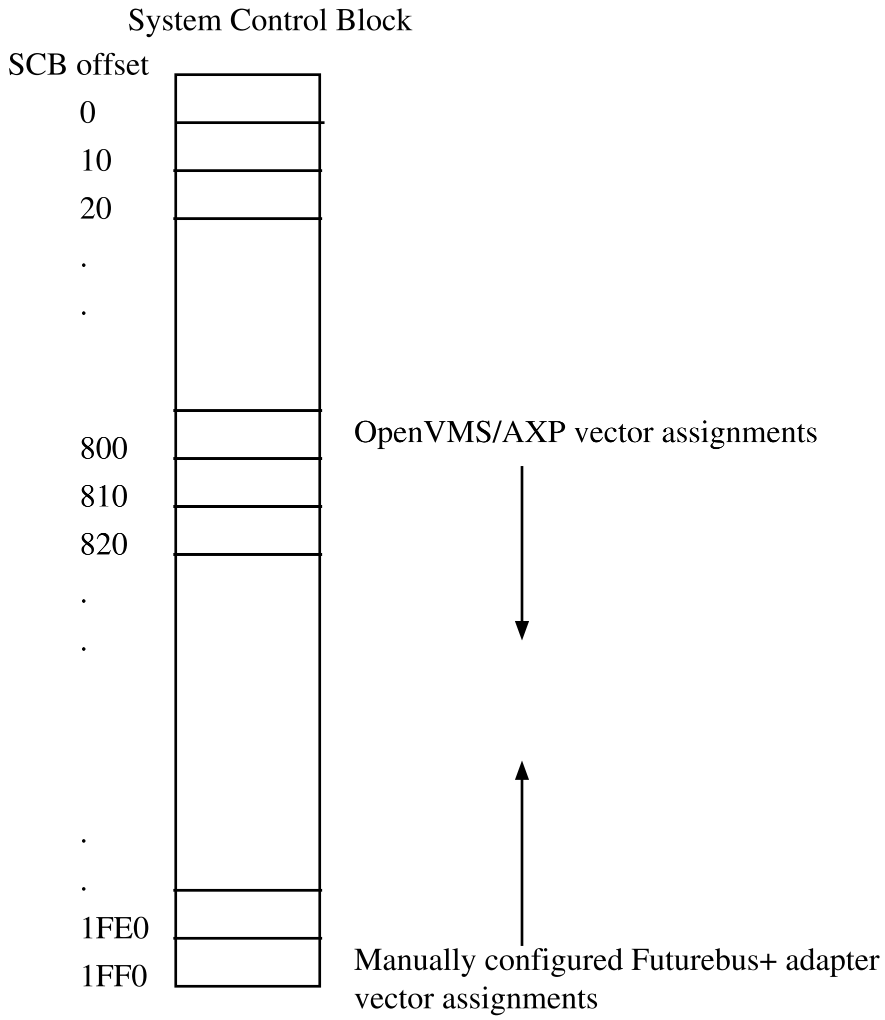
The "vec" parameter specifies the interrupt vector to be used by this Futurebus+ adapter. If the adapter and driver use multiple interrupt vectors, they must be explicitly listed as follows:

/vec = (vec1, vec2, vec3, ...)

You must be careful when specifying interrupt vectors not to use one that is already in use by another driver/adapter. OpenVMS/AXP maintains a data structure which keeps track of which SCB vectors are in use. This is pictured below:



Using SDA, you can look at the SCB Reservation bit vector to determine which interrupt vectors are already in use. Since OpenVMS/AXP starts vector assignment at SCB offset 800 (hex) and proceeds to higher vector numbers, it is suggested that for manually configured Futurebus+ adapters, you start at the opposite end of the SCB at offset 1FF0 and work your way towards lower vector numbers, as shown in the following diagram. This reduces any chance of a conflict with a previously assigned vector.



Note that interrupt vectors assigned using SYSMAN IO CONNECT commands will not be reflected in the SCB Reservation bitmap.

7.10 Futurebus+ Bus Probing During Booting

During booting, INI$IOMAP constructs an ADP list that represents the I/O adapters present in the system. For the Futurebus+, an ADP (representing the Futurebus+ bridge) and associated Bus Array are allocated. The size of the Futurebus+ Bus Array is based on the number of physical Futurebus+ backplane slots. The bus array is made large enough to contain an entry for the maximum number of Futurebus+ nodes that could be present in the system, which is twice the number of physical backplane slots.

After setting up the Futurebus+ bridge ADP and Bus Array, INI$IOMAP tests each potential Futurebus+ node to determine if an adapter is present. For each Futurebus+ node, INI$IOMAP reads the TEST_STATUS register in Core CSR space. If the node responds, INI$IOMAP then reads locations in the node ROM space to identify the node. Specifically, INI$IOMAP searches node ROM space for a ROM location that identifies the module manufacturer. Normally this will be one of MODULE_VENDOR_ID or NODE_VENDOR_ID (the Futurebus+ specifications say that only one of these locations should be present in the ROM). The Futurebus+ specifications also define MODULE_SPEC_ID, NODE_SPEC_ID, and UNIT_SPEC_ID for the case when a board from manufacturer X requires a driver from vendor Y--the spec_id identifies the manufacturer that supplies the driver (vendor Y, in this example). If a spec_id is found, INI$IOMAP uses it rather than the vendor_id to identify the board manufacturer.

INI$IOMAP also searches the node ROM space for one of MODULE_SW_VERSION, NODE_SW_VERSION, or UNIT_SW_VERSION (again, the Futurebus+ specifications claim that only one of these locations should be present in the ROM). The sw_version identifies the driver required to support the Futurebus+ adapter.

INI$IOMAP concatenates the vendor_id and sw_version to form a 64 bit identifier, and stores it in the BUSARRAY$Q_HW_ID field in the corresponding bus array entry. This 64 bit identifier uniquely identifies a Futurebus+ adapter. It is used by Digital bus configuration routines to associate an adapter and driver.

The bus specific fields of a Futurebus+ node bus array entry are initialized as follows:

- BUSARRAY$Q_HW_ID -- the concatenation of the vendor_id and sw_version is stored in this field.
- BUSARRAY$Q_CSR -- the base of Futurebus+ Initial Node Space is stored in this field.
- BUSARRAY$L_NODE_NUMBER -- the Futurebus+ node number is stored in this field.

The information in the Futurebus+ bus array is displayed by the SYSMAN IO SHOW BUS command, and is useful for manual CONNECTion of devices.

7.11 Futurebus+ on DEC 4000

This section explains some of the details of the DEC 4000 Futurebus+ implementation. More information can be found in the DEC 4000 System Specifications.

7.11.1 The DEC 4000 Futurebus+ Bridge

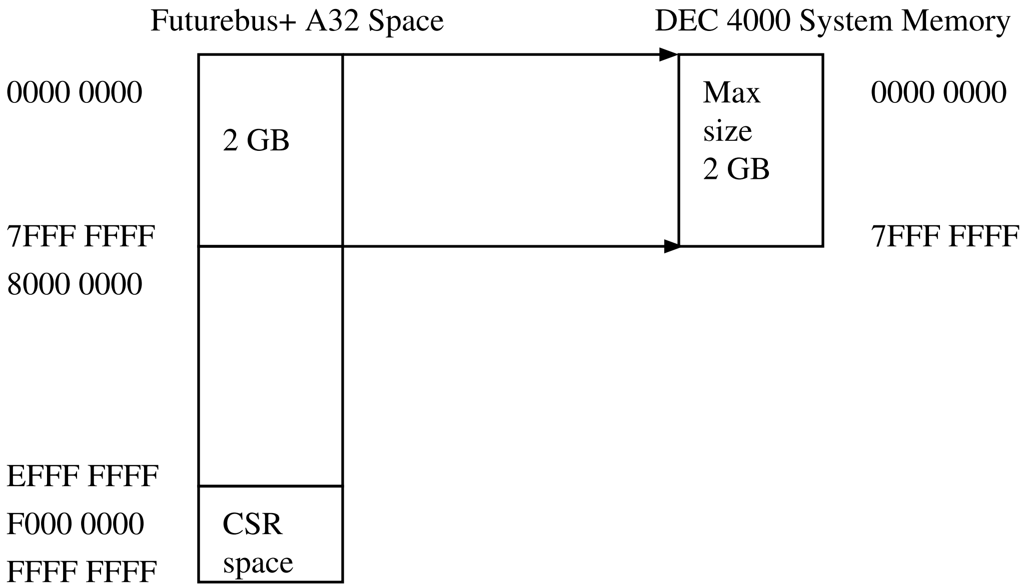
The DEC 4000 I/O Module implements a Local I/O subsystem and a Futurebus+ bridge. The Futurebus+ bridge portion of the I/O module occupies Futurebus backplane slot 0. Backplane slots 1-6 are available for user Futurebus+ options.

As a Futurebus+ master, the I/O module is capable of generating transactions using A32 or A64 address width and D32 or D64 data width.

As a Futurebus+ slave, the I/O module only responds to A32 address width transactions. It does not support A64 addressing as a Futurebus+ slave.

7.11.2 DEC 4000 Futurebus+ Address Space

The DEC 4000 Futurebus+ bridge maps Futurebus+ A32 address space to DEC 4000 system memory space as shown below:



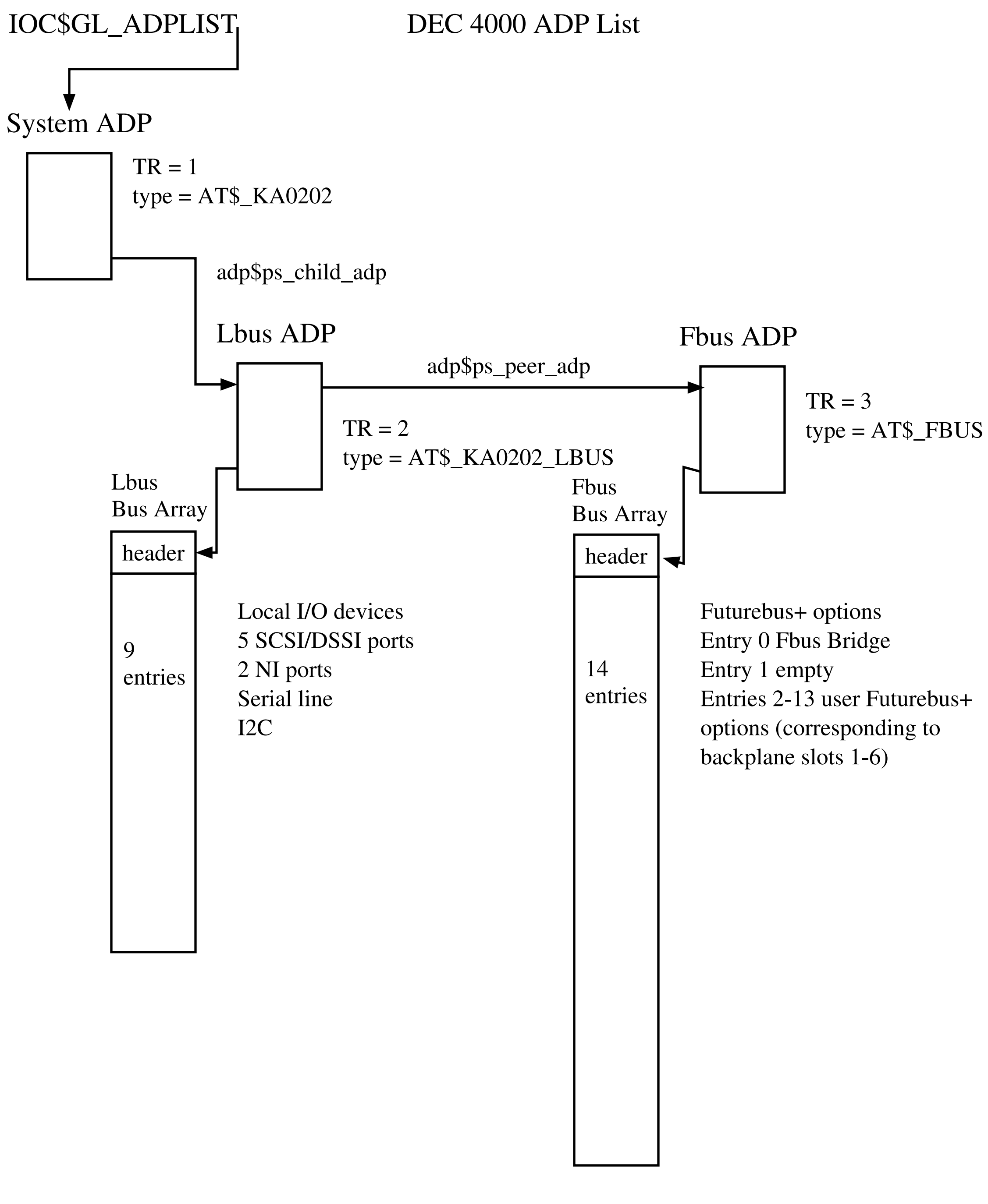
The DEC 4000 I/O module responds as a Futurebus+ slave to A32 addresses in the range 0000 0000 to 7FFF FFFF. A32 transfers in this address range will be passed to system memory by the I/O module. Since the maximum size of DEC 4000 system memory is 2 GB (31 address bits), a Futurebus+ A32 address has enough bits to access the entire DEC 4000 system memory.

If a Futurebus+ adapter requires Futurebus+ A32 space (in addition to its Initial Node Space), the driver must call IOC$RESERVE_FBUS_A32 to reserve a region of A32 space. On DEC 4000, IOC$RESERVE_FBUS_A32 will allocate 16 MB pieces of address space in the region from A32 address 8000 0000 through EFFF FFFF.

If the adapter requires Futurebus+ A64 space, the driver must call IOC$RESERVE_FBUS_A64 to reserve a region of A64 space. Since the DEC 4000 I/O module does not respond to A64 addresses, the entire A64 address space is available to adapters.

7.11.3 DEC 4000 ADP List

During booting, INI$IOMAP creates an ADP list that describes the DEC 4000 I/O subsystem. The DEC 4000 ADP list appears as follows:


IOC$GL_ADPLIST
DEC 4000 ADP List
System ADP
TR = 1
type = AT$_KA0202
adp$ps_child_adp
Lbus ADP
adp$ps_peer_adp
Fbus ADP
TR = 3
type = AT$_FBUS
TR = 2
type = AT$_KA0202_LBUS
Lbus
Bus Array
Fbus
Bus Array
header
header
9
entries
Local I/O devices
5 SCSI/DSSI ports
2 NI ports
Serial line
I2C
14
entries
Futurebus+ options
Entry 0 Fbus Bridge
Entry 1 empty
Entries 2-13 user Futurebus+
options (corresponding to
backplane slots 1-6)

7.12 Futurebus+ on DEC 10000/7000

This section gives some of the details of the DEC 10000/7000 Futurebus+ implementation.

7.12.1 The DEC 10000/7000 Futurebus+ Bridge

The DEC 10000/7000 platform contains an IOP module which interfaces to the DEC 10000/7000 I/O subsystem. The IOP supports up to 4 bridges to remote I/O buses. Currently the supported I/O buses on DEC 10000/7000 are XMI and Futurebus+.

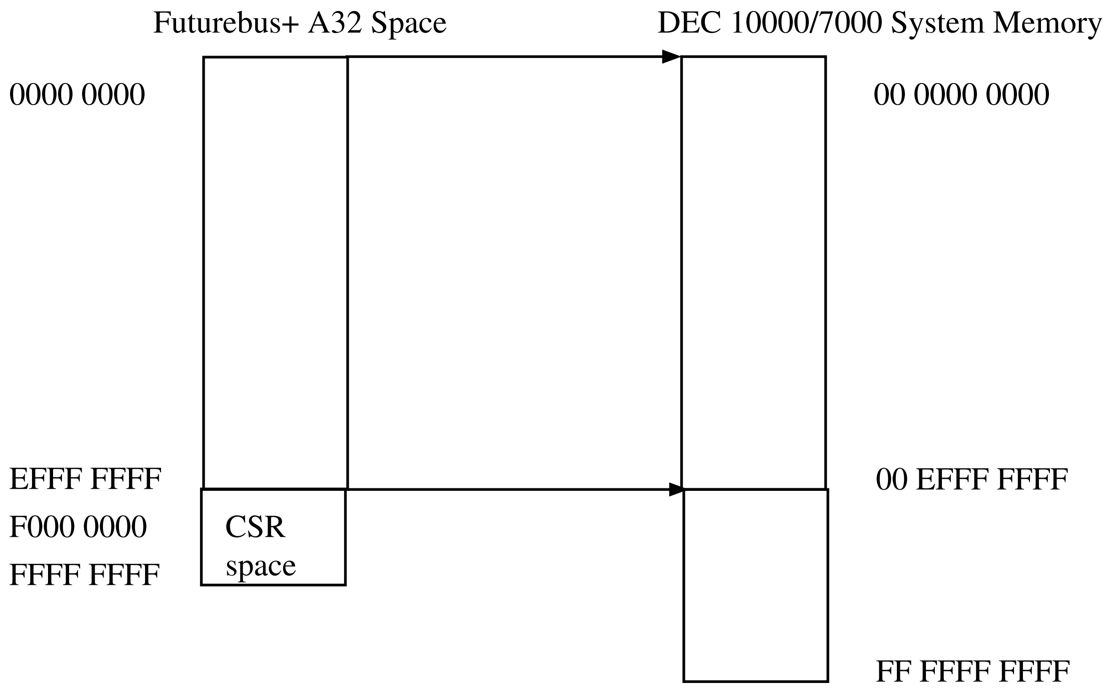
The Futurebus+ bridge on DEC 10000/7000 is implemented in a distinct Futurebus+ module called the Flag. The DEC 10000/7000 Futurebus+ backplanes are 10 slots. The Flag must be installed in slot 5. Slots 0-4 and 6-9 are available for user Futurebus+ options.

As a Futurebus+ master, the Flag can generate A32 and A64 address width transactions, and D32 and D64 data width transactions.

As a Futurebus+ slave, the Flag accepts both A32 and A64 address width transactions, and D32 and D64 data width transactions.

7.12.2 DEC 10000/7000 Futurebus+ Address Space

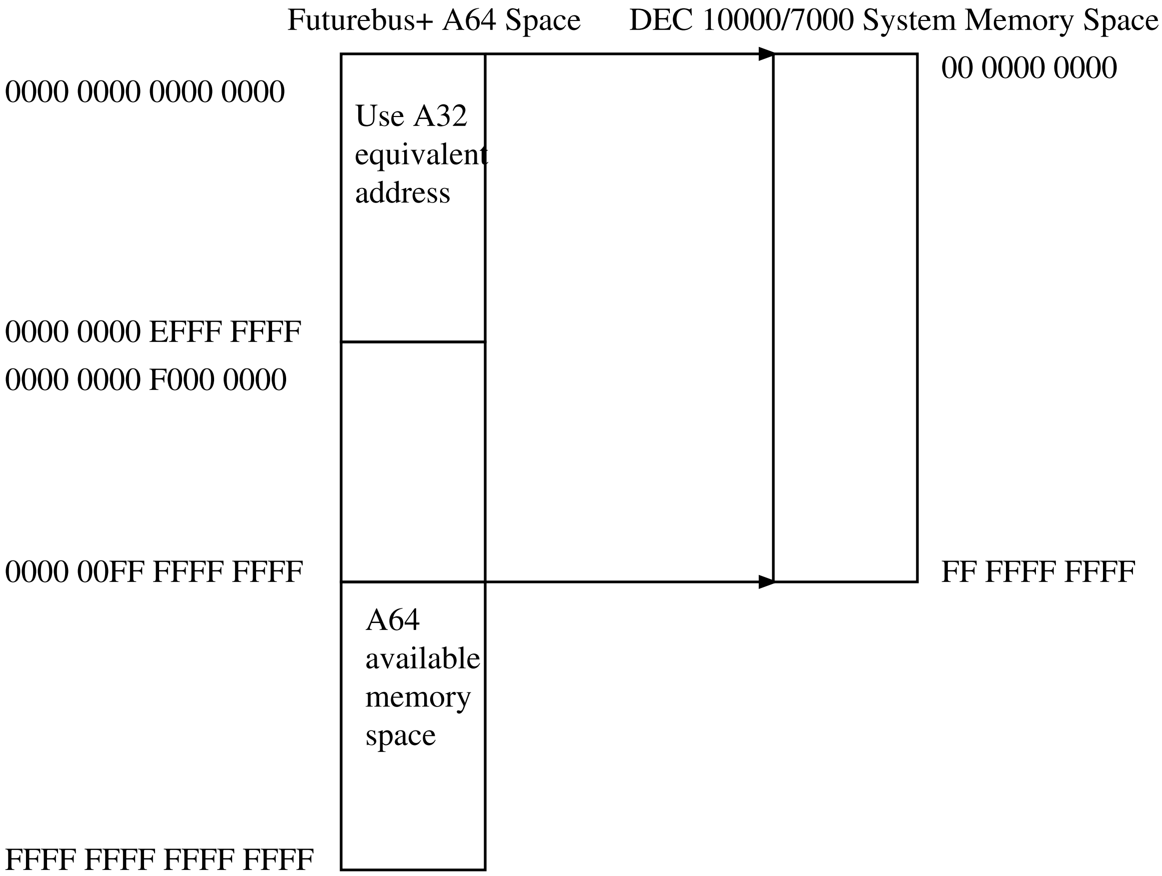
The DEC 10000/7000 system supports a 40 bit (1 TB) memory address space. The Flag module maps Futurebus A32 space to DEC 10000/7000 memory space as shown below:



The Flag responds as a Futurebus+ slave to A32 transfers from 00000000 to EFFFFFFF in Futurebus+ 32 bit address space and passes the transfer to DEC 10000/7000 system memory.. DEC 10000/7000 system memory above 00EFFFFFFF is not accessible from the Futurebus+ using 32 bit addressing.

Since the Flag module consumes the entire A32 address region (except for CSR space), Futurebus+ adapters which require A32 space (that is, which require A32 address space in addition to their CSR space) will not work on DEC 10000/7000.

The Flag module maps Futurebus+ 64 bit address space as shown below:



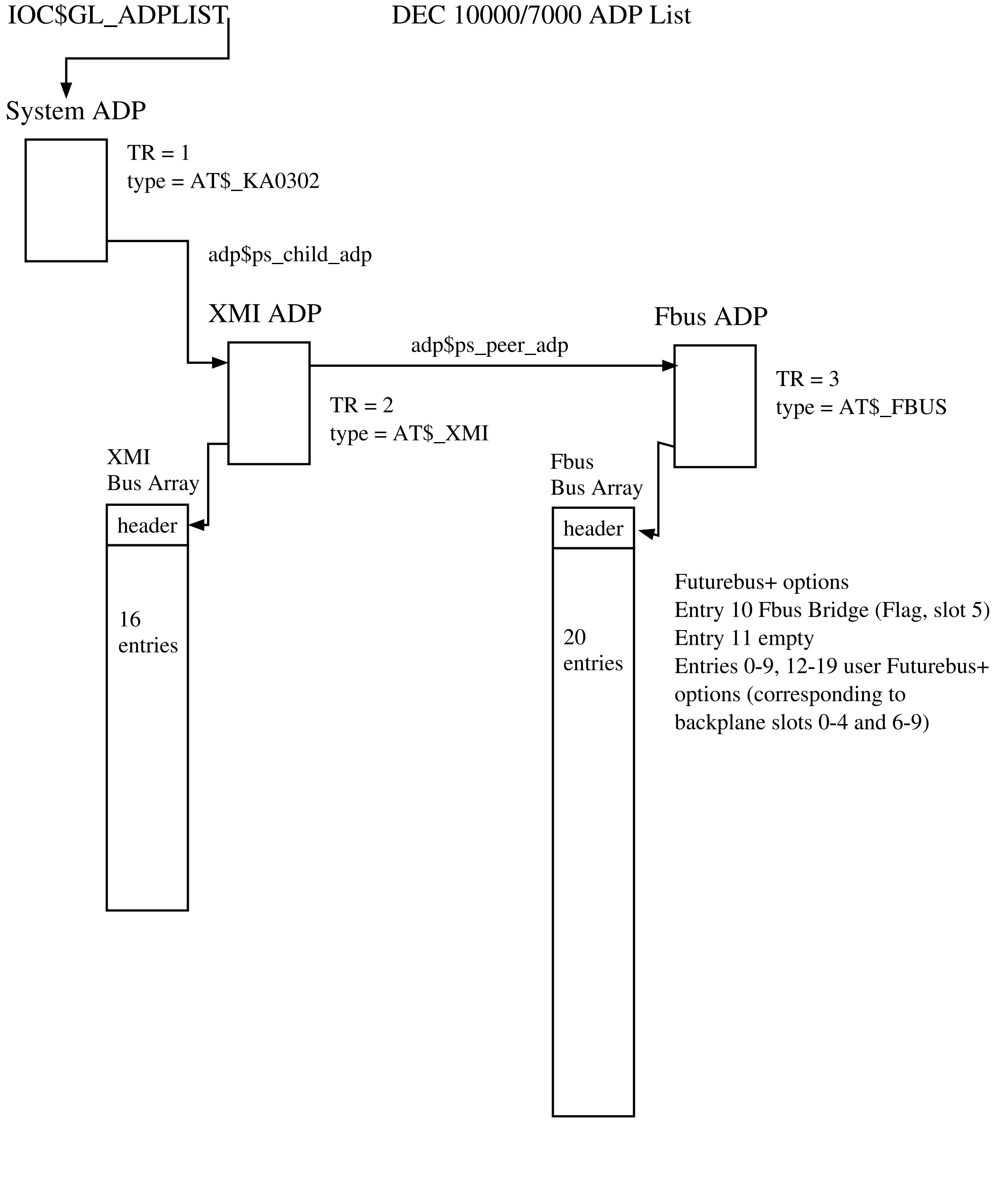
The Flag responds as a Futurebus slave to A64 transactions from 0000 0000 F000 0000 to 0000 00FF FFFF FFFF and passes the transfer to DEC 10000/7000 system memory. The Flag does not respond to A64 transactions directed to 64 bit address space from 0 to 0000 0000 EFFF FFFF. Adapters should not generate A64 accesses in this region anyway, as this region overlaps A32 address space. The equivalent A32 address should be used in the address range below EFFF FFFF. See the Flag specification and the IEEE Futurebus+ specifications for more information.

If a Futurebus+ adapter requires A64 space, the driver must call IOC$RESERVE_FBUS_A64 to reserve a region of A64 space. The A64 address region at 100 0000 0000 through FFFF FFFF FFFF FFFF (which is the address range beyond the space consumed by the Flag module) is available to adapters.

7.12.3 DEC 10000/7000 ADP List

During booting, INI$IOMAP creates an ADP list describing the DEC 10000/7000 I/O subsystem. The exact format of the DEC 10000/7000 ADP list depends on which I/O bridges and backplanes are installed in the system. An example of a DEC 10000/7000 ADP list with an XMI and a Futurebus+ backplane is shown below:

DEC 10000/7000 ADP List



8 E/ISA Bus Overview

This section will describe the history of evolution of the EISA bus and give some backround information on the ISA bus. The following sections will discuss the EISA bus on the DEC 2000 system.

8.1 Evolution of the bus

The EISA (Extended Industry Standard Architecture) Bus is an extension of the ISA (Industry Standard Architecture) Bus. ISA started as the 8 bit bus in the early IBM PCs. It was based on the INTEL 8088 chip. The I/O cards used an 8 bit data bus, and a 16 bit address bus. To save decode logic on the cards, they only decoded the lower 10 address bits for I/O space. Most cards had their resources (IRQ, DMA, IO Ports, Memory Range) hardwired into them. There were 8 interrupt levels (provided by an INTEL 8259 chip), and 4 DMA Channels (provided by an INTEL 8237 chip) available. Obviously the configuration of the machine was limited by these resources.

As the CPU chips got faster, and system size increased, the ISA bus functionality was increased to offer 15 interrupt lines, 7 DMA channels, and a 16 bit data bus to/from I/O. This is known as 16 bit ISA. The cards still decoded only 10 bits in I/O space. To differentiate between memory and I/O space addresses the cards used the AEN signal distributed by the CPU. If AEN was asserted, the cards were enabled to look at the address on the bus. If it matched their IO Port address range, they responded. While DMA was happening AEN was deasserted so that no card would accidentally decode a DMA address as an IO space access. At this point, IBM archictected a new I/O bus, Microchannel Architecture (MCA), which was totally incompatable with the ISA bus. This led the industry to design the Extended Industry StandardArchitecture (EISA) bus. EISA was designed to accept both 8 and 16 bit ISA cards as well as EISA cards, allowing customers to protect their current investment in PC hardware. EISA extended the data path to 16 bits, added software readable product ID's (essential to automatic configuration), slot specific I/O addressing, and active-low level sensitive interrupts (allowing true sharing of IRQ levels).

To allow ISA cards to work in the new EISA systems, slot specific AEN signals were added. In I/O space if a[9:8] were not equal to 00, it was assumed that the access was intended for an old style ISA card, and AENx was asserted at all slots to allow the proper card to decode it. If I/O address bits a[9:8] = 00, then the reference is assumed to be to a new EISA card, and a[15:12] are used to select which slot answers the access (via AENx).

Note that the ISA bus is the prevalent bus seen in PCs today. There are still both 8 and 16 bit ISA cards in many machines. EISA offers higher bandwidth, due to the increased data path size, thus is more suited to the faster microprocessors available today. Also note that the EISA bus functionality is a superset of the ISA bus. Anything that works on the ISA bus will also work on the EISA bus, however, the reverse is not true. EISA cards will not work on an ISA bus.

8.2 Intel 82350DT EISA Chipset

INTEL offers a chip set which is designed to provide an interface between a CPU and the EISA Bus. This chipset provides a 15 level Interrupt Controller(2 logical 8259 chips cascaded), a 7 channel DMA controller(2 logical 8237 chips cascaded), timers, buffers, and logic to convert the 8 and 16 bit ISA protocols to the 32 bit EISA protocol. There is also a chip designed for use by adapter boards to interface to the EISA bus. The INTEL 82355 Bus Master Interface chip is intended for use by adapters wishing to talk EISA protocol, and works in conjunction with the 82350 chip set. This chip set is described in detail in the INTEL specs. For more details about interrupts, DMA channel manipulation, or timer use, see the INTEL spec.

8.3 EISA Bus Resources

The EISA bus has a limited set of system resources. Each adapter is designed to use some subset of the available EISA resources:

- Interrupt Request Levels (IRQ's)
- DMA Channel
- I/O Port Addresses
- EISA Memory space addresses

Due to the limited number of these resources, the large supply of E/ISA adapter vendors, and the need to plug up to 15 adapters into the EISA bus, the configuration of EISA based machines is a very complicated task. Configuration of the system refers to the assignment of resources in a conflict free manner, allowing all adapters to work properly. Adapters are typically designed to use a subset of each of the resource types. The assignment of these resources to adapters is done by an EISA Configuration Utility (ECU) which is run before booting the machine. Next a quick description of each of these resources is given, followed by a description of the ECU.

8.3.1 IRQs

Each EISA slot connector has 15 pins dedicated to the IRQ levels, assuring each card that it can use any of the available IRQ levels. These wires are routed to the Interrupt Controller chip which performs priority resolution, and notification of the CPU. See below for a more detailed description of the interrupt flow. EISA cards typically have a software programmable register that the driver must set up to inform the hardware which IRQ line to drive. ISA cards typically have jumpers on the board which must be set up to tell the hardware which IRQ line to use. The assignment of an IRQ level to a board is done by the Configuration Utility, discussed below.

8.3.2 DMA Channel

For boards that do not have the Bus Master capability (ie, are not complex enough to take control of the EISA bus) the 82350 chip set provides seven DMA channels. The driver must set up the registers in the 82357 chip (base address, count, etc.) corresponding to the assigned

DMA channel. See the section below on DMA for a more detailed description. The ECU is responsible for assigning the DMA channels to the boards that require them.

8.3.3 I/O Port Addresses

This resource applies to ISA cards. As mentioned earlier, when the ISA bus was developed, cards were designed to respond to specific 10 bit addresses. As ISA became more popular, and vendors increased, the I/O addresses each board recognized started to overlap. In order to avoid 2 cards answering to the same I/O address, boards began to provide a jumper to select a range of I/O addresses they would answer to, increasing flexibility of configuration. Thus the ECU is responsible for assigning ISA boards an I/O Port starting address. See the System Address Map in a later section for a description of the available ISA I/O ports. EISA cards generally do not need to have I/O ports assigned to them as they can do geographical slot-based addressing using the AENx signal in the EISA protocol.

8.3.4 EISA Memory Addresses

On some PC machines, system memory is accessed directly over the EISA bus. On Digital machines, the EISA bus is used as an I/O bus. Digital machines provide access to EISA memory space in order to access the on-board memory, such as frame buffers, on some EISA option cards. See the System Address Map in a following section for a description of these address ranges.

8.3.5 EISA Configuration Utility

The ECU is a standalone program that is run prior to booting the machine. The purpose of the ECU is to assign resources to all the cards plugged into the EISA bus in a conflict free manner. The ECU uses configuration files (.CFG files) to get resource requirements for the EISA devices in the system, and prompts for user input to get resource requirements for the ISA devices in the system. The EISA protocol added a software readable ID at a predefined offset on the board (C80-C83). The board manufacturer is free to choose any ID they want for the offset, but, typically, companies are assigned a 3 letter code which uniquely identifies their boards. It is then up to the manufacturer to keep track of it's own IDs in order that each board is uniquely identifiable. For details on the contents of the configuration files, see the EISA spec. The .cfg file contains initialization information and resource requirements. The ECU uses the initialization information to write the specified I/O ports with the specified bit stream to set the board up in a known state. It then will determine the requirements of all the options and find a conflict free assignment of the resources if possible. The ECU also has the capability of prompting the user to enter configuration information about ISA cards, which typically have no configuration information files. The ECU then writes the configuration information into NVRAM on the system board, using console (or BIOS) callback routines. This information in NVRAM is made available to the drivers via the operating system. (IOC$NODE_FUNCTION, IOC$NODE_DATA are the routines provided by OpenVMS AXP). If the ECU can not find a conflict free assignment of resources, it informs the user that a different configuration must be used. After writing the configuration information to NVRAM, the ECU instructs the user to power off the system, set any jumpers/switches on the ISA boards, remove the ECU diskette, and power on. At this point the drivers must call the available routines to determine which resources they have been assigned.

Note that the ECU need only be run when the system configuration is changed, that is, when boards are added, deleted, or moved around in the EISA slots. It is not necessary to run the ECU before each boot, as the configuration is stored in NVRAM.

8.4 EISA Interrupts

The interrupt mechanism on EISA is designed around the requirements of the INTEL 8259 interrupt controller chip. An EISA I/O adapter is designed to work using a small subset of the available interrupt levels on EISA. The cards are designed this way to allow many different configurations to be supported. What this means to the driver writer is you need a method of finding out which IRQ has been assigned to the board so that you can program the board to work with the IRQ (or set a jumper if the board is an ISA board). The IOC$NODE_DATA routine provides this functionality and is described in a later section.

The following is an overview of the sequence of events surrounding an EISA device interrupt.

- The EISA Device requires an interrupt, and asserts the programmed EISA irq line. These lines can be programmed for either edge-triggered (active high) or level-sensitive (active low for sharing irqs) mode.
- The 82357 sees at least one bit in the Interrupt Request Register go high and sets an interrupt pin on the CPU.
- PALcode is invoked and determines that the interrupt is an EISA I/O interrupt
- An INTA (Interrupt Acknowledge) command is sent out over the EISA bus to the 82357.
- This INTA causes the 82357 to lock the Interrupt Request Register for prioritization of the requesting interrupts, and sets a bit in the In Service Register denoting the interrupt selected for service.
- The CPU sends another INTA command over the bus to the 82357, and this command causes a vector identifying the highest priority IRQ requesting service to be sent back over the EISA bus to the CPU.
- PALcode recieves the vector and vectors through the appropriate SCB vector.
- The Driver Interrupt Service Routine services the interrupt and returns to the Operating System Support EISA support code.
- The EISA support code then issues an End Of Interrupt (EOI) command to the 82357 which clears the In Service Register, allowing interrupts of equal and lower priority to occur again. This EOI command is done automatically by either PALCode or Bus Support code in OpenVMS AXP. Operating system code then REIs back to the interrupted thread.

Note that if software does not perform the EOI, all future interrupts of equal or lower priority will be disabled.

8.5 EISA DMA Support

The 82357 Chip provides 7 independently programmable DMA channels for use by EISA cards that do not have Bus Master capability, and cannot drive the necessary signals to perform DMA on their own. These seven channels are implemented using the logic of 2 INTEL 8237 chips, with one cascading into the other (Channel 4 is used to cascade the two controllers together).

Any channel can be programmed for 8, 16, or 32 bit DMA device size, and ISA compatible "type a", "type b", or burst dma "type c" modes. The EISA Bus Controller chip handles the data size translation. The DMA addressing circuitry supports full 32 bit addresses for DMA devices. Each channel includes a 16 bit Current register, a low Page register and a high page register. Both page registers are 8 bits, and between the 3 cover the 32 bits. The channels can be programmed for one of four different transfer mode:single, block, demand, and cascade. The DMA controller also offers buffer chaining, auto-initialization, and support for a Ring buffer Data Structure in memory. Buffer Chaining is not yet supported in OpenVMS AXP.

See the INTEL spec for more details on the usage of a DMA channel.

8.6 EISA I/O Address Map

For a detailed description of the register addresses on the INTEL 82350DT Chip Set, refer to the INTEL specification. These registers include all the Interrupt Controller CSRs, the DMA controller CSRs, and the Interval timer CSRs. Note all addresses and offsets are hexadecimal unless otherwise specified.

EISA I/O space is slot specific for EISA boards, with address bits <15:12> signifying the slot number the board is in. Thus each slot has available 12 bits, or 4K, of I/O space (1000h). As a side effect of keeping the EISA bus backwards compatible with the ISA bus, certain ranges of this 4K of I/O space are used only for ISA boards. These ranges are called "alias" addresses in the documentation. Since the ISA boards react to any 10 bit address which matches their assigned address, EISA boards are designed to use only addresses with 0s in bits 8 and 9. ISA boards react only to addresses with bits <9:8> not equal to 00. Thus, if this restriction is followed, there will be no conflicts between boards. Address aliases mean that the two address ranges address the same physical board: 0100 is the same as 0900 if you use only the first 10 bits. In the following list, all addresses are hexadecimal, and "z" (the EISA slot number) can range from 0-F.

- 0000 - 00FF are system board (slot 0) addresses
- 0100 - 03FF are ISA board addresses
- 0400 - 04FF are system board (slot 0) addresses
- 0500 - 07FF are "alias" ISA board addresses
- 0800 - 08FF are system board addresses
- 0900 - 0BFF are ISA alias addresses
- 0C00 - 0CFF are system board addresses

- 0D00 - 0FFF are ISA alias addresses
- z000 - z4FF are slot z EISA board addresses
- z500 - z7FF are slot z ISA board addresses
- z800 - z8FF are slot z EISA board addresses
- z900 - zBFF are ISA alias addresses
- zC00 - zCFF are slot z EISA board addresses
- zD00 - zFFF are ISA alias addresses

Just to clarify the "alias" addressing even more (it confused me when I first encountered it!) I will provide an example using the DEC 2000 system. Lets say that we have a DEC 2000 with an EISA board plugged into slot 1, and an ISA board in slot 4. Assume the ISA board has been configured using jumpers to respond to starting I/O port address 03F0 (as is the floppy on DEC 2000). Referencing addresses z3F0, z7F0, zFF0, where z is 0-6 on DEC 2000, will access the same register on the floppy. If the EISA board plugged into slot 1 was designed improperly, and included a register at address 3F0, both boards would try to respond to a reference to address 13F0.

9 EISA Bus Support on DEC 2000

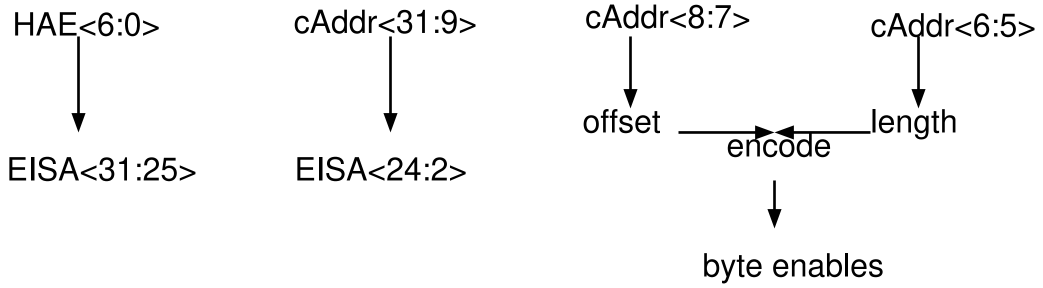
The following sections describe the EISA Bus Support offered on DEC 2000. First a description of the DEC 2000 System Address map is given, then a description of the addressing scheme used for byte access to EISA registers, followed by detailed descriptions of interrupt processing, DMA channels, and finally the OpenVMS AXP provided bus support routines.

9.1 DEC 2000 System Address Map

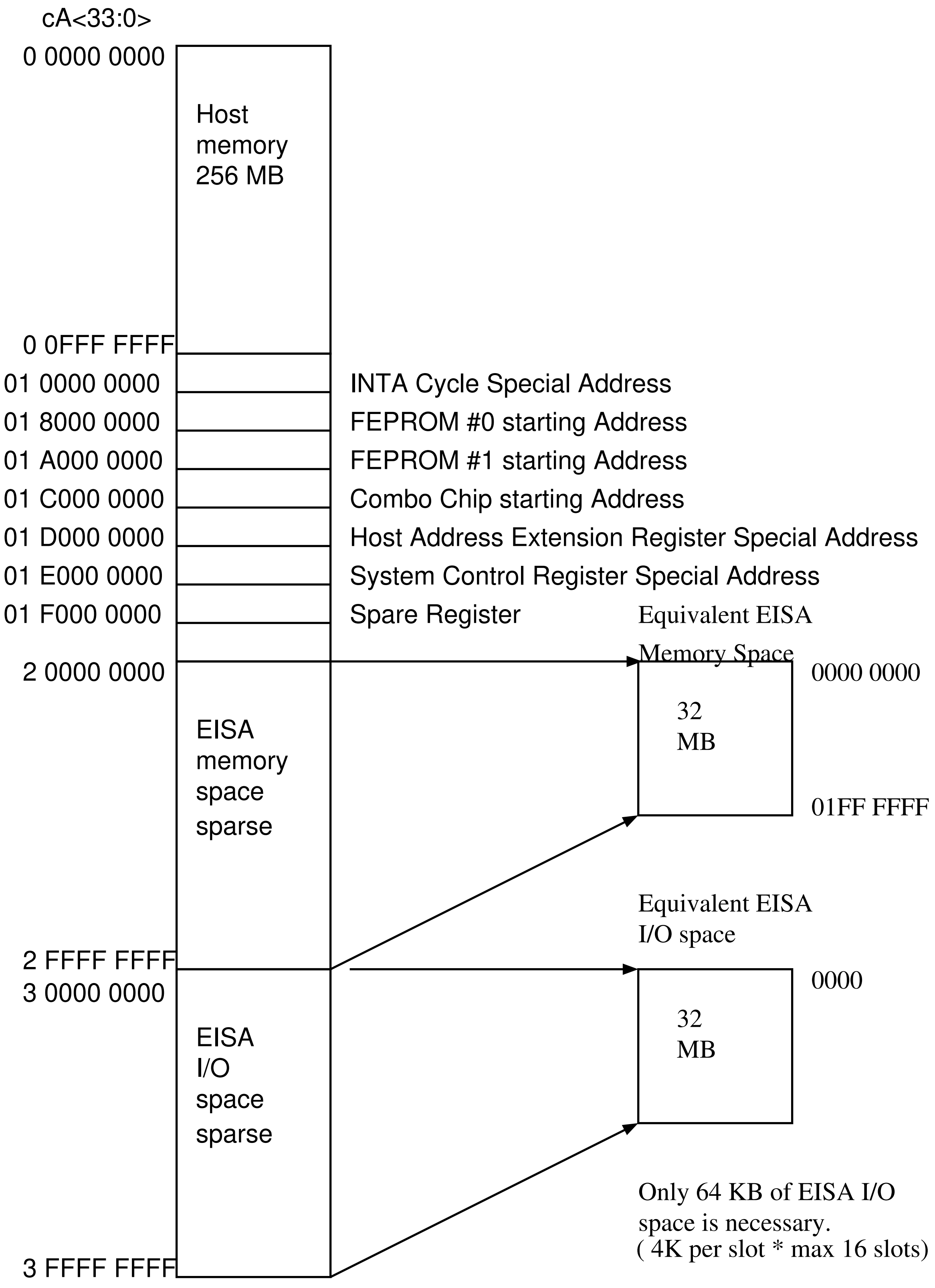
The addressing scheme on DEC 2000 is somewhat tricky, and will need much explaining and some pictures to describe. The DEC 2000 system address map is shown in the following table.

cpu Addr<33:32>	cpu Addr <31>	cpu Addr <30>	cpu Addr<29:28>	Effect
00	MBZ	MBZ	MBZ	Local Memory cAddr<31:28> MBZ
01	1	MBZ	MBZ	EISA INTA cycle (I/O space) cA<31:5> MBZ cA<4:0> SBZ Data appears in low byte
01	1	0	0x	FEPROM #0 cA<28:9> = addr for up to 1MB of ROM, cA<8:0> SBZ Data in low byte
01	1	0	1x	FEPROM #1 cA<28:9> = addr for up to 1MB of ROM cA<8:0> SBZ Data in low byte
01	1	1	00	Combo Chip ComboAddr<..9> from cA<..9> cA<8:0> SBZ Data in low byte
01	1	1	01	Host Address Extension Reg cA<27:0> SBZ
01	1	1	10	System Control Reg cA<27:0> SBZ
01	1	1	11	Spare Register
10	x	x	xx	EISA Memory A<31:25><- HAE<6:0> A<24:2> <-cA<31:9> Length/Offset from cA<8:5>
11	x	x	xx	EISA IO A<31:25><- HAE<6:0> A<24:2> <-cA<31:9> Length/Offset from cA<8:5>

9.1.1 A Pictorial view of DEC 2000 Address Space



DEC 2000 address map as seen by CPU



9.1.2 DEC 2000 System Memory (0-FFF.FFFF)

The DEC 2000 system will support up to 256MB of main memory, addressable using cpu address bits <27:0>. Note that the memory will not respond to access by any device other than the CPU in the upper half-megabyte of the first megabyte of memory (0.5-1MB). This mean that no DMA is allowed into that address range. This restriction is to allow E/ISA devices with on board memory buffers to reside in the first megabyte of EISA memory space.

9.1.3 INTA Cycle Access (1.0000.0000)

The interrupt scheme defined for this machine includes an access to the 82357 chip in order to find out which IRQ level is requesting an interrupt. The mechanism by which this is accomplished on the DEC 2000 is via the special address, 1.0000.0000 . Reading this address will cause hardware to issue an INTA cycle over the EISA bus, which is responded to by the 82357 Interrupt Controller. The first INTA cycle causes the controller to lock the request register for priority resolution, and the second read to this address causes the controller to return a vector in the low byte. This vector indicates the EISA irq currently being serviced (irq + 8 is returned). Device drivers should never need to access this register. The INTA cycle is performed by PALcode.

9.1.4 NVRAM Access (1.8000.0000, 1.A000.0000)

The Flash Rom on DEC 2000 stores console code and configuration information. It is divided up into logical blocks of size 64K. The SRM console is contained in the first 6 logical blocks, with the SRM EISA configuration information contained in the 7th logical block. The Windows/NT console is contained in blocks 8 through 12, with the NT EISA configuration data contained in block 13. The remaining blocks contain the Failsafe Loader. Device drivers should never access NVRAM directly. Bus Support routines are provided as an interface to the configuration data blocks (IOC$NODE_DATA, IOC$NODE_FUNCTION).

9.1.5 VTI VL82C106 Combination Chip (1.C000.0000)

The Combo chip is addressed starting at physical address 1.C000.0000. All the port addresses given in the VTI Combo Chip spec are offset from that base address. Refer to the Combo Chip spec for more details on register offsets. The data returned from reads to this space is returned in the low byte, regardless of the byte offset. This space is mapped into the processor's virtual address space by the Bus support code, and the VA can be found in the ADP$Q_CSR offset of the COMBO ADP (TR # = 3), or in the IDB$Q_CSR field if your driver is loaded via autoconfigure.

It is important to note that the Combo Chip addresses are also "massaged" after being put out on the bus. The combo chip offsets should be shifted up by 9 bits before being put out over the bus by the CPU. This hardware then shifts them down when presenting them to the Combo chip. For example, byte offset 3BC in combo chip space is used to address a parallel port register. In order to get the proper physical address to map the following translation must take place:

PA = 1.C000.0000 + 3BC ^ 9 = 1.C007.7800

Or, using the base VA from the Combo ADP, ADP$Q_CSR + 77800 would access the register. The CRAM routines perform this address shifting for you when used for register access.

9.1.6 Host Address Extension Register (1.D000.0000)

This register is used to extend the addressable space on DEC 2000. This 8 bit register is used to form address bits <31:25> on the EISA bus. For the DEC 2000 implementation, this register is set to 0 by console, and is not used afterwards. There is a possibility that this register may be used to relocate the 1 MB Frame Buffer of the VGA card above System Memory. However, if this is done, there are synchronization issues that will have to be dealt with. Device drivers should never access this register. OpenVMS AXP assumes that this register is always zero.

9.1.7 System Control Register Access (1.E000.0000)

The System Control Register contains Memory configuration information in bits <7:4>, and a code for the LED's in bits <3:0>.

9.1.8 EISA Memory Space Access (2.0000.0000 - 2.FFFF.FFFF)

On the DEC 2000, this address space is used to access the RAM buffers which are built into some EISA adapter boards. For instance, the DE422 card contains a 64KB buffer on board which is accessed via EISA Memory Space. The memory addresses are assigned to the boards that need them by the EISA Configuration Utility. These starting addresses are no larger than F8000. From the EISA adapter point of view, that address appears to be in system memory space. That is the primary reason that the system board does not recognize EISA addresses in the range 0.5 - 1 MB as DMA addresses. The following figures are intended to show the mapping of EISA Memory space, and then give several examples describing how a particular board uses this mapping. It is important to note that devices can not use this memory region at will.* They must be assigned a particular address range by the ECU to avoid overlap. An example of how an EISA memory address would be formed may help to illustrate the bit shifting: Assume that the DE422 board is assigned a 64K buffer in EISA memory space, starting at address C0000. To access a longword at address C0008 the CPU would have to map the following PA:

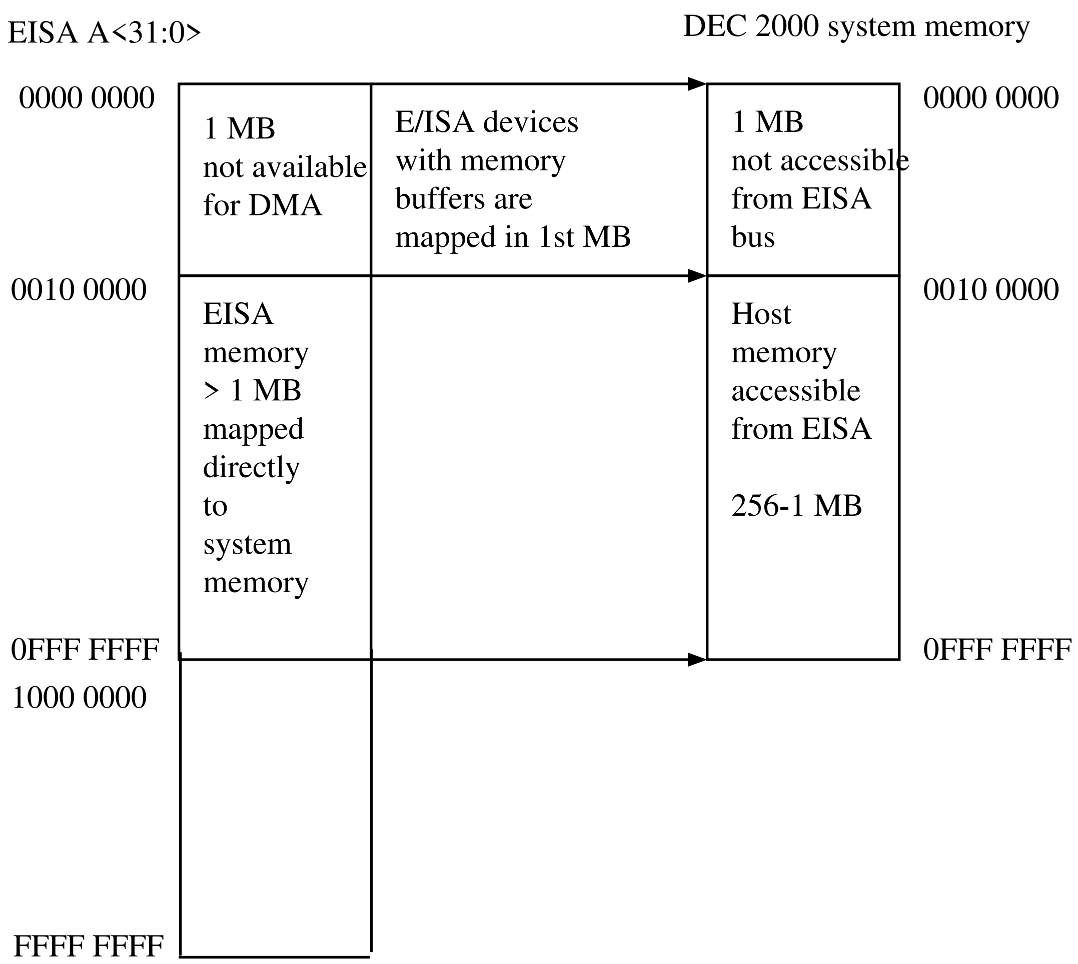
PA = base_addr + offset_shifted_by_7 + length_constant_for_longword

2.0000.0000 + C0008 ^ 7 + 060 = 2.0600.0460

The base address is 2.0000.0000, the offset is shifted by 7 bits, putting the byte offset into bits <8:7>, and the length constant for longword is put into bits<6:5> by adding 060. Mapping this PA would yield a VA which would access the longword on the DE422 on-board RAM starting at offset 8.

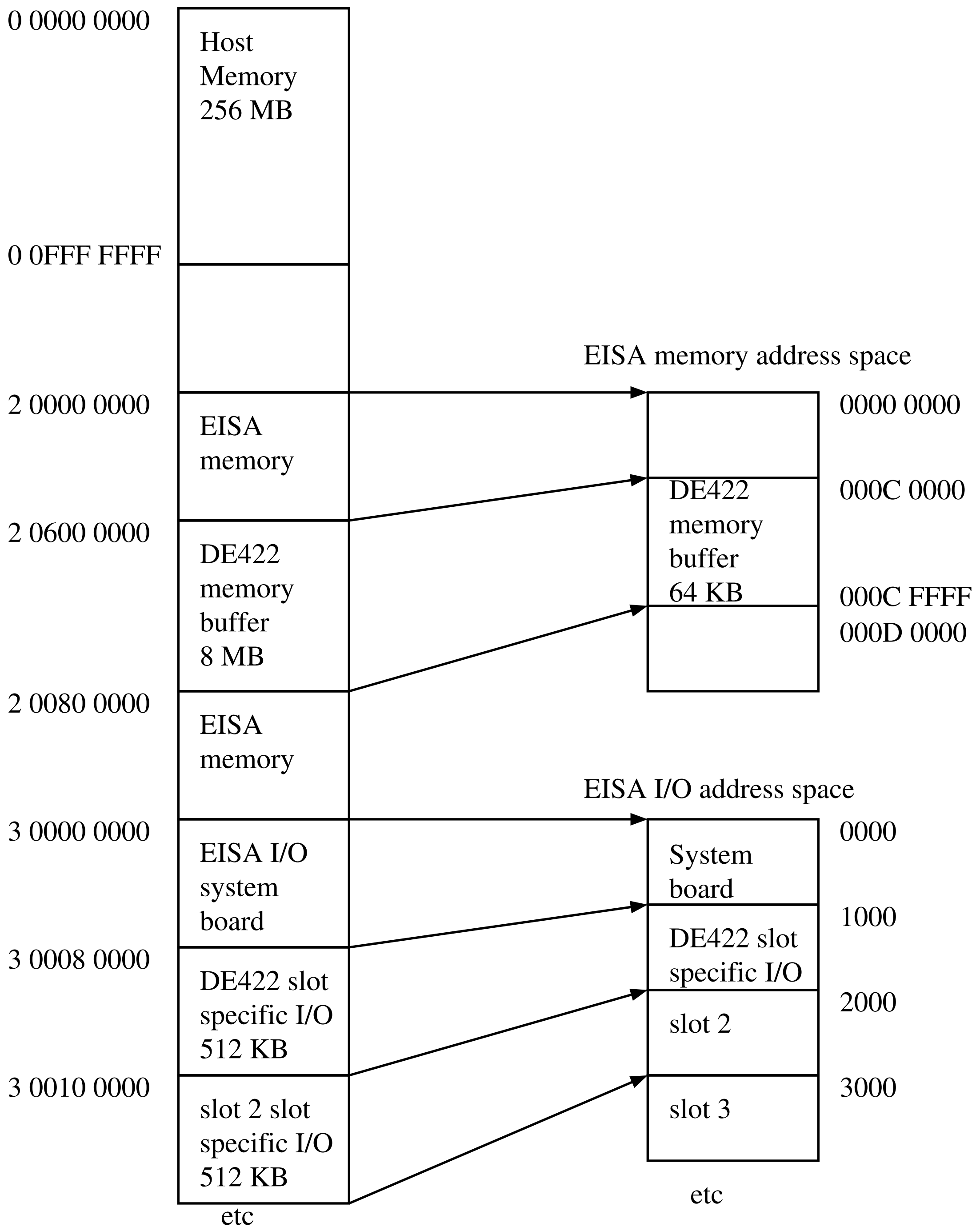
***Programmers take note:** This means you can not access E/ISA card buffer space as you did in the INTEL architecture machines. In fact, unless your card is addressable at an EISA address between 0x80000 and 0x100000, you risk corruption of system memory if you attempt to access the E/ISA board buffer. The next diagram shows this overlap of memory addresses. Note also that EISA memory space is subject to the same "sparse space addressing" as the EISA IO space. There are solutions available for cards that can't meet the above requirements on addressing, but they involve losing part of the available system physical memory.

EISA memory address space as seen by EISA adapter on DEC 2000



Install a DE422 EISA Ethernet card in a DEC 2000/EISA slot. The DE422 uses slot specific EISA I/O space for control registers, and requires a 64 KB EISA memory buffer.

DEC 2000 address map as seen by CPU. DE422 in EISA slot 1, DE422 memory buffer at EISA address C0000.



9.1.9 EISA I/O Space Access (3.0000.0000 - 3.FFFF.FFFF)

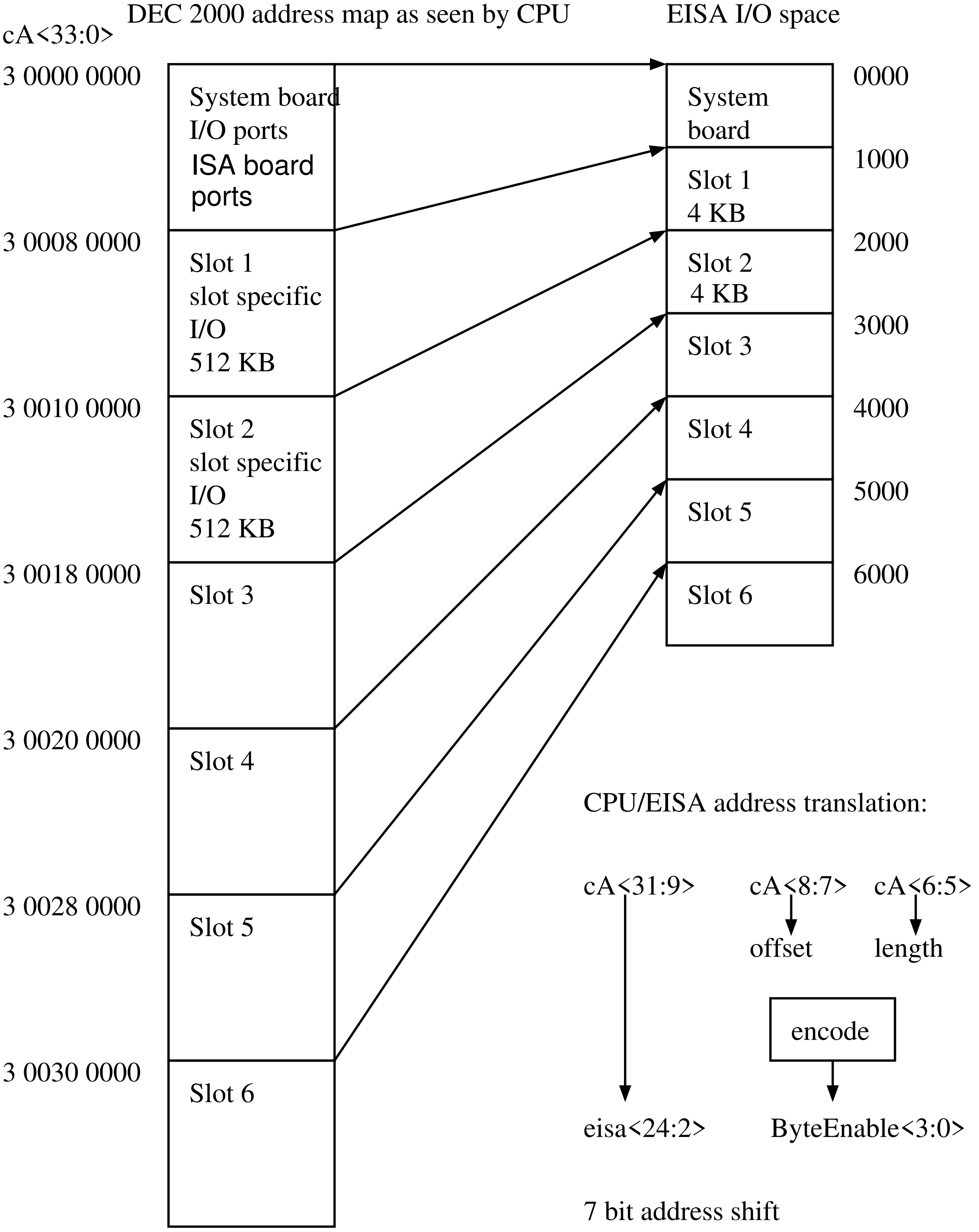
EISA I/O space is mapped into the address space accessed by CPU addresses 3.0000.0000 thru 3.FFFF.FFFF. As mentioned in the historical discussion of the EISA/ISA busses, the cards in the system decode only 10 I/O address bits, with the EISA cards using bits <15:12> to provide geographical addressing (they choose the slot the reference is directed to). Since each slot can take up a maximum of 4Kbytes, the maximum I/O space needed on this system is 24 Kbytes. Unfortunately, due to the "funny address tricks", this requirement gets shifted up by 7 bits and turns into 3Mbytes of EISA I/O address space. An expanded view of the mapping of EISA I/O space to the DEC 2000 Address Map is shown on the following page. As an example of how the address for an EISA I/O space access is formed, assume that the Adaptech 1742a board is in EISA bus slot 4, and you want to read the product ID longword. The PID is stored starting at offset C80. The address would be formed as follows:

```
PA = base_addr+( slot *  1000 + slot_offset) ^7 + length_constant_for_longword
        3.0000.0000 +(  4C80 ^ 7)                  +  060
        3.0000.0000 +  264000                      +  060 = 3.0026.4060
```

Mapping this PA would give access to the longword containing the PID for any board in slot 4. Note that the CRAM routines provided by the Bus Support Code insulate the user from all this bit manipulation. The interface to these handy routines is described in an earlier chapter. Note that the CRAM routines expect the user to use the proper byte lane when reading or writing data from the CRAM$Q_RDATA/CRAM$Q_WDATA fields of the CRAM. Specifically, if it is intended to write a byte at byte offset 1 of a quadword, that data byte must be placed in byte lane 1 of the CRAM$Q_WDATA quadword.

Note that for all ISA devices the slot number is considered to be 0. All ISA devices respond to slot 0 addresses regardless of what is put into address bits <15:12> on the EISA bus. If the slot offset has bits <9:8> = 00, then the ISA devices don't respond at all, if bits <9:8> != 00 then the ISA devices respond, no matter what the "slot" bits (<15:12>) are set to.

Expanded view of DEC 2000/EISA I/O space



9.2 Sparse Space

All I/O access on DEC 2000 is done in "sparse" space. There is no dense space. Address bits <8:5> are used to select the length of the transaction and the starting longword offsets shown in the following table:

Addr bits 8:7	Addr bits 6:5	Access type	Byte Enables	length constant
00	00	byte access to byte 0	f f f T	000
00	01	word access to word 0	f f T T	020
00	10	tribyte access to tribyte 0	f T T T	040
00	11	longword access to LW 0	T T T T	060
01	00	byte access to byte 1	f f T f	080
01	01	word access to word 1	f T T f	0A0
01	10	tribyte access to tribyte 1	T T T f	0C0
01	11	reserved		
10	00	byte access to byte 2	f T f f	100
10	01	word access to word 2	T T f f	120
10	10	reserved		
10	11	reserved		
11	00	byte access to byte 3	T f f f	180
11	01	reserved		
11	10	reserved		
11	11	reserved		

The following example shows how the EISA I/O space is "fragmented" to represent the Product ID Longword Register for an arbitrary EISA board. Assume we are trying to read the Product ID of the DE422 board in slot 1. The byte offset for the Product ID board is zC80 - zC83. We can read it as 4 separate bytes, 2 word, a tribyte and a byte, or a longword.

The starting address is 3.000E.4000. That PA points to EISA I/O address 1C80, and specifies a byte transaction. The PID register holds the following value for the DE422 : 2042A310

3.000E.4000				10	EISA Address 1C80	Byte Enables <00>	f f f T
3.000E.4020			A3	10		<01>	f f T T
3.000E.4040		42	A3	10		<10>	f T T T
3.000E.4060	20	42	A3	10		<11>	T T T T
3.000E.4080			A3		EISA Address 1C81	<00>	f f T f
3.000E.40A0		42	A3			<01>	f T T f
3.000E.40C0	20	42	A3			<10>	T T T f
						<11>	
3.000E.4100		42			EISA Address 1C82	<00>	f T f f
3.000E.4120	20	42				<01>	T T f f
						<10>	
						<11>	
3.000E.4180	20				EISA Address 1C83	<00>	T f f f
						<01>	
						<10>	
						<11>	

Note that the blank bytes in the I/O space map above represent unknown data on the EISA bus, and may come back with stale data. Consumers of EISA bus data should pay careful attention to the bytes requested, and mask off any unrequested bytes that are returned with the wanted data. Also notice that some of the accesses are restricted. Any access that would attempt to cross a longword boundary is illegal, returning no data.

9.3 Register Access

Register access on the DEC 2000 is done via the Direct Access method described earlier. There is no mailbox hardware on DEC 2000. However, to make the interface easier for the driver writer, the CRAM routines are provided, and simply perform direct access of registers, as opposed to a mailbox access. Usage of these routines is highly recommended, as the EISA bus will be found on newer platforms, and may use different shift amounts than DEC 2000. If the driver uses the CRAM routines for register access, there is one less thing that may need to change to move the driver between EISA platforms.

9.3.1 Direct Register Access

Daring authors are certainly able to do the direct access of registers without using CRAMs. Be sure to pay strict attention to the setting of the length/offset bits <8:5>, and be sure to understand the address bit shifting that is occuring over the various buses in the DEC 2000. In the case of an EISA bus access, the cycle time of the EISA bus is so slow relative to the EV4, that the overhead of a routine call for the CRAM routines is negligible.

9.3.2 CRAM Register Access

CRAM register access on the DEC 2000 works similarly to the other AXP machines. The same CRAM routine interface (IOC$CRAM_INIT, IOC$CRAM_CMD, IOC$CRAM_IO, etc)is provided, and it works in the same fashion. Obviously there is quite a lot of bit shifting going on behind the scenes of the routine calls. Be sure to byte lane the data in the CRAM RDATA and WDATA quadwords before using. Note that this is not true Hardware Mailbox access, as defined in the SRM. The DEC 2000 has no mailbox hardware, the CRAM routines use direct access methods. That is, the actual access is performed using a load or store instruction to a virtual address.

9.4 DMA on DEC 2000

There is no Scatter/Gather Map on DEC 2000. A DMA engine is provided for those devices that don't have Bus Master capability. 7 DMA channels exist in the 82357 chip. The channels are assigned as resources by the ECU. It is up to the driver to set up the channel for operation via writes to the 82357 DMA registers. IOC$NODE_DATA can be used to get the DMA channel assigned to a board. The IOC$NODE_DATA interface is described below. A brief example is given, showing one usage of a DMA channel. For complete details, please refer to the INTEL 82357 specification.

9.4.1 DMA example

This example shows how the floppy driver on DEC 2000 made use of one of the provided DMA channels. The floppy chip (82077) is resident on the Adaptech SCSI board, but is considered to be a seperate device, using the ISA protocol. It is hardcoded to use IRQ6, I/O ports 3F2h-3F7h, and DMA channel #2.

The driver allocates and maps a contiguous buffer to store the DMA data in between the floppy device and the requestor. Then for each access it must set up a number of DMA registers in the 82357 chip to program the channel for the access. One sequence used to set up a channel looks as follows:

- Setup the channel in the desired mode, enable it
  - Command register, offsets 8D0, enable the channels. Note channel 4 must be enabled if device is using channels 0-3, as they are cascaded through channel 4.
  - Mode Register, offsets B,D6, set the desired mode: chan 4 = cascade, and set the assigned channel to be either read or write, depending on the command to be issued.
- Set up the target address
  - Clear the byte pointer, offset C. This clears a byte pointer flip-flop which indicates whether the high or low byte was accessed last. This must be done prior to R/W of the address or word count registers.
  - Load the base addr register, offset 4. This byte register is written twice to load the 16 bits of the base address register.
  - Load the low page register, offset 81. This byte register contains bits <16:23> of the 32 bit target address
  - Load the high page register, offset 481. This is the high byte of the target address.
- Load the transfer count registers
  - Clear the byte pointer once again
  - Load base count register, offset 5. Note it is necessary to load #_of_bytes_to_transfer - 1 in the count register due to the HW of the 82357 chip.
  - Load the count register, offset 405. Load the 2nd byte of the count register.
- Enable the channel, fill the command silo of the 82077 chip, and let it rip!
  - Set Mask Write register, offset A. Set the bits enabling channel 2 to do DMA.
  - Perform the DMA action.

Note that some of this setup is required for each transfer.

9.5 I/O Interrupts on DEC 2000

An earlier section described the general flow of an E/ISA device interrupt. Here we describe the OpenVMS AXP view in more detail. It is assumed that the reader has read the previous section on EISA interrupts.

9.5.1 EISA IRQs

There are 15 IRQ levels available on DEC 2000. Each E/ISA device reports its interrupts through one of these IRQ levels. The default set-up for the IRQ's is to be edge-triggered. The ECU/console is responsible for assignment of the IRQ level to a board. The assignment of the IRQ levels depends on the order boards are plugged into the EISA slots. In some cases one IRQ level can be given to one of a subset of boards. The first board encountered during resource assignment (board in the earliest EISA slot) will be given the IRQ. The others boards that may use that IRQ are assigned another available IRQ.

Today the console assigns the IRQ's as follows:

- IRQ0 - 82357 Timer Interrupt
- IRQ1 - Combo chip Parallel Port Interrupt
- IRQ2 - not a valid interrupt, used to cascade the slave 8259 IRQ's into the master 8259
- IRQ3 - Not assigned to any device
- IRQ4 - Not assigned to any device
- IRQ5 - Assigned to the first DE422 board encountered in system
- IRQ6 - Not assigned by console, given to the Floppy by Bus Support Code
- IRQ7 - Not assigned, this is the spurious interrupt IRQ level, see 82357 spec for details
- IRQ8 - Not a valid interrupt, it is asserted low, and not used on DEC 2000
- IRQ9 - Compaq VGA card uses IRQ9, DEFEA card can also use IRQ9
- IRQ10 - Second DE422 is assigned this IRQ
- IRQ11 - First DEFEA card found is assigned this IRQ
- IRQ12 - First Adaptech SCSI board found is assigned this IRQ
- IRQ13 - Not assigned to any device, used for Buffer Chaining by DMA engine
- IRQ14 - Assigned to the second Adaptech SCSI board found
- IRQ15 - Assigned to the second DEFEA card found in the system

Note that the COM ports on the COMBO chip do not report via an EISA IRQ. They are wired directly to an Interrupt Request Level on the EV4 chip.

9.5.2 SCB Vectors

On OpenVMS AXP EISA device I/O interrupts are dispatched through SCB vectors 800h-8F0h. Each IRQ level is assigned a vector using the following formula:

SCB = 800h + (IRQ * 10h)

So IRQ 0 reports through SCB vector 800, IRQ1 through 810, etc. In the special DEC 2000 release version, interrupt dispatching is indirectly vectored, and all SCB vectors 800-8F0 point to the indirect dispatcher, which will then call the appropriate driver's Interrupt Service Routine.

9.5.3 EOI

The End Of Interrupt command is used to release an interrupt on DEC 2000. The release of an interrupt enables future interrupts of equal or lower priority, thus is critical to the interrupt flow. The EOI command is sent out over the EISA bus to the 82357 chip, which sends it to the associated logical 8259 chip to release the interrupt.

OpenVMS AXP uses indirect dispatching to control the dispatch of interrupts. The indirect dispatcher is responsible for issuing the EOI command for all EISA interrupts.

9.6 EISA Bus Interface Registers

9.6.1 Interrupt Enable Register

EISA I/O interrupts are enabled or disabled via the Interrupt Enable Register. This 82357 register is accessed via the IOC$NODE_FUNCTION routine using the IOC$K_ENABLE_INTR and IOC$K_DISABLE_INTR function codes.. When a driver is ready to handle device interrupts, it should call the IOC$NODE_FUNCTION routine with the IOC$K_ENABLE_INTR function code. This will cause a bit in the Interrupt Enable register to be asserted, enabling EISA interrupts from the IRQ assigned to the device driver calling the routine. The interrupts can also be disabled by using the IOC$K_DISABLE_INTR command index. This will clear the enable bit in the Interrupt Enable Register, prohibiting further interrupts from the IRQ associated with the device. As an example call of this routine is shown below:

```
status = IOC$NODE_FUNCTION (crb_address,
                            ioc$k_enable_intr);
```

9.6.2 End of Interrupt Command

As mentioned above the EOI command is critical to the success of the EISA interrupts. This 82357 register is accessed via the IOC$NODE_FUNCTION routine with the command index IOC$K_ISSUE_EOI. This routine call will write the EOI register with the IRQ level assigned to the calling device, releasing the interrupt. Early test versions of OpenVMS AXP required drivers to use this call in the driver interrupt service routine to release the interrupt, however, the EOI is now done by the indirect interrupt dispatcher. It is not necessary for drivers to isse the call to IOC$NODE_FUNCTION to cause an EOI.

9.6.3 IOC$NODE_FUNCTION & IOC$NODE_DATA

9.6.3.1 IOC$NODE_FUNCTION

On DEC 2000 this routine is used to enable/disable EISA interrupts. The parameters are CRB address and function code. The available functions are ioc$k_enable_intr and ioc$k_disable_intr.

Both ioc$k_enable_intr and ioc$k_disable_intr work on the Interrupt Mask Register in the Interrupt Controller Chip (82357) for the EISA bus. In the driver unit or controller init routine, after the device is set up to handle interrupts, the driver should call IOC$NODE_FUNCTION with the ioc$k_enable_intr code. Note that many EISA devices also have interrupt enable bits resident on the boards themselves which the driver must explicitly enable also.

The IOC$NODE_FUNCTION function codes are defined in [LIB.LIS]IOCDEF.SDL.

9.6.3.2 IOC$NODE_DATA

This routine is used by a driver to get the resources assigned to its device. Currently the console assigns the resources to the supported boards. Soon this will be done using an EISA Configuration Utility (ECU) from MCS Corporation. The resources which are assigned to EISA boards are Interrupt Request Level (IRQ), DMA Channel #, EISA IO Port address, and EISA Memory address. **NOTE WELL:** If the console doesn't support your board, then these routines will not work until the ECU is supported. You will have to hardcode your resources in the driver until the Configuration Utility is available. (See the paragraph below on manually connecting a driver for available resources). Note that the console assigns only a single resource to a board, thus this routine returns only a single resource. However, the ECU may assign multiple resources for a board, depending on what is in the manufacturers configuration file. It is suggested that the callers of this routine allocate a buffer large enough to hold the information for all the resources. That is to say, if the board can be assigned more than 1 Memory Buffer, say 4 (as is the case for the COMPAQ VGA card), the caller ought to be sure to allocate 4 times the suggested buffer size given below. All buffer sizes given below are for a single resource. When the ECU is supported, the format of the data returned will change to match what is written to NVRAM by the ECU.

The parameters to IOC$NODE_DATA are:

```
crb         - Address of the CRB which contains the EISA slot number
fnc_code    - Function code to be performed. This routine supports only
               4 of the available function codes: ioc$k_eisa_irq,
               ioc$k_eisa_dma_chan, ioc$k_eisa_io_port,
               and ioc$k_eisa_mem. Any
              of the other function codes will return bad status.
user_buffer - Address of a user supplied buffer to contain the data. It   is assumed
               that the user has allocated enough space  for the return data.
```

Function code ioc$k_eisa_irq requires a longword buffer, and returns the IRQ assigned to the board in that buffer. Valid EISA IRQ's for DEC 2000 are 0,1,3-7,9-12,14,15. Boards are typically designed to work with a subset of these IRQ's, and need to be told which IRQ they will be using in the current system configuration. Drivers typically need to program the board to use the IRQ it has been assigned, and can use this routine to find that out.

Function code ioc$k_eisa_dma_chan requires a longword buffer and returns the EISA DMA Channel assigned to the board. There are 7 DMA Channels available on DEC 2000, 0-3 and 5-7. Not all boards do DMA, nor do all DMA boards use the DMA engine provided by the Intel 82350DT Chip set.

Function code ioc$k_eisa_io_port requires only a word length buffer. It returns to the user the starting address of the assigned IO port. Many ISA boards, and some EISA boards are designed to use some number of different address ports for CSR access. This routine will return the starting IO port address assigned to this board.

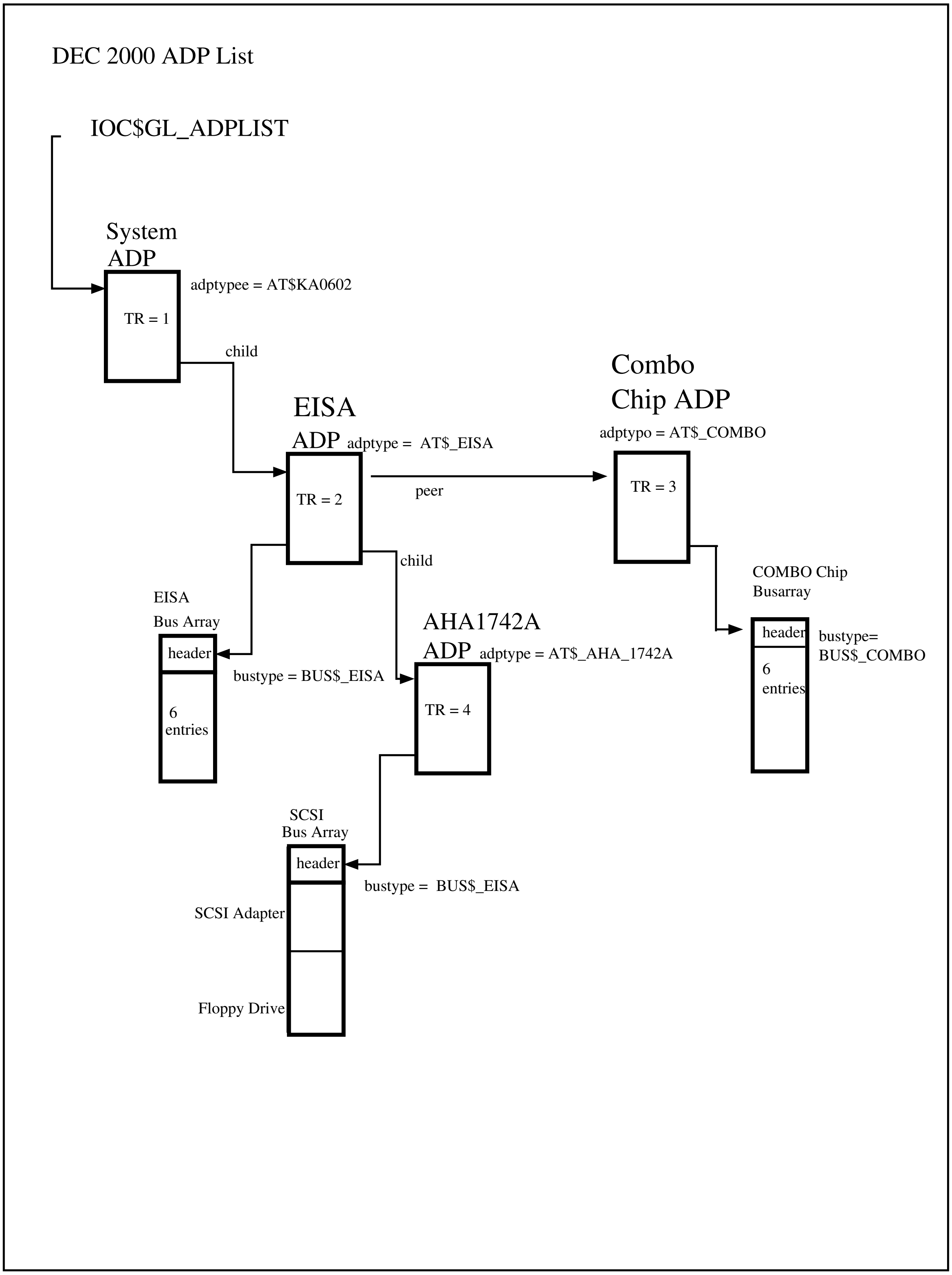
Function code ioc$k_eisa_mem require a quadword buffer. It will return the assigned starting address of EISA memory in the first longword, and the size of the allocated buffer in the second longword. These addresses are used to access on board RAM on DEC 2000 adapters. Available EISA memory addresses on DEC 2000 are A0000 - EFFFF, and are given out in chunks of 32 or 64K. Note the driver is responsible for mapping the PA of the EISA board mem buffer (=2.0000.0000 + (start_addr ^ 7)). Currently there are only 3 boards supported which request memory buffers: DE422, DEFEA, and VGA card. However, depending on how many of each of these cards are in the system, the available memory resources can be exhausted. The VGA card is assigned addresses A0000-C7FFF(3 chunks). The NI and FDDI cards compete for the same memory buffer addresses, E0000/D0000. The NI card requires a 64K buffer, and asks for a starting address of either E0000, or D0000. The FDDI card requires either a 32K or 64K buffer, and asks for a starting address of either C8000, or E0000 then D0000. So, we can not have 2 NI and 2 FDDI cards in the system at one time. The resources limit the configuration to 1 NI & 2 FDDI or 1 FDDI & 2 NI. See the later section on resource assignment for all the details of available resources.

9.6.3.3 CRB$L_NODE

The CRB$L_NODE field contains information critical to the success of these routines. If the driver is manually connected, the command must include the "/NODE=%xyyyyzzzz" qualifier, where yyyy is the IRQ assigned, and zzzz is the slot number the board is plugged into. For instance, if the console assigns the board IRQ 5, and it is plugged into slot 6, the CONNECT command would include /NODE=%x00050006. If the driver is autoconfigured, then this field is filled in by the driver loading code.

9.7 DEC 2000 I/O Space Map

The Bus Support Code is responsible for creating a map of the I/O space. This is done during bus probe time, as mentioned earlier. The ADP map created for DEC 2000 looks as follows:


DEC 2000 ADP List
IOC$GL_ADPLIST
System
ADP
adptypee = AT$KA0602
TR = 1
child
EISA
ADP
adptype = AT$_EISA
TR = 2
peer
Combo
Chip ADP
adptypo = AT$_COMBO
TR = 3
child
EISA
Bus Array
header
6
entries
bustype = BUS$_EISA
AHA1742A
ADP
adptype = AT$_AHA_1742A
TR = 4
COMBO Chip
Busarray
header
6
entries
bustype=
BUS$_COMBO
SCSI
Bus Array
header
bustype = BUS$_EISA
SCSI Adapter
Floppy Drive

9.8 Configuring a Device on DEC 2000

If your device is not supported by the Bus Support code on DEC 2000, it will need to be manually configured. The method to do this is similar to all other AXP platforms, with the exception of the CRB$L_NODE field. Below is given the command to issue in order to configure a device and load the driver for an EISA adapter. Each of the command parameters is discussed in detail.

```
$ MCR SYSMAN IO CONNECT xxA0 /driver=xxdriver
                             /vector=(irq*4)
                             /node=%x000z000a
                             /adap=2
                             /csr=b
```

Specify the device name and driver name as you would for any system.

9.8.1 Vector parameter

/vector=(irq*4): The vector is slightly tricky. Because DEC 2000 uses indirect dispatching, the vector is computed, similar to the other indirectly dispatched platforms, using the IRQ as follows: vector=(irq*4). IE, if your device is going to use IRQ 4, the vector parameter would be 16. Choosing an available IRQ is the tricky part. You must choose an IRQ that your board can work with that hasn't already been assigned by the console. Until we get an EISA Configuration Utility, the console will be assigning resources to only the following supported boards as shown:

Adaptech 1742 SCSI board #1 - IRQ 12

Adaptech 1742 SCSI board #2 - IRQ 14

DE422 NI board #1 - IRQ 5 EISA MEM ADDR E0000, D0000 if FDDI #2 already has E0000
DE422 NI board #2 - IRQ 10 EISA MEM ADDR D0000

DEFEA FDDI board #1 - IRQ 9 EISA MEM ADDR C8000

DEFEA FDDI board #2 - IRQ 11 EISA MEM ADDR E0000, D0000 if NI #1 already has E0000
VGA board, currently no IRQ assigned, EISA MEM ADDR A0000-C7FFF.

Floppy (if configured) - IRQ 6, DMA Channel 2

In a typical system, where there is only 1 SCSI board, 1 NI board, and a VGA card the user is free to choose any IRQ for the board except for:

- IRQ0 - assigned to the 82357 Timer
- IRQ2 - used to cascade IRQs 8-15 into IRQs 0-7
- IRQ5 - assigned to the Ethernet card

- IRQ6 - assigned to the Floppy Chip, resident on the Adaptech 1742a card
- IRQ8 - this is active low and not used on DEC 2000
- IRQ9 - assigned to the Compaq VGA card
- IRQ12 - assigned to the Adaptech 1742a SCSI controller
- IRQ13 - used by the DMA controller, not available on DEC 2000

The user is free to choose any DMA channel except:

- DMA channel 2, which is used by the Floppy controller
- DMA channel 4, which is used to cascade channels 0-3 into 4-7.

If your board needs EISA MEM space, you are restricted to choosing something in the range 80000h - 100000h, other than:

- A0000-C7FFF - assigned to the Compaq VGA card
- D0000-DFFFF - assigned to the DE422 card.

Note that the DMA channel and the EISA Memory resource are not specified in the CONNECT command anywhere. There is no system software checking for conflicts in usage of the resources. It is assumed that the user will select a conflict free assignment. When the ECU is available, it will check for conflicts before boot time, and the driver is assured of a conflict free assignment by using the IOC$NODE_DATA function to get ECU assigned resources.

9.8.2 Node parameter

/node=%x000z000a:

Here "z" is the IRQ level that is chosen, and "a" is the slot number the board is plugged into. The slots are numbered from 1 - 6. Note that without this qualifier, the IOC$NODE_DATA and IOC$NODE_FUNCTION routines will NOT work.

9.8.3 CSR parameter

/csr = b: The CSR address can be found using the MCR SYSMAN IO SHOW BUS command. For each adapter plugged into the system, the EISA Bus Support code maps that adapters IO Space. This is the base VA which is used by the CRAM routines to access CSR's on the board. It is strongly encouraged that you use CRAM routines (IOC$CRAM_CMD and IOC$CRAM_IO) to access CSR's, as then the bus support code performs all the funny address bit shuffling for you. The CSR parameter is copied into the IDB$Q_CSR field.

9.8.4 Resource Assignment on DEC 2000

9.8.4.1 IRQ's

In a typical system, where there is only 1 SCSI board, 1 NI board, and a VGA card the user is free to choose any IRQ for the board except for:

- IRQ0 - assigned to the 82357 Timer
- IRQ1 - assigned to the Parallel Port on the Combo chip

- IRQ2 - used to cascade IRQs 8-15 into IRQs 0-7
- IRQ5 - assigned to the Ethernet card
- IRQ6 - assigned to the Floppy Chip, resident on the Adaptech 1742a card
- IRQ8 - this is active low and not used on DEC 2000
- IRQ9 - assigned to the Compaq VGA card
- IRQ12 - assigned to the Adaptech 1742a SCSI controller
- IRQ13 - used by the DMA controller, not available on DEC 2000

Explicitly, that would be IRQs: 3,4,7,10,11,14,15.

9.8.4.2 EISA Memory Addresses

If your board needs EISA MEM space, you are restricted to choosing something in the range 80000h - 100000h. Depending on the configuration, choose something other than:

- A0000-C7FFF - assigned to the Compaq VGA card
- D0000-DFFFF - assigned to the DE422 card, or a second DEFEA card
- E0000- EFFFF - assigned to the DEFEA FDDI card if present, or a second DE422 card

Note that these memory requirements prohibit an EISA bus configuration consisting of 2 DE422 cards and 2 DEFEA cards. There is not enough memory space to go around.

9.8.4.3 ISA I/O Port Addresses

The only port addresses being used on the DEC 2000 system are that of the floppy port. The floppy is using ports 3F2-3F7h. If more ISA devices are plugged into the EISA bus, they will also be assigned port addresses.