CCU Architecture

A Common Command Unit Architecture
based on P1212.1 Shareable List DMA

X3T9.2/92-131r1 / D0.52 13Aug92

Abstract

This document specifies the I/O process that provides communication between a I/O driver and an I/O service. The scope of the specification includes the formats of command and status lists, the list-update operations, device registers, and ROM entries. This document assumes that nodes are connected using the P1394 Serial Bus and that devices fully comply with the IEEE Std 1212-1991 CSR Architecture. This document is based on the concepts developed within the P1212.1 DMA Framework working-group draft.

NOTE: Copies of this document may be purchased from:
Global Engineering Documents, 2805 McGaw, Irvine, CA 92714
(800) 854-7179 or (714) 261-1455.
Please refer to document X3T9.2/92-131

draft proposed
American National Standard for information systems -

Common Command Unit Architecture
CCU

13Aug92

Secretariat

Computer and Business Equipment Manufacturers Association

This is a draft proposed American National Standard of Accredited Standards Committee X3. As such, this is not a completed standard. The X3T9 Technical Committee may modify this document as a result of comments received during X3 approval as a standard.

Ed Note: The appropriate legal stuff has to be put here.

Comments on this Draft should be addressed to the current Editor:
either of the Editors:

Scott Smyers
Apple Computer, Inc.
20705 Valley Green Drive, MS: 60-AR
Cupertino, CA 95014
voice: 408/974-7057
fax: 408/446-9154
email: smyers.s@applelink.apple.com

Comments may also be addressed to the previous (and still somewhat active) editors:

Daniel C. O'Connor
Apple Computer, Inc.
20705 Valley Green Drive, MS: 60-AR
Cupertino, CA 95014
voice: 408/974-6753
fax: 408/446-9154
email: doconnor@apple.com

Dave James
Apple Computer, Inc.
20450 Stevens Creek Blvd., MS: 60-AR
Cupertino, CA 95014
voice: 408/974-1321
fax: 408/974-9793
email: dvj@apple.com

Contents

Listings

Figures

Tables

Foreword by the Chairman of the Working Group

Historical Perspective

This standard is based on the I/O architecture specified by the IEEE Std 1212-1991 CSR Architecture. This standard has been fortunate to share working group members with the former P1212 working group, as well as members of the IEEE Std 1596-1992 Scalable Coherent Interface working group.

TBD - additional comments should reflect the continuing technical and organizational history of this project.

Acknowledgments

TBD - since a final list is only possible after everyone has had a chance to contribute.

Working Group Members

The specification has been developed with the combined efforts of many volunteers. The following is a list of those who were members of the Working Group while the draft and final specification were compiled:

TBD - List to be created at the end of the working group refinement process.

1. Introduction

1.1 I/O Process Overview

1.1.1 I/O Process Components

This document specifies the I/O process, initiator-target command/status protocol based on the Shareable List DMA model described in the IEEE project P1212.1 DMA Framework draft document. This specification covers the necessary algorithms, transaction types, data structures, registers, and ROM entries to be a participant in the I/O process. This document assumes that the underlying bus is specified by the IEEE P1394 Serial Bus draft document and that devices connecting to it comply to IEEE Std 1212-1991 CSR Architecture.

This document builds on the DMA Framework to provide a unit architecture, called the Common Command Unit Architecture (CCU Architecture), that participates in this command/status protocol and defines management functions that are not specified in the DMA Framework. This document is self-sufficient with respect to the DMA Framework (P1212.1). It is not, however, self-contained with respect to P1394 Serial Bus or to IEEE Std 1212-1991 CSR Architecture.

The major elements in a simple I/O process are illustrated in Figure 1-1. An initiator is a function that has the intelligence to generate commands. The initiator appends a command entry to the command list of the appropriate target. The command entry contains queuing information, command information, data buffer pointer(s), and status information pointer(s).

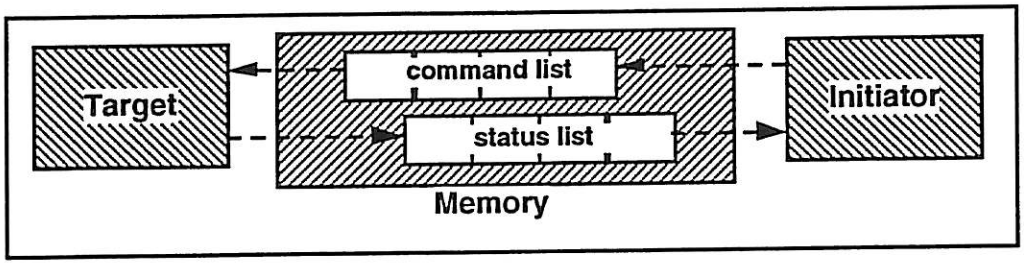

Figure 1-1: I/O Process System Overview

A target is a function that has one or more I/O services, such as a disk controller or network interface, as its client(s). When the target is notified that there is new information in one of its command lists, it extracts the command entry (updating the list pointers as required) and passes a copy of the command entry to the appropriate I/O service.

When the target's I/O service completes the command-entry-specified action, it may be required to return status. The target updates the contents of the command-entry-specified status entry and appends that status entry to the command-entry-specified status list, and may notify the command-entry-specified initiator that new information was added to that list.

1.1.2 Steps in the I/O Process

To better illustrate the I/O process concepts, consider the operation of the I/O process in the context of a simple single initiator and a single target configuration. The steps involved in performing a DMA transfer are then illustrated in Figure 1-2.

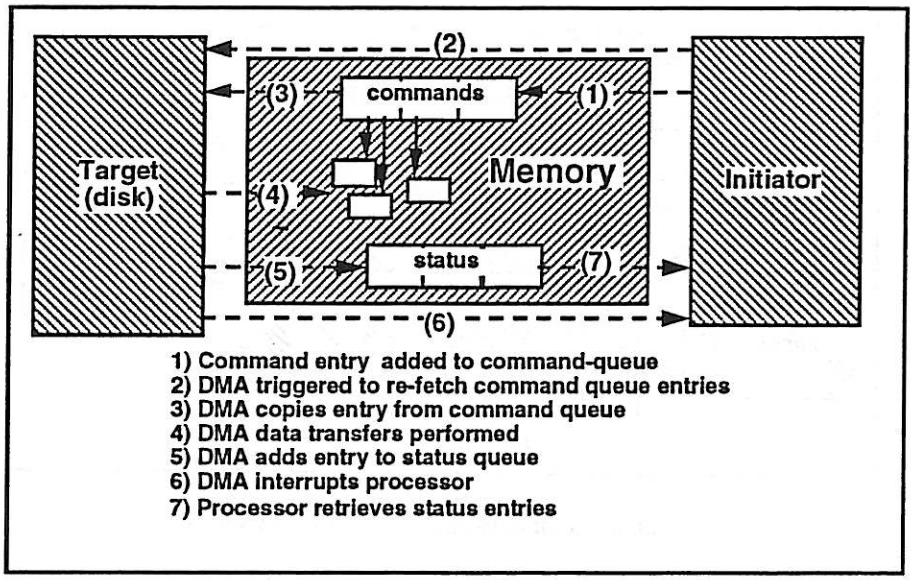

Figure 1-2: Steps in the I/O Process

These steps in the I/O process are summarized below:

1) Command Append. The initiator initializes the contents of a command entry data structure. The command entry is then appended to the tail of the target's command list.

2) Target Wakeup. A write to a specialized target *wakeup* register re-activates the target's processing of additional command-list entries.

3) Target Extract. The target extracts one or more command entries from the command list, possibly queueing them in internal storage before they are processed.

4) Command Execution. The target executes the data-transfer commands contained within the command entry. These commands typically transfer data between system memory space and device-specific addresses (such as locations on a disk).

5) Status Append. After the execution of a command entry completes, its affiliated status block is updated and appended into a memory-resident status list.

6) Initiator Wakeup. After the status has been appended to the status list, a write to a specialized initiator *wakeup* register re-activates the initiator's processing of additional status-list entries.

7) The initiator's I/O driver software processes the returned status entries. After each status entry is processed, the command entry, status entry, and affiliated data buffers are released for other system uses.

1.1.3 Shared-Queue Transfers

Although the I/O process often involves one initiator and one target, the command and status list structures can be shared. For example, consider two processors (called *initiator[0]* and *initiator[1]*) which access three I/O devices (called *target[a]*, *target[b]*, and *target[c]*), as illustrated in Figure 1-3.

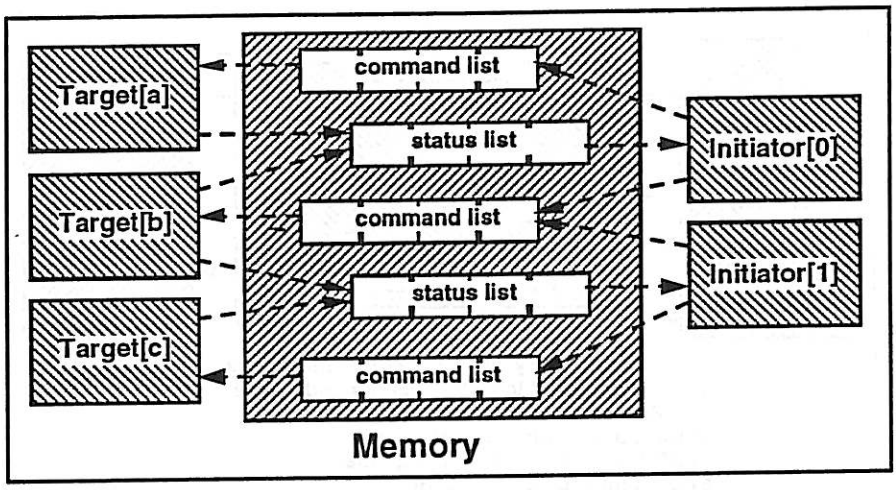

Figure 1-3: Shared Queue Overview

In this example, the command list for *target[b]* is shared by *initiator[0]* and *initiator[1]*, which can actively share the device. Similarly, the status lists for *initiator[0]* are shared by its dedicated *target[a]* and shared *target[b]* devices; the status lists for *initiator[1]* are shared by its dedicated *target[c]* and shared *target[b]* devices.

In many cases, the initiator that generates a command entry and the initiator that processes the returned status entry will be the same. However, since the address of the status list is specified within each command entry, the command entry has the possibility of being routed to a different status list.

As shown, the command and status lists may reside anywhere in bus-accessible space, that is in the addressable space common to the targets and initiators. Also, an initiator may append command entries to any number of command lists; a target, in an analogous fashion, may append status to any number of status lists. In one system, there could be many targets accessed by only one initiator, as in a CPU and peripherals configuration. In a different system, there may be many initiators accessing one target, as in a shared printer configuration.

1.1.4 Initiator and Target Resources

The initiator and targets maintain internal state, as needed to access the command and status lists. Both the initiator and the target are required to provide an externally accessible wakeup register, which is used to reactivate command and status entry processing in the target and initiator respectively, as illustrated in Figure 1-4.

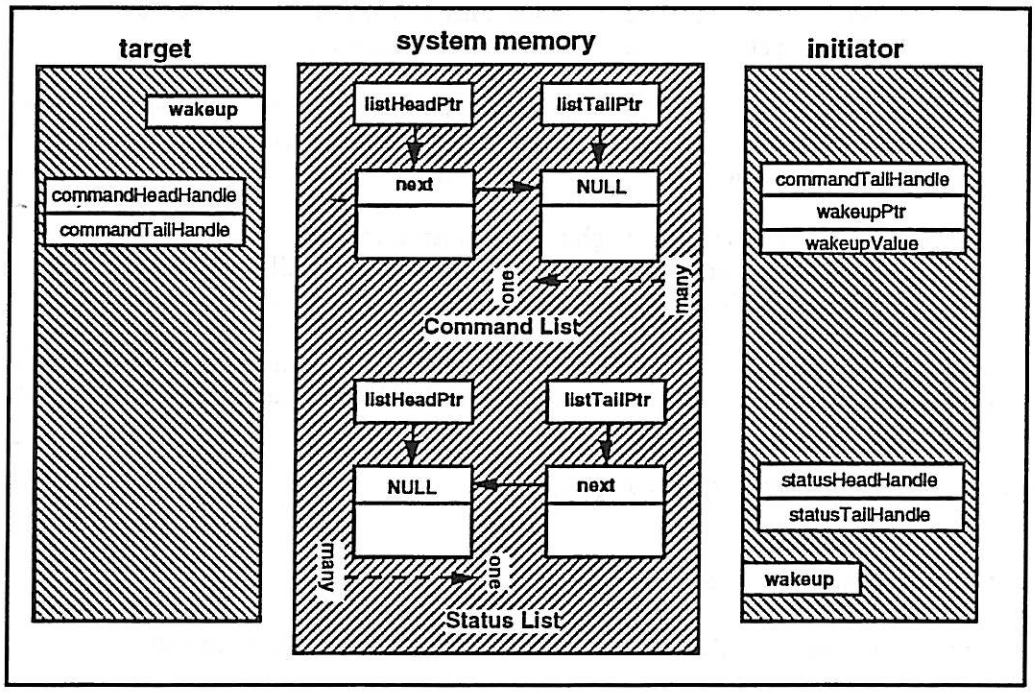

Figure 1-4: Initiator and Target Resources

To append command-list entries, the initiator also knows the addresses of the command-list tail pointer (*commandTailHandle*), the address of the wakeup register (*wakeupPtr*), and the value to use when writing to the command-list-affiliated wakeup register (*wakeupValue*). This information is needed for each target that may be accessed.

To extract command-list entries, the target knows the addresses of the command-list head and tail pointers (*commandHeadHandle* and *commandTailHandle*). The target does not save status-list related parameters, since the status-queue related parameters (*statusTailHandle*, *wakeupPtr*, and *wakeupValue*) are included as parameters within the command-list entries.

1.1.5 Design Capabilities

The functional properties of the CCU Architecture's command and status list design, which are unavailable on many alternative DMA architectures, include the following:

1) High Performance. Most of the initiator's read and write transactions that update command and/or status lists involve efficient accesses of local system memory. Remote SerialBus control register accesses are generally much less efficient, particularly for read bus transactions. The CCU Architecture is optimized for this environment; control register reads are not normally used and only one write transaction is needed to initiate the target's command-list processing.

2) Device Autonomy. Because the target autonomously fetches its DMA commands from system memory, the size of initiator-generated command lists is not limited by the target's physical buffer size. The target paces the flow of commands, so the transient rates of command generation (by the initiator) and command consumption (by the target) need not be matched.

3) Scalable. The scalability properties of the CCU Architecture include the following:

   a) Shared Queues. Multiple initiators may share a single target's command list and multiple targets may share an initiator's status list.

   b) Generalized. In addition to supporting traditional processor-to-I/O communications, the shared command/status queue structures are an efficient mechanism for sending processor-to-processor messages through shared memory.

   c) High-End Support. Although the CCU Architecture was designed primarily for low-cost SerialBus applications, the architecture is equally applicable to high-speed backplanes, such as provided by the IEEE Std 1596 Scalable Coherent Interface.

4) Flexible. Because commands and data are located in target-accessible shared memory, special capabilities (like flow-control, command-list looping, and command-list transfers) are easily supported.

1.2 Glossary and Notation

1.2.1 Conformance Levels

Several keywords are used to differentiate between different levels of requirements and optionality, as follows:

expected. A keyword used to describe the behavior of the hardware or software in the design models assumed by the CCU standard. Other hardware and software design models may also be implemented.

may. A keyword that indicates flexibility of choice with no implied preference.

shall. A keyword indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other CCU Architecture conformant products.

should. A keyword indicating flexibility of choice with a strongly preferred alternative. Equivalent to the phrase "it is recommended."

1.2.2 Glossary of Terms

A large number of bus and interconnect-related technical terms are used in this document. These terms are described below:

bus transaction. A data exchange between two nodes, consisting of a request subaction and a response subaction. The request subaction transfers a read/write/lock transaction code from a requester to a responder. The response subaction returns a completion-code from a responder to the requester. Depending on the transaction code, the data may be transferred in the request (for a write), in the response (for a read), or in both (for a lock).

byte. Eight bits of data, used as a synonym for octet.

command entry. An entry in a command list that contains command information and may optionally contain address information.

command list. A singly-linked list of command entries. One or more initiators may append command entries to a command list. A single target is responsible for extracting command entries from a command list.

compareSwap. An indivisible bus transaction that conditionally stores a *new* argument to a specified data address and returns the previous data value from that address. The store occurs when the addressed memory value and a second *arg* value are equal. In the CSR Architecture, this is called a compare_swap transaction.

CSR Architecture. IEEE Std 1212-1991, Control and Status Register Architecture.

CCU Architecture. A term which refers to the contents of this document.

doublet. A data format or data type that is 2 bytes in size.

DMA Framework. A draft document being produced by the IEEE P1212.1 working group.

fullSwap. A form of the maskSwap bus transaction, in which the mask value is all ones.

hexlet. A data format or data type that is 16 bytes in size. The name hexadeclet would more accurately describe these 16-byte formats, but for notational convenience this abbreviated term is used throughout this standard.

indivisible access. A data access for which the entire datum is read or written as a whole, with no possibility of being partial interleaved with another data access.

initiator. An entity that generates commands and delivers them to a target using the I/O process. The initiator may also receive status from the target upon command completion.

I/O. An abbreviation for the terms *Input and Output*.

I/O process. TBD.

I/O service. TBD.

list access mechanism. TBD.

lock transaction. A bus transaction that transfers data and sub-command from the requester to the responder and returns data from the responder. The subcommand indicates whether this is a maskSwap or compareSwap transaction.

maskSwap. An indivisible bus transaction that stores bits of a *new* argument to a specified data address and returns the previous data value from that address. The affected bits are specified by a *mask* argument. In the CSR Architecture this is called a mask_swap transaction.

node. An entity associated with a particular set of control register addresses (including identification ROM and reset command registers) that is initially defined in a 4Kbyte (minimum) initial node address space. In normal operation each node can be accessed independently (a control register update on one node has no effect on the control registers of another node). Physically, a node is most commonly a device attached to SerialBus, although such a device may in fact contain more than one node.

octlet. A data format or data type that is 8 bytes in size. Not to be confused with an octet, which has been commonly used to describe 8 bits of data. In this document, the term byte, rather than octet, is used to describe 8 bits of data.

quadlet. A data format or data type that is 4 bytes in size.

read operation. The data-transfer phase in the execution of a read command entry that copies data from the device-specific space into bus-accessible space. A read operation consists of one or more write transfers.

read transaction. A bus transaction that returns data from the responder to the requester.

read transfer. A transfer of data from a contiguous range of bus addresses into device-specific address space. Read transfers are performed by the target in the process of executing a write (memory-to-target) or copy (memory-to-memory) command-entry.

requester. A term which describes the node that initiates a bus transaction and transfers an address and command to the responder.

responder. A term which describes the node that completes a bus transaction and returns status to the requester.

scatter array. A contiguous array of elements that specifies a set of discontiguous address spaces to which a single logical data transfer is performed. A scatter array consists of one or more elements, where each scatter element contains a 64-bit address pointer and an unsigned 32-bit count.

SerialBus. Refers to the draft document being produced by the IEEE P1394 working group. This defines an inexpensive serial interconnect that can be used as an alternate control or diagnostic path, as an I/O connection, or even in place of a parallel bus in some systems.

Scalable Coherent Interface. A term which refers to the IEEE Std 1596-1992 Scalable Coherent Interface.

status entry. An entry in a status list that contains status information and may optionally contain address information.

status list. A singly-linked list of status entries. One or more targets may append status entries to a status list. A single initiator is responsible for extracting status entries from a status list.

target. An entity that receives commands from one or more initiators, acts on those commands and optionally returns status, using the I/O process.

transaction. See *bus transaction.*

unit. A sub-component of the node that provides a processing, memory, or I/O functionality. After the node has been initialized (typically by generic software), the unit provides the register interface which is accessed by I/O driver software. The units normally operate independently of each other, and do not affect the operation of the node upon which they reside. Note that one node could have multiple units (for example: processor, memory, and SCSI controller).

write operation. The data-transfer phase in the execution of a write command entry that copies data from the bus-accessible space into device-specific space. A write operation consists of one or more read transfers.

write transaction. A bus transaction that transfers data from the requester to the responder.

write transfer. A transfer of data from a device-specific address space into a contiguous range of bus addresses. Write transfers are performed by the target in the process of executing a read (memory-to-target) or copy (memory-to-memory) command-entry.

1.3 Bit, Byte, and Quadlet Ordering

This document defines registers and memory locations which are 4 bytes (or larger) in size. To ensure interoperability across bus standards, the ordering of the bytes within these locations is defined by their relative addresses, not their time slot or physical position on the bus. Bus bridges are similarly expected to route data bytes from one bus to another based on their addresses, not their physical position on a bus. The routing of data bytes based on their address is called address-invariance.

To support the address-invariance model, this standard specifies the mapping of data-byte addresses to bytes within the multi-byte registers and memory-resident data locations. For a quadlet location, the data byte with the smallest address is the most significant, as illustrated in Figure 1-5.

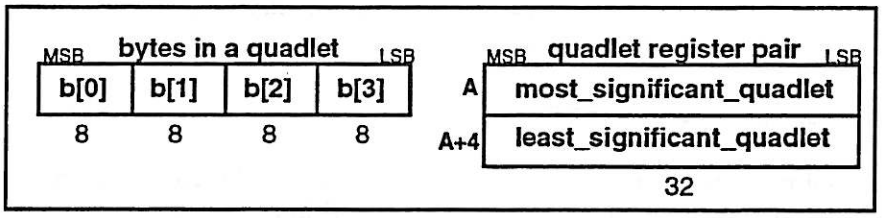

Figure 1-5: Byte and Quadlet Ordering

Since 64-bit addressing is supported throughout this standard, many values are stored as quadlet-register pairs. For consistency, the quadlet register with the smallest address is also the most significant, as illustrated above.

For most of the quadlet locations, the size of all fields within the quadlet are specified; the bit position of each field is implied by the size of fields to its right or left. This labelling convention is more compact than bit-position labels, and avoids the question of whether 0 should be used to label the most or least significant bit.

1.4 Numerical Values

Decimal, hexadecimal, and binary numbers are used within this document. For clarity, decimal numbers are generally used to represent counts, hexadecimal numbers are used to represent addresses, and binary numbers are used to describe bit patterns within binary fields.

Decimal numbers are represented in their standard 0, 1, 2, ... format. Hexadecimal numbers are represented by a string of one or more hexadecimal (0-9,A-F) digits followed by the subscript 16. Binary numbers are represented by a string of one or more binary (0,1) digits, followed by the subscript 2. Thus the decimal number "26" may also be represented as "$1A_{16}$" or "11010_2".

1.5 Address Pointers

1.5.1 4/8-byte Aligned Pointers

SerialBus uses 64-bit addresses to access objects in bus address space. In general, a 64-bit address of a data structure can be arbitrarily aligned.

The I/O process defines special objects that have certain data alignment requirements. An example is the wakeup register (described in Section 5.1.2) which must be located at 4-byte-aligned addresses. When used within CCU-Architecture-defined data structures, these addresses have a the format illustrated in Figure 1-6.

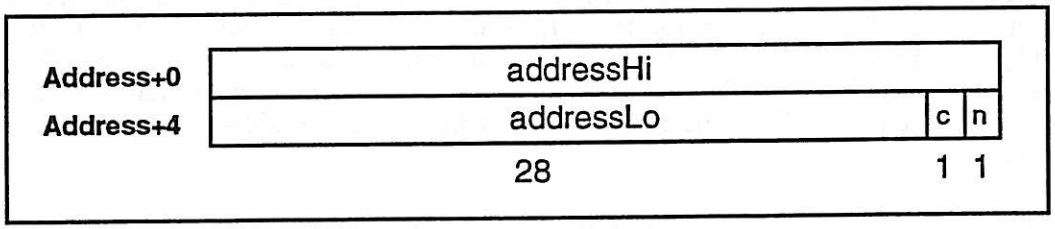

Figure 1-6: 4/8-Byte Aligned Pointer Format

The null bit **n** is 0 or 1 if the address pointer is valid and invalid respectively. The cache bit **c** is 0 or 1 if the address pointer is valid and invalid respectively.

1.5.2 16-byte Aligned Pointers

The I/O process defines other data objects which are required to be 16-byte aligned, such as the command block (described in Section 4.1). When used within the CCU-Architecture-defined data structures, these addresses have the format illustrated in the Figure 1-7.

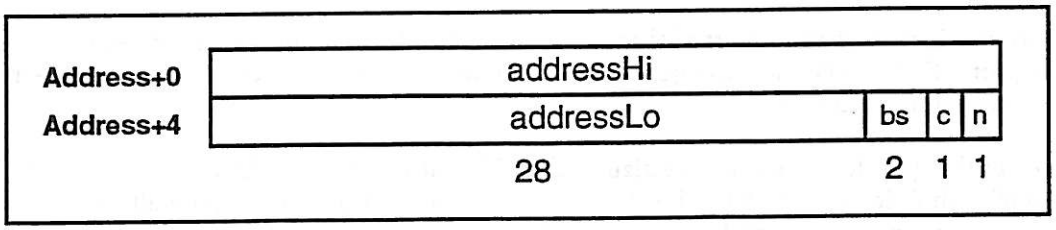

Figure 1-7: 16-Byte Aligned Pointer Format

For byte-aligned pointers, the associated *bs*, *c*, and *n* fields are contained in listControl, a nearby quadlet location in the command entry. Regardless of their location, these fields limit the bus transactions that are permitted when accessing the data to which the pointer points.

The null bit **n** is 0 or 1 if the address pointer is valid or invalid respectively. The cache bit **c** is 0 or 1 if the address pointer is valid or invalid respectively.

For non-coherent transfers, the 2-bit block-size field **bs** field values specifies the maximum bus-transaction size, as specified in Table 1-1, below.

n	c	bs	description
0	0	0	Uncached; read4 and write4 transactions maximum
"	"	1	Uncached; read16 and write16 transactions maximum
"	"	2	Uncached; read64 and write 64 transactions maximum
"	"	3	Uncached; transactions of any size up to maximum for bus speed
0	1	all	Cache-coherent access (not applicable to SerialBus)
1	0,1	all	Null pointer (address is invalid)

Table 1-1: Standard Aligned-Address Field Values

1.6 Memory-Access Operations

1.6.1 List and Management Operations

The maintenance of the lists and the other management functions specified by the I/O process require the underlying bus to support certain memory-access operations. Since many portions of the list-access protocols are specified as C code, these memory-access operations are also defined by C-code routines; prototypes for these routines are listed in Table 1-2, below.

```
Quadlet Read4(Octlet address);

Octlet  Read8(Octlet address);

void    Read64(Octlet address, Byte *dataBlock);

void    Write4(Octlet address, Quadlet data);

void    Write8(Octlet address, Octlet data);

void    Write64(Octlet address, Byte *dataBlock);

Octlet  MaskSwap8(Octlet address, Octlet data, Octlet mask);

Octlet  CompareSwap8(Octlet address, Octlet data, Octlet test);
```

Table 1-2: Memory-Access Routines

The `Read4()` routine shall return the `Quadlet` stored in the location specified by `address`. The `address` value shall be an integer multiple of 4.

The **Read8()** routine shall return the octlet stored in the location specified by address. The address value shall be an integer multiple of 8.

The **Read64()** routine returns 64 bytes of data, starting from the location specified by address. The size of the dataBlock array shall contain at least 64 bytes. The address value shall be an integer multiple of 16.

The **Write4()** routine shall store the quadlet data value in the location specified by address. The address value shall be an integer multiple of 4.

The **Write8()** routine shall stores the octlet data value in the location specified by address. The address value shall be an integer multiple of 8.

The **Write64()** routine moves the 64 bytes in dataBlock to locations starting at address. The size of the dataBlock array shall contain at least 64 bytes. The address value shall be an integer multiple of 16.

The **MaskSwap8()** routine performs a partial swap of data with the value in the location specified by address. Data is stored in those bits of the addressed location corresponding to 1 bits in the mask value.

The **CompareSwap8()** routine compares the octlet test value with the octlet in the location specified by address; if they are equal, the octlet data value is stored in the location specified by address. In all cases, the previous octlet value at that address is returned. The address value shall be an integer multiple of 8.

The memory-access routines listed in Table 1-5 generate actual bus activity, arbitration, request packet, etc. However, their functionality can be illustrated by the following C-code, which assumes all resources are memory-mapped. The C-code is shown in Listing 1-1, and the necessary support code can be found in the Appendix.

```
/* Listing 1-1: Memory-Access Routines          */
/* CODE_BEGIN */
 Quadlet
Read4(Octlet address)
{  Quadlet result;

   /* Generates a read4 transaction, with the following effects */
   assert((address%4)==0);
   result= *((Quadlet *)address);
   return(result);
};

 Octlet
Read8(Octlet address)
{  Octlet result;

   /* Generates a read8 transaction, with the following effects */
   assert((address%8)==0);
   result= *((Octlet *)address);
   return(result);
};
```

```
 void
Read64(Octlet address, Byte *dataBlock)
{  int i;

   /* On SerialBus, generates read64 transaction, with the following effects.
    * On aligned buses, like NuBus or Scalable Coherent Interface, four
    * read16 transactions are used on addresses that are not 64-byte aligned */
   assert((address%16)==0);
   for (i= 0; i<64; i+= 1)
      *(dataBlock+i)= *((Byte *)address+i);
};

 void
Write4(Octlet address, Quadlet data)
{
   /* Generates a write4 transaction, with the following effects */
   assert((address%4)==0);
   *((Quadlet *)address) = data;
};

 void
Write8(Octlet address, Octlet data)
{
   /* Generates a write8 transaction, with the following effects */
   assert((address%8)==0);
   *((Octlet *)address) = data;
};

 void
Write64(Octlet address, Byte *dataBlock)
{  int i;

   /* On SerialBus, generates write64 transaction, with the following effects.
    * On aligned buses, like NuBus or Scalable Coherent Interface, four
    * write16 transactions are used on addresses that are not 64-byte aligned*/
   assert((address%16)==0);
   for (i= 0; i<64; i+= 1)
      *((Byte *)address+i) = *(dataBlock+i);
};

 Octlet
MaskSwap8(Octlet address, Octlet data, Octlet mask)
{  Octlet old;

   /* Generates a maskSwap8 transaction, as defined by CSR Architecture.
    * On SerialBus, the (data&mask) operation is performed by the
    * requester before the request subaction is generated. However,
    * this transaction always has the following (bus-independent) effects */
   assert((address%8)==0);
   old= *((Octlet *)address);
   *((Octlet *)address)= (data & mask) | (old & ~mask);
   return(old);
};
```

```
 Octlet
CompareSwap8(Octlet address, Octlet data, Octlet test)
{  Octlet old;

   /* Generates a compareSwap8 transaction, with the following effects */
   assert((address%8)==0);
   old= *((Octlet *)address);
   if (old==test) *((Octlet *)address) = data;
   return(old);
};
/* CODE_END - BUS_TRANSACTIONS */
```

2. List Update Protocols

2.1 Command and Status List Structures

2.1.1 List Structures

Both the command and status lists have the same structure, shown in Figure 2-1, and are manipulated by the same algorithms. The only difference between the two lists is the information in the `entryInfo` field. All of the elements of the list shall be located in bus-accessible space and shall be 16-byte aligned.

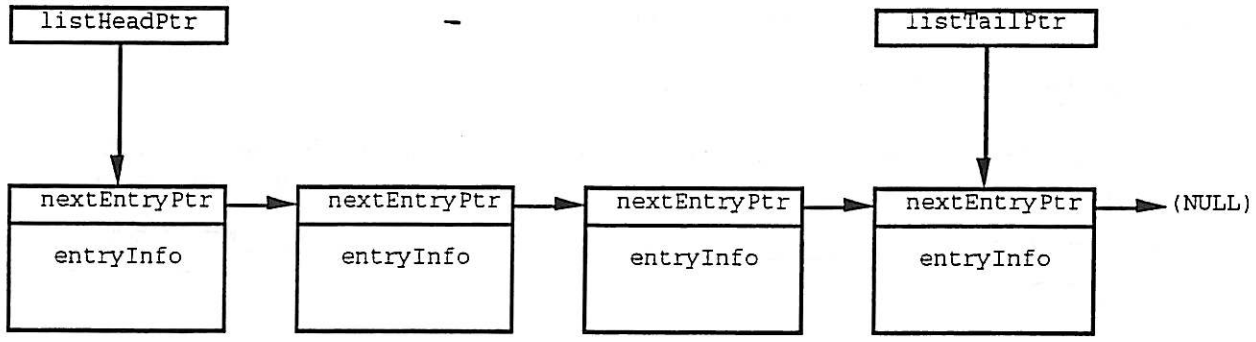

Figure 2-1: List Structure

The `listHeadPtr` element is the pointer to the first entry in the list. In the context of a command list, this element is called a command head pointer, `commandHeadPtr`. In the context of a status list, this element is called a status head pointer, `statusHeadPtr`.

A list entry consists of two parts. The next entry pointer, `nextEntryPtr`, and the entry-dependent information, `entryInfo`. In the context of a command-list entry, the next entry pointer is called a next command pointer, `nextCommandPtr`, and the entry-dependent information is called the command information, `commandInfo`. In the context of a status-list entry, the next entry pointer is called a next status pointer, `nextStatusPtr`, and the entry-dependent information is called the `statusInfo`.

The `listTailPtr` element is the pointer to the last entry in the list. In the context of a command list, this element is called a command tail pointer, `commandTailPtr`. In the context of a status list, this element is called a status tail pointer, `statusTailPtr`.

The summary of list element names is given in Table 2-1.

generic list	command list	status list
listHeadPtr	commandHeadPtr	statusHeadPtr
listTailPtr	commandTailPtr	statusTailPtr
nextEntryPtr	nextCommandPtr	nextStatusPtr
entryInfo	commandInfo	statusInfo

Table 2-1: List Element Names

2.1.2 Empty (Zero-Entry) List Structures

The list is initially empty, in that it contains no entries. In an empty list the value of `listTailPtr` is the address of `listHeadPtr` and the value of `listHeadPtr` is NULL, as illustrated in Figure 2-2.

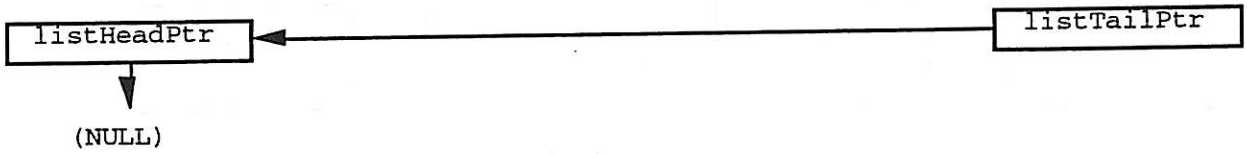

Figure 2-2: Empty List

2.2 List Append Operation

2.2.1 Appending One Entry

To append entry into the list, the appender allocates the entry and initializes its `nextEntryPtr` value to NULL. The appender then swaps the address of the new entry with the list's `listTailPtr` value and saves the returned value (the old `listTailPtr` value) in a local `oldlistTailPtr` location. This sequence is shown in Figure 2-3.

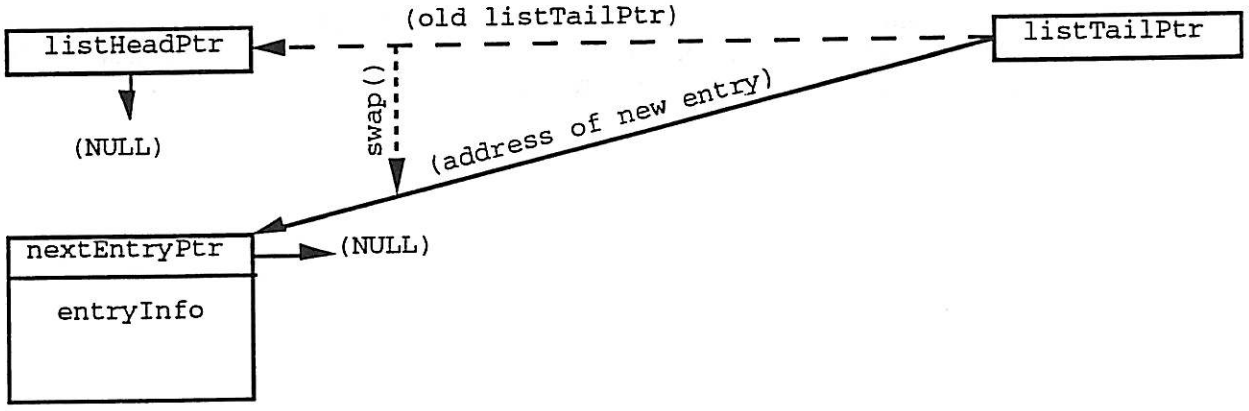

Figure 2-3: Adding First Entry to List, Phase 1

To complete the appending of the new entry, the appender writes the address of the new entry to the address specified by the previously returned `oldListTailPtr` value. After the append completes, the appender typically signals the associated extracter by writing a pre-specified quadlet wakeup value to a pre-specified wakeup register address, as illustrated in Figure 2-4.

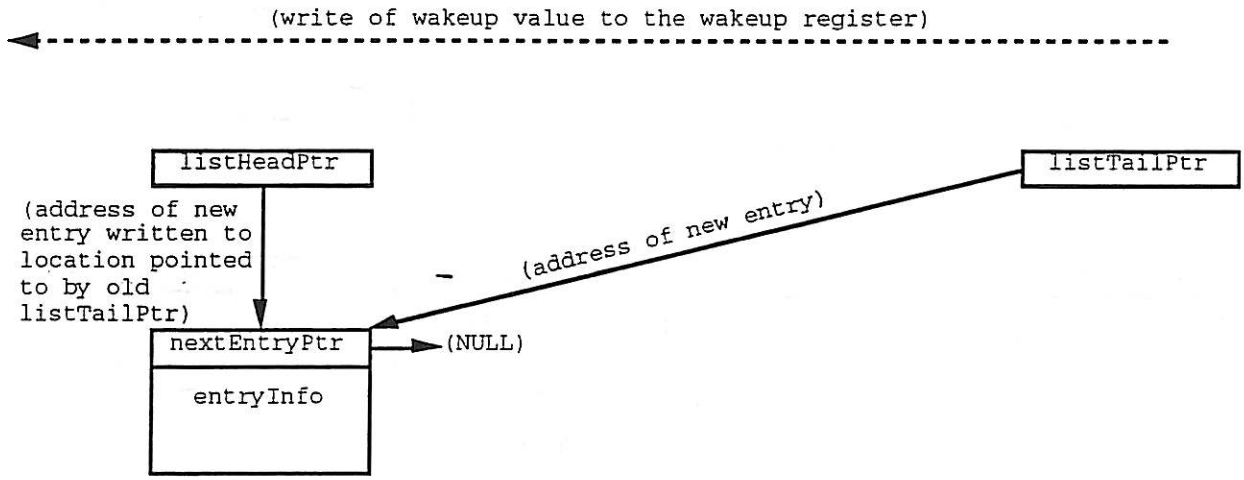

Figure 2-4: Appending First Entry to List, Phase 2

2.2.2 Appending Multiple Entries

When the appender appends multiple entries simultaneously, the append algorithm is modified slightly. The appender pre-links the entries to be added to the list, as shown in Figure 2-5. The appender then swaps the address of the last entry in the mini-list with the list's `listTailPtr` value and saves the returned value (the old `listTailPtr` value) in a local `oldTailPtr` location, as shown in Figure 2-5.

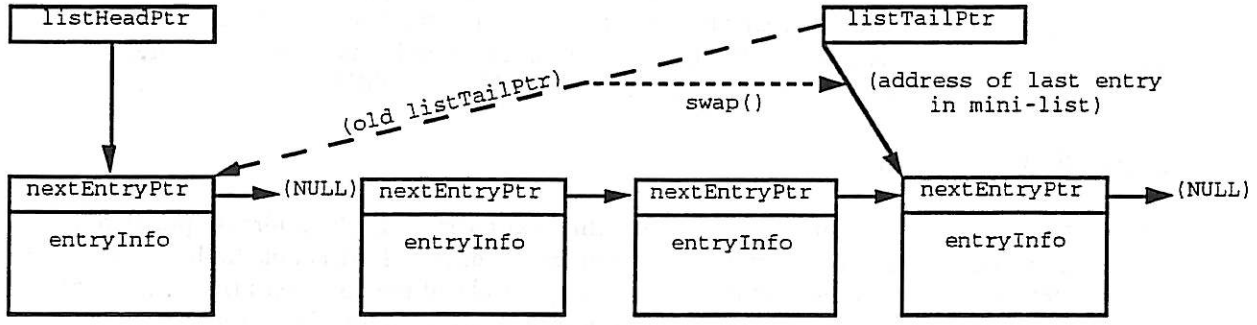

Figure 2-5: Adding a Mini-List to List, Phase 1

To complete the appending, the appender then writes the address of the first entry in the mini-list to the address specified by the previously returned `oldListTailPtr` value. After the append is completed, the appender may send a wakeup signal to the extracter, as illustrated in Figure 2-6.

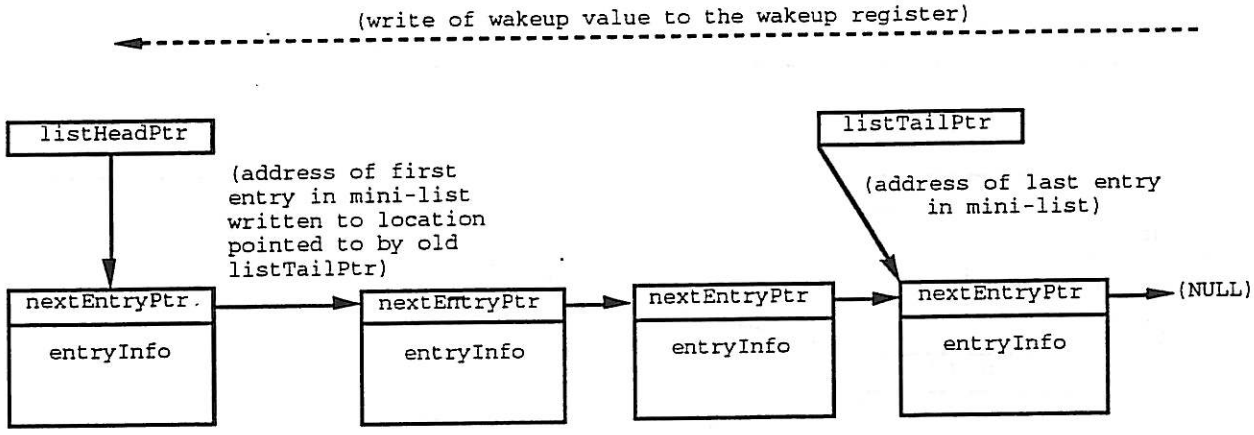

Figure 2-6: Appending a Mini-List to List, Phase 2

2.3 List Extract Operation

2.3.1 Extractor Wakeups

For the extractor, the wakeup register address and the wakeup value are expected to uniquely identify which list contains the new entry. When checking that list, the extracter reads the `listHeadPtr` to see if it is valid, i.e. non-NULL. When `listHeadPtr` is NULL, the extractor waits for the next wakeup signal.

When `listHeadPtr` is not NULL, the extractor attempts to extract one or more entries, as described in the following sections. To simplify this concept, the textual descriptions consider the case where only one entry is extracted. A more efficient extraction process can be used to extract multiple command entries, as specified by the C-code specification of Listing TBD.

2.3.2 List Extractions

If `listHeadPtr` is not NULL, the extractor reads the next entry from the address specified by the `listHeadPtr`. If the `nextEntryPtr` value in this entry is not NULL (it is not the last list entry), the extractor fetches any extended portions of the entry into local memory, sets the memory-resident `listHeadPtr` to the `nextEntryPtr` value, and processes the local copy of the entry.

If the `nextEntryPtr` value is NULL, this extractor initially assumes this is the last entry in the list. The extracter saves the shared `listHeadPtr` value in a local `queueHeadShadowPtr` location and writes a NULL value to the shared `listHeadPtr` location. Using the indivisible `CompareSwap8` operation, the `listTailPtr` value is compared to the `queueHeadShadowPtr` value and (if they are equal) the `listTailPtr` value is set to the address of `listHeadPtr`.

If the `CompareSwap8` operation succeeds, the extractor fetches any extended portions of the entry into local memory and processes the local copy of this entry. The extractor waits for the next wakeup signal.

If the `CompareSwap8` operation fails (because other entries are in the process of being appended), the extractor copies the `queueHeadShadowPtr` value back to the `listHeadPtr` location and waits for the next wakeup signal.

2.4 Append and Extract Specification

2.4.1 Append and Basic Extract

There are three basic algorithms that describe how to append an entry to a list and how to extract an entry from a list: `Append()`, `ExtractOne()`, and `ExtractMany()`. The `ExtractOne()` routine illustrates how individual command-list entries can be simply extracted; the `ExtractMany()`. routine illustrates how extraction of multiple command-list entries can be more efficient. The C-code for both are shown in Listing 2-1. The necessary simulation environment for compiling and testing this code is located in the Appendix.

```
/*              Listing 2-1: Append and Extract List Operations              */
/* BEGIN_CODE */
#include "CCU.h"
#include "BusXacts.h"
#include "Address.h"
#define NULL (~(Octlet)0)                          /* All 1's is NULL value */
#define NULL_BIT 1                                 /* LSB is the null bit */
#define NULL_SET 1                                 /* 1 is the null-bit value */
#define ADDRESS_MASK (~(Octlet)0XF)                /* Four LSBS are "special" */
enum {                                             /* Append status codes */
   DONE_GOOD,                                      /* Completed successfully */
   DONE_NULL                                       /* Tail pointer was null */
} ListUpdateStatus;

/* Append function assumes that the client has allocated an entry and
 * filled in the entry_info and set the *next_entry to NULL. */
 ListUpdateStatus
Append(Octlet queueTailHandle,Octlet firstEntry,Octlet lastEntry)
{  Octlet oldTailPtr, newTailPtr;

   /* Update tailPtr to point to end of appended list,
    * but leave the null bit unchanged. */
   oldTailPtr= MaskSwap8(queueTailHandle,lastEntry,~((Octlet)NULL_BIT);
   if ((oldLastEntry&NULL_BIT)==NULL_SET)
      return(DONE_NULL)
   newTailPtr= oldTailPtr & ADDRESS_MASK;          /* Ignore 4 LSBs */
   Write8(newTailPtr,firstEntry);
   return(DONE_GOOD);

   /* The wakeup write is done at the next highest level,
    * since some appenders may not always send a wakeup.
    */
};
```

```
/* The ExtractOne() function is called after a wakeup indicates the
 * list state may have changed. The routine "ProcessEntry()" is called to
 * fetch any extended portions of the entry, to "process" the
 * entry contents and to return a pointer to another usable data block. */
 ListUpdateStatus
ExtractOne(Octlet queueHeadHandle, Octlet queueTailHandle, Entry *localEntry)
{  Octlet queueHeadShadowPtr, oldQueueTailPtr;

   if ((queueHeadHandle & NULL_BIT) == NULL_SET)
      return(DONE_NULL);                          /* Don't use a null handles */

   /* queueHeadShadowPtr is the extracter's local copy of *queueHeadPtr */
   queueHeadShadowPtr= Read8(queueHeadHandle);

   /* If the head pointer is null, don't do anything */
   if ((queueHeadShadowPtr & NULL_BIT) == NULL_SET)
      return(DONE_GOOD);

   /* If the head pointer is valid, copy the entry into local storage */
   Read64(queueHeadShadowPtr, (Byte *)localEntry);

   /* While there are more known entries, process the localEntry copy.
    * Then update local queueHeadShadowPtr address and read next list entry. */
   if (((localEntry->nextEntry & NULL_BIT) != NULL_SET) {
      queueHeadShadowPtr= localEntry->nextEntry;
      /* Process localEntry data, returning pointer to additional storage */
      (void)ProcessEntry(localEntry);
      Write8(queueHeadHandle,queueHeadShadowPtr);
      return(DONE_GOOD);
   }

   /* The previously-fetched entry appears to be the last list entry.
    * Set queueHeadPtr value to NULL, in expectation of emptying the list */
   Write8(queueHeadHandle,(Octlet)NULL);

   /* Perform the compare&swap on the tail to empty the list */
   oldQueueTailPtr=
    CompareSwap8(queueTailHandle, queueHeadHandle, queueHeadShadowPtr);

   if (oldQueueTailPtr!= queueHeadShadowPtr) {
      /* If compare&swap fails, write the local queueHeadShadowPtr value
       * to its memory-resident queueHeadPtr storage location. */
      Write8(queueHeadHandle,queueHeadShadowPtr);
      return(DONE_GOOD);
   }
   /* If the compare and swap succeeds, process last entry & return */
   localEntry = ProcessEntry(localEntry);
   return(DONE_GOOD);
};
```

```
/* The ExtractMany() function is like ExtractOne(), but the routine
 * ProcessEntry() is called repetitively while processing command entries */
 ListUpdateStatus
ExtractMany(Octlet queueHeadHandle, Octlet queueTailHandle, Entry *localEntry)
{  Octlet queueHeadShadowPtr, oldQueueTailPtr;

   if ((queueHeadHandle & NULL_BIT) == NULL_SET)
      return(DONE_NULL);       /* Don't use a null handles */

   /* queueHeadShadowPtr is the extracter's local copy of *queueHeadPtr */
   queueHeadShadowPtr= Read8(queueHeadHandle);

   /* If the head pointer is null, don't do anything */
   if ((queueHeadShadowPtr & NULL_BIT) == NULL_SET)
      return(DONE_GOOD);

   /* If the head pointer is valid, copy the entry into local storage */
   Read64(queueHeadShadowPtr, (Byte *)localEntry);

   /* While there are more known entries, process the localEntry copy.
    * Then update local queueHeadShadowPtr address and read next list entry.
*/
   if (((localEntry->nextEntry & NULL_BIT) != NULL_SET) {
      queueHeadShadowPtr= localEntry->nextEntry;
      /* Process localEntry data, returning pointer to additional storage */
      localEntry= ProcessEntry(localEntry);
      if ((localEntry)==NULL) {     /* Local storage exhausted, */
                              Write8(queueHeadHandle,queueHeadShadowPtr);
                              return(DONE_GOOD);      /* Save state and return
*/
      }
      /* Mask the 4 LSBs of entry pointers; these have special meanings. */
      Read64(queueHeadShadowPtr & ADDRESS_MASK, (Byte *)localEntry);
   }

   /* The previously-fetched entry appears to be the last list entry.
    * Set queueHeadPtr value to NULL, in expectation of emptying the list */
   Write8(queueHeadHandle,(Octlet)NULL);

   /* Perform the compare&swap on the tail to empty the list */
   oldQueueTailPtr=
    CompareSwap8(queueTailHandle, queueHeadHandle, queueHeadShadowPtr);

   if (oldQueueTailPtr!= queueHeadShadowPtr) {
      /* If compare&swap fails, write the local queueHeadShadowPtr value
       * to its memory-resident queueHeadPtr storage location. */
      Write8(queueHeadHandle,queueHeadShadowPtr);
      return(DONE_GOOD);
   }
   /* If the compare and swap succeeds, process last entry & return */
   localEntry = ProcessEntry(localEntry);
   return(DONE_GOOD);
}
/* END_CODE */
```

2.5 Wakeup Registers

To improve efficiency and ensure forward progress, the standardized wakeup registers supports the immediate acceptance of an arbitrary number of quadlet-write transactions. Although all wakeup events are immediately queued, the processing of the queued events may be delayed (based on vendor-dependent wakeup priority servicing protocols).

To ensure interoperability between processors and I/O controllers provided by different vendors, the CSR Architecture restricts the functionality of wakeup registers. Wakeup events are constrained to be **write4** transactions (which transfer 32-bit data values), whose address is a register within the affected unit architecture. The wakeup register address is 64 bits in size, since the wakeup register could be located on any node, and unit-dependent wakeup registers could be most any place on the node.

For the target unit, the ROM entries specify where the unit's register set (which includes the wakeup register) is located. For the initiator unit, the command entry specifies which wakeup register address and data value shall be used by the target.

2.5.1 Target Wakeup Register Model

To simplify the design target units, one wakeup register is shared for all of the target's command lists, as illustrated in Figure 2-7. The value which is written to this wakeup register selects which of N wakeup bits is set, where N is the number of command lists supported by the unit. Each write can be processed immediately, by setting the addressed wakeup bit, so there is never a need to return a busy status.

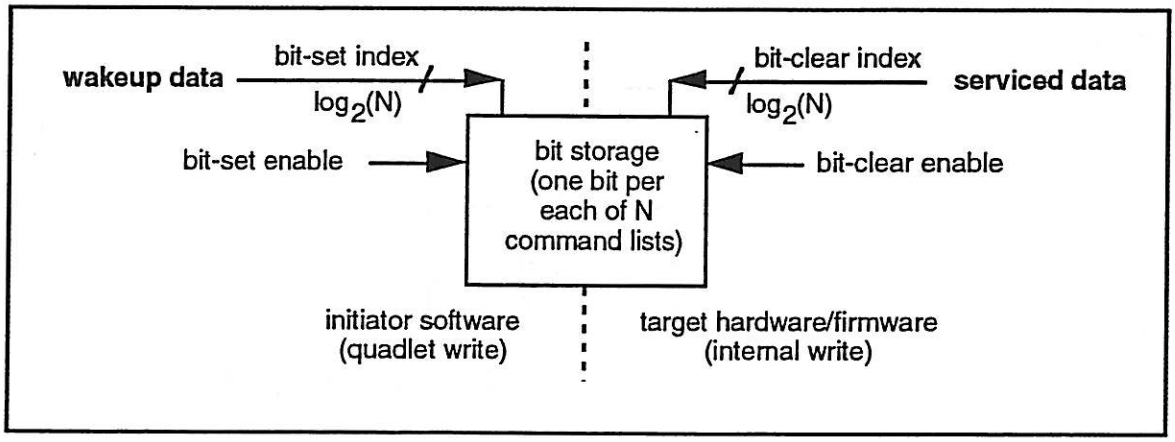

Figure 2-7: Bit-Indexed Wakeup Model

2.5.2 Initiator Wakeup Register Model (Broadcast Capable)

Since the command entry specifies which wakeup register address and data value shall be used, a variety of initiator wakeup register formats can be supported. The initiator is expected to support a few status lists, one for each priority level. Simple initiators are expected to support the standardized INTERRUPT_TARGET register (as defined in the CSR Architecture), since this can support broadcast as well as directed interrupts.

The INTERRUPT_TARGET register provides one bit of storage for each of 32 interrupt groups; an interrupt is queued by setting the corresponding interrupt-group bit. In bus-accessible writes to this INTERRUPT_TARGET register, the data bits are OR'd with the 32 interrupt-pending bits. In initiator-internal updates of the INTERRUPT_TARGET register, the data bits can be selectively cleared, as illustrated in Figure 2-8.

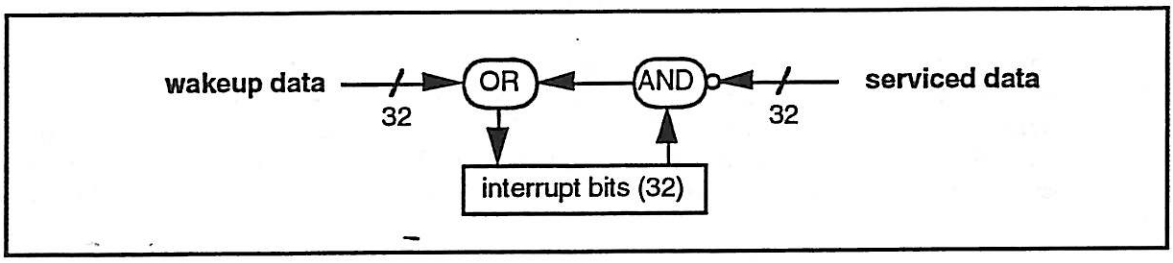

Figure 2-8: Bit-Mapped Wakeup Model

2.5.3 Alternate Initiator Wakeup Register Model (Directed Only)

The initiator is not required to support the (optional) INTERRUPT_TARGET register, nor is it required to support bit-mapped interrupt storage. Since the command entry specifies which wakeup register address and data value shall be used, an initiator can request that an arbitrary 32-bit value (called an interrupt vector) be returned to its wakeup register.

Initiators which support interrupt vectors are expected to provide a FIFO to save the received interrupt-vector values. A write to the wakeup data register saves the data value in the FIFO, and a unit-internal service access removes the most-recent data value from the FIFO, as illustrated in Figure 2-9.

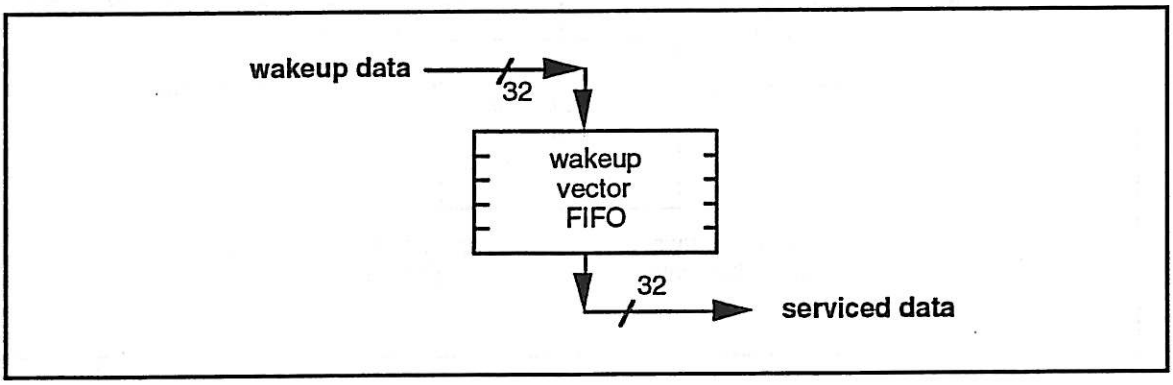

Figure 2-9: Vector-Fifo Wakeup Model

Saving interrupt vectors simplifies the common I/O driver software, which can can quickly and simply dispatch to the proper interrupt routine. However, saving these vectors requires special hardware FIFO support and (since the size of the FIFO is finite), software needs to ensure that the FIFO does not overflow. These overflow-avoidance constraints complicate the software and limit the number of simultaneously active status entries to the size of the interrupt-vector FIFO.

3. Command-List Structure

3.1 Command Entries

A command entry is a data structure that the initiator creates for the target. The information contained in a command entry describes some action for the target to perform. Section 4 contains the complete definition of the fields in a command entry.

Command entries always occur in linked lists. The first field in all command entries, called the nextCommandPtr, contains the address of the next command entry in the linked list. For command entries that have no command entries following them, the next command entry pointer contains a NULL value.

A command entry may completely describe an action for a target to perform. In this case, the command contains sufficient information for the target to act and optionally return status to the initiator upon command completion.

It is also possible to create a linked list of command entries which are logically related to each other. Such a linked list is referred to as a command group. The following section describes command groups.

3.2 Command Groups

A command group is a set of one or more command entries for which there is one status entry. Initiators may choose to use a command group instead of a linked list of individual commands in situations where more than one command is required to describe a given operation. In other words, the commands in a command group are (in normal operation) executed together.

The first and last command entries in a command group differ in how the statusEntryPtr and statusTailHandle fields are filled in. The *first* command-group entry has a valid statusEntryPtr value and a NULL statusTailHandle value. The *last* command-group entry has a NULL statusEntryPtr value and a valid statusTailHandle value. All commands in between the first and last entries have NULL statusEntryPtrs and statusTailHandle values. Figure 3-1 shows this relationship graphically.

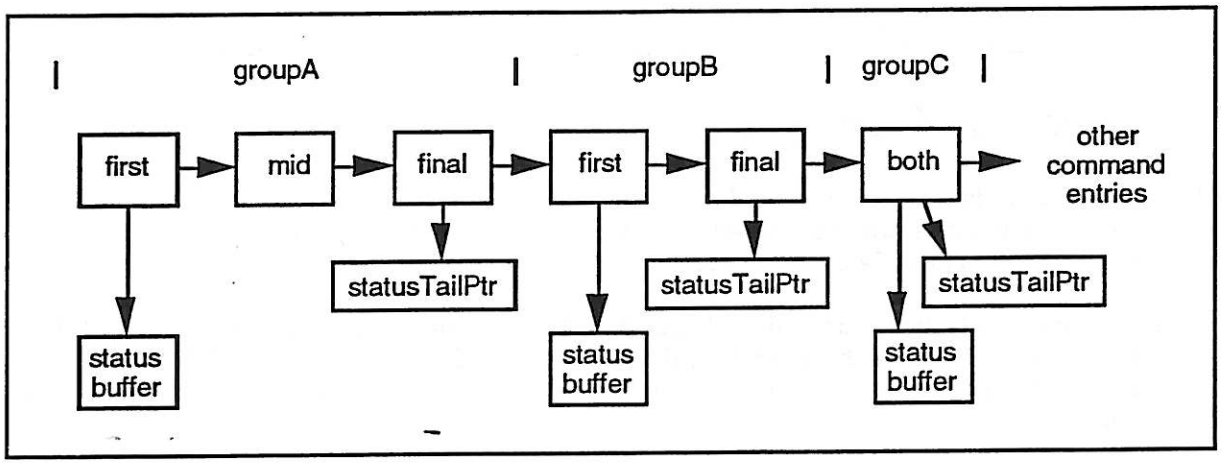

Figure 3-1: Command-Group Structures

The above figure shows a linked list of command groups. Notice that the last command group contains exactly one command entry. By convention, all commands are members of a command group, but in some cases the command group contains only one entry.

For any command entry in a command group, if the initiatorWakeupPtr value is not NULL, the target shall generate a wakeup to the initiator by writing the initiatorWakeupValue to the address contained in the initiatorWakeupPtr field.

When the target detects the beginning of a command group (signalled when a command entry is encountered in which the statusEntryPtr value is valid, and the statusTailHandle is NULL), it shall store the statusEntryPtr value, then process the command entry. The target continues processing subsequent commands, accumulating status in the status entry pointed to by the saved statusEntryPtr value, if necessary.

If an error occurs during the processing of command-group entries, the target skips subsequent command-list entries in the command group until it finds the final command-group entry (i.e., the next command entry containing a valid statusTailPtr value). The target then appends the updated status entry (at the previously-specified statusEntryPtr address) to the desired status list (whose address is specified by the statusTailHandle value). If the initiatorWakeupPtr field in the last command group command entry is valid, the target writes the initiatorWakeupValue to the address contained in the initiatorWakeupPtr field. The target may then continue processing subsequent command-list entries in the list.

If the target encounters no errors during the processing of commands in the command group, it continues until it encounters the last entry in the command group (a command entry in which the statusTailHandle value is not NULL). The target processes this last command, then appends the status entry pointed to by the saved statusEntryPtr value to the status list whose tail handle is contained in the statusTailHandle value in the last entry. If the initaitorWakeupPtr field in the last entry is not NULL, the target writes the value in the initiatorWakeupValue field to the address contained in the initiatorWakeupPtr field.

The initiator shall pre-initialize the contents of the status buffer with an expected target-status value. The target should only update the status buffer in cases where it must return an unexpected status value.

3.3 Data Transfer Addresses

3.3.1 Direct and Indirect Address Blocks

The read and write data-transfer commands copy data between the initiator's bus-accessible space and the device-specific space. For example, the device-specific address space may map into logical disk addresses. The initiatorBufferPtr and initiatorBufferLength fields in the command entry always describe a buffer (or buffers) in the bus-accessible space, while the targetBufferPtr and targetBufferLength fields in the read and write command entries describe a buffer (or buffers) in the device-specific address space.

The initiator and target buffer pointer and length fields may describe a physically contiguous buffer, or one that is scattered within the address space. A physically contiguous buffer is one that begins at the address contained in the buffer pointer field and extends to higher addresses for the number of bytes contained in the buffer length field.

A scattered data buffer is one that is composed of multiple smaller contiguous blocks of memory. Each block of memory in the buffer is described by a separate buffer pointer and length field. Each buffer pointer and length field pair form a single entry in a scatter array. The scatter array is a data structure that is contiguous and which contains one or more scatter entries. The format of a scatter array is described in section 4.

If the initiator or target buffer is scattered, the corresponding buffer pointer and buffer length fields contain the start address and length of the scatter list. Either the target buffer or the initiator buffer or both may be scattered.

For example, the *initiatorBufferPtr* address could point to a contiguous data block and the *targetBufferPtr* address could point to a scatter list, as illustrated in Figure 3-2:

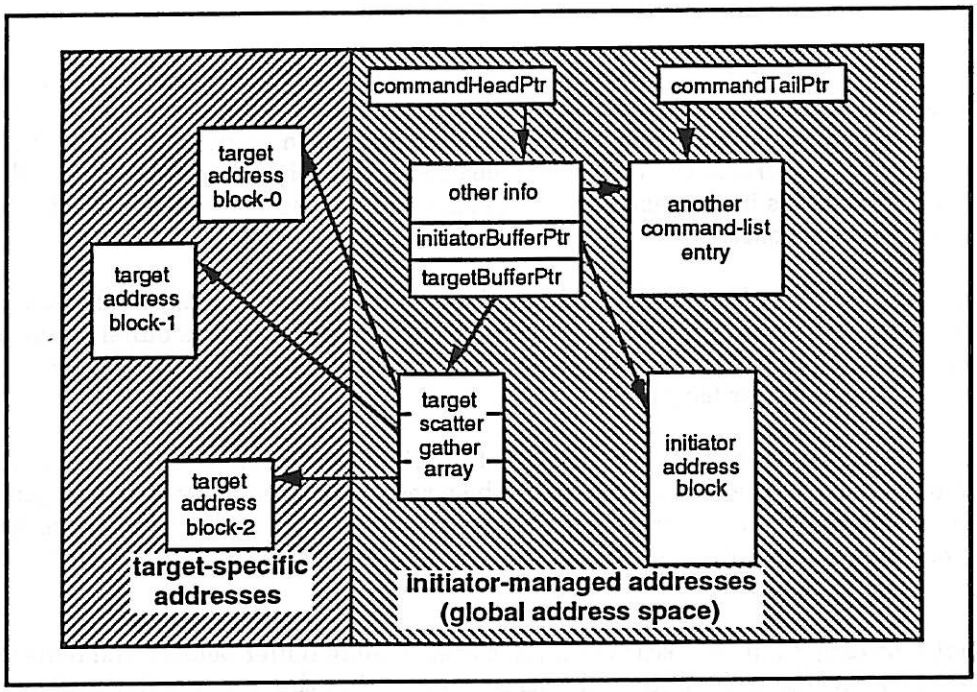

Figure 3-2: Address-Block Structures

Data transfers may be performed between two identically-sized sets of address blocks, and either address block set may contain direct or indirect address blocks. To better illustrate this concept, several forms of data-block specifications are illustrated in the following sections. Within the context of these examples, all numerical values are represented in hexadecimal format and the data bytes to be transferred are labelled with the letters *a* through *v*.

3.3.2 Data-Block Transfer Examples

3.3.2.1 Direct-Initiator/Direct-Target Transfers

Data may be transferred between a directly-specified range of initiator addressees and a directly-specified range of target addresses, as illustrated in Figure 3-3. The command entry fields for the transfer are shown in Table 3-1. Although arbitrarily-aligned initiator addresses shall be supported, transfers are expected to be more efficient when the initiator addresses are 64-byte aligned. The target address alignment restrictions are device dependent.

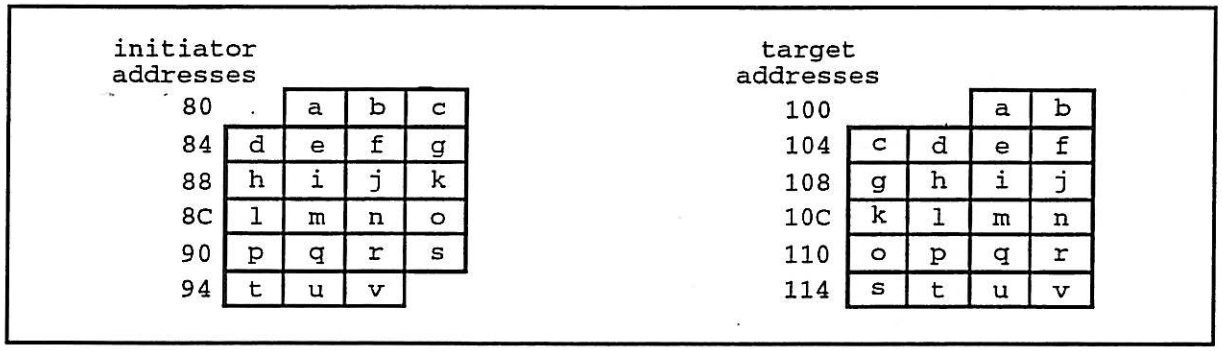

Figure 3-3: Data Addressing, Direct-Initiator/Direct-Target Transfers

field	value
initiatorBufferPtr	81_{16}
initiatorBufferLength	16_{16}
targetBufferPtr	102_{16}
targetBufferLength	16_{16}

Table 3-1: Command Entry Fields, Direct-Initiator/Direct-Target Transfers

3.3.2.2 Indirect-Initiator/Direct-Target Transfers

Data may be transferred between an indirectly-specified set of initiator addresses and one directly-specified range of target addresses, as illustrated in Figure 3-4. The command entry fields for the transfer are shown in Table 3-2. The *listControl.iSa* bit within the standard-header portion of the command entry specifies whether the initiator addresses are contiguous or scattered; see Sections 4.1. and 7.1 for details.

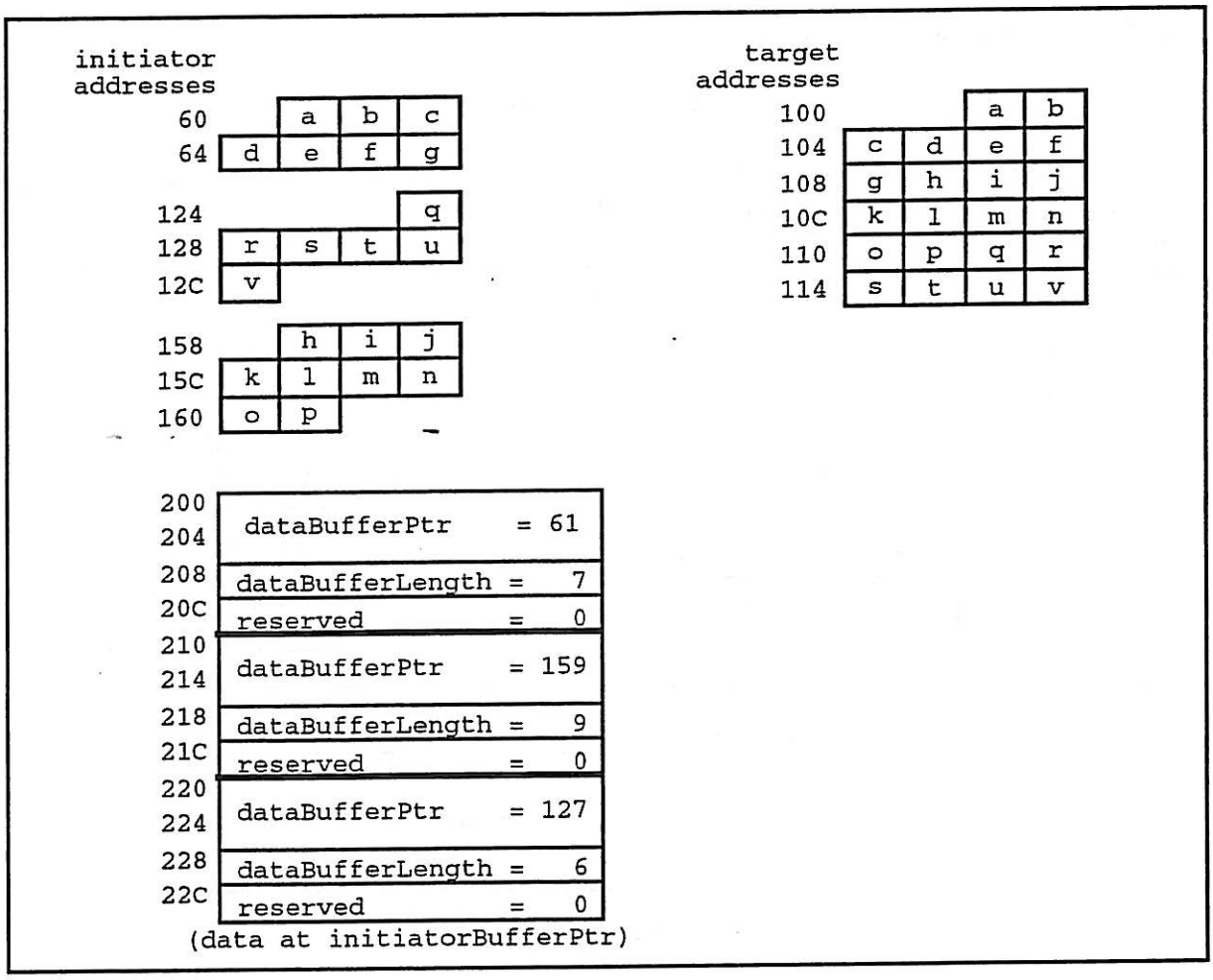

Figure 3-4: Data Addressing, Indirect-Initiator/Direct-Target Transfers

field	value
initiatorBufferPtr	200_{16}
initiatorBufferLength	30_{16}
targetBufferPtr	102_{16}
targetBufferLength	16_{16}

Table 3-2: Command Entry Fields, Indirect-Initiator/Direct-Target Transfers

3.3.2.3 Indirect-Initiator/Indirect-Target Transfers

Data may be transferred between an indirectly-specified set of initiator addressees and an indirectly-specified set of target addresses, as illustrated in Figure 3-5. The command entry fields for the transfer are shown in Table 3-3.

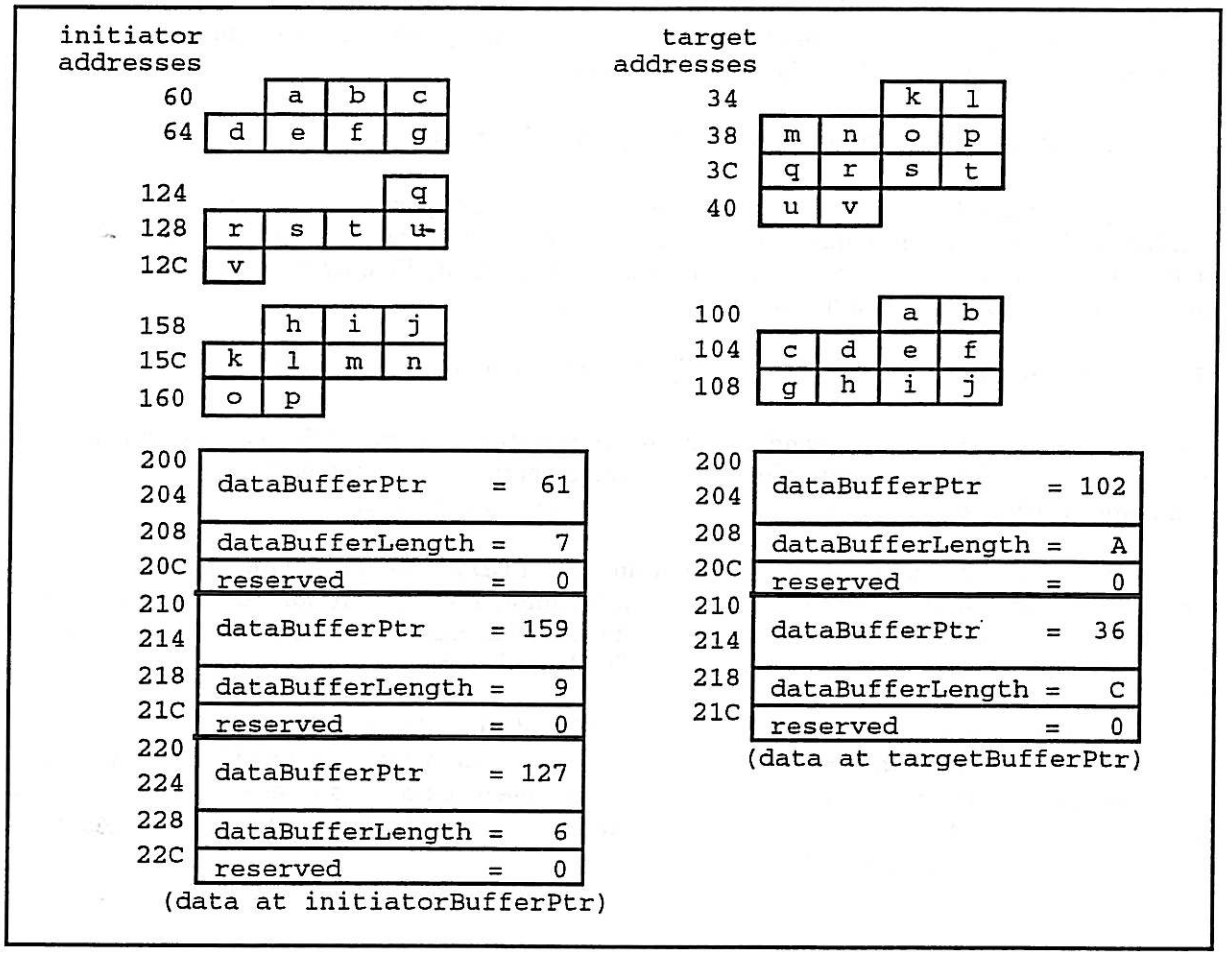

Figure 3-5: Data Addressing, Indirect-Initiator/Indirect-Target Transfers

field	value
initiatorBufferPtr	200_{16}
initiatorBufferLength	30_{16}
targetBufferPtr	360_{16}
targetBufferLength	20_{16}

Table 3-3: Command Entry Fields, Indirect-Initiator/Indirect-Target Transfers

3.4 Constant Transfers

3.4.1 Initiator Constant used with Read Command

For a read command, the first half of the command-dependent portion of the command entry may contain a 12-byte constant value, rather than a pointer to bus-accessible space. This specifies the data value(s) which are copied into the bus-accessible space specified by the command entry's targetBufferPtr and targetBufferLength components.

3.4.2 Initiator Constant used with Copy Command

For a copy command, the first half of the command-dependent portion of the command entry may contain a 12-byte constant value, rather than a pointer to bus-accessible space. This specifies the data value(s) which are copied into the bus-accessible space specified by the command entry's targetBufferPtr and targetBufferLength components.

3.4.3 Target Constant used as Device-Dependent Command

For the read commands, the second half of the command-dependent portion of the command entry may contain a 12-byte constant value, rather than a pointer to device-specific. This specifies the command which is executed by the device as part of the data transfer.

For example, the 12-byte constant could contain a SCSI CDB data value, which specifies a disk-block and transfer length. The read command and the initiatorBufferPtr/initiatorBufferLength values control the bus-accessible addresses generated by the DMA hardware; the 12-byte constant controls the device-specific addresses generated by the DMA hardware.

This access mode supports the use of traditional SCSI devices, for which a variety of special operations (disk format, tape rewind, etc.) may need to be supported. Passing CDB values is also useful for supporting special reads or writes, for example disk accesses where the media is formatted to support 520-byte blocks. However, most devices are expected to be optimized for passing a scatter list of device-specific transfer addresses and lengths.

3.5 Data-Transfer Constraints

3.5.1 Unaligned or Cross-Block Transfers

Many large bus-accessible space transfers are expected to have addresses and sizes that are cache- or page-aligned. However, unaligned read transfers are also required to efficiently support smaller transfers, network or terminal traffic. For example, Figure 3-6 illustrates a transfer that does not fall on an even boundary at either the start or end.

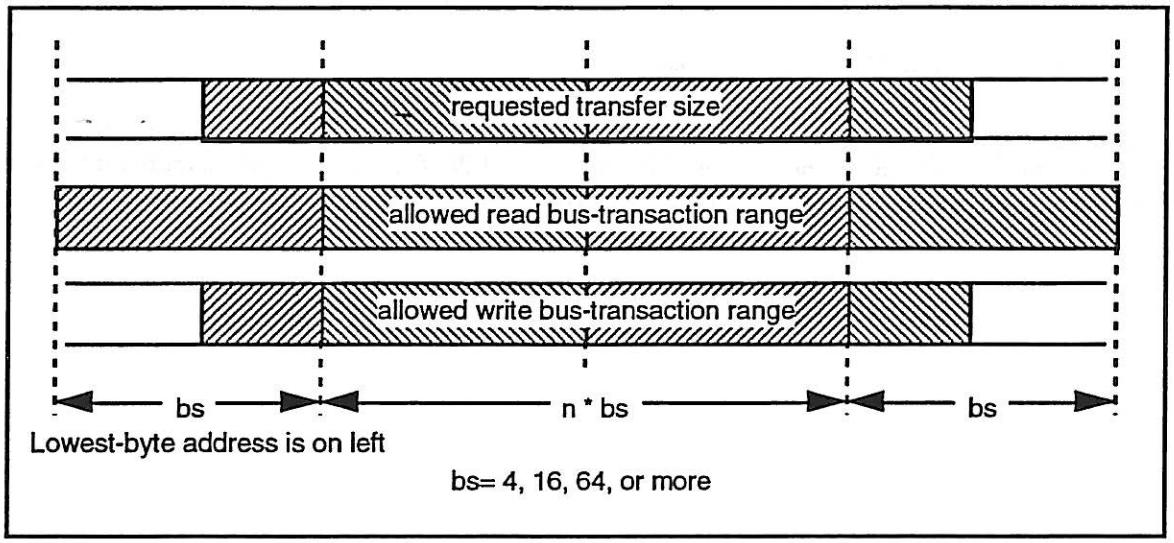

Figure 3-6: Cross-Block Data Transfers

Targets are expected to implement such unaligned read transfers as a sequence of aligned block transfers of the largest convenient size (e.g., read16, preferably read64, and most preferably block read transactions of the maximum size permitted for the given bus speed). The maximum transfer size is configurable, since simple bridges between the target and its memory may not support the large block-transfer sizes.

The rule to follow is that **read transfers** shall not cross a block-size-aligned boundary. For example, a 64-byte block read transaction may not cross a 64- byte-aligned boundary. The effect that this restriction has on normal operation is that for large data buffer read operations, in the worst case, the first and last block read must access only the residual between the beginning/end of the buffer up to the next alignment boundary.

For read transactions, the device issuing the read request may optionally begin the first block read on a boundary that is before the actual beginning of the desired data. For the last block read, the read transaction may continue up to the next block boundary following the end of the data of interest.

For SerialBus, the most bandwidth-efficient read transactions access only the portions of the data that are needed. A bridge to another block-aligned bus (like Scalable Coherent Interface) is expected to convert these partial first and final transfers into block-size transactions, discarding the data that is not used.

Unaligned **write transfers** are harder to perform, since data at adjacent addresses shall not be modified. Targets are expected to implement such unaligned write transfers as an initial unaligned transfer, a sequence of block-aligned transactions, and a final unaligned transfer. DMA units are expected to use write transactions of the largest convenient size during the intermediate, aligned portion of the block copy, again subject to the pre-specified block-size maximum.

For SerialBus, the most bandwidth-efficient read and write transactions access only the portions of the data which are needed. A bridge to another block-aligned bus (like Scalable Coherent Interface) is expected to convert these partial first and final transfers into one or more of its supported bus-block-aligned transactions, updating only the data which is addressed.

3.5.2 Aligned Sub-Block Transfers

The uncached 1/2/4/8/16/64-byte read transfers that are contained within one address block generate exactly one read1/2/4/8/16/64 transaction. The uncached 1/2/4/8/16/64-byte write transfers that are contained within one address block generate exactly one write1/2/4/8/16/64 transaction.

4. Data Formats

The I/O Process requires three data structures: command entry, status entry, and scatter array. The following sections describe these data structures .

4.1 Command Entry

Figure 4-1 shows the command entry. The initiator appends command entries to a list and the target extracts them from a list. The command entry describes an operation that the initiator wishes the target to perform. The fields in the command entry are described in the following text.

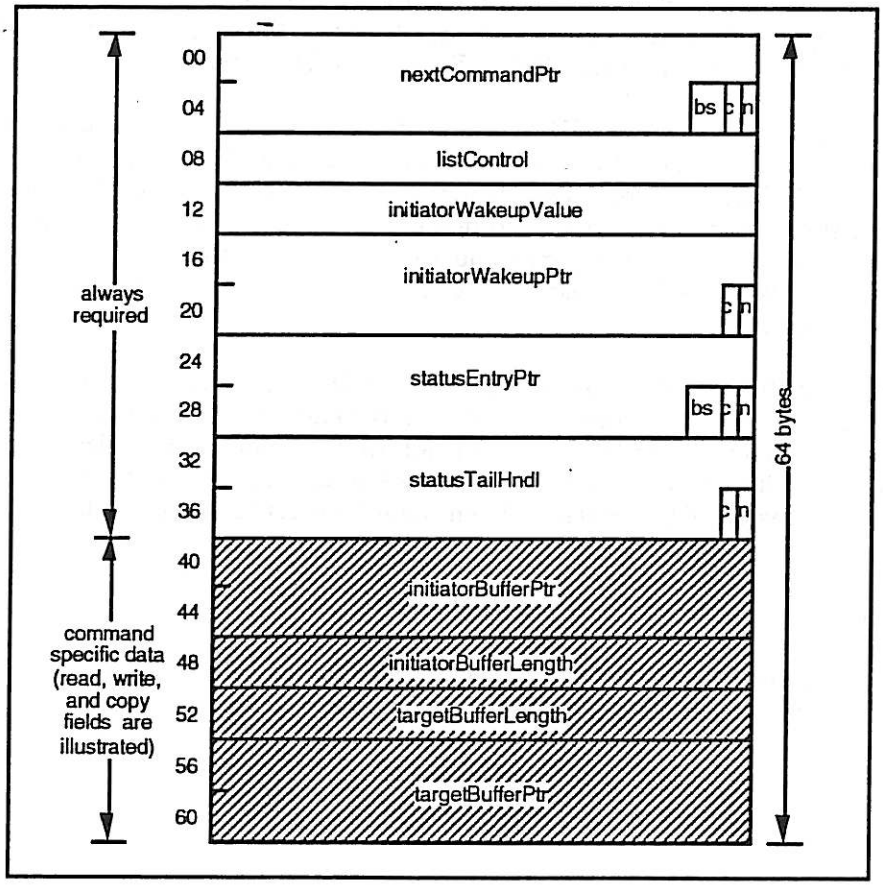

Figure 4-1: Command Entry Fields

The standard command-entry formats are defined to be 64 bytes long, but non-standard command-entry formats may be larger (up to 4K bytes), as specified by a field within the listControl quadlet, see Section 4.1.1.1 for details.

The **nextCommandPtr** field is either NULL or contains the address of the next command entry in the list. When an initiator appends a mini-list, the nextCommandPtr in all but the last command entry in the mini-list shall point to the next command entry in the mini-list. The initiator shall set the nextCommandPtr field in a single command entry or in the last mini-list entry, to NULL. The actual address of the next command entry is the nextCommandPtr value with the four least significant bits set to zero (note that command entries are always 16-byte aligned).

The **listControl** field specifies which DMA command is performed and provides additional command-dependent parameters. See Section 4.1.1.1 for details.

The **initiatorWakeupValue** is the value that the target writes to the initiator wakeup register after it has appended a status entry to the designated status list. The address of the initiator wakeup register and the pointer needed to append a status entry to a status list are all contained in the command entry (see below).

The **initiatorWakeupPtr** value is either NULL or it contains the address of the initiator wakeup register. When this field is not NULL, the target shall write the value contained in the initiatorWakeupValue field to the address contained in the initiatorWakeupPtr field when the target has completed the command described in this command entry. The actual address of the initiator wakeup register is the initiatorWakeupPtr value with the two least-significant bits set to zero (note that the wakeup register is always 4-byte-aligned). The c and the n bits are described in section 1. When the n bit is set to 1, the **initiatorWakeuPtr** is NULL (i.e., not valid).

The **statusEntryPtr** value is either NULL or it contains the address of the status entry associated with this command entry. The format of a status entry is described in a later section. If the status entry pointer is not NULL, the target shall return status for this command after completely processing the last command in the current command group (note that the current command group may contain as few as one command entry). The actual address of the status entry is the statusEntryPtr value with the four least significant bits set to zero (note that status entries are always 16-byte-aligned).

The **statusTailHandle** value is either NULL or it contains the address of the tail pointer for a status list. The target uses this value to append a status entry to the status list upon command completion. The actual address of the status list tail pointer is the statusTailHandle value with the two least significant bits set to zero (note that status list tail pointers are always 8 byte aligned). The c and the n bits are described in section 1. When the n bit is set to 1, the **statusTailHandle** is NULL (i.e., not valid).

All fields that follow those listed above are available for **command-dependent information**. Note that there are 24 bytes available for command specific information in a 64 byte command entry. The allowable sizes for a command entry are limited by the CESize field in the listControl field. This field and all other fields in the listControl field are described in the next section.

4.1.1 listControl Field

4.1.1.1 listControl Format

The format of the listControl field is shown in Figure 4-2.

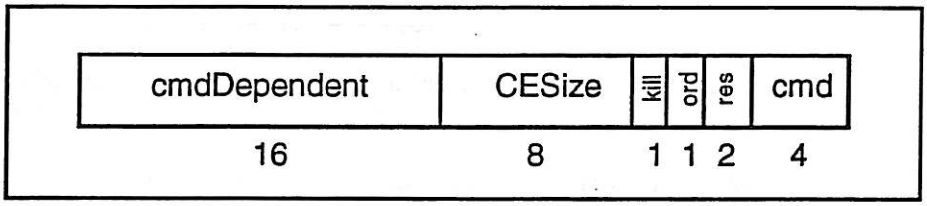

Figure 4-2: listControl Format

The **cmdDependent** field is available for command specific information. As such, the definition of this field varies by command.

The 1-bit **kill** field allows the initiator to kill an active command-group entry, without corrupting the remainder of the command-entry fields.

The 8-bit **CESize** field gives the size, as the number of 16-byte blocks minus 1, of the entire command entry. A value of 3 in this field indicates that the entire command entry is 64 bytes long, which is the case for all standard command entries. When the cmd field indicates that this is an *other* command, the command can be larger.

The **ord** field is a 1 bit field that determines allowable reordering of this command relative to other command entries in the same list. When the ord field contains a value of 0, this command entry may be reordered without restriction relative to other commands in the same list. When the ord field contains a value of 1, this command entry shall be treated using the same ordering rules that apply to an ordered tagged command in the SCSI-3 queuing model.

The 2-bit **res** field is reserved.

The following section describes the 4-bit **cmd** field.

4.1.1.2 listControl.cmd Values

The 4-bit **cmd** field indicates to the target what operation is performed by the I/O service. The values and meanings are defined in Table 4-1.

value	name	description
0000_2	write	Data is copied from the initiatorBufferPtr-specified bus-accessible space into the targetBufferPtr-specified device-specific space (memory-to-device transfers).
0001_2	read	Data is copied from the targetBufferPtr-specified device-specific space into the initiatorBufferPtr-specified bus-accessible space (device-to-memory transfers).
0010_2	copy	Data is copied from the initiatorBufferPtr-specified bus-accessible space into the targetBufferPtr-specified bus-accessible space (memory-to-memory transfers).
0011_2	attach	Two pointer arguments identify the head and tail of an attached list. The third pointer argument identifies the list to which this is attached.
0100_2	loop	A 64-bit argument value is subtracted from an internal loopCount value, and the result is stored at an argument-specified address. If the result is positive, an argument specifies the address of the next command entry.
0101_2	kill	Search the specified command list, terminate processing of the specified command group, and return status. This command is appended to the management list to kill commands in the other command-group lists.
0110_2	noOp	The target shall not transfer any data as a result of this command. The target shall return a "normal termination" status for this command.
0111_2-1110_2	reserved	These encodings are reserved for future standard definitions.
1111_2	other	The command operation is indicated by the fields after the standard header (starting at command-entry byte-offset 40).

Table 4-1: listControl.cmd Field Values

There are no commands for explicitly copying data between device-specific spaces, but the equivalent functionality can be provided by concurrent execution of dependent data-transfer lists; see Section 6.1.1 for details.

4.2 Status Entry

4.2.1 Status Entry Format

The status entry, shown in Figure 4-3, is appended by the target and extracted by the initiator. It carries status information from the I/O service to the driver.

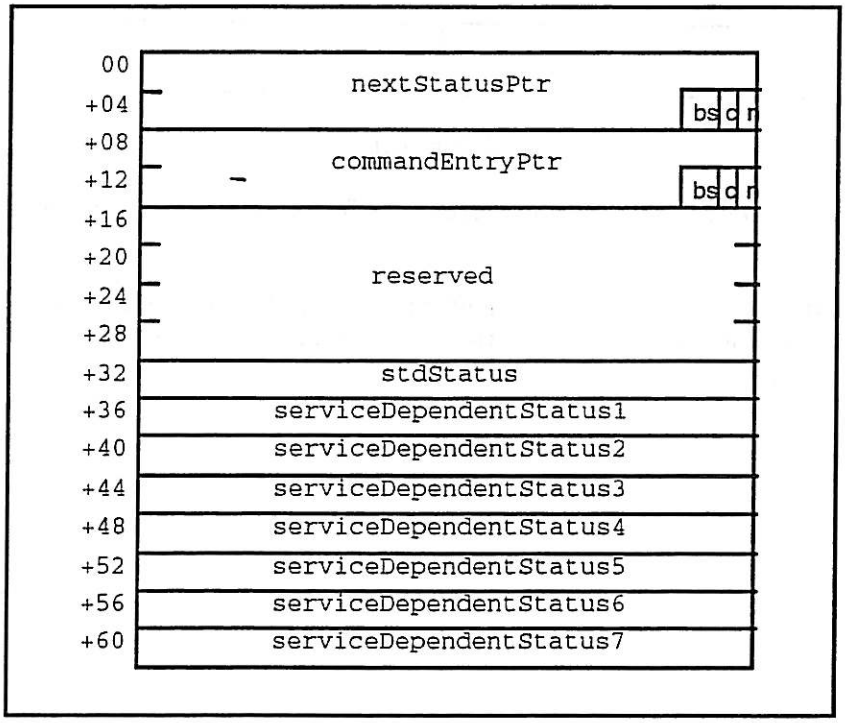

Figure 4-3: Status Entry Fields

The **nextStatusPtr** field is either NULL or it contains the address of the next status-list entry. When a target appends a mini-list, the nextStatusPtr value in all but the last status entry shall point to the next status-list entry in the mini-list. The last nextStatusPtr in a single entry or the nextStatusPtr in the last mini-list entry shall be set by the initiator to NULL. The actual address of the next status entry is the value in the nextStatusPtr field with the low order 4 bits set to zero (note that status entries must be 16-byte-aligned).

The **commandEntryPtr** shall be initialized by the initiator to the address of the first command group entry; the target shall not modify this field.

The **stdStatus** value is preset to "normal termination" by the initiator. When necessary (to return another completion-status value), the target may modify this value. The format of the stdStatus0 value is described in a following section.

The **serviceDependentStatus1** through **serviceDependentStatus7** locations provide space for service dependent information to be passed from the target to the initiator.

4.2.2 stdStatus Values

The format of the stdStatus field is shown in Figure 4-4.

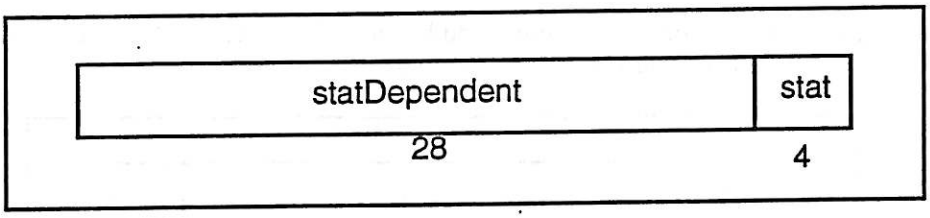

Figure 4-4: stdStatus Format

The **statDependent** field is available for status-specific information. As such, the definition of this field varies by stat value.

The stat field contains I/O service independent status. It shall be preset to *good* by the initiator and shall only be modified by the target if the stat for the command (group) is not *good*. Its four LSBs are defined in Table 4-2.

value	name	description
0	good	Normal termination indicates that the command was processed correctly and no errors occurred.
1	abnormalTermination	An error occurred that could not be categorized by the target.
2	dataTransferError	A bus-transaction error occurred. Either a timeout or a data CRC error was detected.
3	reserved	Reserved for extensions to the CCU Architecture.
4	dataLengthError	The size of the initiator data buffer(s) is insufficient for the total transfer size of the command.
5	targetLengthError	The size of the target data buffer(s) is insufficient for the total transfer size of the command.
6	initiatorDataAddressError	An address within the initiator data buffer(s) could not be accessed by the target.
7	targetDataAddressError	An address within the target data buffer(s) does not exist within the target.
8	killed	This command-group entry was killed by a kill command in the management list or (when the command group was processed) the listControl.kill bit of the first command entry was 1.
9	killFound	Kill-command status: the command entry was terminated.
10	killNotFound	Kill-command status: the command entry was not found and all command-list entries were searched.
11	killFragment	Kill-command status: the command entry was not found; the command list was fragmented and not all command-list entries could be checked.

Table 4-2: StdStatus.stat Values

4.3 Scatter Array

A scatter array (which is pointed to by the command entry) allows a single command to specify a transfer of a logically contiguous memory buffer that maps to a discontiguous set of physically-addressed memory blocks.

A scatter array consists of one or more elements located at contiguous physical addresses. The format of a scatter array differs slightly depending on whether it is a scatter array for the initiator or for the target. Both formats of scatter arrays are described in the following sections.

4.3.1 Target's Scatter-Array Elements

The Figure 4-5 shows the format of the first two elements in a target's scatter array at the address specified by targetBufPtr. Two entries are shown for illustrative purposes; a minimal scatter array consists of only one element.

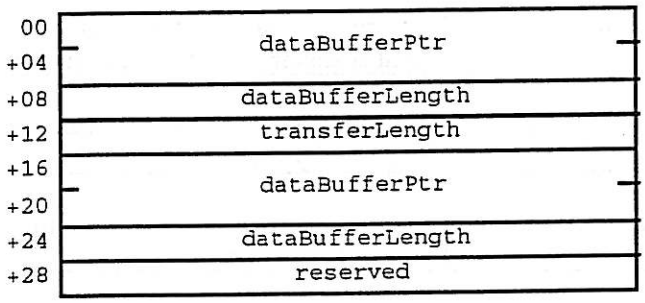

Figure 4-5: Initiator's Scatter-Array Format

The **dataBufferPtr** field contains the address of a 1-byte-aligned physically contiguous block of memory. The command entry's listControl field contains bits that affect the meaning of this dataBufferPtr field.

The **dataBufferLength** field contains the size in bytes of the block of memory whose address is contained in the dataBufferPtr field.

The **transferLength** field is the final quadlet in the first of the target's scatter array elements, and specifies the number of data bytes involved in the command entry's data transfer. An error condition shall be reported if this value exceeds the acumulative sum of the dataBufferLength fields for this array's scatter elements.

In all but the first of the target's scatter elements, the final quadlet is **reserved**.

4.3.2 Initiator's Scatter-Array Elements

The Figure 4-6 shows the format of the first two elements in a initiator-specific scatter array at the address specified by initiatorBufPtr. Two entries are shown for illustrative purposes; a minimal scatter array consists of only one element.

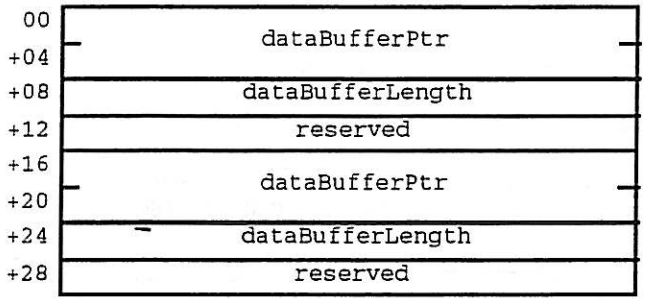

Figure 4-6: Initiator's Scatter-Array Format

The **dataBufferPtr** field contains the address of a 1-byte-aligned physically contiguous block of memory. The listControl field contains bits that affect the meaning of this dataBufferPtr field.

The **dataBufferLength** field contains the size in bytes of the block of memory whose address is contained in the dataBufferPtr field.

In all of the initiator's scatter elements, the final quadlet is **reserved**.

5. Registers and ROM Entries

5.1 DMA Register Addressing

5.1.1 Unit Address Spaces

The address structure of a CCU unit architecture is shown in Figure 5-1.

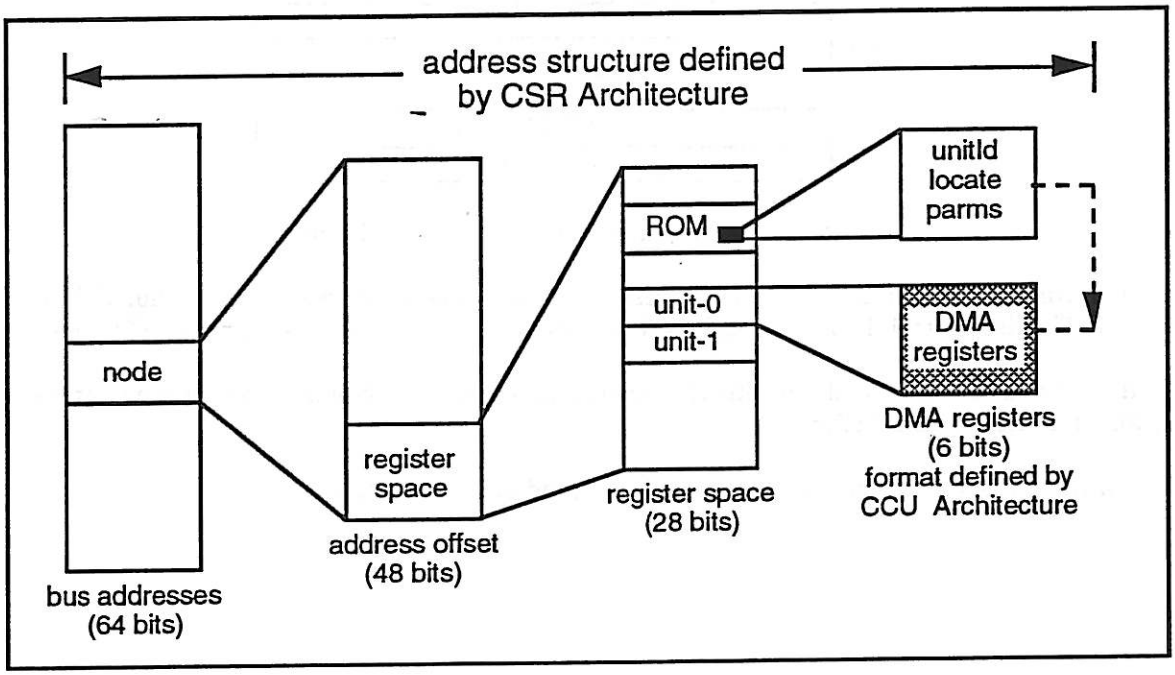

Figure 5-1: CCU Unit Addressing

The addressing structure is defined by the CSR Architecture.

ROM contains a unitId, which identifies the proper I/O Software interface. Locate specifies the location and size of the DMA-related registers.

A variety of unit-dependent parameters (parms) may also be provided, but the format and content of these are beyond the scopes of the CSR and CCU Architectures. These driver-dependent parameters could specify the structure and number of DMA list groups supported by the target.

The unit's ROM identifies the location and size of its DMA registers. The CCU Architecture defines the format and function of the initial DMA registers. The remaining registers may be device dependent.

5.1.2 Control Register Structure

The control register is used to control the overall activity of the list to which it is related. The format for the control register is shown in Figure 5-2.

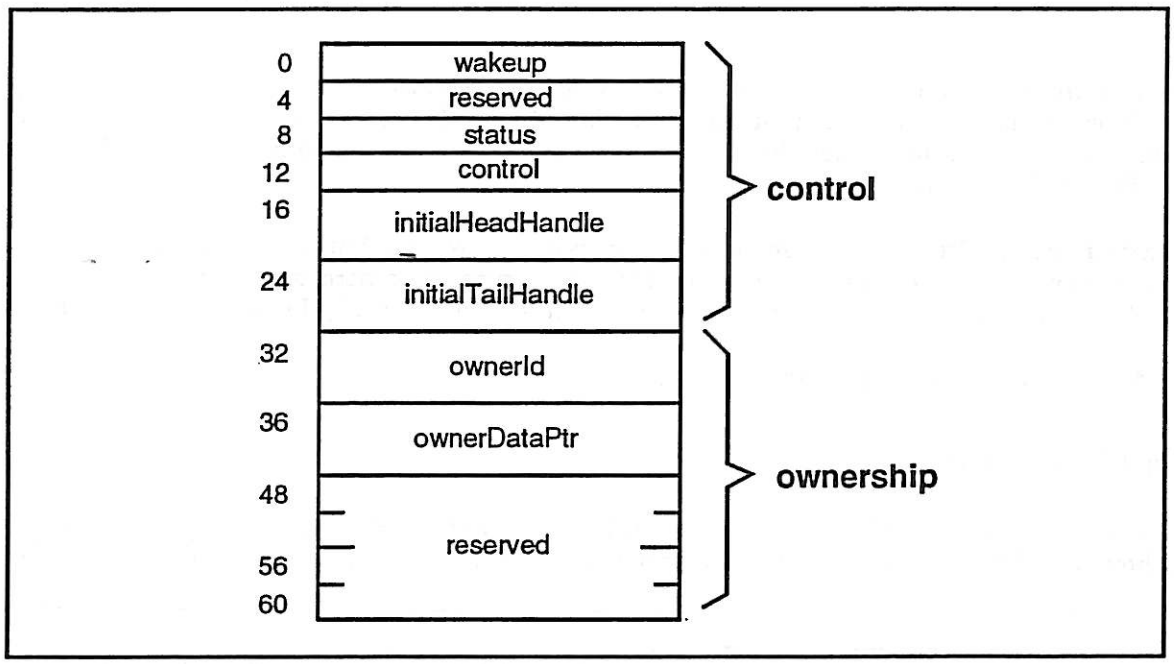

Figure 5-2: Unit Register Organization

5.2 Control Registers

The **wakeup** register provides access to an internal array of wakeup bits, where one wakeup bit is provided for each of the target's command lists. The wakeup register is typically written to by an initiator after the DMA command-list parameters have changed. Any one of these wakeup bits may be individually set by writing the bit's index value to the wakeup register. The wakeup register is cleared by the affiliated I/O process before the command-list status is re-checked.

This has a different format than the INTERRUPT_TARGET register defined by the CSR Architecture, whose contents are OR'd with up to 32 internal wakeup bits. The difference in format and function are possible because 1) broadcast capability is not needed and 2) More than 32 internal wakeup bits may exist on high-capability unit architectures.

Reads from the **status** register indicate the state of the DMA process: stopped, initializing, running, and halted. DMA halts when processing of the highest-level event list has a fatal error (typically a bus error). Errors in processing of other DMA lists are reported through the event list.

Writes to the **control** register are used to start, stop, and resume DMA operations.

The values in the read/write **initialHeadHandle** and **initialTailHandle** registers are transferred to internal commandHeadHandle and commandTailHandle locations at the beginning of the DMA-start process. This information is sufficient to initiate management-list processing.

5.3 Ownership Registers

Ownership registers are read/write registers, initially zero, that can be accessed using any of the read4, write4, and compareSwap8 transactions. By I/O driver software convention, they are used to control ownership of the unit or driver-dependent portions of the unit. They have no special side effects.

Owners are expected to leave an owner-unique 64-bit value in the **ownerId** register, to identify which owner has control of the unit resources. The 64-bit value consists of the 24-bit companyId value (as defined in the CSR Architecture) concatenated with the company-unique 40-bit component specifier of the owning device.

The **ownerDataPtr** register specifies the address of the owner's shared data structure. The data contained within this data structure is used to synchronize and control shared unit resources. Further definition of this data structure and conventions for its use by I/O driver software are TBD.

5.4 DMA List Groups

5.4.1 List Groups

DMA lists are organized into list groups. This list group has the structure illustrated in Figure 5-3 Although multiple transfer lists are illustrated, a minimal list group contains only one transfer list.

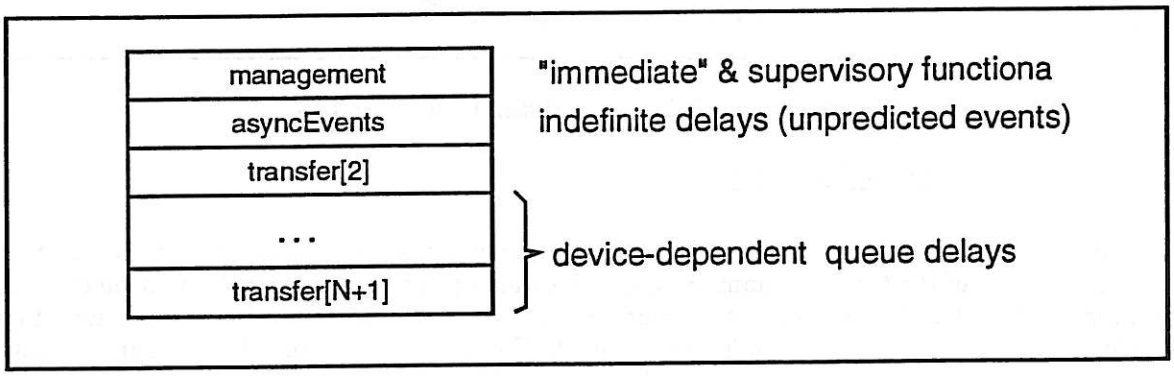

Figure 5-3: List Group Components

The commands in the **management** list are expected to affect lists within the same list group, or lists at a lower level within a hierarchical list-group structure. However, all management list commands are expected to be completed almost immediately, independent of the affiliated device operations.

For example, a kill command is placed in the management queue, even though it has the side-effect of killing a pending command in the n'th transfer queue, transfer[n].

The commands in the **asyncEvents** list are expected to be read commands that (on demand) return unexpected event status. Only one or two read commands are expected to be queued, since a queued command is not pre-allocated to any list, but can be used to return event status from any of the affiliated lists. Note that the commands in the asyncEvents list may remain for an indefinite period, depending on the arrival rate of unexpected events.

The commands in the **transfer[2]** through **transfer[N+1]** lists are expected to be used individually to support half-duplex traffic or in pairs to support full-duplex traffic. The commands in these lists are typically transient, in that they are constantly consumed as I/O operations are performed. However, some I/O operations (such as terminal reads or flow-controlled writes) may have an indefinite lifetime.

5.4.2 Unit Architectures

A minimal unit architecture supports only one list group. A unit architecture may also support multiple list groups, where a hierarchical list-group structure typically reflects the functional capabilities of the physical device attachments. Such a hierarchical list-group structure is illustrated in Figure 5-4

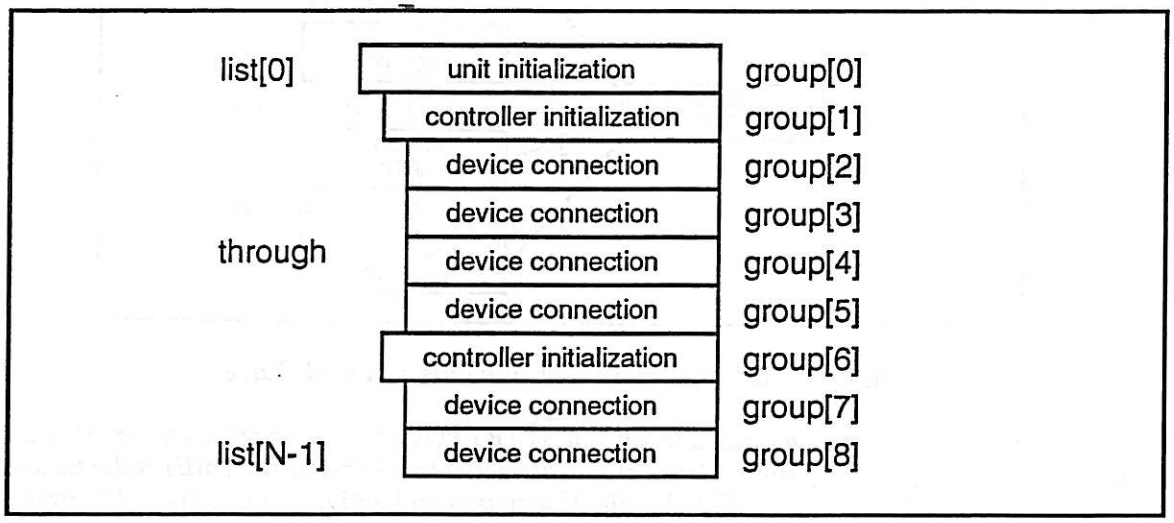

Figure 5-4: Multiple List Groups

The number of levels in the hierarchy as well as the detailed structure of each level within the hierarchy is unit dependent and beyond the scope of the CCU Architecture.

5.5 Internal State

This section describes how internal unit state is mapped into the target's address space and updated indirectly through read and write DMA commands. An alternative proposal is located in Section 8, which maps these resources to registers within the unit's address space. The reader is encouraged to consider the ramifications of both design models.

A unit has a significant amount of context that may be associated with each of the target's command lists, including the *commandHeadHandle*, the *commandTailHandle*, the *bufSize* parameter (if flow-control is supported) and the *intLoopCount* value (if command-list looping is supported). This internal state is mapped into the target's address space and is accessed indirectly, using a read or write command entry.

5.6 Unit Initialization and Control

A reset of the node clears the unit's registers to zero. For the status register, this is interpreted as the INITIALIZING state. After the node has been initialized, the status register state changes to READY, as illustrated in Figure 5-5.

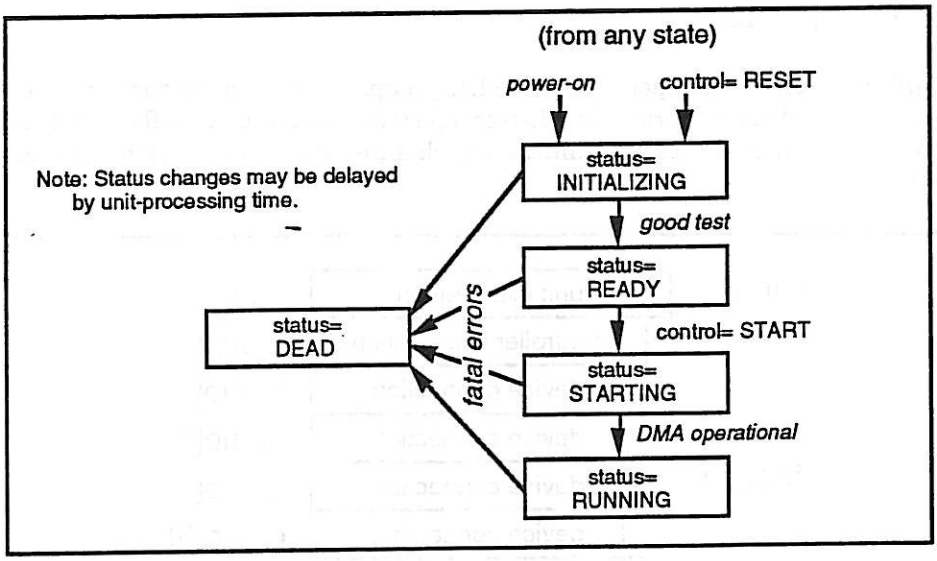

Figure 5-5: Changes in a Unit's Operational State

The I/O driver software is then expected to write a START value to the control register. This starts the DMA operation, by copying the contents of *initialHeadHandle* and *initialTailHandle* into the *internalHeadHandle[0]* and *internalTailHandle[0]* registers and activating command-list processing.

Further description of the initialization process is TBD.

6. Special Operations

6.1 Dependent Data Transfers

6.1.1 Serialized Data Transfers

In some cases, there is the need to transfer data directly from one device to another. These device-to-device transfers have often been performed by transferring the data through intermediate memory buffers, as illustrated in Figure 6-1. A producer's command list copies the data from the producer to memory. After the producer's transfer completes, the data is transferred from memory to the consuming device.

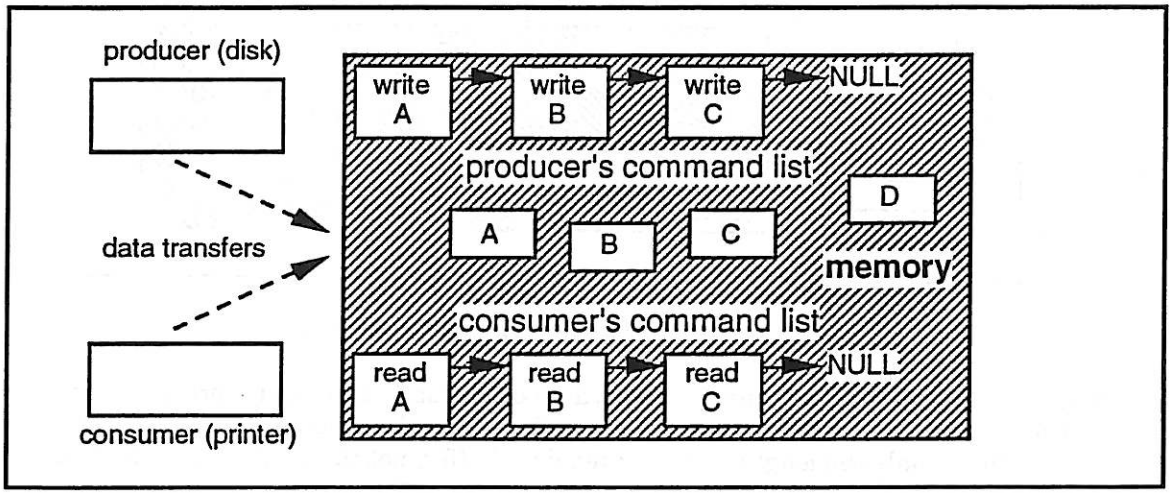

Figure 6-1: Sequential Dependent Transfers

Although this technique works in many applications, it has the disadvantage that a large memory buffer (which can be expensive) must be pre-allocated and the data-transfer latency (which may be unacceptable in real-time applications).

A large transfer could be decomposed into a sequence of smaller data transfers, but software would have to intervene and append additional mini-lists as previous mini-lists complete. Software intervention complicates the I/O software and may place critical real-time requirements on the interrupt-service times.

Note that the efficiency of these operations can be improved dramatically if the memory buffers (A, B, and C) are physically located in the consuming device. The normal pair of read and write memory transactions are then reduced to a single write transaction (with a target address in the consumer's address space).

6.1.2 Flow-Controlled Transfers

The producer and consumer are programmed to agree on a common buffer size; in this example bufSize is the size of data buffer A. The producer and consumer transfers then begin, updating memory-resident produceCount and consumeCount locations as the transfers progress. These components are shown in Figure 6-2.

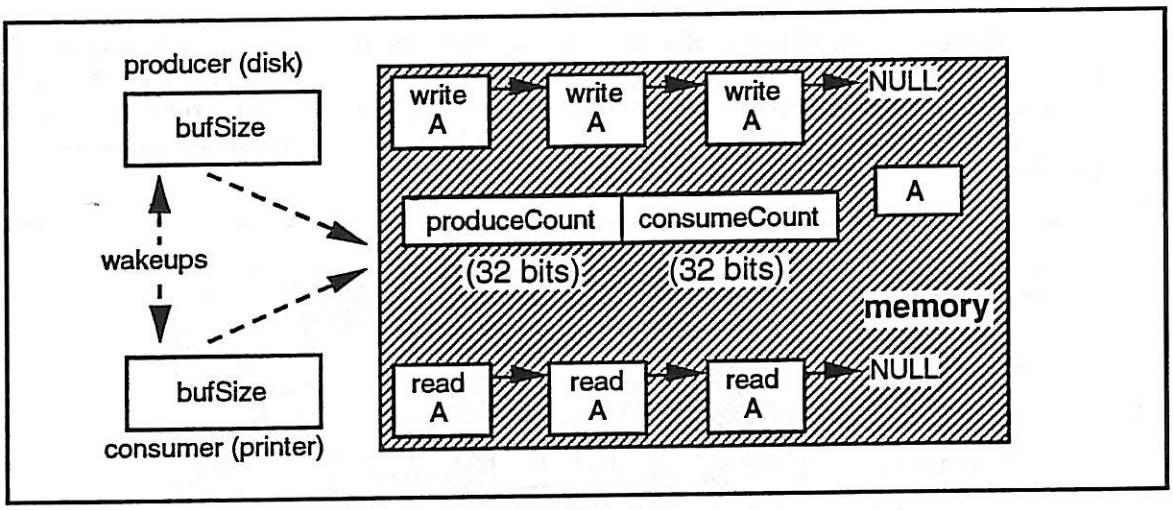

Figure 6-2: Flow-Controlled Transfers

The values of produceCount and consumeCount are both initially zero, and represent the number of bytes transferred into (or out of) the shared data buffer. Since the 32-bit number can wrap around, the 32-bit size limits only the length of the shared data buffer, not the total transfer length.

The producer writes data into system-memory addresses until the buffer becomes full. The producer then sends a wakeup signal to the consumer, and then the producer sleeps. After a wakeup, the producer rechecks the buffer counts and (if more buffer space is available) activates another data transfer. The amount of free buffer space is specified by the following equation:

```
freeBufferSpace= bufSize - (unsigned)(produceCount-consumeCount);
```

The consumer reads data from system-memory addresses until the buffer becomes empty. The consumer then sends a wakeup signal to the producer and then the consumer sleeps. After a wakeup, the consumer rechecks the buffer counts and (if more buffer space is available) activates another data transfer. The amount of filled buffer space is specified by the following equation:

```
filledBufferSpace= (unsigned)(produceCount-consumeCount);
```

The updates of the produceCount and consumeCount values may occur more frequently, such as before every lengthy disk seek operation. The detailed timing of such intermediate checkpoints is beyond the scope of the CCU standard.

To improve system efficiency, the produceCount and consumeCount values are required to be at adjacent quadlet addresses and the address of the first (produceCount) quadlet shall be a multiple of 8. With this restriction, one producer-generated transaction can be used to simultaneously write produceCount and read consumeCount. Equivalently, one consumer-generated transaction can be used to simultaneously read produceCount and write consumeCount.

To minimize the number of SerialBus transactions, the produceCount and consumeCount values could be physically located in either the producer or the consumer. One of the two count-update transactions (generated by the producer or consumer) can then be performed locally.

6.1.3 Command-List Looping

On address-less devices, like a printer or communication channel, the target address need not be updated as the data transfer progresses. Thus, two or more of the command blocks may contain identical information. To support large contiguous data transfers to such devices, a command-list loop capability is provided, as illustrated in Figure 6-3.

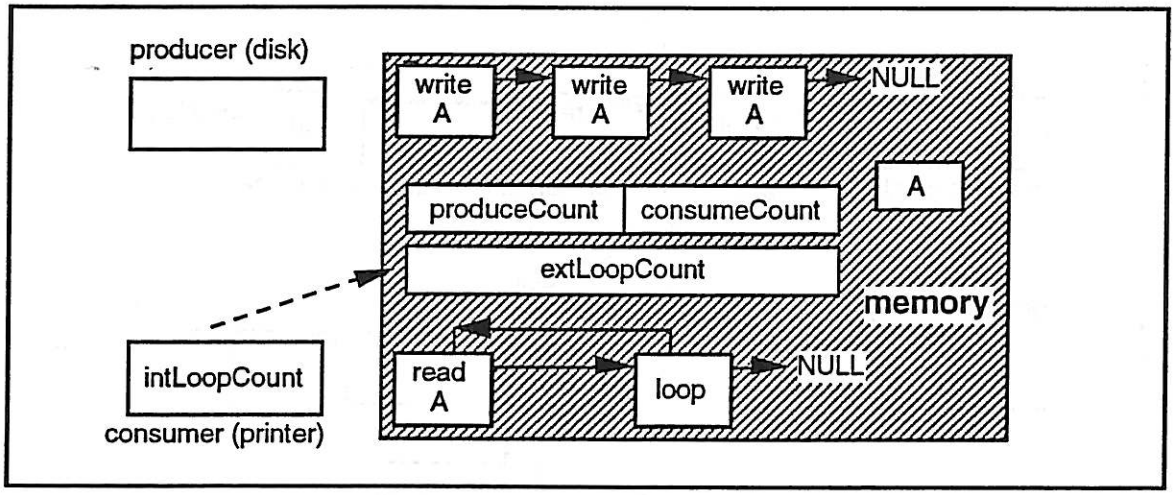

Figure 6-3: Command-List Looping

Each loop "iteration" performs an add of a command-entry resident constant with an internal 64-bit *intLoopCount* value and saves the result in a command-entry specified external *extLoopCount* location. If the loop condition is met, the command-entry specified *loopCommandPtr* value specifies the address of the next command entry which is fetched. The looping stops when *extLoopCount* becomes negative, or when a data transfer error causes all but the last command-group entry to be skipped.

6.1.4 Command-List Attachments

In the case of a printer and disk, it is desired to control the spooling order from the printer and to activate the disk data transfers as required. To control the order of print listings, the printer enforces the data-transfer orders, and the disk has no preferred ordering. The attach command supports this capability, providing the functionality illustrated in Figure 6-4.

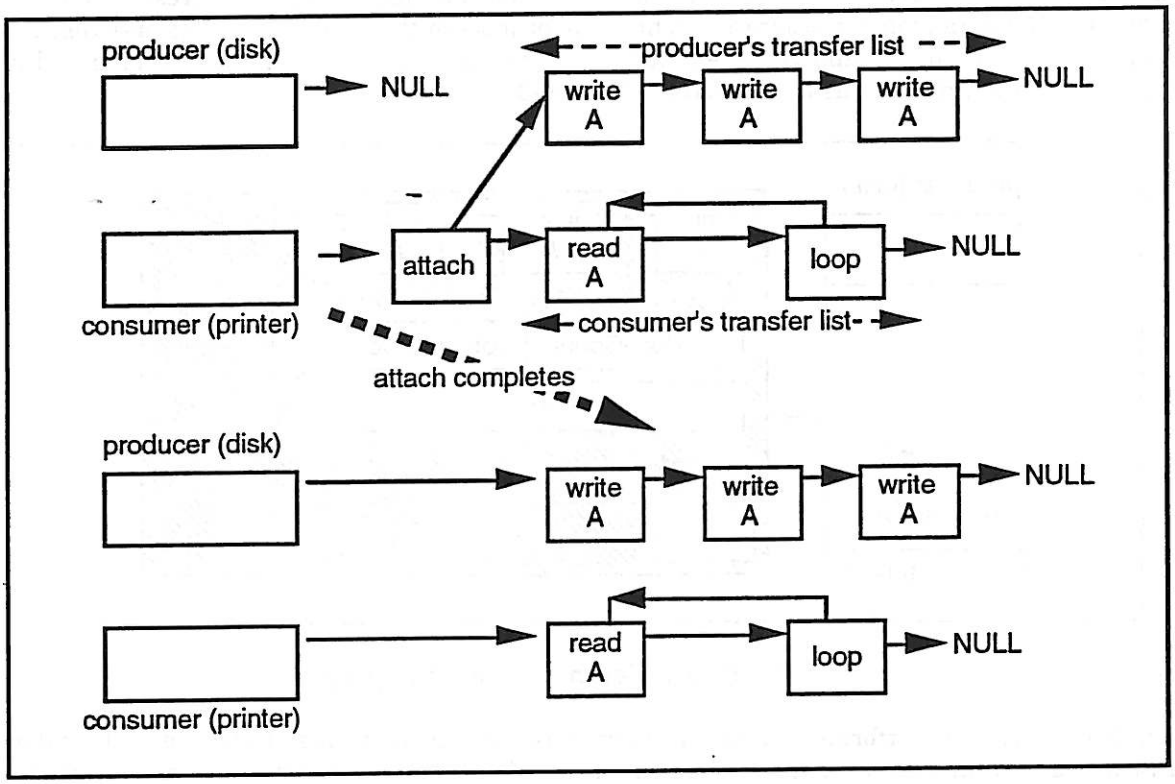

Figure 6-4: Command-List Attachments

6.2 Supervisory Kill Command

The expected use of a kill command is to abort a previously-appended command entry, so that the command entry, status entry, and affiliated data-transfer buffers can be released for other system uses. Within the context of this illustration, the kill command is said to kill an affected command group, which is located in one of the affected command lists (an event or transfer list).

Before appending a kill command-entry to the management list, the initiator is expected to set the listControl.kill bit in the first of the affected command-group entries. This inhibits further processing of the affected command group, which could otherwise occur before or while the supervisory-list kill command-entry is processed by the target.

The initiator then appends the kill command entry to the management queue. When the target processes this a kill command, it stops processing commands in the affected list. The initiating kill command is completed immediately, with one of the following status conditions:

1) *killFound*. The affected command-group was terminated, in either of the following ways:

   a) Stopped. The affected command-group was being processed when the management's kill command was received; the affected command-group processing was terminated immediately (with a killed status).

   b) Queued. The affected command group had not been processed, but was found in a forward search of the affected command list.

2) *killNotFound*. The affected command group was not being processed and could not be found in a search to the end of a stable command list. The address of the final command-list entry was equal to the commandTailPtr value (the affected list appeared to be stable).

2) *killFragment*. The affected command group was not being processed and could not be found in a search of the affected command list. The address of the final command-list entry was different than the commandTailPtr value (an append to the affected list was in progress).

After the initiating kill command is processed, the target continues processing commands in the affected list. When it encounters the command entry which has a listControl.kill bit value is one, the target returns a *killed* status for the affected command group and continues normal processing of command groups from its (affected as well as other) command lists.

7. Standardized DMA Commands

This section describes commands that have a standardized format. It is expected that devices implementing the sharable list DMA model will implement the commands described in this section.

7.1 Read and Write Command Entries

The read and write command entries are used to transfer data between bus-accessible space and device-specific space. The format of the command-specific portion of the read or write command entry is illustrated in Figure 7-1.

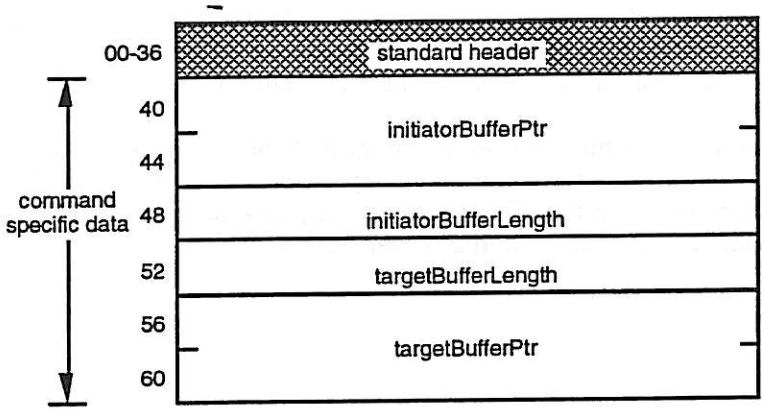

Figure 7-1: Read and Write Command Entry Formats

The **initiatorBufferPtr** field is a part of the initiator buffer descriptor. Depending on the listControl field values, this field contains a constant, the address of a data-transfer block in system memory, or the address of a scatter array in system memory.

The **initiatorBufferLength** field contains the size, in bytes, of the object pointed to by the initiatorBufferPtr value. If the *listControl.iSa* bit (contained in the standard header) is 0 (direct buffer descriptors), then the initiatorBufferLength field is ignored.

The **targetBufferLength** field contains the size, in bytes, of the object pointed to by the targetBufferPtr value. If the *listControl.tSa* bit(contained in the standard header) is 0 (direct buffer descriptor), then the targetBufferLength field contains the total transfer size, in bytes, for this command entry. If the *listControl.tSa* bit is one (indirect buffer descriptor), then the total transfer size, in bytes, is contained in the first entry in the target's scatter array.

The **targetBufferPtr** field is a part of the target buffer descriptor. Depending on the listControl field values, this field contains a constant, the address of a data-transfer block in target address space, or the address of a scatter array in system memory.

For the read and write command entries, the format of the command-dependent portion of the listControl quadlet (contained in the standard header) is illustrated in Figure 7-2.

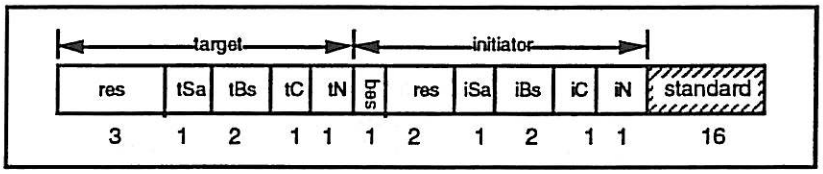

Figure 7-2: Read and Write listControl Formats

If the value of the initiator-scatter-array field **iSa is 0**, the data transfer involves one contiguous bus-accessible space and the iN, iC and iBs fields specify how the data are accessed, as listed in Table 7-1.

iN	iC	iBs	description
0	0	0	Uncached data buffer; block size is 4
"	"	1	Uncached data buffer; block size is 16
"	"	2	Uncached data buffer; block size is 64
"	"	3	Uncached data buffer; block size is bus maximum
0	1	0-3	Cached data buffer (not applicable to SerialBus)
1	0	0	Fixed bus address, 4-byte reads and writes
"	"	1	Fixed bus address, 16-byte reads and writes
"	"	2	Fixed bus address, 64-byte reads and writes
"	"	3	Fixed bus address, 256-byte reads and writes
1	1	0-3	Immediate constant, use the initiatorBufferPtr value itself

Table 7-1: iN, iC, iBs Fields, Contiguous Initiator Space

If the value of the initiator-scatter-array field **iSa is 1**, the data transfer involves a scattered bus-accessible space and the iN, iC and iBs fields specify how the data accessed, as listed in Table 7-2.

iN	iC	iBs	description
0	0	0	Uncached scatter array; block size is 4
"	"	1	Uncached scatter array; block size is 16
"	"	2	Uncached scatter array; block size is 64
"	"	3	Uncached scatter array; block size is bus maximum
0	1	0-3	Cached scatter array (not applicable to SerialBus)
1	0,1	0-3	Null pointer (address is invalid)

Table 7-2: iN, iC, iBs Fields, Scattered Initiator Space

If the value of the target-scatter-array field **tSa is 0**, the data transfer involves one contiguous device-specific space and the tN, tC and tBs fields specify how the data accessed, as listed in Table 7-3.

tN	tC	tBs	description
0	0,1	0-3	Data buffer in target address space (tC and tBs are ignored)
1	0	0	Fixed target address, 4-byte reads and writes
"	"	1	Fixed target address, 16-byte reads and writes
"	"	2	Fixed target address, 64-byte reads and writes
"	"	3	Fixed target address, 256-byte reads and writes
1	1	0-3	Immediate constant, use the initiatorBufferPtr value itself

Table 7-3: tN, tC, tBs Fields, Contiguous Target Space

The values of the target-scatter-array field **tSa is 1**, the data transfer involves one contiguous device-specific space and the tN, tC and tBs fields specify how the data accessed, as listed in Table 7-4.

tN	tC	tBs	description
0	0	0	Uncached scatter array; block size is 4
"	"	1	Uncached scatter array; block size is 16
"	"	2	Uncached scatter array; block size is 64
"	"	3	Uncached scatter array; block size is bus maximum
0	1	0-3	Cached scatter array (not applicable to SerialBus)
1	0,1	0-3	Null pointer (address is invalid)

Table 7-4: tN, tC, tBs Fields, Scattered Target Space

For the fixed address options, the address is not incremented during the specified data transfer and the bus transaction size shall equal the block size, as specified by the tBs field.

If the value in the field **seq is 0**, the target may transfer data to the initiator out of order.

If the value in the field **seq is 1**, the target shall transfer data to the initiator in order, i.e., beginning at the address specified in the initiator buffer pointer, or the address specified in the first initiator buffer scatter/gather array containing address information, and proceeding to higher addresses. The target shall appear to have transferred data off of the media in order, i.e., beginning at the address specified in the target buffer pointer, or the address specified in the first target scatter/gather array containing address information and proceeding to higher addresses.

7.2 Copy Command Entries

The format of the command-specific portion of the copy command entry is illustrated in Figure 7-3. and described in the following text:

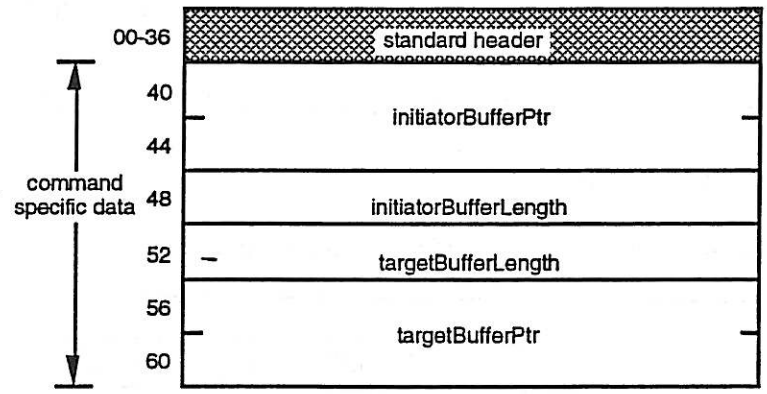

Figure 7-3: Copy Command-Entry Format

The **initiatorBufferPtr**, **initiatorBufferLength**, and **targetBufferLength** fields for the read, write, and copy commands are identically defined.

The **targetBufferPtr** field is a part of the target buffer descriptor. Depending on the listControl field values, this field contains a constant, the address of a data-transfer block in bus-accessible space, or the address of a scatter array in system memory.

The values of **tN**, **tC**, **tBs**, and **tSa** (for the copy command) are the same as those defined for the iN, iC, iBs, and iSa in the read and write commands.

7.3 Kill Command

The kill command is used to delete a command group from a list or to inform the target to stop processing a command group that is currently being processed. The kill command is a one-entry command group, and shall not be included within part of another command group. The kill command may only be appended to a management list. The format of the command-specific portion of the kill command entry is illustrated in Figure 7-4.

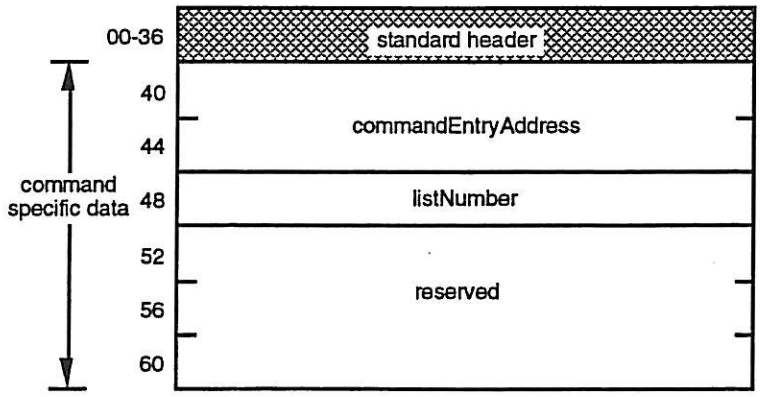

Figure 7-4: Kill Command-Entry Format

The **commandEntryAddress** contains the address of the first command in a command group to kill. Only entire command groups may be killed (note that a command group may contain only one command entry).

The **listNumber** field is a 32 bit identifier for the list containing the command entry to kill. This value determines if the command entry to kill is in the async list or one of the transfer lists associated with the management list.

Note that there are no command dependent bits defined for the kill command.

7.4 Loop Command

The format of the command-specific portion of the loop command entry is illustrated in Figure 7-5.

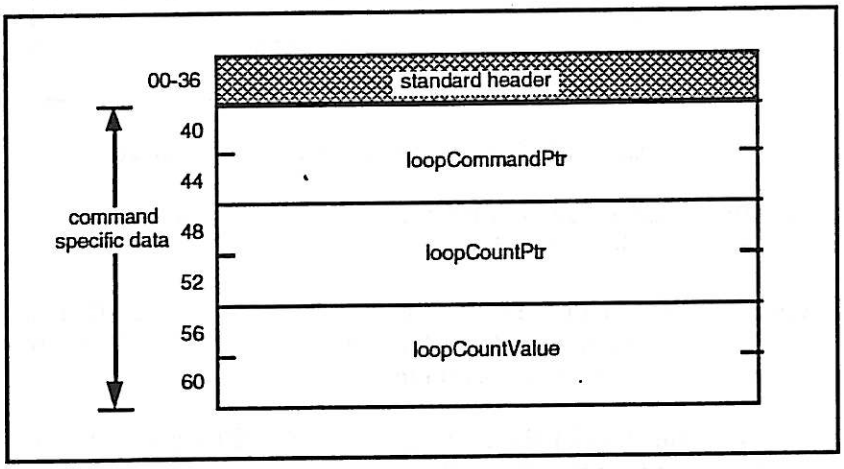

Figure 7-5: Loop Command-Entry Format

If the loop condition is satisfied, the **listCommandPtr** value specifies the address of the next command entry that is executed.

The value of **loopCountValue** shall be added to the internal intLoopCount value, and the result shall be written to the address specified by **loopCountPtr** (unless the pointer is null). The loop condition is satisfied if the result is positive and an error condition has not initiated skipping of command entries.

7.5 Attach Command

The format of the command-specific portion of the attach command is illustrated in Figure 7-6.

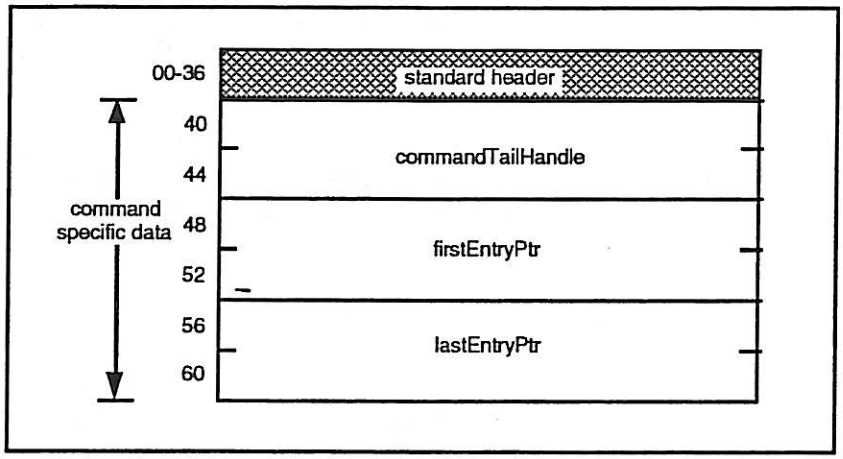

Figure 7-6: Attach Command-Entry Format

The specified list is appended to the command list whose listTailPtr address is specified by **commandTailHandle**. The first and last entries of the to-be-attached list are specified by **firstEntryPtr** and **lastEntryPtr** respectively.

8. Design Alternatives

In some cases, a final decision between two (or more) design alternatives has not been made. To maintain continuity while reading this document, only one of the design alternatives is contained in the main body of this document. The reader should understand that other design alternatives are being actively considered.

These alternative proposals will be maintained until there is a clear technical reason or organizational mandate to eliminate a design alternative. By formally including descriptions of active design alternatives, the significant ramifications of both alternatives can be evaluated in the context of (eventually detailed) technical documentation.

8.1 Direct-Mapped DMA Resources

Background: There is a significant amount of context that may be associated with each of the target's command lists, including the commandHeadHandle, the commandTailHandle, the bufSize parameter (if flow-control is supported) and the intLoopCount value (if command-list looping is supported).

Proposal: The information associated with each of the target's command lists should be directly accessible through memory-mapped control registers contained within the unit's address space. The cost of supporting the memory-mapped access capability is offset by the additional diagnostic and control capabilities that are provided.

Arguments: A better technical specification of the indirect-mapped (defined in Section 5) and direct-mapped DMA resources (defined here) is needed, as well as an understanding of when these resources are updated.

9. Appendix

9.1 C-Code Specification

The real specification in this document is in the C-code. It has been run using GNU CC 2.1 on a Solbourne 5E/500 running SunOS4.1.1. The compile command used was "`gcc -O -ansi -Wall`".

9.1.1 Code Extraction

Developers are expected to extract portions of this specification during their system development. To assist in this process, comments are included in the first and last line of the C code routines. To obtain the executable coherence code, the electronic form (Microsoft Word for the Macintosh) of this document should first be saved using the "Save As..." command, specifying the file name as "ascii_ccu" and the "File Format" as "Text Only with Line Breaks".

In the UNIX environment, the text-only "ascii_ccu" file can be processed by the stream editor (sed) to extract the ccu-code headers, "`ccuHeadX.h`" and the coherence-code C routines, "`ccuCode.c`". The UNIX shell program that performs this conversion is illustrated in Listing 9-1.

```
/*                    Listing 9-1: make_cache shell-file                    */
cat asciiSci | sed -n -f sedCmd1 > ccuHead1.h
cat asciiSci | sed -n -f sedCmd2 > ccuHead2.h
cat asciiSci | sed -n -f sedCmd3 > ccuHead3.h
cat asciiSci | sed -n -f sedCmd3 > ccuHead4.h
cat asciiSci | sed -n -f sedCmd3 > ccuHead5.h
cat asciiSci | sed -n -f sedCmdc > ccuCodeA.c
cat ccuHead1.h ccuHead2.h ccuHead3.h ccuHead4.h ccuHead5.h \
 ccuCodeA.c > ccuCode.c
```

This shell program assumes the existence of the stream-editor command files: "sedCmd1", "sedCmd2", "sedCmd3", "sedCmd4", "sedCmd5", "sedCmdc". The file sedCmd1 is used to extract the code located between (and including) the lines containing the commented words `HEAD1_BEGIN` and `HEAD1_END`. The other files sedCmd2(through sedCmd5) are used to similarly extract the code located between the commented words `HEAD2_BEGIN` (through `HEAD5_BEGIN`) and `HEAD2_END` (through `HEAD5_END`).

The file sedCmdc is used to extract the code located between (and including) the lines containing the commented words CODE_BEGIN and CODE_END. Special nonprinting characters are used in this section to disable the code-extraction process on the keywords used in this section. The first and final command files (sedCmd1 and sedCmdc) are illustrated in Listing 9-2 and Listing 9-3.

```
/*                  Listing 9-2: sedCmd1 sed-command files                  */
: start
n
s/HEAD1_BEGIN/HEAD1_BEGIN/
t first
b start
: first
p
: more
n
p
s/HEAD1_END/HEAD1_END/
t start
b more
```

```
/*                       Listing 9-3: sedCmdc sed-file                      */
: start
n
s/CODE_BEGIN/CODE_BEGIN/
t first
b start
: first
p
: more
n
p
s/CODE_END/CODE_END/
t start
b more
```

9.1.2 Simulation Environment

TBD - The old code-simulation environment has not been updated to reflect recent changes in the document. The old code should be updated and included in this document, to provide a simulation environment in which lint and gcc will produce no errors and the code functionality can be successfully tested.

9.2 Native SerialBus Adapters

The SerialBus uses a 64-bit Fixed addressing model, as defined within the CSR Architecture. This addressing model is scalable, in the since that it can also be applied to processor-to-memory applications as well as low-cost device-connection applications. For example, the Scalable Coherent Interface, which is optimized for multiprocessor-and-memory connections, uses the same 64-bit Fixed addressing model.

9.2.1 Address-Space Mappings

A bridge between a Scalable-Coherent-Interface-based HostBus and SerialBus need not perform address translations, since both interconnects support compatible 64-bit Fixed address space models. Such a bridge would selectively routes transactions, based on the value of the nodeId portion (the most-significant 16-bits) of their 64-bit physical address. The routing decision would be made by comparing the nodeId to one or several software-settable base/bound on the bridge.

A simple bridge could partition the system addresses into low and high spaces, where the high address space would correspond to host-bus addresses and the low address space would correspond to SerialBus addresses. Transactions with low addresses would be routed from SerialBus to HostBus; addresses with high addresses would be routed from HostBus to SerialBus.

TBD - provide a picture illustrating how HostBus and SerialBus addresses are mapped.

The HostBus-to-SerialBus transactions might be performed indirectly, to avoid the high latency associated with these transactions, as described in Section XX. However, these latencies are not a concern during normal system operation, since writes are the only HostBus-to-SerialBus transactions used during normal system operation, and these can be buffered in the bridge.

9.2.2 Buffered Write Transactions

In a pipelined high-performance system, write-transaction status may be returned to the host when the transaction has been queued by the HostBus side of a bridge but before it has been completed by the SerialBus responder. These are called buffered writes, since the write-transaction data is temporarily held in a bridge buffer.

For the read and write transactions specified by the CSR Architecture, the response status is expected to be returned from the responder. An implementation of a bridge can safely buffer these writes, if the response still appears to have been returned from the responder (i.e. the buffered writes are transparent to software).

For simple bus topologies, two ordering constraints are sufficient to make buffered writes transparent:

1) Request Ordering. After a buffered write is accepted, other read/lock/write requests (which pass through the bridge in the same direction) cannot bypass the buffered write.

2) Response Ordering. After a buffered write is accepted, other read/lock response subactions (which pass through the bridge in the same direction) cannot pass it.

To illustrate the need for the request ordering constraint, consider a processor which performs a CSR write (1) to change the NODE_IDS register on a SerialBus node (i.e. the SerialBus node address is changed). After the write has been buffered, the processor could reads a different value (2) from within that newly-assigned space, as illustrated in Figure 9-1. In the absence of ordering constraints, the read request can bypass the buffered write and return and incorrect address_error status. Note that simple address checks, which return the most-recent data from the buffered write, fail when the CSR write has an effect on other address spaces.

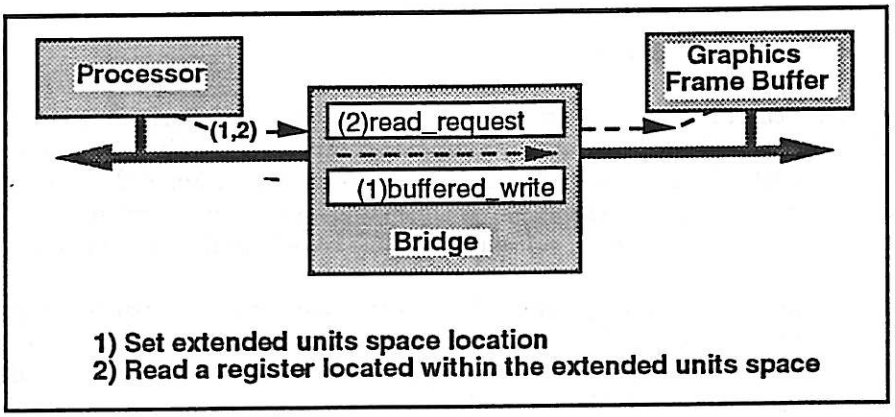

Figure 9-1: Request Ordering Violation

To illustrate the need for a response ordering constraint, consider two DMA controllers (target-A and target-B), where target-A is the producer, target-B is the consumer. In this example, the target-A-to-target-B data transfer is performed through an intermediate memory buffer (located in memory-B) and are controlled by a flow-control location (located in memory-A), as illustrated ion Figure 9-2.

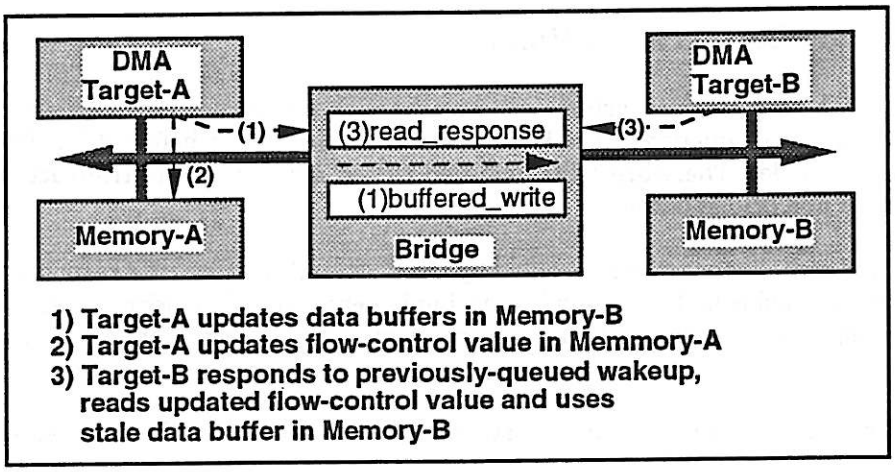

Figure 9-2: Response Ordering Violation

After target-A (1) completes the write which updates the shared memory buffer in memory-B, target-A (2) updates the flow--control location in memory-A. Note that the write to the shared memory buffer may be temporarily buffered in the bridge. When responding to a previously-queued wakeup, target-B could read the updated flow-control value in memory-A and incorrectly use the stale data in memory-B, while the original data-buffer write remains buffered in the bridge.

Note that buffered writes are only transparent in the absence of errors, since the responder's error status errors cannot be easily or uniformly returned to the requester. After a buffered HostBus-to-SerialBus write error, the bridge is expected to block other HostBus-to-SerialBus accesses (which may depend directly or indirectly of the buffered-write effects) until the error condition is explicitly cleared (presumably by higher-level I/O driver software). Blocking other accesses is necessary to safely avoid using potentially corrupted data.

9.2.3 Incoming Page Tables

Transactions can be transparently forwarded from SerialBus to HostBus, based on the transaction's address. However, the bridge between SerialBus and HostBus is expected to provide page-table entries for checking incoming system addresses. These tables would control:

1) Protection. Access to HostBus addresses could be selectively enabled.

2) Performance Options. The use of optional bus capabilities (such as optional 256-byte SCI transactions) could be selectively disabled, to avoid using optional transactions addressed to unsupportive address spaces.

3) Caching Modes. Specifies whether the SerialBus read/write/lock transactions should be mapped directly to SCI (using equivalent non-coherent transactions) or should be mapped to an adapter cache (whose updates generate coherent transaction sequences).

4) Swap Conversions. Selectively enables the conversion of SerialBus maskSwap transaction to a more-efficient HostBus list-append transaction sequence.

These host-software-settable protection tables would be located in system memory that is protected from remote system accesses, as illustrated in Figure 9-3. A bridge register would specify the base address of the page table, and page table entries could be cached in a small number of bridge-resident TLB entries.

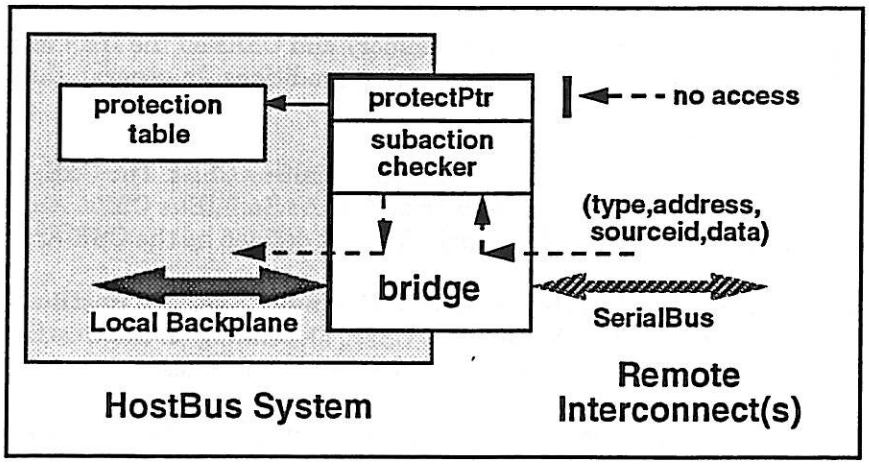

Figure 9-3: Remote Subaction Checking

By checking fields in the request or response subaction headers, protection tables could selectively enable remote system accesses based on the source's 16-bit nodeId value (**sourceId**), the transaction **type** (read, write, or lock), or the HostBus page address (**address**).

9.3 Foreign SerialBus Host Bus Adapter (HBA)

This section discusses a specific architecture of a Host Bus Adapter (HBA) for connecting an existing HostBus to one or several remote SerialBuses. The concepts presented apply to any HostBus which supports read and write operations.

An HBA is typically a printed circuit board that plugs into an expansion bus. Examples of expansion busses include NuBus, MCA, SBus and the AT bus. The HBA contains two sets of control-and-status-registers (CSRs), one set for implementing HostBus registers and one set for each of the attached SerialBuses, as illustrated in Figure 9-4.

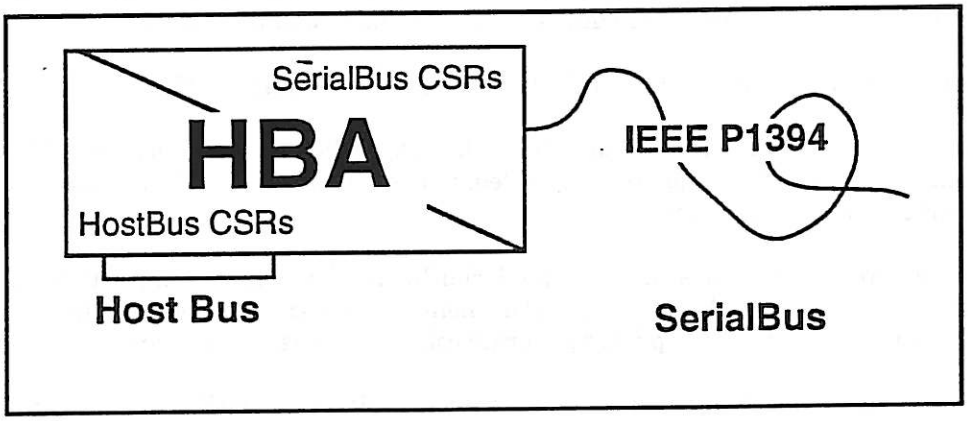

Figure 9-4: Host Bus Adapter (HBA) Components

The HBA provides an adaptation function from the HostBus to SerialBus. Although SerialBus defines several lock transactions (e.g., maskSwap8 and compareSwap8), the HostBus does not need to support those facilities, since the lock transactions (as used within the CCU Architecture) can be indirectly generated by the HBA hardware.

The HostBus resident processor updates system memory and CSRs on the HBA as needed to initiate and complete DMA operations and to monitor exception conditions. In general, the host processor would not be physically located on the HBA.

From a SerialBus perspective, the HBA is a node with SerialBus CSRs. The CSR Architecture and SerialBus standards constrain the function and format of the SerialBus CSRs. Access to ROM locations can be used to identify this node's capabilities, as defined by the CSR Architecture.

From a HostBus perspective, the HBA is a node with HostBus CSRs. The HostBus Architecture constrains the function and format of the HostBus CSRs. Depending on the HostBus Architecture, access to ROM locations may be used to identify this node's capabilities.

During normal operation, the HBA is involved in the following phases of DMA operations,:

1) Processor access of SerialBus CSR's. These SerialBus ROM registers are accessed during system initialization; the wakeup registers are accessed during normal system operation.

2) Processor-initiated appending to command lists in system memory.

3) SerialBus-initiated transfers to data buffers in system memory.

4) SerialBus-initiated appending of entries to status lists in system memory.

5) SerialBus-initated transfer of wakeup value to processor on HostBus.

This section describes one HBA design. However, the design of a HBA is beyond the scope of the CCU Architecture and is not constrained by this section.

This HBA implements several registers for indirectly generating all SerialBus transactions and maskSwap transaction sequences on HostBus. The names and sizes of these registers are illustrated in Figure 9-5. Note that an actual HBA may implement an additional register for host processor interrupt vectoring.

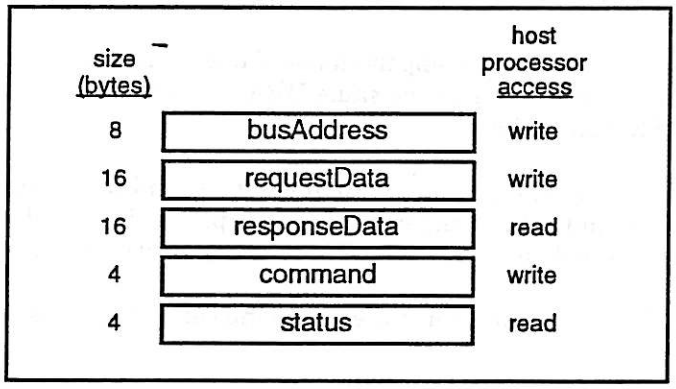

Figure 9-5: Special HBA Registers (HostBus Port)

The host processor uses these registers to indirectly access SerialBus CSR's and to append command entries to a shared list on SerialBus. The sections below describe these operations in more detail.

Note that the HBA could also provide a simple DMA interface for fetching its transaction-generation commands from system memory. A simple circular-queue DMA interface would be sufficient, since these transaction-generation commands can (without loss in performance) can be fetched in FIFO order and all transaction-generation commands can be quickly executed.

9.3.1 Processor-Initiated Accesses of SerialBus CSRs

Every SerialBus node provides a minimal set of CSRs, which can be accessed using the read4 and write4 transactions. The CCU Architecture requires additional CSR's to be implemented, some of which are accessed using the compareSwap8 transaction.

These SerialBus-resident CSRs can be accessed indirectly by the host-processor, by accessing the CSRs on the HostBus side of the HBA, as follows:

1) Address Setup. The processor writes the transaction-address value to the 8-byte *busAddress* register.

2) Request-Data Setup. The processor writes the request-data values to a portion of the 16-byte *requestData* registers. For a write transaction, this is the data values which are written. For a maskSwap transaction, this is the data and mask arguments for the atomic update.

3) Transaction Start. The SerialBus transaction is initiated by a processor-initiated write to the *command* register. The HostBus processor is interrupted when the transaction has completed.

4) Transaction End. The processor reads the *status* register to verify the completion of the SerialBus transaction.

5) Response-Data Return. The processor reads the returned data values from a portion of the 16-byte *responseData* registers. For a read transaction, this is the data values which are returned in the response. For a maskSwap transaction, this is the old memory value.

9.3.2 Processor-Initiated Appending To Command Lists

The host processor also uses the HBA to append a list of one or more command entries to a shared command list. The append operation uses the same HBA registers as those used for generating the previously-described CSR transactions.

The host processor first creates one or more command entries and links them into a command-group list. The structure of command entries and command groups are described elsewhere in this document. The host processor then updates the CSRs on the HostBus side of the HBA, as follows:

1) Address Setup. The processor writes the commandTailPtr address to the 8-byte *busAddress* register.

2) Request-Data Setup. The processor writes the addresses of the first and last command-group entries to the 16-byte *requestData* registers.

3) Transaction Start. The processor triggers a command-list-append operation by writing to the *command* register. The processor is interrupted when the command-list-append operation completes.

4) Completion Check. The processor reads the *status* register to confirm the successful completion of the command-list-append operation.

The HBA performs the append operation according to the protocol described elsewhere in this document. The append transaction sequence is equivalent to generating maskSwap8 and write8 transactions. The append sequence is divisible from the perspective of the processor, but (since it is always performed indirectly by the HBA), it appears to be indivisible from the perspective of SerialBus nodes.

9.3.3 SerialBus Transfers to HostBus-Resident System Memory

When a SerialBus target transfers data between itself and host memory, it performs SerialBus read and write transactions. In the example SerialBus HBA, these SerialBus read and write transactions result in direct access to system memory across the HostBus . The only requirement placed on the HostBus is that it support reads and 1-byte writes.

The SerialBus read and write transactions are forwarded transparently through the HBS, without processor intervention.

9.3.4 SerialBus Appending to Host-Resident Status Lists

If directed to do so, a SerialBus target appends a status entry to a status list when it completes a command., using the append protocol defined previously within this document.

From the perspective of the HostBus processor, the maskSwap8 transaction to HostBus memory is divisible. To avoid conflicts on statusTailPtr accesses, the processor (by software convention) always leaves one status entry in the status list. Since the processor never accesses the statusTailPtr value and all SerialBus maskSwap8 transactions are serialized through the HBA, the SerialBus maskSwap8 transactions appear to be performed indivisibly.

As part of the list append, the HBA may translate the write8 transaction into one or more HostBus writes. If more than one HostBus write is required, the HBA ensures that the contents of the highest address is updated last. Thus, the null status entry pointer remains null (i.e., the null bit remains set) until the full 8 bytes are valid.

9.3.5 SerialBus Wakeup of HostBus Processors

If directed to do so, a SerialBus target writes to the initiator wakeup register when it completes a command. The wakeup's write4 transaction is translated by the HBA into a HostBus processor interrupt signal.

The method by which the HBA interrupts the host processor depends on the HostBus characteristics and the HBA implementation. Some HBA's may use the write4 transaction to trigger the assertion of bus-dependent interrupt-request signals.

9.4 List Append/Extract Overview

This appendix contains an overview and examples of the shared list append and extract operations in a variety of scenarios. The examples in this section are for illustration and tutorial purposes only. These examples rely on the append and extract protocols defined elsewhere in this document.

There exists no situation under which a command append will fail due to insufficient target resources. This is true because the command append algorithm (outlined below) does not require any target resources, other than access to the appropriate CSR registers. This eliminates the need for special retry or error recovery software to handle boundary conditions.

9.4.1 Appending Command-Group Entries; Single Initiator

To append a new command group to an existing target command list, the initiator performs the following steps. Assume that the initiator has already constructed a command group consisting of one or more command entries, as described elsewhere in this document.

Step A Perform a maskSwap on the target's listTailPtr, providing the address of the last command-group entry being appended. Note that the nextEntryPtr field of the final command-group entry is null. The maskSwap places the address of the first command-group entry into the target's listTailPtr; the old listTailPtr value is returned to the initiator.

Step B Perform a write8 to the address specified by the old listTailPtr value. This action places the address of the first command-group entry into the address previously specified by tailPtr.

Step C Write this command-list's wake-up value to the target's wake-up register. This write activates the target by indicating that entries may have been appended to its command list.

These three steps are illustrated graphically in the following diagram and further described in Figure 9-6

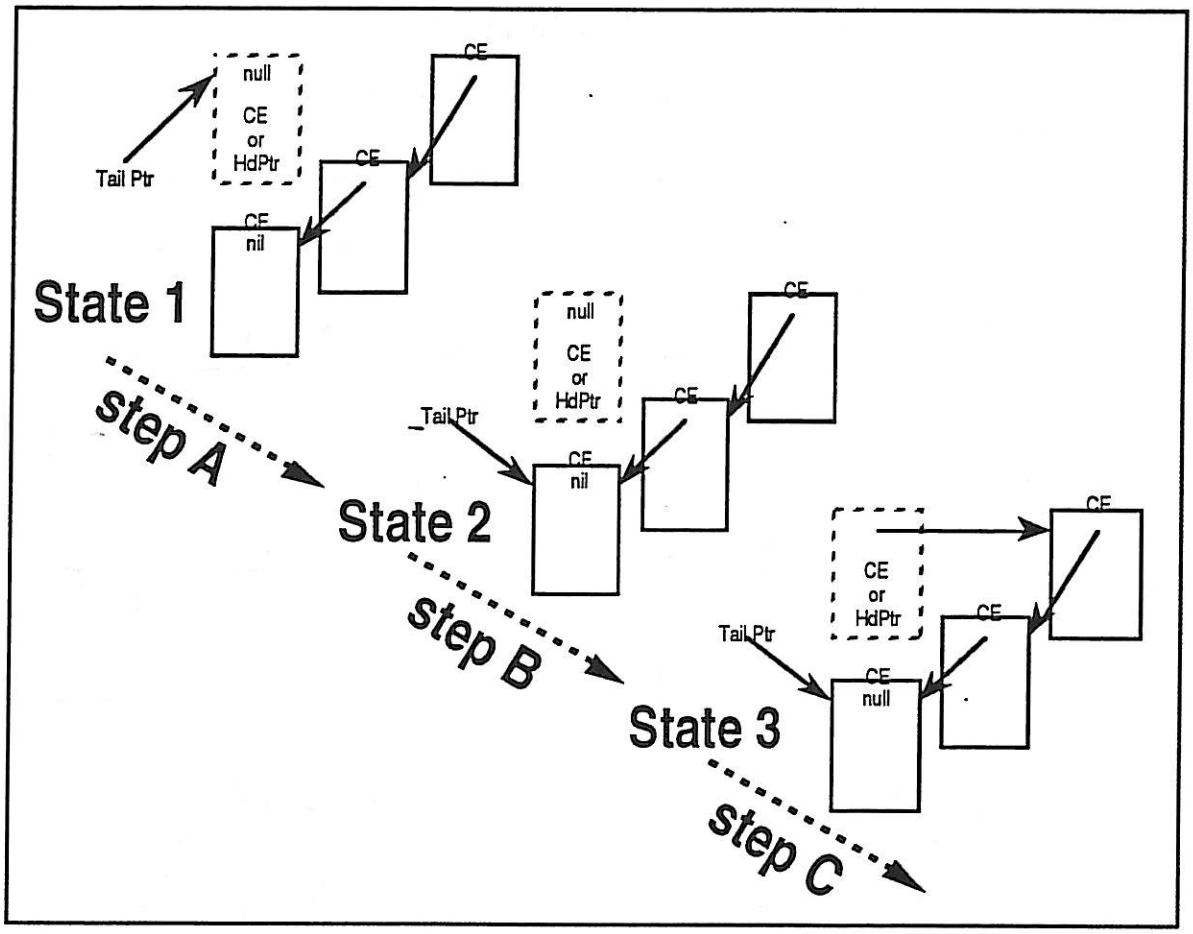

Figure 9-6: Command-Group Appending; Single Initiator

State 1 The listTailPtr points to either a commandEntry or to the listHeadPtr. The listTailPtr normally points to the location where the initiator will write the address of the next commandEntry being appended to the list.

State 2 The initiator has performed the swap on the listTailPtr thereby causing the listTailPtr to point to the last command-group entry being appended.

State 3 In step 2, the result returned from the swap on the listTailPtr was the address of the location to which the listTailPtr was previously pointing. The initiator writes the address of the first command-group entry to the address specified by the previous listTailPtr value, thereby adding a new command group to the existing target's command list.

Note that the state of the list prior to the above append operation is not relevant. Also note that the listHeadPtr plays no significant role in the list operation. It merely serves as a place holder for a pointer to the first command-list entry, in situations where the list is empty. When the list is not empty, the place holder is the nextEntryPtr field of the last command-list entry.

9.4.2 Appending Command-Group Entries; Multiple Initiators

Extending this algorithm to the case of 2 initiators concurrently appending command groups to the same target command list, refer To Figure 9-7 and the following text.

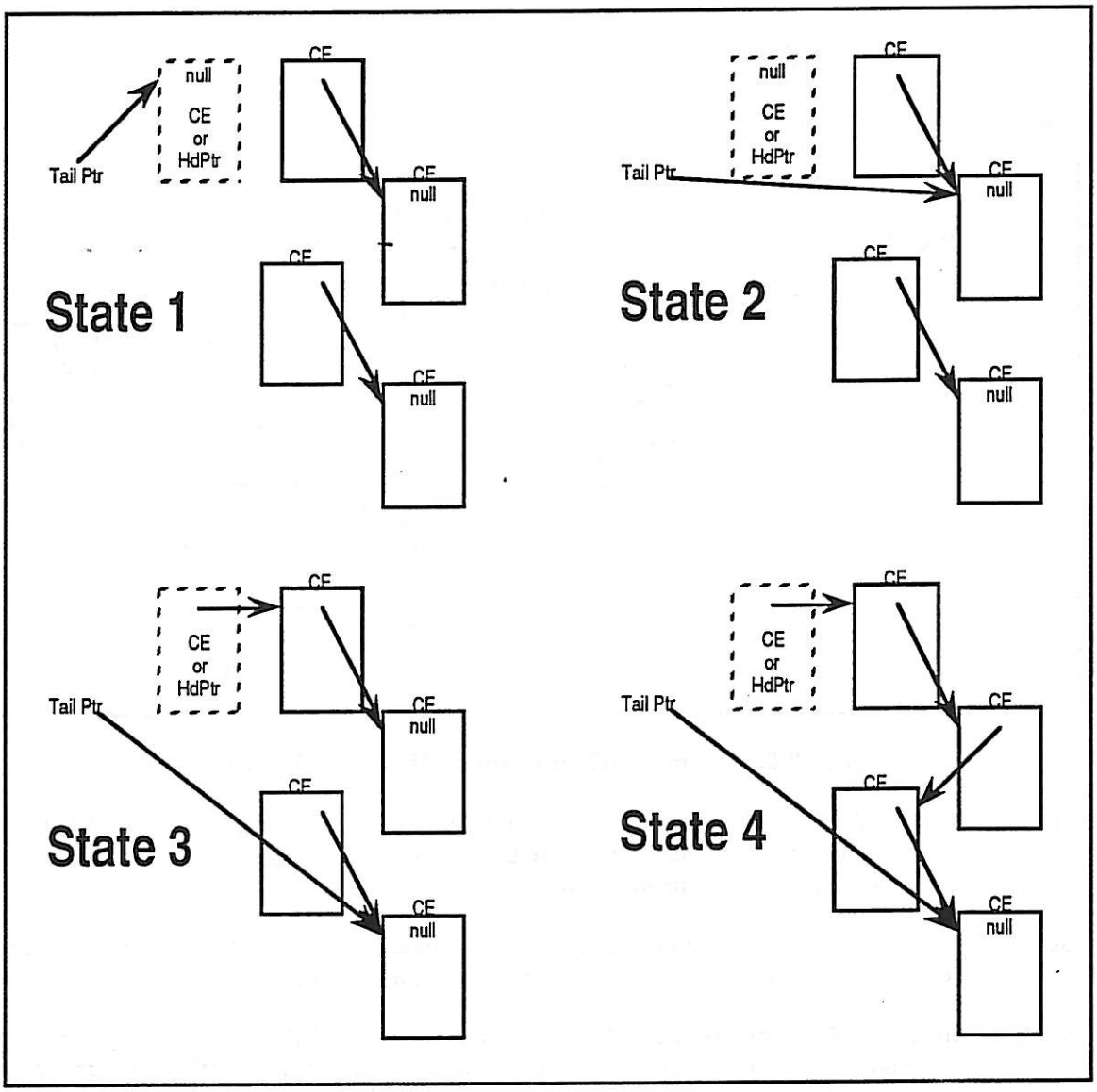

Figure 9-7: Command-Group Appending; Multiple Initiators

State 1 Initial state. The listTailPtr points to either the listHeadPtr or the nextEntryPtr in the last command-list entry. The value which the listTailPtr addresses contains a null pointer (i.e., end of list).

State 2 After Initiator-1 performs a swap on listTailPtr. The listTailPtr points to the last of initiator-1's command-group entries. When initiator-1 performs this swap, its response returns the old listTailPtr value; in this case, this is the address of either the listHeadPtr or the address of the last entry in the previously-existing command list.

State3 After initiator-2 performs a swap on listTailPtr. The listTailPtr points to the last initiator-2's command-group entries. When initiator-2 performs this swap, its response returns the old listTailPtr value; in this case, this is the address of the last entry in initiator-1's newly-appended command group.

Also after initiator-1 performs a swap on the address specified by the previously-returned listTailPtr value. The result of the last swap was used as an address on which to perform the next write. This results in the previously null pointer pointing to the first of initiator-1's command-group entries.

State 4 Initiator-2 performs a write8 on the address specified by the previously-returned listTailPtr value. The nextEntryPtr value in initiator-1's last command-group entry now points to the first of initiator-2's command-group entries.

After initiator-1 and initiator-2 have performed the above operations, they write to the target's wake-up register. This may have no effect if the target is already awake (fetching previously-queued commands), may wakeup the target once, or may wakeup the target twice. The number of serviced wakeup events is not important, since at least one wakeup is received after the final command-group has been appended

Note that there is a window of time during which the pointers are not in a consistent state. This occurs in state 3 where the listTailPtr points to the last of initiator-2's command-group entries, the previous "listHeadPtr" points to the first of initiator-1's command-group entries, and the last entry in initiator-1's command-group does not yet point to the first of initiator-2's command-group entries. The extract mechanism, described in the next section, explains how the target detects this condition, and how it ensures the protocol functions correctly.

9.4.3 Extracting Command-List Entries

This section covers the steps that a target goes through to read the linked list of commandEntries and remove items from the list. The extract protocol complements the append protocol described in the above section. As the initial state, assume that the target has just been awakened by an initiator writing to the target's wake-up register. At this point, the target performs the following steps:

Step 0 Wait for a write to the target's wakeup register.

Step 1 Read the listHeadPtr. This contains the address of the next commandEntry to be processed.

Step 2 Read and process commandEntries until the nextEntryPtr field is null. Do not process this command.

Step 3 Write to the listHeadPtr, replacing its old value with null.

Step 4 Perform a compareSwap on the listTailPtr. The arguments for this operation are as follows:

test = Address of last commandEntry read (but not yet processed)

data = Address of listHeadPtr

Step 5 If the compareSwap of step 4 succeeds , process the last commandEntry. Continue by returning to step 0.

Step 6 If the compareSwap of step 4 fails, write the address of the last commandEntry (which has not yet been processed) into listHeadPtr.
Continue by returning to step 0.

10. TBDs

The section contains brief descriptions of issues that should be addressed in the future. Readers are encouraged to submit proposed solutions (which reduce the size of this list) or additional issue descriptions (which increase the size of this list).

1) Initialization. More work is needed to unambigously define the initialization process, which should be well defined without unnecessarily restricting implementation technologies. Specific considerations include:

   a) Shared Boot Devices. How can a boot device (located on a shared SerialBus) be shared by two or more processors (located on separate processor/memory buses)? Well defined software-based interlock protocols are needed.

   b) Corner Cases. Describe how updates of the unit's control registers are synchronized. More specifically, define the affect of writing a second value to the control register before the processing of the first value has completed. Perhaps this is undefined, except for the RESET cmd value, which has precedence.

2) Access Control. Although command lists can be physically shared, access protocols may be required to properly synchronize the use of these shared command lists. The current thought is that command lists could have the following sharing properties:

   Free - command list is not being used.
   Shared - command list is shared, higher-level access control is not required.
   Checked - command list cannot be accessed directly, but command lists are attached (after access rights have been validated) by the list owner.
   Exclusive - command list may only be accessed by the list owner.

   Note that a unit may have many lists, and the owners of these lists are not required to be the same.

3) Re-joining Attach Lists. Consider in more detail the completion of the dependent command lists which are appended to a secondary target. How are the status reports from the primary and secondary targets merged, before being returned to the primary initiator?

4) Reserved. Accurately define reserved (and its abbreviated name res). Should always be set to zero when written, and should always be ignored when read.

5) Exact Data Transfers. We should clearly state how the target can be programmed to access memory using the read1/2/4/8/16/64 and write1/2/4/8/16/64 transactions.

11. Index