1394
Open Host Controller Interface Specification

Release 1.00
October 20, 1997

PREFACE

Notice

Intellectual Property

Implementation of this Specification is governed by the terms of the 1394 Open Host Controller Interface Patent License Agreement.

This specification may contain and sometimes even require the use of intellectual property owned by others. Rights to such intellectual property are not conveyed except as provided by the 1394 Open HCI Promoters agreement and the 1394 Open HCI Adopters agreement.

Information

An on-line copy, updates, and notices regarding this specification will be maintained on the following FTP site:
ftp://www.austin.ibm.com/pub/chrptech/1394ohci

Questions, comments, and issues concerning this document should be directed to the 1394 Open HCI reflector:
1394ohci-l@austin.ibm.com

Promoters

The Promoters of record on October 20, 1997, the date of publication of the 1394 Open Host Controller Interface Specification, Release 1.00, are:

Apple Computer, Inc.
Compaq Computer Corporation
Intel Corporation
Microsoft Corporation
National Semiconductor Corporation
Sun Microsystems, Inc.
Texas Instruments, Inc.

Contributors

This specification was developed using Apple Computer's *Pele* design as a starting point. The *Pele* contributors were Jim Baldwin, Kevin Christiansen, Nikhil Jayaram, Michael Johas Teener and Rahoul Puri. The original Editor of the 1394 OpenHCI specification up through Draft 0.7, was Michael Johas Teener.

The following is a list of key contributors to the 1394 Open Host Controller Interface specification.

Lee Wilson, *Chair*
Diana Klashman, *Editor*

Eric W. Anderson
Richard Baker
Joe Bennett
Mike Eneboe
John Fuller
Jerry Hauck
Robert Macomber
Rahoul Puri
Michael Johas Teener
Peter Teng
Scott Smyers
Erik Staats
David Wooten

The following is a list of other major participants (those who attended at least three meetings and/or conference calls).

Larry Blackledge	Vasanta Madduri
Dmitriy L. Budko	Neil Morrow
Josh Collier	Yehuda Peled
Nobuo Furuya	Gerhard Ringel
Carl Humphreys	Curtis Stevens

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Related documents

The following documents may be useful in understanding the terms and concepts used in this specification. The documents are for general background purposes only and are not incorporated into and do not form a part of this specification.

[A] IEEE 1394-1995 High Performance Serial Bus
IEEE, 1995

[B] ISO/IEC 13213:1994 Control and Status Register Architecture for Microcomputer Busses
International Standards Organization, 1994

[C] IEEE P1394a
IEEE Draft Standard for a High Performance Serial bus (Supplement), Work-in-Progress

All references to 1394 in this document refer to IEEE 1394-1995 ([A] above) unless otherwise specified.
Following IEEE conventions, the term “quadlet” is used throughout this document to specify a 32-bit word.

1.2 Overview

The 1394 Open Host Controller Interface (**Open HCI**) is an implementation of the link layer protocol of the 1394 Serial Bus, with additional features to support the transaction and bus management layers. The 1394 Open HCI also includes DMA engines for high-performance data transfer and a host bus interface.

IEEE 1394 (and the 1394 Open HCI) supports two types of data transfer: asynchronous and isochronous. Asynchronous data transfer puts the emphasis on guaranteed delivery of data, with less emphasis on guaranteed timing. Isochronous data transfer is the opposite, with the emphasis on the guaranteed timing of the data, and less emphasis on delivery.

1.2.1 Asynchronous functions

The 1394 Open HCI can transmit and receive all of the defined 1394 packet formats. Packets to be transmitted are read out of host memory and received packets are written into host memory, both using DMA. The 1394 Open HCI can also be programmed to act as a bus bridge between host bus and 1394 by directly executing 1394 read and write requests as reads and writes to host bus memory space.

1.2.2 Isochronous functions

The 1394 Open HCI is capable of performing the cycle master function as defined by 1394. This means it contains a cycle timer and counter, and can queue the transmission of a special packet called a “cycle start” after every rising edge of the 8 kHz cycle clock. The 1394 Open HCI can generate the cycle clock internally (required) or use an external reference (optional). When not the cycle master, the 1394 Open HCI keeps its internal cycle timer synchronized with the cycle master node by correcting its own cycle timer with the reload value from the cycle start packet.

Conceptually, the 1394 Open HCI supports one DMA controller each for isochronous transmit and isochronous receive. Each DMA controller can be implemented to support up to 32 different DMA channels, referred to as *DMA contexts* within this document.

The isochronous transmit DMA controller can transmit from each context during each cycle. Each context can transmit data for a single isochronous channel.

The isochronous receive DMA controller can receive data for each context during each cycle. Each context can be configured to receive data from a single isochronous channel. Additionally, one context can be configured to receive data from multiple isochronous channels.

1.2.3 Miscellaneous functions

Upon detecting a bus reset, the 1394 Open HCI automatically flushes all packets queued for asynchronous transmission. Asynchronous packet reception continues without interruption, and a token appears in the received request packet stream to indicate the occurrence of the bus reset. When the PHY provides the new local node ID, the 1394 Open HCI loads this value into its Node ID register. Asynchronous packet transmit will not resume until directed to by software. Because target node ID values may have changed during the bus reset, software will not generally be able to re-issue old asynchronous requests until software has determined the new target node IDs.

Isochronous transmit and receive functions are not halted by a bus reset, instead they restart as soon as the bus initialization process is complete.

A number of management functions are also implemented by the 1394 Open HCI:

a) A global unique ID register of 64 bits which can only be written once. For full compliance with higher level standards, this register must be written before the boot block is read. To make this implementation simpler, the 1394 Open HCI optionally has an interface to an external hardware global unique ID (GUID, also know as the IEEE EUI-64). An example device is the Dallas Semiconductor DS2501-EUI-64.
b) Four registers that implement the compare-swap operation needed for isochronous resource management.

1.3 Hardware description

Figure 1-1 provides a conceptual block diagram of the 1394 Open HCI, and its connections in the host system. The 1394 Open HCI attaches to the host via the host bus. The host bus is assumed to be at least 32 bits wide with adequate performance to support the data rate of the particular implementation (100Mbit/sec or higher plus overhead for DMA structures) as well as bounded latency so that the FIFO's can have a reasonable size.

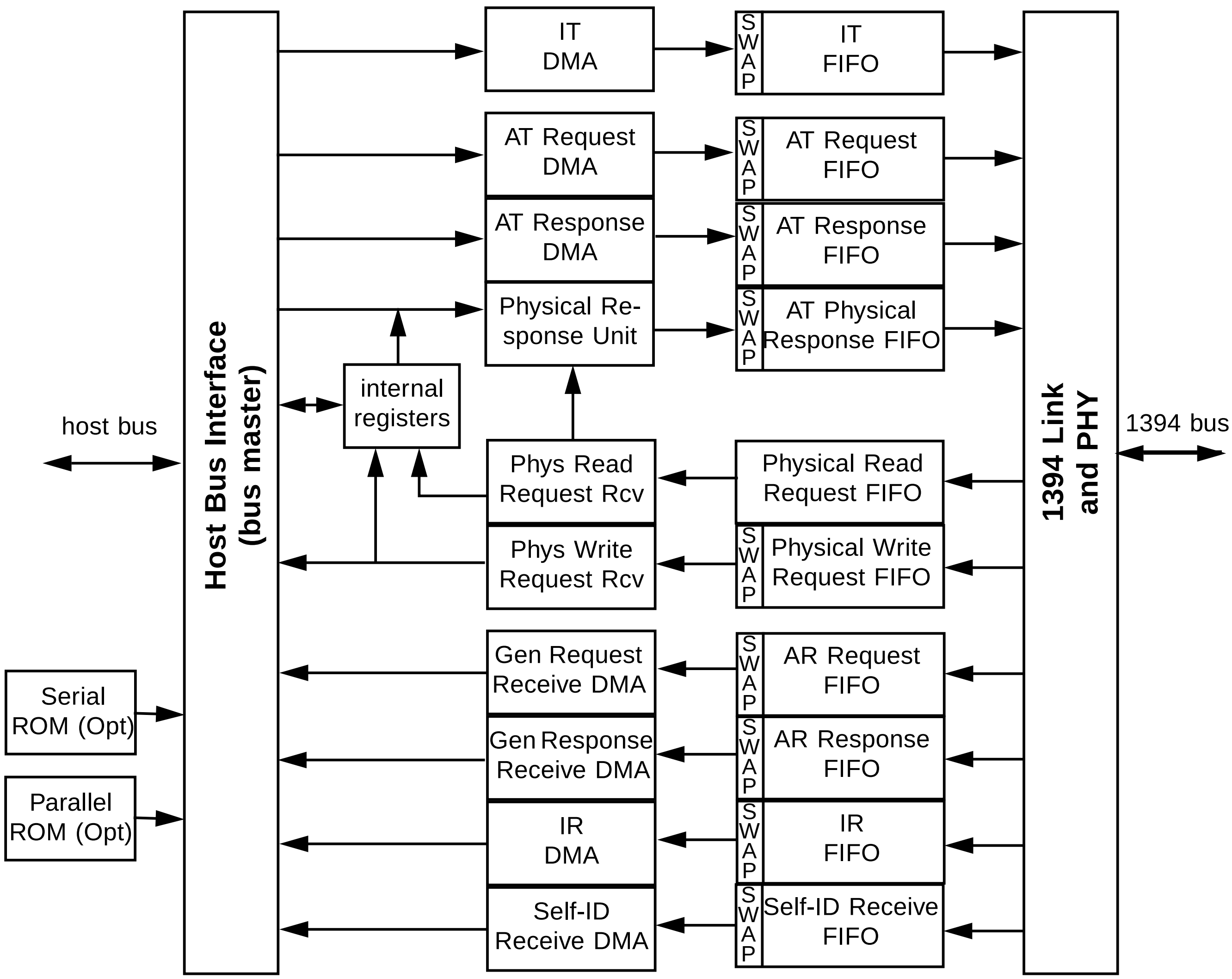


Figure 1-1 — 1394 Open HCI conceptual block diagram

1.3.1 Host bus interface

This block acts both as a master and a slave on the host bus. As a slave, it decodes and responds to register access within the 1394 Open HCI. As a master, it acts on behalf of the 1394 Open HCI DMA units to generate transactions on the host bus. These transactions are used to move streams of data between system memory and the devices, as well as to read and write the DMA command lists.

1.3.2 DMA

The 1394 Open HCI supports seven types of DMA. Each type of DMA has reserved register space and can support at least one distinct logical data stream referred to as a *DMA context*.

Table 1-1 — DMA types and contexts

DMA type	number of contexts
Asynchronous Transmit	1 Request, 1 Response
Asynchronous Receive	1 Request, 1 Response
Isochronous Transmit	4 minimum, 32 maximum
Isochronous Receive	4 minimum, 32 maximum
Self-ID Receive	1
Physical Receive & Physical Response	0 (not programmable like those above)

Each asynchronous and isochronous context is comprised of a buffer descriptor list called a *DMA context program*, stored in main memory. Buffers are specified within the DMA context program by *DMA descriptors*. Although there are some differences from controller to controller as to how the DMA descriptors are used, all DMA descriptors use the same basic format. The DMA controller sequences through its DMA context program(s) to find the necessary data buffers. This frees the system from stringent interrupt response requirements after buffer completions. The mechanism for sequencing through DMA contexts differs somewhat from one controller to the next and is described in detail for each type of DMA in its respective chapter.

The Self-ID receive controller does not utilize a DMA context program and consists instead of a pair of registers; one to be configured by software, and one to be maintained by hardware.

The 1394 Open HCI also has physical request DMA controller that processes incoming requests that read directly from host memory. This controller does not have a DMA context, it is instead controlled by dedicated registers.

1.3.2.1 Asynchronous transmit DMA

Asynchronous transmit DMA (AT DMA) utilizes three data streams, one each for AT DMA request, AT DMA response, and the Physical Response Unit. These three functions can share resources.

AT DMA request and AT DMA response move transmit packets from buffers in memory to the corresponding FIFO (request transmit FIFO or response transmit FIFO). For each packet sent, it waits for the acknowledge to be returned. If the acknowledge is *busy*, the DMA context will resend the packet up to a software-configurable number of times for single-phase retry, or up to a software-configurable time limit for dual-phase retry.

When the receive DMA indicates that a physical read has been received, the Physical Response Unit takes over to send the response packet. The Physical Response Unit can only interrupt the AT DMA response controller or AT DMA request controller between packets.

The asynchronous transmit DMA supports either the single-phase retry protocol (retry_X) or the dual-phase retry protocol (retry_1/retry_A/retry_B).

1.3.2.2 Asynchronous receive DMA

The asynchronous receive DMA (AR DMA), contains two DMA controllers: the Physical Request Unit and the AR DMA controller.

The Physical Request Unit takes control when a request with a physical address is received. There are three types of physical addresses: host memory addresses (corresponding to the 4Gbyte address of a typical 32-bit CPU), compare-swap management addresses, and the bus_info_block.

The AR DMA controller handles all incoming asynchronous packets not handled by the other functions in the AR DMA. It consists of two contexts, one for asynchronous response packets, and one for asynchronous request packets. Each packet is copied into the buffers described by the corresponding DMA context program. Note that received lock requests not targeted to one of the four compare-swap management registers are always handled by the AR DMA request context.

It is recommended that Open HCI asynchronous receive support dual-phase retry.

1.3.2.3 Isochronous transmit DMA

The isochronous transmit DMA controller supports a minimum of four isochronous transmit DMA contexts and can be implemented to support up to 32 isochronous transmit DMA contexts. Each context is used to transmit data for a single isochronous channel. Data can be transmitted from each IT DMA context during each isochronous cycle.

1.3.2.4 Isochronous receive DMA

The isochronous receive DMA controller supports a minimum of four isochronous receive DMA contexts and can be implemented to support up to 32 isochronous receive DMA contexts. All but one IR DMA context is used to receive packets from a single isochronous stream (channel). One context, as selected by software, can be used to receive packets from multiple isochronous streams (channels).

Isochronous packets in the receive FIFO are processed by the context configured to receive their respective isochronous channel numbers. Each DMA context can be configured to strip packet headers or include the headers and trailers when moving the packets into the buffers. In addition, each DMA context can be configured to concatenate multiple packets into its buffers (bufferFill mode) or to place just a single packet into each set of buffers (packet-per-buffer mode).

1.3.2.5 Self-ID receive DMA

Self-ID packets (received during the bus initialization self-ID phase) are automatically routed to a single designated host memory buffer by 1394 Open HCI self-ID receive DMA. Each time bus initialization occurs, the new self-ID packets will be written into the self-ID buffer from the beginning of the buffer, thereby overwriting the old self-ID packets.

1.3.3 Global unique ID (GUID) interface

The optional GUID (EUI-64) interface is intended to interface to an external ROM device from which the 1394 64-bit "node_unique_ID" may be loaded. If this interface is provided and an external device is present, the GUID_ROM bit in the Version Register is set and the GUID will be automatically loaded from the external ROM device following a hardware reset. This interface is required for Host Controllers that are intended to be used on add-in cards. The specifics of the interface to the external ROM device are outside the scope of this specification.

1.3.4 FIFOs

Data entering or leaving the FIFO's is conditionally byte-swapped. The 1394 Open HCI is designed to run in both little-endian environments (x86/PCI) and byte-swapped big-endian environments (PowerMac/PCI). Note, however, that the 1394 standard specifies that data is treated as big-endian, with the most significant byte of a doublet, quadlet, or octlet transmitted first. This means that the data coming through the FIFOs should be byte swapped if it is intended for a byte-swapped little-endian PCI like the PowerMac (two byte-swap operations leaves the data in the original big-endian 1394 format). Little-endian x86 systems may or may not want the data byte swapped, so there is an Open HCI control flag to enable byte swapping for 1394 packet data.

1.3.4.1 Asynchronous transmit FIFOs

The asynchronous transmit FIFOs are temporary storage for non-isochronous packets that will be sent from the Host Controller to devices on 1394. The asynchronous request FIFO is loaded by the asynchronous request DMA unit, the asynchronous response FIFO is loaded by the asynchronous response DMA unit and the physical response FIFO is loaded by the physical DMA response unit.

It is not required that these FIFOs be implemented as separate physical entities. A single FIFO can be used for all asynchronous transmit packets as long as the implementation prevents pending asynchronous requests and asynchronous responses from blocking each other. For example, if a read request is being sent to a 1394 device that is returning ack_busy, this should not prevent responses from either the physical DMA unit or the asynchronous response unit from being sent. Furthermore, a busied response from the asynchronous response unit should not block responses from the physical DMA unit. Other sections of this specification will provide implementation guidelines that will help ensure that the non-blocking requirements can be met with a single asynchronous transmit FIFO.

1.3.4.2 Isochronous transmit FIFO

The isochronous transmit FIFO, is temporary storage for the isochronous transmit data. It is filled by the ITDMA and is emptied by the transmitter.

1.3.4.3 Receive FIFOs

Conceptually there are several receive FIFOs for handling incoming asynchronous requests, asynchronous responses, isochronous packets and self-ID packets. The FIFOs are used as a staging area for packets which will be routed to the appropriate handler. There is no requirement on the number of hardware FIFOs that must be implemented to provide the required functionality set forth in this document. However, any specific FIFO implementation must ensure that physical requests, asynchronous requests, asynchronous responses, isochronous packets, and self-ID receive contexts proceed independently and do not block each other.

For example, if a unified receive FIFO is used and the transaction layer request queue is busy or stopped, all other received packet types (physical requests, asynchronous responses, isochronous packets, and self-ID packets) must still pass through the FIFO and be delivered to the transaction layer or host bus interface. Other sections of this specification will provide implementation guidelines that will help ensure that the non-blocking requirements can be met with a single receive FIFO.

1.3.5 Link

The link module sends packets which appear at the transmit FIFO interfaces, and places correctly addressed packets into the receive FIFO. It includes the following features.

- Transmits and receives correctly formatted 1394 serial bus packets.
- Generates the appropriate acknowledge for all received asynchronous packets, including support for both the single and dual phase retry protocol for received packets.
- Performs the function of cycle master.
- Generates and checks 32-bit CRC.
- Detects missing cycle start packets.
- Interfaces to Open-HCI-compliant PHY.

- Receives isochronous packets at all times (does not ignore isochronous packets received outside of the expected period between cycle start and a subaction gap). This supports asynchronous streams and allows isochronous data to be received even if there is a CRC error in a received cycle start.
- Ignores asynchronous packets received during the isochronous phase (such packets are not ack'ed and isochronous phase continues).

The acknowledges generated by the link depend on the type of received packet, the address and the state of the OpenHCI FIFOs:

Table 1-2 — Link generated acknowledges

Acknowledge	Condition
ack_complete	A packet with good CRC in both the header and data block (if there is one) and which also falls into one of the following classifications: a) Any response that is accepted from 1394. b) A write request with the offset address between 48'h0 and the configurable (optional) PhysicalUpperBound-1 or 48'0000_FFFF_FFFF when i) *posted writes* are enabled, ii) the request will be handled as a physical request, and iii) the number of outstanding posted writes is within the implementation specific limit. c) A write request with the offset address between either the configurable (optional) PhysicalUpperBound or 48'h0001_0000_0000, and 48'hFFFE_FFFF_FFFF that can be fully copied into the host memory receive buffer. **NOTE:** For further information on implementation requirements for posted writes, see Section 3.3.3.
ack_pending	A packet with good CRC in both the header and data block (if there is one) and which also falls into one of the following classifications: a) Any read request that can be fully loaded into the receive buffer. b) Any lock request that can be fully loaded into the receive buffer. c) Any block request with a non-zero extended tcode. d) A write request with the offset address between 48'hFFFF_0000_0000 and 48'hFFFF_FFFF_FFFF (the top 4GB, which includes the register space) that can be fully loaded into the receive buffer.
ack_busy_X, ack_busy_A, ack_busy_B	Any received packet with a good CRC in both the header and data block (if there is one) that cannot be fully loaded into the receive buffer. (The choice of _X, _A, or _B depends on the choice of acknowledge algorithm and the particular "rt" value of the received packet.)
ack_data_error	Any received packet with a good header CRC and a bad data CRC.
ack_type_error	For a block write request with a good CRC in both the header and data block, this error ack: • May be returned when the data_length is larger than the size indicated in the max_rec field of the Bus_Info_Block of the Host Controller. • Shall be returned if data_length is larger than max_rec *and* the request is not handled by the physical response unit. For a block read request with a good CRC in the header, this error ack may be returned when the data length is larger than the size indicated in the max_rec field of the Bus_Info_Block of the Host Controller and the request is handled by the physical response unit.

1.4 Software interface overview

There are three basic means by which software communicates with the 1394 Open HCI: registers, DMA, and interrupts.

1.4.1 Registers

The host architecture (PCI, for example) is responsible for mapping the 1394 Open HCI's registers into a portion of the host's address space.

1.4.2 DMA operation

DMA transfers in the 1394 Open HCI are accomplished through one of two methods:

a) DMA. Memory resident data structures are used to describe lists of data buffers. The 1394 Open HCI automatically sequences through this buffer descriptor list. This data structure also contains status information regarding the transfers. Upon completion of each data transfer, the DMA controller conditionally updates the corresponding DMA Context Command and conditionally interrupts the processor so it can observe the status of the transaction. A set of registers within the 1394 Open HCI is used to initialize each DMA context and to perform control actions such as starting the transfer.

b) Physical response DMA. The 1394 Open HCI can be programmed to accept 1394 read and write transactions as reads and writes to host memory space. In this mode, the 1394 Open HCI acts as a bus bridge from 1394 into host memory.

The formats for the data sent and received in all these modes are specified in the applicable chapters.

1.4.3 Interrupts

When any DMA transfer completes (or aborts) an interrupt may be sent to the host system. In addition to the interrupt sources which correspond to each DMA context completion, there is also a set of interrupts which correspond to other 1394 Open HCI functions/units. For example, one of these interrupts could be sent when a selfID packet stream has been received.

The processor interrupt line is controlled by the IntEvent and IntMask registers. The IntEvent register indicates which interrupt events have occurred, and the IntMask register is used to enable selected interrupts. Software writes to the IntEventClear register to clear interrupt conditions in IntEvent.

In addition, there are registers used by the isochronous transmit and isochronous receive controllers to indicate interrupt conditions for each context.

1.5 1394 Open HCI Node Offset (Address) Map

OpenHCI divides the 48-bit node offset space as depicted below:

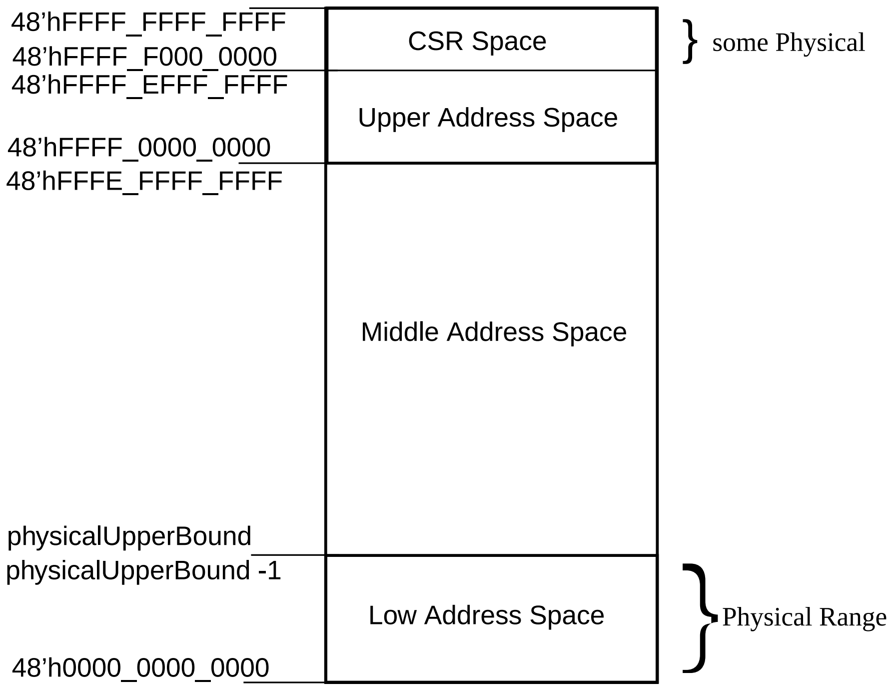


Figure 1-2 — Node Offset Map

Low Address Space is from 48’h0 up to physicalUpperBound. Asynchronous read and write requests into this range can be handled by the Physical Request/Physical Response units, providing an efficient mechanism for moving asynchronous data. Whether or not a request can be handled in this manner depends on a set of criteria as described in section 12. For write requests which are handled by the Physical Request unit, the Host Controller may issue an ack_complete immediately, even before the data has been written to host memory, to maximize packet transaction efficiency (this is referred to as a *Posted Write*). Or, depending on circumstances, the Host Controller may instead issue an ack_pending for such requests.

physicalUpperBound is an optional register that some Host Controllers may implement which provides a means to change the upper bound of the low address space. If not implemented, the Host Controller uses a default physical upper bound of 48’h0001_0000_0000, which provides a physical range of 4GB. If implemented, 64-bit systems can use the physicalUpperBound register to increase the size of the Physical Range.

Middle Address Space is from physicalUpperBound through 48’hFFFE_FFFF_FFFF. Packets with destination offsets within this range are not candidates for handling by the Physical Request/Response units, and are instead passed to software for processing. Although there will be added latency while software performs processing, the Host Controller nevertheless issues an ack_complete for all write requests within this range which normally require an ack (e.g., broadcast write requests are never ack’ed). This is to maximize packet transaction efficiency. However, although the node that issued the write request is informed (via the ack_complete) that the write succeeded, it is possible that an error may occur and that the write does not in fact reach its destination. This address range is best suited to protocols such as TCP/IP for example which have their own mechanisms for detecting and recovering from lost packets.

Upper Address Space is from 48’hFFFF_0000_0000 to 48’hFFFF_EFFF_FFFF. Packets with destination offsets within this range are not candidates for handling by the Physical Request/Response units, and are instead passed to software for processing. The Host Controller will respond to write requests to this range with an ack_pending, and software will issue a write response with resp_complete only after the data has been written to its specified destination. This range is best suited to protocols that do not tolerate lost packets.

CSR Space is from 48’hFFFF_F000_0000 to 48’FFFF_FFFF_FFFF providing a range of 256MB. This range is the reserved register space as specified in ISO/IEC 13213:1994. Most packets with destination offsets within this range are not candidates for handling by the Physical Request/Response units, and are instead passed to software for processing. Some however are handled directly by the Host Controller without involving software and are listed in section 12.

1.6 System Requirements

This Host Controller specification is intended to be largely independent of the type of system to which it is attached. The intent is that Host Controller designs that follow this specification may be built for many different types of systems and still adhere to the same programming model. The required system facilities are:

a) Host Controller must be able to initiate accesses of host system memory,
b) Host Controller must be able to modify system memory with byte granularity,
c) Host Controller must be able to signal an exception/interrupt to the host CPU,
d) access of 32-bit entities in either system memory or on the Host Controller must be endian neutral and atomic. No 8-bit or 16-bit access to Host Controller registers are supported.

The 1394 Open HCI does not preclude a system from having multiple 1394 Open HCI controllers.

1.7 Alignment

1.7.1 Data alignment

The 1394 Open HCI must perform these two alignment functions:

a) Translate between the byte alignments of the host-based data and the quadlet aligned FIFO. For instance, if a 5 byte 1394 data packet is to be stored at host bus address 6, then the first two bytes of the first data quadlet in the FIFO must be stored at host bus address 6 and 7 using a single quadlet write, then the next two bytes of the first quadlet in the FIFO combined with the first byte of the next quadlet in the FIFO are written to host bus address 8, 9, and 10.
b) Stuff extra zero bytes into the transmit FIFO when the number of bytes to transmit is not an integral number of quadlets

1.7.2 Memory structure and buffer alignment

Alignment requirements for host memory data structures and host memory buffers can be found in sections of this document where those elements are described.

2. Conventions - Notation and Terms

2.1 Notation

2.1.1 Numeric Notation

Unless otherwise specified, numbers will be represented in Verilog language style. In particular, numbers with a "'h" prefix are hexadecimal, "'b" are binary, and "'d" or those without a prefix are decimal. If a number precedes the " ' ", then it indicates the length of the number in bits. For example, 4'h8 is the binary number 'b1000.

2.1.2 Register Notation

There are two types of registers described in this document; read/write registers and set and clear registers. The notation used for each is described below, as well as notation used for register reset values and reserved fields and registers.

2.1.2.1 Read/Write registers

Read/write registers are registers for which a single address is defined and for which fields may be defined with one or more of the following attributes:

Table 2-1 — read/write register field access tags

access tag (rwu)	name	meaning
r	read	field may be read
w	write	field may be written from the host bus
u	update	field may be autonomously updated by Open HCI hardware

2.1.2.2 Set and Clear registers

Throughout this document there are Host Controller registers that are identified as *Set and Clear* registers. These registers have the property of having two addresses by which they may be referenced by the host. Unless otherwise stated in the description of the register, a host read of either address will return the current contents of the register. Host writes, however, have different effects when addressing the different addresses.

When the host writes to the *Set* address the value written is taken as a bit mask indicating which bits in the underlying register are to be set to one. A one bit in the value written indicates that the corresponding bit in the register is to be set to one, while a zero bit in the value written indicates that the corresponding bit in the register is not to be changed. Similarly, host writes to the *Clear* address specify a value that is a bit mask of bits to clear to zero in the underlying register, a one bit means to clear the corresponding bit while a zero bit means to leave the corresponding bit unchanged. It is intended that writing zero bits to these addresses has no effect on the corresponding bits in the underlying register, including transient effects that could affect the operation of the Host Controller.

There are several reasons to use this type of register:

- The host doesn't need to do both a read and a write to affect only a single bit.
- The host doesn't risk the Host Controller modifying a bit while the host does a read-modify-write operation, thus causing unintended effects.
- The host doesn't have to serialize its access to frequently used registers in order to ensure that conflict with another process doesn't cause unintended effects.

For set and clear registers that have an undefined value following a reset, it is recommended that software write all ones to the Clear address to ensure the register has a known value.

Table 2-2 — Set and Clear register field access tags

access tag (rscu)	name	meaning
r	read	field may be read
s	set	field may be set from the host bus
c	clear	field may be cleared from the host bus
u	update	field may be autonomously updated by Open HCI hardware

2.1.2.3 Register Reset Values

Register field descriptions may be tagged with one or more of the following reset values. This column indicates the value of the field immediately following a software reset or hardware reset. Except where otherwise noted, the results from a software reset and hardware reset are the same. Note that the reset column is for software and hardware resets only and does not include bus reset values (those are discussed as needed in the applicable text).

Table 2-3 — Register field reset values

reset value	meaning
x'by or x'hy	Indicates the value (in binary or hexadecimal) of the field upon completion of a reset. For description of Verilog notation see section 2.1.1.
undef	Following a reset, the value of this field is undefined and may contain (any combination of) zero(s) or one(s).
N/A	Not applicable. A reset does not have any effect on this field.

Unless otherwise specified, all fields will remain unchanged after a 1394 bus reset.

2.1.2.4 Reserved fields

All reserved fields (indicated by a hatched or grayed-out pattern) are read as zeros (but must be ignored) and must be written as zeros.

2.1.2.5 Reserved registers

Addresses within the OpenHCI Register Address space that are marked as reserved must return zeros when read and must ignore writes.

2.1.2.6 Register field notation

In descriptions which refer to specific register fields, the notation Rrrrr.*fffff* will be used where Rrrrr refers to the register name and *fffff* refers to the referenced field within that register.

2.2 Terms

The following terms and acronyms are used throughout this document.

AR DMA	**A**synchronous **R**eceive **DMA**.
AR DMA Request	Refers to the asynchronous receive DMA context that handles all incoming request packets not handled by the *physical request unit*.
AR DMA Response	Refers to the asynchronous receive DMA context that handles all incoming response packets.
asynchronous stream packet	A stream packet for which only a channel has been reserved at the isochronous resource manager. An asynchronous stream packet shall be transmitted during the asynchronous period and not during the isochronous period. For the same channel number, there is no restriction on multiple talkers nor upon a single talker sending multiple asynchronous stream packets. Fair arbitration rules govern the transmission of these packets. See also *isochronous stream packet* and *stream packet*.
AT DMA	**A**synchronous **T**ransmit **DMA**.
AT DMA Request Unit	Refers to the asynchronous transmit DMA subunit which moves transmit packets from buffers in memory to the request transmit FIFO.
AT DMA Response Unit	Refers to the asynchronous transmit DMA subunit which moves transmit packets from buffers in memory to the response transmit FIFO.
big endian	A term used to describe the arithmetic significance of data-byte addresses. With big-endian, the data byte with the largest address is the least significant.[a]
bridge	A hardware adapter that forwards transactions between buses.[a]
channel	Refers to an *isochronous channel* number.
CSR architecture	ISO/IEC 13213: 1994 [ANSI/IEEE Std 1212, 1994 Edition], *Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses*. The CSR architecture supports the concept of bus bridges, which can transparently forward transactions from one compliant bus to another.
Config ROM	A portion of a node's 1394 address space defined by clause 8 of ISO/IEC 13213:1994 [ANSI/IEEE Std 1212, 1994 Edition]. The region contains information describing the node and it's units. The region is read-only to other 1394 nodes. See also *GUID ROM* and *PCI Expansion ROM*.
DMA context	A distinct logical stream (not necessarily physical) through the Open HCI which can be described by a *DMA context program* and a minimum of two registers: ContextControl and CommandPtr.
DMA context program	A list of *DMA descriptors* which identify buffers used for data transfer.
DMA controller	Refers to the mechanism used in support of a specific DMA function. Each controller utilizes and maintains its own set of registers to perform its specified functionality.
DMA descriptor	A data structure used to describe buffers and buffer-list control.
DMA descriptor block	A group of DMA descriptors that are contiguous in host memory and can therefore be prefetched by the Host Controller. The last DMA descriptor in a block contains the address of the next block as well as a count of the number of descriptors contained in the next block. This count is referred to as the Z value.
EUI-64	Extended Unique Identifier. See *Global Unique ID* below.
Global Unique ID	See *GUID*.
GUID	**G**lobal **U**nique **ID** -A 64-bit node unique identifier, comprised of a 24-bit node company ID and a 40-bit chip ID

GUID ROM A hardware component that holds the EUI-64 of the node and is automatically loaded into the GlobalUniqueID registers of the controller when power is applied. Additional information may be stored in the GUID ROM and is available via the controller's GUID ROM register. See also *Config ROM* and *PCI Expansion ROM.*

hardware reset Refers to a host power reset.

HC **H**ost **C**ontroller. The device whose interface is defined by this specification.

HCI **H**ost **C**ontroller **I**nterface. The interface defined by this specification.

INPUT_* Abbreviated notation for INPUT_MORE and INPUT_LAST DMA descriptor commands.

INPUT_LAST* Abbreviated notation for INPUT_LAST and INPUT_LAST-Immediate descriptor commands.

INPUT_MORE* Abbreviated notation for INPUT_MORE and INPUT_MORE-Immediate descriptor commands.

IR DMA **I**sochronous **R**eceive **DMA.**

isochronous channel Within the packet header of an IEEE 1394 isochronous packet there is a 6 bit channel number. Receivers “listen” for packets transmitted with particular channel number(s).

isochronous stream packet A stream packet for which both channel and bandwidth have been reserved at the isochronous resource manager. Only one talker may transmit an isochronous stream packet during a single isochronous cycle. Isochronous stream packets shall not be transmitted outside of the isochronous period. See also *asynchronous stream packet* and *stream packet*.

IT DMA **I**sochronous **T**ransmit **DMA.**

ITF **I**sochronous **T**ransmit **F**IFO.

link layer (LINK) The layer, in a stack of three protocol layers defined for the Serial Bus, that provides the service to the transaction layer of one-way data transfer with confirmation of reception. The link layer also provides addressing, data checking, and data framing. The link layer also provides an isochronous data transfer service directly to the application.[c]

little endian A term used to describe the arithmetic significance of data-byte addresses. With little-endian, the data byte with the smallest address is the least significant.[a]

Node ID This is a unique 16-bit number, which distinguishes the node from other nodes in the system.[c]

OHCI **O**pen **H**ost **C**ontroller **I**nterface.

OUTPUT_* Abbreviated notation for OUTPUT_MORE and OUTPUT_LAST DMA descriptor commands.

OUTPUT_LAST* Abbreviated notation for OUTPUT_LAST and OUTPUT_LAST-Immediate descriptor commands.

OUTPUT_MORE* Abbreviated notation for OUTPUT_MORE and OUTPUT_MORE-Immediate descriptor commands.

PCI **P**eripheral **C**omponent **I**nterconnect. Specification that defines the PCI bus. This bus is intended to define the interconnect and bus transfer protocol between highly-integrated peripheral adapters that reside on a common local bus on the system board (or add-in expansion cards on the PCI bus).[b]

PCI Expansion ROM A hardware component on a PCI add-in card that contains the x86 BIOS and/or Open Firmware required by the device. See also *Config ROM* and *GUID ROM*.

PHY Abbreviation for the physical layer.[c]

physical layer The layer, in a stack of three protocol layers defined for the Serial Bus, that translates the logical symbols used by the link layer into electrical signals on the different Serial Bus media. The physical layer guarantees that only one node at a time is sending data and defines the mechanical interfaces for the Serial Bus.[c]

Physical Request Unit **P**hysical **Req**uest **U**nit. Refers to the asynchronous receive DMA subunit that handles physical requests.

Physical Response Unit Refers to the asynchronous transmit DMA subunit that handles physical responses.

posted write A write request received by the Host Controller for which the Host Controller sends an ack_complete before the data is actually written to system memory.

ROM See *Config ROM*, *GUID ROM* and *PCI Expansion ROM*.

RQTF **R**e**q**uest **T**ransmit **F**IFO. Refers to the FIFO used for asynchronous transmit requests.

RSTF **R**e**s**ponse **T**ransmit **F**IFO. Refers to the FIFO used for asynchronous transmit responses. Used for AT DMA responses and physical responses.

stream packet A 1394 primary packet with transaction code 4'hA. See also *asynchronous stream packet* and *isochronous stream packet*.

quadlet A 32-bit word.

RDMA **R**eceive **DMA**.

ROM **R**ead **O**nly **M**emory.

software reset Refers to a Host Controller reset that is initiated by host software. See section 5.7, "HCControl registers (set and clear)."

Z block See *DMA descriptor block*.

a. *Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses*, ISO/IEC 13213 [1994], The Institute of Electrical And Electronics Engineers, Inc., New York, NY.

b. Shanley, T. and Anderson, D. [February 1995], *PCI System Architecture*, Addison-Wesley, Reading, MA.

c. IEEE Standard for a High Performance Serial Bus, Std 1394-1995, The Institute of Electrical And Electronics Engineers, Inc., New York, NY.

3. Common DMA Controller Features

The 1394 Open HCI provides several types of DMA functionality:

a) General-purpose DMA handling asynchronous transmit and receive packets and isochronous transmit and receive packets.
b) An inbound bus bridge function that allows 1394 devices to directly access system memory called "physical DMA."
c) A separate write buffer for the received self-ID packets.
d) A mapping between a 1K byte block in system memory and the first 1K of 1394 Configuration ROM.

This section will describe the common controller features and attributes.

3.1 Context Registers

A context provides the basic information to the Host Controller to allow it to fetch and process descriptors for one of the several DMA controllers. All contexts (except for SelfID) minimally have a ContextControl Register and a CommandPtr Register. The format of the ContextControl Registers is DMA controller specific but all ContextControl registers minimally have the bits as shown in figure 3-1 and described in table 3-1. The CommandPtr Registers for all controllers are the same and follow the format shown in figure 3-2 and described in table 3-3.

3.1.1 ContextControl register

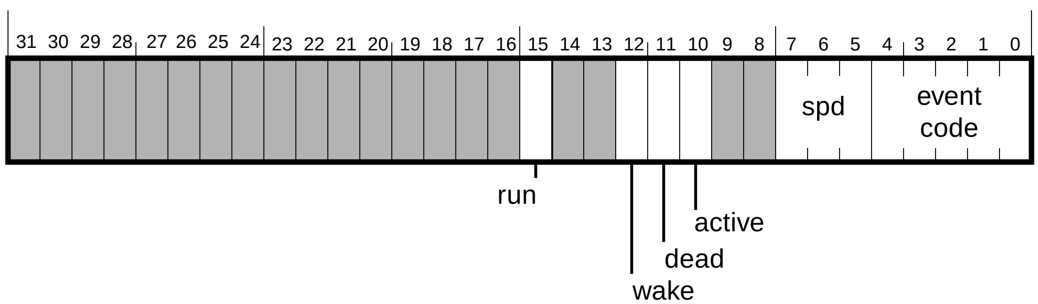


Figure 3-1 — ContextControl (set and clear) register format

Table 3-1 — ContextControl (set and clear) register description

Field	rscu	reset	Description
run	rscu	1'b0	The run bit is set by software to enable descriptor processing for a context and cleared by software to stop descriptor processing. The Host Controller will only change this bit on a hardware or software reset to set it to 0. See section 3.1.1.1 for details.
wake	rsu	undef	Software sets this bit to 1 to cause the Host Controller to continue or resume descriptor processing. The Host Controller will clear this bit on every descriptor fetch. See section 3.1.1.2 for details.
dead	ru	1'b0	The Host Controller sets this bit when it encounters a fatal error. The Host controller clears this bit when software clears the run bit. See section 3.1.1.4 for details.
active	ru	1'b0	The Host Controller sets this bit to 1 when it is processing descriptors. See section 3.1.1.3 for details.

Table 3-1 — ContextControl (set and clear) register description

Field	rscu	reset	Description
spd	ru	undef	This field indicates the speed at which the packet was received. 3’b000 = 100 Mbits/sec, 3’b001 = 200 Mbits/sec and 3’b010 = 400 Mbits/sec. All other values are reserved. Spd only contains meaningful information for receive contexts. Software should not attempt to interpret the contents of this field while the ContextControl.*active* or ContextControl.*wake* bits are set.
event code	ru	undef	This field holds the acknowledge sent by the Link core for this packet, or an internally generated error code (evt_*) if the packet was not transferred successfully. All possible event codes are shown in Table 3-2, “Packet event codes,” below.

The packet event codes shown in the table below are possible values for the five-bit ContextControl.*event* field. This field may contain either a 1394 defined ack code or an Open HCI generated event code. As described later in this document, bits 0-15 of the ContextControl register may be written into host memory to indicate packet and/or DMA descriptor status. However, all possible event codes which may appear in a particular context’s ContextControl register may not necessarily ever be written into host memory for a packet or DMA descriptor status, depending on circumstances and the functionality of the context.

1394 ack codes are denoted by the high (fifth) bit set to 1 followed by the 1394 four-bit ack code as received from 1394 (e.g., 1394 ack_pending = 4’h2, OpenHCI ack_pending = 5’h12). The list of ack codes provided in the table below is informative not normative; i.e., for asynchronous packets the event code may be set to any ack code specified in current and future 1394 standards.

OpenHCI generated event codes have an “evt_” prefix and are denoted by a code with the high (fifth) bit equal to 0. In some cases for isochronous I/O Open HCI may generate a 1394 style ack code for ContextControl.*event*.

Table 3-2 — Packet event codes

Code	Name	DMA	Meaning
5’h00	evt_no_status	AT,AR IT,IR	No event status.
5’h01	*reserved*		
5’h02	evt_long_packet	IR	The received data length was greater than the buffer’s data_length.
5’h03	evt_missing_ack	AT	A subaction gap was detected before an ack arrived *or* the received ack had a parity error.
5’h04	evt_underrun	AT, IT	Underrun on the corresponding FIFO. The packet was truncated.
5’h05	evt_overrun	IR	A receive FIFO overflowed during the reception of an isochronous packet.
5’h06	evt_descriptor_read	AT,AR IT,IR	An unrecoverable error occurred while the Host Controller was reading a descriptor block.
5’h07	evt_data_read	AT, IT	An error occurred while the Host Controller was attempting to read from host memory in the data stage of descriptor processing.
5’h08	evt_data_write	AR,IR IT	An error occurred while the Host Controller was attempting to write to host memory either in the data stage of descriptor processing (AR, IR), or when processing a single 16-bit host memory write (IT).
5’h09	evt_bus_reset	AR	Identifies a PHY packet in the receive buffer as being the synthesized bus reset packet. (See section 8.4.2.3).
5’h0A	evt_timeout	AT	Indicates that the asynchronous transmit response packet expired and was not transmitted.
5’h0B	evt_tcode_err	AT, IT	A bad tCode is associated with this packet. The packet was flushed.

Table 3-2 — Packet event codes

Code	Name	DMA	Meaning
5’h0C- 5’h0D	*reserved*		
5’h0E	evt_unknown	AT,AR IT,IR	An error condition has occurred that cannot be represented by any other event codes defined herein.
5’h0F	evt_flushed	AT	Sent by the link side of the output FIFO when asynchronous packets are being flushed due to a bus reset.
5’h10	*reserved*		Reserved for definition by future 1394 standards.
5’h11	ack_complete	AT,AR IT,IR	For asynchronous request and response packets, this event indicates the destination node has successfully accepted the packet. If the packet was a request subaction, the destination node has successfully completed the transaction and no response subaction shall follow. The event code for transmitted PHY, isochronous, asynchronous stream and broadcast packets, none of which yields a 1394 ack code, will be set by hardware to ack_complete unless an event occurs.
5’h12	ack_pending	AT,AR	The destination node has successfully accepted the packet. If the packet was a request subaction, a response subaction will follow at a later time. This code is not returned for a response subaction.
5’h13	*reserved*		Reserved for definition by future 1394 standards.
5’h14	ack_busy_X	AT	The packet could not be accepted after max ATRetries (see section 5.4) attempts, and the last ack received was ack_busy_X.
5’h15	ack_busy_A	AT	The packet could not be accepted after max ATRetries (see section 5.4) attempts, and the last ack received was ack_busy_A.
5’h16	ack_busy_B	AT	The packet could not be accepted after max AT Retries (see section 5.4) attempts, and the last ack received was ack_busy_B.
5’h17 - 5’h1A	*reserved*		Reserved for definition by future 1394 standards.
5’h1B	ack_tardy	AT	The destination node could not accept the packet because the link and higher layers are in a suspended state.
5’h1C	*reserved*		Reserved for definition by future 1394 standards.
5’h1D	ack_data_error	AT,IR	The destination node could not accept the block packet because the data field failed the CRC check, or because the length of the data block payload did not match the length contained in the data_length field. This code is not returned for any packet that does not have a data block payload.
5’h1E	ack_type_error	AT,AR	A field in the request packet header was set to an unsupported or incorrect value, or an invalid transaction was attempted (e.g., a write to a read-only address).
5’h1F	*reserved*		Reserved for definition by future 1394 standards.

3.1.1.1 ContextControl.run

The ContextControl.*run* bit is set by software when the Host Controller is to begin processing descriptors for the context. Before software sets ContextControl.*run*, ContextControl.*active* must not be set, and the CommandPtr Register for the context must contain a valid descriptor block address and a Z value that is appropriate for the descriptor block address.

Software may stop the Host Controller from further processing of a context by clearing ContextControl.*run*. When a ContextControl.*run* is cleared, the Host Controller will stop processing of the context in a manner that will not impact the operation of any other context or DMA controller. The Host Controller may require a significant amount of time to safely stop processing for a context but when the Host Controller does stop, it will clear ContextControl.*active*. If software clears a ContextControl.*run* for an isochronous context while the Host Controller is processing a packet for the context, the Host Controller will continue to receive or transmit the packet and update descriptor status. The Host Controller will, however, stop at the conclusion of that packet. If ContextControl.*run* is cleared for a non-isochronous context, the Host Controller may stop processing at any convenient point as long as the context and descriptors end up in a consistent state (e.g., status updated if a packet was sent and acknowledged).

Clearing ContextControl.*run* may have other side effects that are DMA controller dependent. These effects are described in the chapters that cover each of the DMA controllers.

When software clears ContextControl.*run* and the Host Controller has stopped, the Host Controller is not necessarily in a state that can be restarted simply by setting ContextControl.*run*. Software should always ensure that CommandPtr.*descriptorAddress* and CommandPtr.*Z* are set to valid values before setting ContextControl.*run*.

3.1.1.2 ContextControl.wake

When software adds to a list of descriptors for a context, the Host Controller may have already read the descriptor that was at the end of the list before it was updated. The value that the Host Controller read may contain a Z value of zero indicating the end of the descriptor list. The ContextControl.*wake* bit provides a simple semaphore to the hardware to indicate that the list may be changed since the last time that Host Controller read a descriptor. Therefore, if the Host Controller had fetched a descriptor and the indicated branch address had a Z value of zero, then the Host Controller should reread the pointer value.

For transmit contexts, and receive contexts in *buffer-fill* mode (a mode described later in which a context can receive multiple packets into one data buffer), if the Z value is still zero, then the end of the list has been reached and the Host Controller should clear ContextControl.*active*. For receive contexts in buffer-fill mode, if the Z value is still zero on the reread, then the packet cannot be accepted. For asynchronous contexts, the Host Controller will return the appropriate ack_busy* code. In addition, the Host Controller will “back out” the packet by not updating the buffer’s byte count (resCount), and will flush the packet from the FIFO. The Host Controller will not go inactive, as there is still buffer space available, and it is expected that software is attempting to provide more buffer space.

For both transmit and receive contexts, if the Z value is now non-zero, the Host Controller will continue processing.

In order to ensure that a wake condition is not missed, the Host Controller should clear ContextControl.*wake* before it reads or rereads a descriptor.

ContextControl.*wake* is ignored when ContextControl.*run* is zero.

3.1.1.3 ContextControl.active

ContextControl.*active* is set and cleared only by the Host Controller. It is set when the Host Controller receives an indication from software that a valid descriptor is available for processing. This indication will occur as a result of software setting the ContextControl.*run* or by software setting ContextControl.*wake* while ContextControl.*run* is set. There are four cases in which the Host Controller will clear ContextControl.*active*: when a branch is indicated by a descriptor but the Z value of the branch address is 0; when software clears ContextControl.*run* and the Host Controller has reached a safe stopping point; while ContextControl.*dead* is set; and after a hardware or software reset of the Host Controller. Additionally, for the asynchronous transmit contexts (request and response), the Host Controller will clear ContextControl.*active* when a bus reset occurs.

When ContextControl.*active* is cleared and ContextControl.*run* is already clear, the Host Controller will set the IntEvent bit for the context. This interrupt is the same interrupt that would have been generated by the context if a completed descriptor had indicated that an interrupt should be generated.

3.1.1.4 ContextControl.dead

ContextControl.*dead* is used to indicate a fatal error in processing a descriptor. When ContextControl.*dead* is set by the Host Controller, ContextControl.*active* is immediately cleared but ContextControl.*run* remains set. In addition, setting ContextControl.*dead* causes an unrecoverableError interrupt event (see Table 6-1) and blocks a normal context event interrupt from being set.

ContextControl.*dead* is immediately cleared when software clears ContextControl.*run* or by either a hardware or software reset of the Host Controller.

Software can determine the cause of a context going dead by checking the ContextControl.*event* code (table 3-2). The defined reasons for the Host Controller to set ContextControl.*dead* are described in section 3.1.2.1 and section 13., “Host Bus Errors.”

3.1.2 CommandPtr register

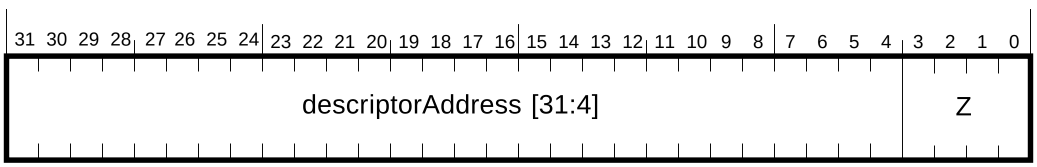


Figure 3-2 — CommandPtr register format

Table 3-3 — CommandPtr register description

Field	rwu	reset	Description
descriptorAddress	rwu	undef	Contains the upper 28 bits of the address of a 16-byte aligned descriptor block. See section 3.1.2 for details.
Z	rwu	undef	Indicates the number of contiguous 16-byte aligned blocks at the address pointed to by descriptorAddress. If Z is 0, it indicates that the descriptorAddress is not valid. Valid values for Z are context specific. Handling of invalid Z values is described in section 3.1.2.1.

Software initializes CommandPtr.*descriptorAddress* to contain the address of the first descriptor block that the Host Controller will access when software enables the context by setting ContextControl.*run*. Software also initializes CommandPtr.*Z* to indicate the number of descriptors in the first descriptor block. Software shall only write to this register when both ContextControl.*run* and ContextControl.*active* are zero. The Host Controller is not required to enforce this rule and its behavior when this rule is violated is undefined.

Since the Host Controller utilizes the CommandPtr register while processing a context, there is a set of guidelines by which software may safely and deterministically read CommandPtr. These guidelines are based on the ContextControl bits as follows (X='don't care'):

Table 3-4 — CommandPtr read values

ContextControl fields				CommandPtr.*descriptorAddress* Value
run	**dead**	**active**	**wake**	
0	0	0	X	A descriptor block address. Either last written or last executed
0	0	1	X	Contents unspecified.
1	0	0	0	Refers to the descriptor block that contains the Z=0 that caused the Host Controller to set active to 0.
1	0	0	1	Contents unspecified.
1	0	1	0	Contents unspecified.
1	0	1	1	Contents unspecified.
1	1	0	X	Points to the descriptor block in which a fatal error occurred.

If ContextControl.*run* is set and ContextControl.*dead* is not set, then the contents of CommandPtr are only specified if both ContextControl.*active* and ContextControl.*wake* are clear. In this instance, CommandPtr.*descriptorAddress* will contain the address of a descriptor within the last descriptor block that was executed. If ContextControl.*run* and ContextControl.*dead* are both set, then descriptorAddress points to a descriptor within the descriptor block in which an unrecoverable error occurred.

Except for the case where software initializes CommandPtr, the value of CommandPtr.*Z* is undefined and Z may contain a value that is implementation dependent.

The value of CommandPtr is undefined after a hardware or software reset of the Host Controller.

3.1.2.1 Bad Z Value

When software sets ContextControl.*run* to 1 and CommandPtr.*Z* contains an invalid value for the controller and context, or if a Z value is invalid for a fetched descriptor block in a running context, the Host Controller:

- will set ContextControl.*dead* to 1
- will set ContextControl.*event* to evt_unknown and
- will not process any descriptors in that context.

3.2 List Management

All contexts use an identical method for controlling the processing of descriptors associated with the context. This presents a uniform interface to controlling software and allows reuse of hardware on the Host Controller.

3.2.1 Software Behavior

3.2.1.1 Context Initialization

Software initializes the context by first checking to see that ContextControl.*run*, ContextControl.*active* and ContextControl.*dead* are all 0. Then, CommandPtr.*descriptorAddress* is written to point to a valid descriptor block and CommandPtr.*Z* is set to a value that is consistent with the descriptor block. Then ContextControl.*run* can be set.

3.2.1.2 Appending to Running List

Software may append to a list of descriptors at any time. Software may append either a single descriptor or a linked list of descriptors. When the to-be-appended list is properly formatted, software updates the branch address and Z value of the descriptor that was at the end of the list being processed by the Host Controller.

When software completes linking process it must set ContextControl.*wake* for the context. This ensures that the Host Controller will resume operation if it had previously reached the end of the list and gone inactive.

3.2.1.3 Stopping a Context

Software can stop a running context by clearing ContextControl.*run*. The context might not stop immediately. To ensure that the context has stopped, software must wait for ContextControl.*active* to be cleared by the Host Controller. This indicates that the Host Controller has completed all processing associated with the context.

3.2.2 Hardware Behavior

The Host Controller has several DMA controllers each of which has one or more contexts. Each DMA controller is expected to examine each of its contexts on a periodic basis and make operational decisions based on the context state as contained in ContextControl. The flow-chart for how a DMA controller uses the ContextControl state to govern descriptor processing is shown below. This process is executed once each time a context is ‘scheduled’. Scheduling of a context is dependent on the DMA controller. For example, an isochronous transmit context will be scheduled once per cycle while an asynchronous request transmit context will only be scheduled once per fairness interval.

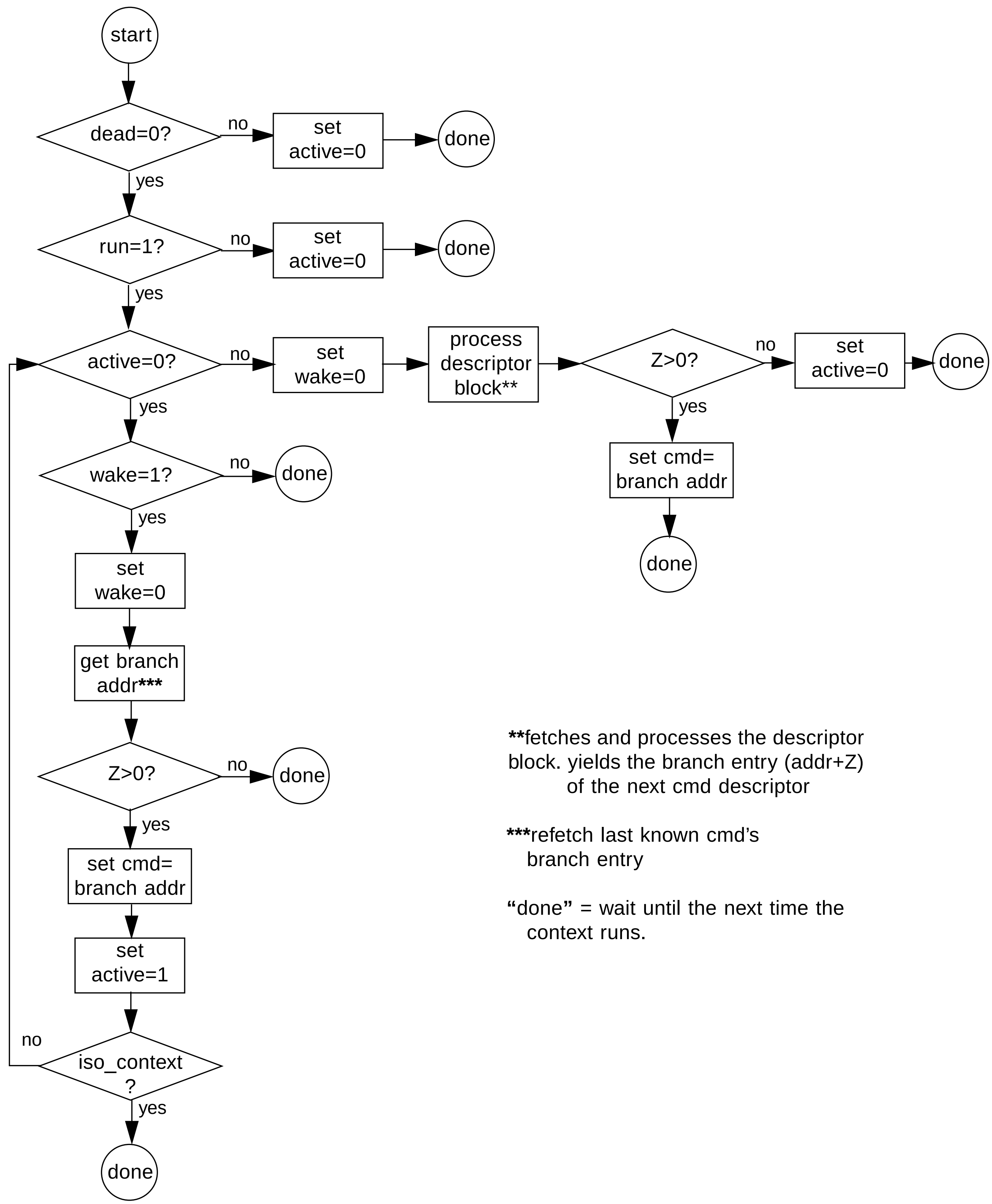


Figure 3-3 — Flow Chart for Processing a DMA Context

3.3 Asynchronous Receive

The Host Controller accepts 1394 transactions and groups them as follows:

1) physical requests - physical requests, including physical read, physical write and lock requests to some CSR registers (section 5.5), are handled directly by the Host Controller and are not made visible to system software. DMA contexts and controllers that are used in a Host Controller for the physical request unit are implementation specific. This specification places no limits on the physical response unit other than its effective address range and the requirement that the Host Controller may not block processing of other transaction types while dealing with physical requests. Chapter 12., “Physical Requests,” provides details on which requests can be processed as physical.
2) self-ID packets - PHY packets with the selfID format can be received at any time. However, only those packets that are received during the selfID phase of bus initialization which immediately follows a bus reset are considered to be selfID packets. Others are considered simply to be PHY packets which are handled like asynchronous requests. The Host Controller can be programmed to accept or ignore selfID packets. When selfID packets are accepted, they are stored in a special memory buffer which has a dedicated controller and context. Because of this special memory buffer, selfID packets can never get ‘stuck’ in a FIFO. See chapter 11., “Self ID Receive,” for more information.
3) asynchronous responses - when the host system initiates a request through the asynchronous transmit request context, the response will be handled by the asynchronous receive response context. The fact that host system software initiates the process and the fact that the Host Controller has a separate context for responses allows system software to budget for all responses which ensures that the Host Controller will always have a place in system memory to store a response when it arrives. In the unlikely event that the Host Controller does not have a place for the response it is allowed to drop the response when it arrives. This will cause a split-transaction timeout which is an error condition with which the software is already able to deal.
4) asynchronous requests - a request may arrive at the Host Controller at any time. Additionally, a request can be of any size up to the limits imposed by the max_rec field in the Bus_Info_Block. Due to the unpredictable nature of this transaction type, it is impractical for the system software to ensure that there is always sufficient buffer space defined in the asynchronous request receive buffers. If the FIFO which is receiving requests becomes full, all subsequent requests will be busied until there is room to receive them.

3.3.1 FIFO Implementation

The limitations and requirements for handling each of the transaction types suggest some ways of simplifying the hardware implementation so that a FIFO is not needed for each of the input transaction types. One simplification would be to place asynchronous requests into a first FIFO and then send all other transaction types (except for physical reads) through a second FIFO. This two FIFO scheme provides the necessary non-blocking behavior because the Host Controller will always be able to remove transactions from the second FIFO whether or not buffer space exists for the transaction. The selfID, isochronous and asynchronous response transactions will either have a buffer defined for the transaction or it is permissible to discard the transaction if no buffer exists to receive it. This leaves requests to be sent to the first FIFO. When that FIFO fills, additional requests will receive ack_busy until system software makes space available to the Host Controller by adding descriptors to the context.

There is an alternative implementation which is to use a single physical FIFO but ensure that it provides the behavior of the multiple FIFO’s. This is a bit more complex than the dual FIFO case but may result in a net savings in hardware. The issue with using a single physical FIFO for all incoming transactions is to make sure that no request is placed in the FIFO unless there is a place for it in system memory. There are several way of accomplishing this with one given as an example here.

On the link side of the input FIFO a counter is maintained. This counter is initialized to 0 when, for the AR DMA request context, ContextControl.*run* is not set. When the system side of the FIFO reads a request descriptor, the reqCount value from the descriptor is passed to the link side of the FIFO. The link side then adds this value to the current count value. When the count value on the link side is greater than zero, the link can accept request data and place it into the FIFO. After each request quadlet is placed in the FIFO, other than those for a physical write request, the link side decrements the counter. When the counter reaches 1, the link checks to see if the end of packet has been reached. If it has, the link uses the last entry for the footer value (cycleCount, speed and ackSent.) If the end of the packet has not been reached, the link places an error value in the last quadlet to indicate that the packet was not totally received and then the link returns an ack_busy to the requestor. The system side of the fifo can indicate that additional space has been made available by writing a new value to the link side. The link side will add these values to the current count value.

The system side of the FIFO will send count values to the link side on two occasions. The first is when a descriptor is initially fetched and the reqCount in the descriptor is sent to the link side. It is required that the Host Controller have a look ahead of at least one descriptor (current plus next). If the Host Controller does not look ahead, the link side will not be able to accept packets that cross descriptor boundaries.

The second instance when the system side of the input FIFO sends a count value to the link side is when the system side sees a packet that has an error. Packets that contain errors (e.g., CRC) are always 'backed out' of the buffer when the context is in buffer fill mode. The AR DMA request context can only be in buffer fill mode so all bad packets must be 'backed out'. When a packet is backed out, the space that was allocated for that packet is made available for other packets and the link side of the FIFO must be informed of the amount of data that has been backed out. A simple implementation of this is to maintain a counter on the system side of the FIFO that is reset at the beginning of each packet. As each quadlet is removed from the FIFO, the counter is incremented. At the end of the packet, the Host Controller checks the error code. If it indicates that there was an error, and the packet was a request, the count value is sent to the link side of the FIFO to indicate the amount of space that has been 'reclaimed'.

The reqCount field in a descriptor may indicate a size as large as 65,532 bytes (16,383 quadlets.) If quadlet counts are maintained this means that 14 bits are required to indicate the maximum number of quadlets (14’h3FFF). To allow for look ahead, the link side counter should be able to hold a value equal to two maximum sized buffers which is 32,766 (15’h7FFE) quadlets or 15 bits. Since the system software is required to allocate buffers that are sized to accept the maximum sized packet (as described in max_rec of the Bus_Info_Block) the Host Controller need only do one level of look ahead on the buffer descriptors to make sure that the maximum sized packet can be accepted.

3.3.1.1 Unrecoverable Error

If an unrecoverable error occurs when the Host Controller is writing to the AR DMA request buffer, a fail indication is sent to the link side of the FIFO. This indicates that the link side should set its count to zero which will busy further read requests and write requests that are destined for the AR DMA request buffer.

If the AR DMA request context has an unrecoverable error, the system side of the FIFO will continue to unload the FIFO even though the AR DMA request context is dead. All asynchronous requests that would have been sent to the AR DMA request queue shall be dropped and no responses for them shall be sent to the initiating node. Dropping requests destined for the AR DMA request queue is acceptable because i) AR DMA read requests are always split transactions (ack_pended), ii) write requests within the physical range have been ack_pended and iii) write requests above the physical range which have been posted (ack_completed) are by definition permitted to fail.

3.3.2 Ack Codes for Write Requests

For write requests that will be handled by the Physical Request controller, the Host Controller may send an ack_complete before the data is actually written to system memory. For a full description of which requests are candidates for Physical Requests, refer to Chapter 12.

The ack_code sent for write requests to offsets in the range of PhysicalUpperBound to 48’hFFFE_FFFF_FFFF when not busied is always ack_complete. The ack_code sent for requests to offsets in the range 48’hFFFF_0000_0000 to 48’hFFFF_FFFF_FFFF and for block requests with a non-zero extended tcode is always ack_pending.

3.3.3 Posted Writes

As described above, a write request that will be handled by the Physical Request controller or which is in the address range PhysicalUpperBound to 48’hFFFE_FFFF_FFFF to be handled by the Asynchronous Request Unit, may generate an ack_complete before the data is actually written to the designated system memory location. These writes are referred to as *posted writes.*

Write requests to the physical memory range of the host may be posted if the host controller supports the PostedWriteAddressLo/Hi error registers (see section 13.2.8.1) and software has enabled posted writes (see section 5.7). If posting is not enabled/supported, the Host Controller must not return a complete indication (ack_complete or resp_complete) until the data has been successfully written to the addressed location in physical memory.

If posting of physical writes is supported and enabled, then the Host Controller is allowed to return ack_complete to a physical write request with certain restrictions.

- A Host Controller implementation is allowed to support any number of posted writes. However, for error reporting purposes a posted write is considered pending until the write is actually completed to the offset address. For each pending posted write, there must be an error reporting register to hold the request’s source node ID and 48-bit offset address should that posted write fail. If the maximum allowed posted writes are pending, the Host Controller must return either ack_pending or ack_busy* for subsequent posted write request candidates and shall only return resp_complete when those writes have actually been performed.
- Read and write requests within the Asynchronous Request FIFO <u>shall not</u> pass <u>any</u> posted writes, whether posted in the Physical *or* Asynchronous Request FIFO’s.
- Within the Physical Request FIFO, read requests <u>may</u> coherently pass posted writes, but writes requests and posted writes <u>shall not</u> pass other writes posted in the Physical Request FIFO. Physical read and write requests <u>may</u> pass writes posted to the Asynchronous Request FIFO.

In conjunction with the ordering rules set forth above for Host Controller implementations, the following protocol restrictions must be adhered to so that proper ordering and therefore data integrity is maintained. The term *visible side-effect* is used to mean an indirect action caused by a request or response which results in the alteration of the contents or usage of host memory outside the address scope of the request or response.

- Write requests within the range PhysicalUpperBound to 48’hFFFE_FFFF_FFFF shall not have 1394 visible side-effects.
- Read or write requests within the range 48’h0 to PhysicalUpperBound-1, whether handled by the Physical Request controller or not, shall not have 1394 visible side-effects.
- Read requests to CSR addresses which are processed autonomously by the Host Controller (see section 5.5) shall not have 1394 visible side-effects

If an error occurs in writing the posted data packet, then the Host Controller sets an interrupt event to notify software and provides information about the failed write in an error reporting register. For more information about error handling of posted writes, refer to section 13.2.8.

3.3.4 Retries

For asynchronous receive, it is recommended that the Host Controller support dual-phase retry for packets that must be busied.

For asynchronous transmit, Host Controller implementations must support the single-phase retry protocol and may optionally support the dual-phase retry protocol. The implemented retry mechanism shall be managed by hardware and invisible to software. Refer to section 7.4 and table 7-12 for details.

3.4 DMA Summary

The following chapters provide details about Open HCI registers and interrupts, and about all the supported DMA types. The table below is a summary of DMA information for reference purposes. Each DMA type is fully described in the indicated chapter.

Table 3-5 — DMA Summary

DMA	Contexts	Per Context Registers	Per Context Interrupts	Receive mode	DMA commands	Z	tcodes (4'h*x*)
Asynchronous Transmit (section 7.0)	1 Request	ContextControl CommandPtr	reqTxComplete		OUTPUT_MORE OUTPUT_MORE-Immediate OUTPUT_LAST OUTPUT_LAST-Immediate	2-8	0, 1, 4, 5, 9, A,E
	1 Response	ContextControl CommandPtr	respTxComplete				2, 6, 7, B
Asynchronous Receive (section 8.0)	1 Request	ContextControl CommandPtr	ARRQ RQPkt	buffer-fill	INPUT_MORE	1	0, 1, 4, 5, 9, E*
	1 Response	ContextControl CommandPtr	ARRS RSPkt				2, 6, 7, B
Isochronous Transmit (section 9.0)	4-32	ContextControl CommandPtr	isochTx isoXmitIntEvent*n* isoXmitIntMask*n*		OUTPUT_MORE OUTPUT_MORE-Immediate OUTPUT_LAST OUTPUT_LAST-Immediate STORE_VALUE	1-8	A
Isochronous Receive (section 10.0)	4-32	ContextControl CommandPtr ContextMatch	isochRx isoRecvIntEvent*n* isoRecvIntMask*n*	packet-per-buffer	INPUT_MORE INPUT_LAST	1-8	A
				buffer-fill	INPUT_MORE	1	
Self-ID (section 11.0)	1	SelfIDBuffer SelfIDCount	SelfIDComplete	buffer-fill		N/A	

E* - this includes packets considered to be PHY packets and the synthesized phy (bus_reset) packet.

For transmit, software may use the tcodes as specified in the table above. The Host Controller hardware shall allow any IEEE 1394-1995 tcode to be transmitted by any asynchronous transmit context.

For receive, the Host Controller shall only receive packets which have tcodes that are defined by an approved IEEE 1394 standard. Packets with undefined tcodes shall be dropped.

4. Register addressing

The 1394 Open HCI's registers occupy a 2048 byte address space. This 2048 byte space is allocated to control registers, common DMA controller registers and individual DMA context registers as indicated below. Registers shall be accessed as 32-bit entities; 8-bit or 16-bit access to Host Controller registers is not supported. Writes to reserved addresses of the 1394 Open HCI address space may have unexpected results and are disallowed. Reads of reserved addresses are undefined. Host processors may only access Host Controller registers with quadlet reads or writes on quadlet boundaries.

Host Controller registers which are written through physical access to the Host Controller will yield unspecified results.

When HCControl.*LPS* is 0, the only accessible registers are Version, VendorID, HCControl, GUID_ROM, GUIDHi and GUIDLo. Access to all other registers is undefined until HCControl.*LPS* is set to 1.

All addresses within this 2K address space are reserved for OpenHCI and not for vendor defined registers.

Annex A. describes how this memory space is accessed from PCI.

Table 4-1 — 1394 Open HCI register space map

Offset (binary)	Space
00R_RRRR_RR00 (11'h000 to 11'h17C)	control register space **R_RRRR_RR** selects register
001_1ccR_RR00 (11'h180 to 11'h1FC)	Asynchronous DMA context register space **cc** = 2'h0-2'h3 selects DMA context **R_RR** selects DMA context register
01t_tttt_RR00 (11'h200 to 11'h3FC)	Isochronous Transmit DMA context register space **t_tttt** = 5'h00-5'h1F selects IT DMA context **RR** selects DMA context register
1vv_vvvR_RR00 (11'h400 to 11'7FC)	Isochronous Receive DMA context register space **vv_vvv** = 5'h00-5'h1F selects IR DMA context **R_RR** selects DMA context register

4.1 DMA Context Number Assignments

The 1394 Open HCI contains up to 68 DMA contexts, 4 for asynchronous and from 8 up to 64 for isochronous. The controller number assignments for asynchronous DMA are illustrated below. Note that these numbers correspond to the "cc" DMA controller select values in the table above.

Table 4-2 — Asynchronous DMA Context number assignments

DMA Context Number	Context Name
2'h0	Asynchronous Transmit Request
2'h1	Asynchronous Transmit Response
2'h2	Asynchronous Request Receive
2'h3	Asynchronous Response Receive

For the isochronous transmit contexts, **t_tttt** represents IT contexts numbered 0-31.
For the isochronous receive contexts, **vv_vvv** represents IR contexts numbered 0-31.

4.2 Register Map

Table 4-3 — Register addresses (Sheet 1 of 4)

Offset	DMA Context	Read value	Write value	See clause
11’h000		Version	-	5.2
11’h004		GUID_ROM	GUID_ROM	5.3
11’h008		ATRetries	ATRetries	5.4
11’h00C		CSRReadData	CSRWriteData	5.5.1
11’h010		CSRCompareData	CSRCompareData	5.5.1
11’h014		CSRControl	CSRControl	5.5.1
11’h018		ConfigROMhdr	ConfigROMhdr	5.5.2
11’h01C		BusID	-	5.5.3
11’h020		BusOptions	BusOptions	5.5.4
11’h024		GUIDHi	GUIDHi	5.5.5
11’h028		GUIDLo	GUIDLo	5.5.5
11’h02C		*Reserved*	*Reserved*	
11’h030		*Reserved*	*Reserved*	
11’h034		ConfigROMmap	ConfigROMmap	5.5.6
11’h038		PostedWriteAddressLo	PostedWriteAddressLo	13.2.8.1
11’h03C		PostedWriteAddressHi	PostedWriteAddressHi	
11’h040		Vendor ID	-	5.6
11’h044 - 11’h04C		*Reserved*	*Reserved*	
11’h050		HCControl	HCControlSet	5.7
11’h054			HCControlClear	5.7
11’h058 - 11’h05C		*Reserved*	*Reserved*	
11’h060	Self ID	*Reserved*	*Reserved*	
11’h064		SelfIDBuffer	SelfIDBuffer	11.1
11’h068		SelfIDCount		11.2
11’h06C		*Reserved*	*Reserved*	
11’h070		IRMultiChanMaskHi	IRMultiChanMaskHiSet	10.4.1.1
11’h074			IRMultiChanMaskHiClear	
11’h078		IRMultiChanMaskLo	IRMultiChanMaskLoSet	
11’h07C			IRMultiChanMaskLoClear	

Table 4-3 — Register addresses (Sheet 2 of 4)

Offset	DMA Context	Read value	Write value	See clause
11'h080		IntEvent	IntEventSet	6.1
11'h084		(IntEvent & IntMask)	IntEventClear	
11'h088		IntMask	IntMaskSet	6.2
11'h08C			IntMaskClear	
11'h090		IsoXmitIntEvent	IsoXmitIntEventSet	6.3.1
11'h094		(IsoXmitIntEvent & IsoXmitIntMask)	IsoXmitIntEventClear	
11'h098		IsoXmitIntMask	IsoXmitIntMaskSet	6.3.2
11'h09C			IsoXmitIntMaskClear	
11'h0A0		IsoRecvIntEvent	IsoRecvIntEventSet	6.4.1
11'h0A4		(IsoRecvIntEvent & IsoRecvIntMask)	IsoRecvIntEventClear	
11'h0A8		IsoRecvIntMask	IsoRecvIntMaskSet	6.4.2
11'h0AC			IsoRecvIntMaskClear	
11'h0B0-11'h0D8		*Reserved*	*Reserved*	
11'h0DC		Fairness Control	Fairness Control	5.8
11'h0E0		LinkControl	LinkControlSet	5.9
11'h0E4			LinkControlClear	
11'h0E8		Node ID	Node ID	5.10
11'h0EC		Phy Control	Phy Control	5.11
11'h0F0		Isochronous Cycle Timer	Isochronous Cycle Timer	5.12
11'h0F4-11'h0FC		*Reserved*	*Reserved*	
11'h100		AsynchronousRequestFilterHi	AsynchronousRequestFilterHiSet	5.13.1
11'h104			AsynchronousRequestFilterHiClear	
11'h108		AsynchronousRequestFilterLo	AsynchronousRequestFilterLoSet	
11'h10C			AsynchronousRequestFilterLoClear	
11'h110		PhysicalRequestFilterHi	PhysicalRequestFilterHiSet	5.13.2
11'h114			PhysicalRequestFilterHiClear	
11'h118		PhysicalRequestFilterLo	PhysicalRequestFilterLoSet	
11'h11C			PhysicalRequestFilterLoClear	
11'h120		PhysicalUpperBound	PhysicalUpperBound	5.14
11'h124-11'h17C		*Reserved*	*Reserved*	
11'h180	Async request transmit	ContextControl	ContextControlSet	3.1, 7.2.2
11'h184			ContextControlClear	
11'h188		*Reserved*	*Reserved*	
11'h18C		CommandPtr	CommandPtr	3.1.2, 7.2.1

Table 4-3 — Register addresses (Sheet 3 of 4)

Offset	DMA Context	Read value	Write value	See clause
11’h190-11’h19C		*Reserved*	*Reserved*	
11’h1A0	Async response transmit	ContextControl	ContextControlSet	3.1, 7.2.2
11’h1A4			ContextControlClear	
11’h1A8		*Reserved*	*Reserved*	
11’h1AC		CommandPtr	CommandPtr	3.1.2, 7.2.1
11’h1B0-11’h1BF		*Reserved*	*Reserved*	
11’h1C0	Async request receive	ContextControl	ContextControlSet	3.1, 8.3.2
11’h1C4			ContextControlClear	
11’h1C8		*Reserved*	*Reserved*	
11’h1CC		CommandPtr	CommandPtr	3.1.2, 8.3.1
11’h1D0-11’h1DF		*Reserved*	*Reserved*	
11’h1E0	Async response receive	ContextControl	ContextControlSet	3.1, 8.3.2
11’h1E4			ContextControlClear	
11’h1E8		*Reserved*	*Reserved*	
11’h1EC		CommandPtr	CommandPtr	3.1.2, 8.3.1
11’h1F0-11’h1FF		*Reserved*	*Reserved*	
11’h200 + 16*n	Isoch transmit n, where “n” = 0 for context 0, 1 for context 1, etc...	ContextControl	ContextControlSet	3.1, 9.2.2
11’h204+ 16*n			ContextControlClear	
11’h208+ 16*n		*Reserved*	*Reserved*	
11’h20C + 16*n		CommandPtr	CommandPtr	3.1.2, 9.2.1

Table 4-3 — Register addresses (Sheet 4 of 4)

Offset	DMA Context	Read value	Write value	See clause
11’h400 + 32*n	Isoch Receive n, where “n” = 0 for context 0, 1 for context 1, etc.	ContextControl	ContextControlSet	3.1, 10.3.2
11’h404 + 32*n			ContextControlClear	
11’h408 + 32*n		*Reserved*	*Reserved*	
11’h40C + 32*n		CommandPtr	CommandPtr	3.1.2, 10.3.1
11’h410+ 32*n		ContextMatch	ContextMatch	10.3.3
11’h414+ 32*n		*Reserved*	*Reserved*	
11’h418+ 32*n		*Reserved*	*Reserved*	
11’h41C+ 32*n		*Reserved*	*Reserved*	

5. 1394 Open HCI Registers

5.1 Register Conventions

Unless otherwise specified, all register fields will initialize as zeros. For software, reads of reserved locations (indicated by a hatched or grayed-out pattern) yield undefined results.

Similarly, unless otherwise specified, all fields will remain unchanged after a 1394 bus reset.

Refer to Section 2.1.2 for an explanation of register notation.

5.2 Version Register

This register contains a 32 bit value which indicates the version and capabilities of the interface. The register is expected to be used to indicate the level of functionality present in the 1394 Open HCI. This register is read only.

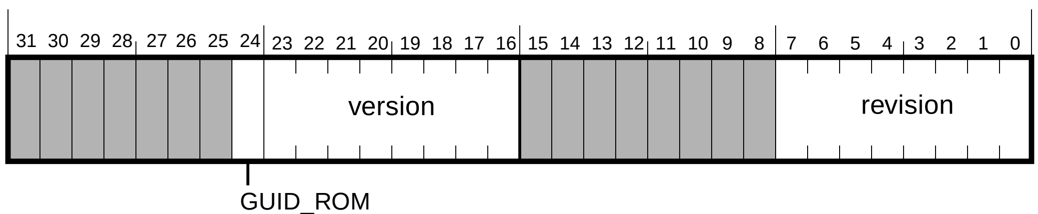


Figure 5-1 — Version register

Table 5-1 — Version register fields

field name	rwu	reset	description
GUID_ROM	r	N/A	The third and fourth quadlets of the bus_info_block will be automatically loaded on hardware reset.
version	r	N/A	Major version of the Open HCI. This field contains the bcd encoded value representing the major version of the highest numbered 1394 OpenHCI specification with which this controller is compliant. For example, a Host Controller implemented to this specification (Release 1.00) will have a version value of 8’h01 and a Host Controller implemented to version 2.25 of this specification will have a value of 8’h02.
revision	r	N/A	Minor version of the Open HCI. This field contains the BCD encoded value representing the minor version of the highest numbered 1394 OpenHCI specification with which this controller is compliant. For example, a Host Controller implemented to this specification (Release 1.00) will have a revision value of 8’h00 and a Host Controller implemented to version 2.25 of this specification will have a value of 8’h25.

5.3 GUID ROM register (optional)

The GUID ROM register is used to access the GUID ROM, and is only present if the Version.*GUID_ROM* bit is set.

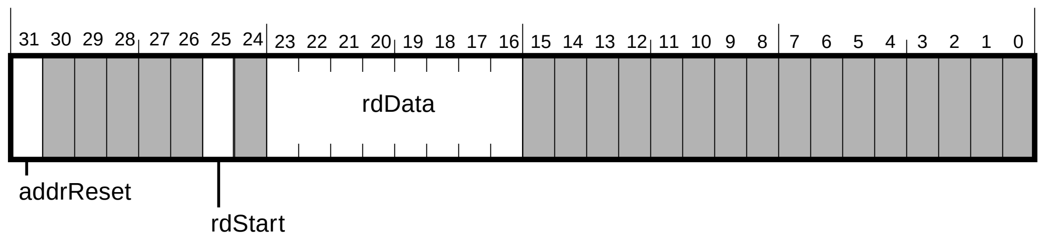


Figure 5-2 — GUID ROM register

Table 5-2 — GUID ROM register fields

field name	rwu	reset	description
addrReset	rsu	1’b0	Software sets this bit to one to reset the GUID ROM address to zero. When the Host Controller completes the reset, it clears addrReset to zero. Upon resetting the GUID ROM address, the host controller does *not* automatically fill rdData with the data from byte address 0.
rdStart	rsu	1’b0	A read of the currently addressed GUID ROM byte is started on the transition of this bit from a zero to a one. When the Host Controller completes the read, it clears rdStart to zero and advances the GUID ROM byte address by one byte.
rdData	ru	undef	The data read from the GUID ROM.

To initialize the GUID ROM read address, software sets GUIDROM.*addrReset* to one. Once software detects that GUIDROM.*addrReset* is zero, indicating that the reset has completed, then software may set GUIDROM.*rdStart* to read a byte. Upon the completion of each read, the Host Controller places the read byte into GUIDROM.*rdData*, advances the GUID ROM address by one byte to set up for the next read, and clears GUIDROM.*rdStart* to 0 to indicate to software that the requested byte has been read.

5.4 ATRetries Register

The AT retries register holds the number of times the 1394 Open HCI will attempt to do a retry for asynchronous DMA request transmit and for asynchronous physical and DMA response transmit. A packet may only be retried when a “busy” acknowledge or ack_data_error is received from the target node, including ack_data_error’s resulting from FIFO underflows. A packet shall not be retried under any other circumstance, including receipt of evt_missing_ack.

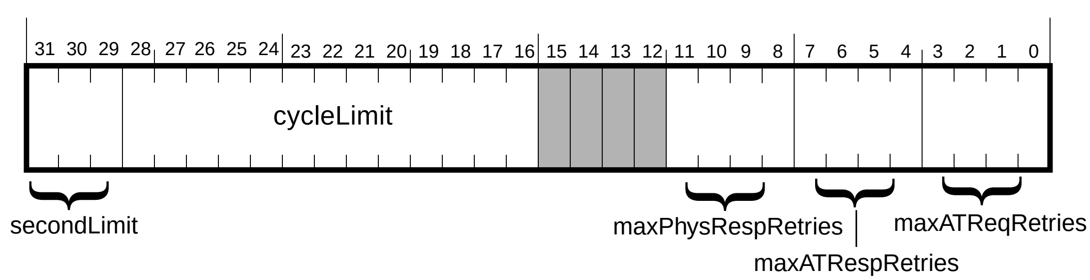


Figure 5-3 — ATRetries register

Table 5-3 — ATRetries register fields

field name	rwu	reset	description
secondLimit	ru *or* rwu	3’h0	Together the secondLimit and cycleLimit fields define a time limit for retry attempts when the outbound dual-phase retry protocol is in use. The secondLimit field represents a count in seconds modulo 8, and cycleLimit represents a count in cycles modulo 8000. If the retry time expires for a physical response, the packet is discarded by the Host Controller. Software is *not* notified. If outbound dual-phase retry is not implemented, both fields shall be read-only and shall read as 16’h0. If outbound dual-phase retry is implemented, both fields shall be read/write, and a value of 0 written to both fields shall disable dual phase retry.
cycleLimit		13’h0	
maxPhysRespRetries	rw	undef	The maxPhysRespRetries field tells the Physical Response Unit how many times to attempt to retry the transmit operation for the response packet. Note that this value is used only for responses to *physical* requests. If the retry count expires for a physical response, the packet is discarded by the Host Controller. Software is *not* notified.
maxATRespRetries	rw	undef	The maxATRespRetries field tells the Asynchronous Transmit Response Unit how many times to attempt to retry the transmit operation for a software transmitted (non-physical) asynchronous response packet.
maxATReqRetries	rw	undef	The maxATRetries field tells the Asynchronous Transmit Request Unit how many times to attempt to retry the transmit operation for an asynchronous request packet.

The Host Controller is required to pace the retries of both requests and responses using fairness intervals as described in P1394A and 1394-1995.

The interrelationship between retries and packet transmission is as follows:

- Retried requests shall not block responses.
- Retried requests may block other requests.
- Retried responses should not block requests.
- Retried AT DMA responses shall not block physical responses.
- Retried AT DMA and physical responses may block AT DMA responses.
- Retried physical responses may block other physical responses.

5.5 Autonomous CSR Resources

The 1394 Open HCI implements a number of autonomous CSR resources. In particular the 1394 compare-swap bus management registers are implemented in hardware, as is the config ROM header, the bus_info_block and access to the first 1K bytes of the configuration ROM. The DMA units handle external 1394 bus requests to these resources automatically, and the following registers manage this function for the local host

5.5.1 Bus Management CSR Registers

1394 requires certain 1394 bus management resource registers be accessible only via "quadlet read" and "quadlet lock" (compare-and-swap) transactions, otherwise ack_type_error shall be sent. These special bus management resource registers are implemented internal to the 1394 Open Host Controller to allow atomic compare-and-swap access from either the host system or from the 1394 bus.

Table 5-4 — Serial Bus Registers

CSR address	csrSel	description	1394-1995 Section #	reset (hardware reset or bus reset)
48'hFFFF_F000_021C	2'h0	BUS_MANAGER_ID	8.3.2.3.6	6'h3F
48'hFFFF_F000_0220	2'h1	BANDWIDTH_AVAILABLE	8.3.2.3.7	13'h1333 ('d4915)
48'hFFFF_F000_0224	2'h2	CHANNELS_AVAILABLE_HI	8.3.2.3.8	32'hFFFF_FFFF
48'hFFFF_F000_0228	2'h3	CHANNELS_AVAILABLE_LO	8.3.2.3.8	32'hFFFF_FFFF

When these bus management resource registers are accessed from the 1394 bus, the atomic compare-and-swap transaction is autonomous, without software intervention. If ack_complete is not received to end the transaction for the generated lock response, IntEvent.*lockRespErr* (table 6-1) shall be triggered.

To access these bus management resource registers from the host, the following registers are used.

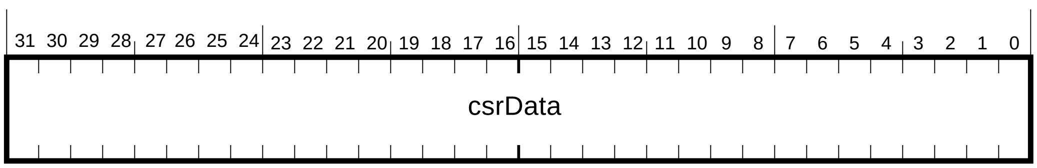


Figure 5-4 — CSR data register

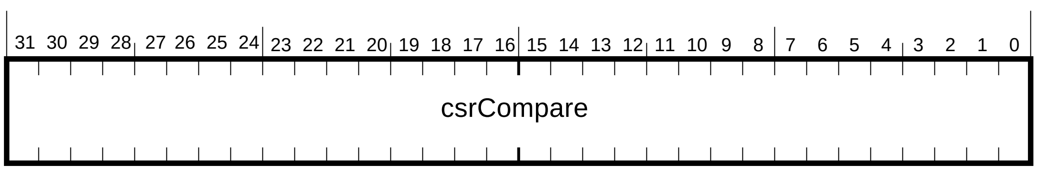


Figure 5-5 — CSR compare register

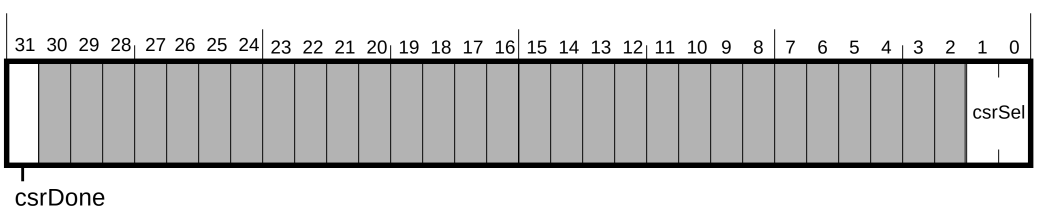


Figure 5-6 — CSR control register

Table 5-5 — CSR registers' fields

field name	rwu	reset	description
csrData	rwu	undef	At start of operation, the data to be stored if the compare is successful.
csrCompare	rw	undef	The data to be compared with the existing value of the CSR resource.
csrDone	ru	1'b1	This bit is set when a compare-swap operation is completed. It is reset whenever this register is written.
csrSel	rw	undef	This field selects the CSR resource: 2'h0 - BUS_MANAGER_ID 2'h1 - BANDWIDTH_AVAILABLE 2'h2 - CHANNELS_AVAILABLE_HI 2'h3 - CHANNELS_AVAILABLE_LO

To access these bus management resource registers from the host bus, first load the CSRData register with the new data value to be loaded into the appropriate resource. Then load the CSRCompare register with the expected value. Finally, write the CSRControl register with the selector value of the resource. A write to the CSRControl register initiates a compare-and-swap operation on the selected resource. When the compare-and-swap operation is complete, the CSRControl register csrDone bit will be set, and the CSRData register will contain the value of the selected resource prior to the host initiated compare-and-swap operation.

Note that an arbitrary update of these resources cannot be done. Only compare-and-swap operations can be used to modify the contents of these internal resource registers.

5.5.2 Config ROM header

The config ROM header register is a 32-bit number that externally maps to the 1st quadlet of the 1394 configuration ROM (offset 48'hFFFF_F000_0400). This register is written locally at the following register (the field names match the IEEE 1394 names):

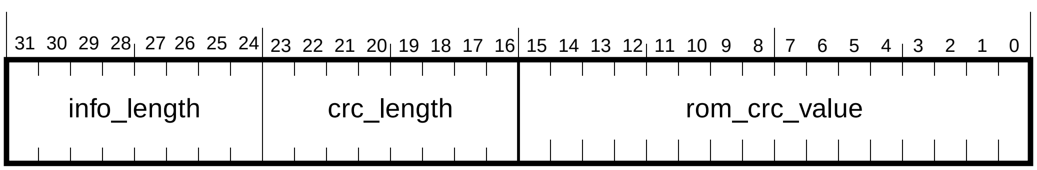


Figure 5-7 — Config ROM header register

Table 5-6 — Config ROM header register fields

field name	rwu	hard reset	soft reset	description
info_length	rwu	8'h0	N/A	IEEE 1394 bus management field. Must be valid at any time the HCControl.*linkEnable* bit is set.
crc_length	rwu	8'h0	N/A	IEEE 1394 bus management field. Must be valid at any time the HCControl.*linkEnable* bit is set.
rom_crc_value	rwu	16'h0	N/A	IEEE 1394 bus management field. Must be valid at any time the HCControl.*linkEnable* bit is set.

For a clarification of the meaning of Config ROM versus GUID ROM versus PCI Expansion ROM, see section 2.2.

5.5.3 Bus identification register

The bus identification register is a 32-bit number that externally maps to the first quadlet of the Bus_Info_Block. This register is read locally at the following register:

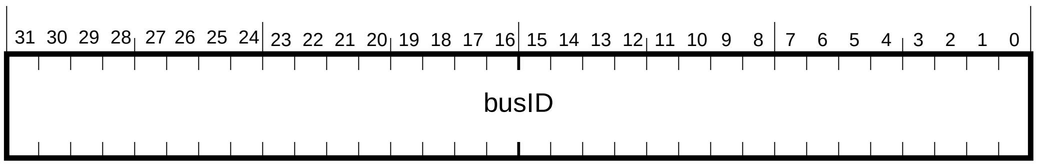


Figure 5-8 — Bus ID register

Table 5-7 — Bus ID register fields

field name	rwu	reset	description
busID	r	N/A	Contains the constant 32'h31333934, which is the ASCII value for "1394".

5.5.4 Bus options register

The bus options register is a 32-bit number that externally maps to the 2nd quadlet of the Bus_Info_Block. This register is written locally at the following register (the field names match the IEEE 1394 names):

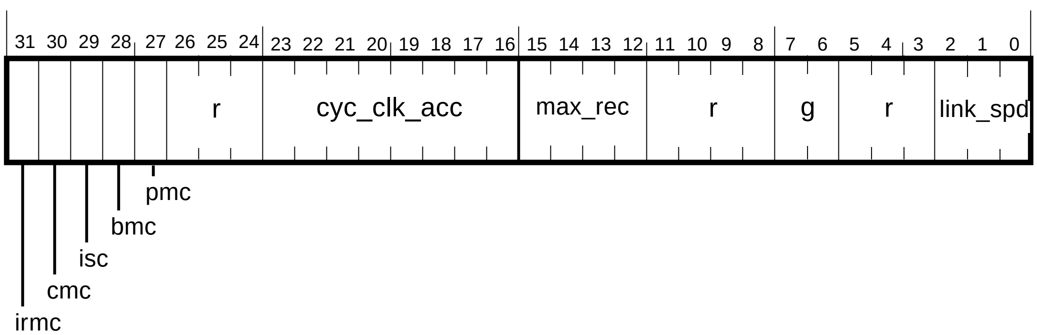


Figure 5-9 — Bus options register

Table 5-8 — Bus options register fields

field name	rwu	reset	description
irmc, cmc, isc, bmc, pmc, cyc_clk_acc	rw	undef	IEEE 1394 bus management fields. Must be valid at any time the HCControl.*linkEnable* bit is set.
max_rec	rw	**	IEEE 1394 bus management field. Hardware shall initialize max_rec to the maximum value supported by the implementation which shall be 512 or greater. Software may change max_rec, however this field must be valid at any time the HCControl.*linkEnable* bit is set to 1. Note that received block write request packets with a length greater than max_rec shall generate an ack_type_error if the request is not handled by the physical response unit, and may generate an ack_type_error otherwise (see table 1-2). ** Reset values: For a hardware reset, max_rec is set to the maximum value supported by the implementation, 512 or greater. For a soft reset, *max_rec* is not changed.
g	rw	undef	Generation counter. This field shall be incremented if any portion of configuration ROM has changed since the prior bus reset.

Table 5-8 — Bus options register fields

field name	rwu	reset	description
link_spd	rwu *or* ru	**	Link speed. **On a hardware reset, link_spd is set by the Host Controller to the maximum speed the link can send and receive. The Host Controller shall support the maximum size asynchronous and isochronous packets for the reported speed. If implemented as read/write, software is permitted to change link_spd to a lower value, which shall cause the link to reject packets arriving at higher speeds. Link_spd may also be implemented as read-only. **On a software reset, the value of link_spd is undefined.
bits 3-5, 8-11 and 24-26	rw	undef	Currently reserved in 1394-1995.

5.5.5 Global Unique ID

The global unique ID (GUID) is a 64-bit number that externally maps to the third and fourth quadlets of the Bus_Info_Block. These registers are written locally at the following registers (the field names match the IEEE 1394 names):

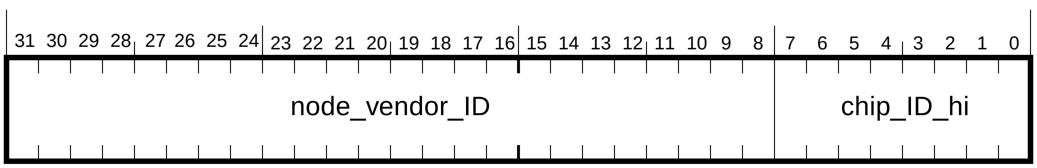


Figure 5-10 — GlobalUniqueIDHi register

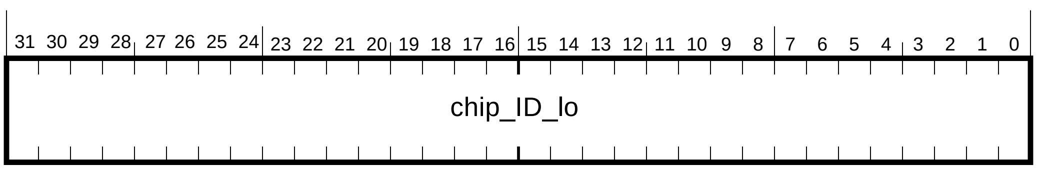


Figure 5-11 — GlobalUniqueIDLo register

Table 5-9 — GlobalUniqueID register fields

field name	rwu	reset	description
node_vendor_ID, chip_ID_hi, chip_ID_lo	rw	**see comments	IEEE 1394 bus management fields. Must be set by firmware or hardware before the HCControl.*linkEnable* bit is set.

**The Global Unique ID (GUID) Registers are reset to 0 after a host power (hardware) reset. A value of 0 is an illegal value. These registers are not affected by a software reset. These GUID registers shall be written only once after host power reset, by either

1) an autonomous load operation from a local, **un-modifiable** resource (i.e., local GUID ROM or local parallel ROM) performed by the 1394 OHCI hardware, or
2) a single host write to each register performed **only by firmware** that is always executed on a hardware reset which affects the Host Controller. This firmware, as well as the GUID value that is loaded, **may not be modifiable by any user action**.

After one of these load mechanisms has executed, the GUID registers are **read-only**.

5.5.6 Configuration ROM mapping register

The configuration ROM mapping register contains the start address within system bus space that will map to the start address of the 1394 configuration ROM for this node. Only quadlet reads to the first 1K bytes of the configuration ROM will map to system bus space, all other transactions to this space will be rejected with a 1394 “ack_type_error”. Since the low order 10 bits of this address are reserved and assumed to be zero, the system address for the config ROM must start on a 1K byte boundary. Note that the first five quadlets of the 1394 config ROM space are mapped to the config ROM header and the bus_info_block, and so are handled directly by the 1394 Open Host Controller as described in sections 5.5.2, 5.5.3, 5.5.4 and 5.5.5. This means that the first five quadlets addressed by the config ROM mapping register are not used.

Software should ensure this address is valid before setting HCControl.*linkEnable* to one.

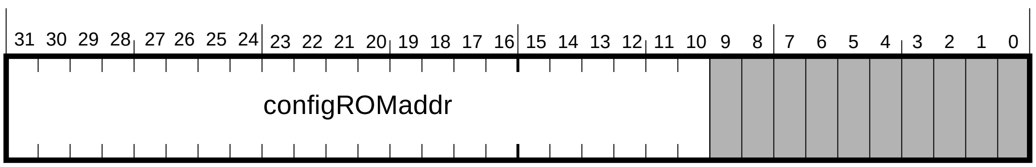


Figure 5-12 — Configuration ROM mapping register

Table 5-10 — Configuration ROM mapping register fields

field name	rwu	reset	description
configROMaddr	rw	undef	If a quadlet read request to 1394 offset 48’hFFFF_F000_0400 through offset 48’FFFF_F000_07FF is received, then the low order 10 bits of the offset are added to this register to determine the host memory address of the returned quadlet.

5.6 Vendor ID register

The vendor ID register holds the company ID of an organization that specified any vendor-unique registers.

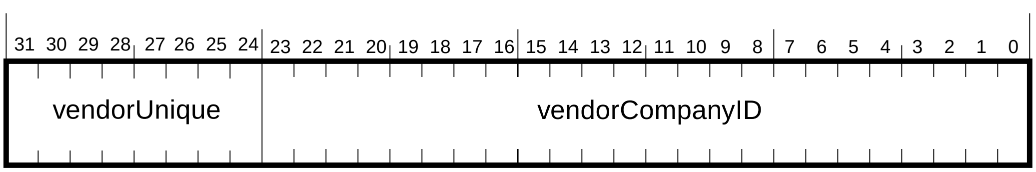


Figure 5-13 — VendorID register

Table 5-11 — VendorID register fields

field name	rwu	reset	description
vendorCompanyID	r	N/A	The company ID of the organization that specified the particular set of vendor unique registers and behaviors of this particular implementation of the 1394 Open HCI. If no additional features are implemented, this field shall be 24’h0.
vendorUnique	r	N/A	Vendor defined.

To obtain a company ID (also known as an Organizationally Unique Identifier, OUI), contact:

Registration Authority Committee
The Institute of Electrical and Electronic Engineers, Inc.
445 Hoes Lane
Piscataway, NJ 08855-1331
USA
(908) 562-3812

Your company need not obtain a company ID if it has been previously assigned an IEEE *48-bit Globally Assigned Address Block* or an IEEE-assigned *Organizationally Unique Identifier (OUI)* for use in network applications. However, be aware that the (left through right) order of the bits within the company ID value is not the same as the (first through last) network-transmission order of the bits within these other identifiers. Consult the IEEE Registration Authority for clarifying documentation.

5.7 HCControl registers (set and clear)

This register provides flags for controlling the Host Controller. There are two addresses for this register: HCControlSet and HCControlClear. On read, both addresses return the contents of the control register. For writes, the two addresses have different behavior: a one bit written to HCControlSet causes the corresponding bit in the HCControl register to be set, while a zero bit leaves the corresponding bit in the HCControl register unaffected. On the other hand, a one bit written to HCControlClear causes the corresponding bit in the HCControl register to be cleared, while a zero bit leaves the corresponding bit in the HCControl register unaffected.

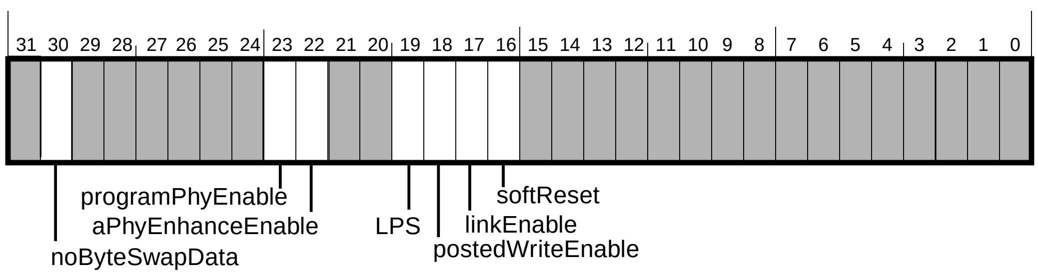


Figure 5-14 — HCControl register

Table 5-12 — HCControl register fields

field name	rscu	reset	description
noByteSwapData	rsc	undef	This bit is used to control whether physical accesses to locations outside the Host Controller itself as well as any other DMA data accesses should be swapped or not. When 0, data quadlets are sent/received in little endian order. When 1, data quadlets are sent/received in big endian order. See the explanation following this table. Software should change this bit only when linkEnable is 0, otherwise unspecified behavior will result. Support of this bit is optional for motherboard implementations and required for all other implementations. See section 5.7.1 below for more information.
programPhyEnable	rc *or* r	*	This bit informs upper-level generic software (e.g., OHCI device driver) if lower-level implementation specific software (e.g., BIOS or Open Firmware) has consistently configured P1394a enhancements in the Link and PHY. When 1 and while linkEnable is 0, generic software is responsible for configuring the P1394a enhancements within the PHY and the aPhyEnhanceEnable bit within the Host Controller Link in a consistent manner. When 0, generic software may not modify the P1394a enhancement configuration in either the Link or PHY and cannot interpret the setting of aPhyEnhanceEnable *On a hardware reset, this bit should be 1 for Host Controllers that can support the enabling of all P1394a PHY enhancements by generic software, and may be 0 for Host Controllers which are always configured by lower-level software. A soft reset and a bus reset shall not affect this bit. See section 5.7.2 below for more information.
aPhyEnhanceEnable	rsc *or* r	**	When the programPhyEnable bit is 1, this bit is used by generic, implementation independent software (e.g., OHCI device driver) to enable the Host Controller Link to use <u>all</u> of P1394a enhancements. Generic software can only modify this bit when the programPhyEnable bit is 1 and the linkEnable bit is 0. This bit is meaningless to software when the programPhyEnable bit is 0. When 0, none of the P1394a enhancements are enabled within the Link. When 1, the set of all P1394a enhancements is enabled within the Link. **On a hardware reset, this bit should be 0 for Host Controllers which initialize without all of the P1394a PHY enhancements enabled, and 1 for those which initialize with all P1394a PHY enhancements enabled. A soft reset and a bus reset shall not affect this bit. See section 5.7.2 below for more information.
LPS	rs	1’b0	This bit is used to control the Link Power Status. Software must set LPS to 1 to permit Link↔PHY communication. Once set, the link can use LREQs to perform PHY reads and writes. An LPS value of 0 prevents Link↔PHY communication. In this state, the only accessible Host Controller registers are Version, VendorID, HCControl, GUID_ROM, GUIDHi and GUIDLo. Access to other registers is not defined. Hardware and software resets clear LPS to 0. Software shall not clear LPS. See section 5.7.3 below for more information.
postedWriteEnable	rsc	undef	This bit is used to enable (1) or disable (0) physical posted writes. When disabled (0) physical writes shall be handled but shall not be posted and instead are ack’ed with ack_pending. Software should change this bit only when linkEnable is 0, otherwise unspecified behavior will result. See Section 12., “Physical Requests,” for information about posted writes.

Table 5-12 — HCControl register fields

field name	rscu	reset	description
linkEnable	rsu	1’b0	Software must set this bit to 1 when the system is ready to begin operation and then force a bus reset. This bit is necessary to keep other nodes from sending transactions before the local system is ready. When this bit is clear the Host Controller is logically and immediately disconnected from the 1394 bus. The link shall not process or interpret any packets received from the PHY, nor shall the link generate any *bus* requests. However, the link may access PHY registers via the PHY control register. This bit is cleared to 0 by a hardware reset or software reset, and shall not be cleared by software. Software should not set the linkEnable bit until the Configuration ROM mapping register (section 5.5.6) is valid. See section 5.7.3 below for more information.
softReset	rsu	***	When set to 1, all Host Controller state is reset, all FIFO’s are flushed and all Host Controller registers are set to their hardware reset values unless otherwise specified. Registers outside of the OpenHCI realm, i.e., host attachment registers such as those for PCI, are not affected. ***The read value of this bit is 1 while a soft reset or a hard reset is in progress. The read value of this bit is 0 when neither a soft reset nor hard reset are in progress. Software can use the value of this bit to determine when a reset has completed and the Host Controller is safe to operate.

5.7.1 noByteSwapData

The 1394 bus is quadlet based big endian. By convention, when quadlets are sent in big endian order, the leftmost byte (bits 31-24) of a quadlet is sent first. When sent in little endian order, the right most byte (bits 7-0) is sent first with the leftmost bit of each byte sent first.

When the Host Controller sends/receives a packet, the header information is always sent/received in big endian order (leftmost byte first). Header information is composed of a sequence of quadlets which is invariant over big and little endian system.

When the HCControl.*noByteSwapData* bit is not set, data quadlets are sent/received in little endian order and when HCControl.*noByteSwapData* is set, data quadlets are sent/received in big endian order. The data quadlets that are subject to swap are:

1) any data quadlet covered by data CRC (tcodes 4'h1, 4'h7, 4'h9, 4'hA an 4'hB)
2) the data quadlet in a quadlet write request (tcode 4'h0)
3) the data quadlet in a quadlet read response (tcode 4'h6)

Since the cycle_time is self contained within the Host Controller, it is never byte-swapped regardless of the setting of the noByteSwapData bit.

The data in a PHY packet (identified internally with tcode 4'hE) is not byte swapped for send or receive.

[**Note**: due to some confusion regarding this bit, an explanation and some examples will be published at a later date and made available on the OpenHCI FTP site.]

5.7.2 programPhyEnable and aPhyEnhanceEnable

After a hardware or software reset, system software must ensure that the PHY and the Link are set to a consistent, compatible set of P1394a enhancements. The programPhyEnable and aPhyEnhanceEnable bits are provided to enable software to accomplish this task.

Since different levels of software may be responsible for ensuring this setup, the programPhyEnable bit is defined to allow communication between implementation specific lower-level software (e.g., BIOS or Open Firmware) and generic, implementation independent upper-level software (e.g., OHCI device driver). If generic software reads this bit as a 1, it is responsible for configuring the P1394a enhancements in both the Link and PHY in a consistent manner (either all enhancements enabled or all enhancements disabled). A 0 value for this bit informs the upper-level system software that no further changes to the P1394a configurations of the Link and PHY are permitted since either: 1) lower-level software has previously performed initialization appropriate to the Host Controller capabilities, or 2) the link has hardwired P1394a capabilities to match the PHY with which it is being used. Note that this bit is only a software flag and does not control any Host Controller functionality.

The programPhyEnable bit may be read-only, returning a zero value, if upper-level software will not be involved in the configuration of P1394a enhancements for the Link and PHY. This is appropriate when the Link and PHY are hardwired with compatible settings or when lower-level software will consistently configure both the Link and PHY. To allow the possibility for upper-level software control of P1394a enhancements, programPhyEnable should be implemented as read/clear with a hardware reset value of 1. Software should clear programPhyEnable once the PHY and Link have been programmed consistently by either lower-level or upper-level software.

When programPhyEnable is set to 1, then the aPhyEnhanceEnable bit allows generic software to enable or disable all P1394a enhancements within the Host Controller Link. A value of 1 for aPhyEnhanceEnable configures the Link to use all P1394a enhancements and is appropriate when software has enabled all of the enhancements within the PHY. Likewise, a value of 0 prevents the Link from using any P1394a enhancements and is appropriate when software has disabled all of the enhancements within the PHY. Note that generic software must not attempt to modify or interpret the setting of the aPhyEnhanceEnable bit if programPhyEnable contains a 0.

The aPhyEnhanceEnable bit may be read-only or read/set/clear depending on options implemented in the hardware. If the aPhyEnhanceEnable bit is read/set/clear, it shall hardware reset to 0 for default compatibility with legacy PHYs. If the aPhyEnhanceEnable bit is read-only, it shall hardware reset to 0 if it only operates with legacy PHYs or shall hardware reset to 1 if it only operates with P1394a PHYs. In either case, the upper-level software will be responsible for programming the PHY consistently (provided programPhyEnable is set).

The following table illustrates the responsibility of generic software for some example Link implementations.

Table 5-13 — programPhyEnable and aPhyEnhanceEnable Examples

Link Capabilities	programPhyEnable	aPhyEnhanceEnable	comments
Legacy-only Link	0 (read-only)	X(meaningless)	Generic software shall not change PHY or Link enhancement configuration.
P1394a-only Link	0 (read/clear)	X (meaningless)	Generic software shall not change PHY or Link enhancement configuration.
	1 (read/clear)	1 (read-only)	Generic software must enable P1394a enhancements in the PHY.
P1394a capable Link	0 (read/clear)	X (meaningless)	Generic software shall not change PHY or Link enhancement configuration.
	1 (read/clear)	0 (read/set/clear)	Generic software may modify aPhyEnhanceEnable and shall configure PHY consistently.
	1 (read/clear)	1 (read/set/clear)	

In all cases, the PHY-Link enhancements shall be programmed only when linkEnable is 0.

5.7.3 LPS and linkEnable

There are three basic tasks and ensuing requirements with respect to the PHY/Link interface:

- Bootstrap of Open HCI.
  This requires a mechanism to configure the link and the PHY prior to receiving any packets or generating any bus requests.
- Recovery from a hung system.
  This requires a mechanism which places Open HCI in a near pre-bootstrap condition, and allows the PHY and link to get back into sync if required.
- Power Management via Suspend/Resume
  This requires a mechanism to inform the PHY that PHY/Link communication is no longer required and, if possible, the PHY can suspend itself if no active ports remain.

To achieve proper behavior in satisfying these requirements, software shall always assert the signals in the following sequence: LPS, then linkEnable, then any other individual context enables or runs. The Host Controller behavior when violating this order is undefined and can produce unreliable behavior. The table below illustrates the progressive functionality as these signals are asserted.

Table 5-14 — LPS and linkEnable assertion

#	LPS	linkEnable	contextControl.*run*	Sequence Comments
a.	Off	Off	Off	Initial State
b.	On	Off	Off	Allows SCLK to start
c.	On	Off	Off	Config PHY/Link registers
d.	On	On	Off	Initiate Bus Reset
e.	On	On	Off	Physical DMA/Cycle Starts Okay
f.	On	On	On	Normal Operation

Following a hardware or software reset, LPS and linkEnable are Off as shown in step *a*. Software proceeds to enable the link power status (*b*) and when SCLK has started, software can configure PHY and Link registers as listed in step *c* (e.g., Self-ID receive DMA registers). Setting linkEnable in step *d* enables some DMA function, and asserting contextControl.*run* (*e*) for the Host Controller contexts then yields full functionality.

5.8 FairnessControl register (optional)

This register provides a mechanism by which software can direct the Host Controller to transmit multiple asynchronous request packets during a fairness interval as specified in P1394a.

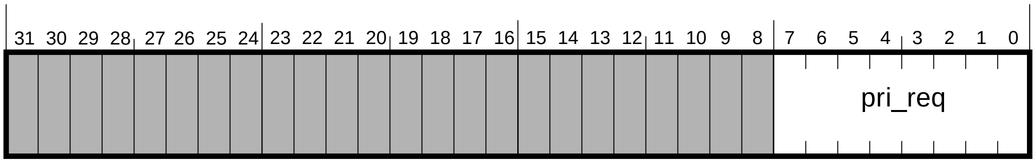


Figure 5-15 — FairnessControl register

Table 5-15 — FairnessControl register fields

field name	rw	hard reset	soft & bus-reset	description
pri_req	rw	undef	N/A	This field specifies the maximum number of priority arbitration requests for asynchronous request packets that the link is permitted to make of the PHY during a fairness interval. A *pri_req* value of 8’h0 is equivalent to the behavior specified by IEEE 1394-1995. The number of implemented bits is variable as per the P1394a specification. Unimplemented bits shall be read-only and shall read as 0’s.

The FairnessControl register is configured by software in conjunction with software support of the Fairness Budget Register specified in P1394a. Transmission of all asynchronous packets via the Asynchronous Transmit Request context shall be governed by the fairness protocol supported by the Host Controller.

5.9 LinkControl registers (set and clear)

This register provides the control flags that enable and configure the link core protocol portions of the 1394 Open HCI. It contains controls for the receiver, and cycle timer. There are two addresses for this register: LinkControlSet and LinkControlClear. On read, both addresses return the contents of the control register. For writes, the two addresses have different behavior: a one bit written to LinkControlSet causes the corresponding bit in the LinkControl register to be set, while a zero bit leaves the corresponding bit in the LinkControl register unaffected. On the other hand, a one bit written to LinkControlClear causes the corresponding bit in the LinkControl register to be cleared, while a zero bit leaves the corresponding bit in the LinkControl register unaffected.

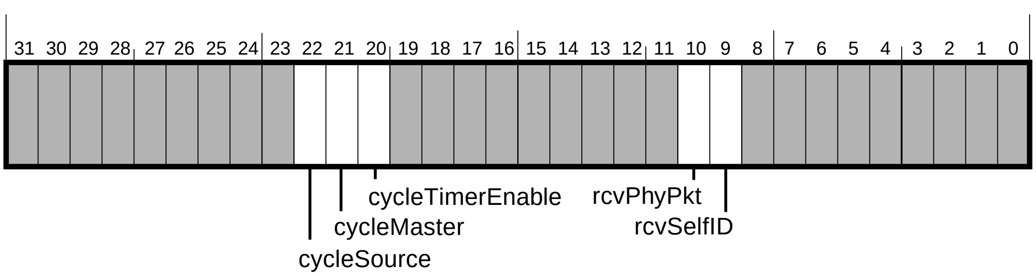


Figure 5-16 — LinkControl register

Table 5-16 — LinkControl register fields

field name	rscu	reset	description
cycleSource	rsc *or* r	**	Optional. When one, the cycle timer will use an external source to determine when to increment cycleCount. When cycleCount is incremented, cycleOffset is reset to 0. If cycleOffset reaches 3071 before an external event occurs, it will remain at 3071 until the external signal is received and is then reset to 0. When the cycleSource bit is zero, the 1394 Open HCI will roll the cycle timer over when the timer reaches 3072 cycles of the 24.576 MHz clock (i.e., 8 kHz). If not implemented, this bit will read as 0. CycleSource has an effect only when cycleMaster is enabled. ** A hardware reset, clears this bit to 0. A software reset has no effect.
cycleMaster	rscu	undef	When one and the PHY has notified the 1394 Open HCI that it is root, the 1394 Open HCI will generate a cycle start packet every time the cycle timer rolls over, based on the setting of the cycleSource bit. When zero, the 1394 Open HCI will accept received cycle start packets to maintain synchronization with the node which is sending them. This bit is automatically zeroed when the IntEvent.*cycleTooLong* event occurs and cannot be set until the IntEvent.*cycleTooLong* bit is cleared.
cycleTimerEnable	rsc	undef	When one, the cycle timer offset will count cycles of the 24.576 MHz clock and roll over at the appropriate time based on the settings of the above bits. When zero, the cycle timer offset will not count.
rcvPhyPkt	rsc	undef	When one, the receiver will accept incoming PHY packets into the AR request context if the AR request context is enabled. This does *not* control either the receipt of self-identification packets during the Self-ID phase of bus initialization or the queuing of synthesized bus reset packets in the AR DMA Request Context buffer (section 8.4.2.3). This does control receipt of any self-identification packets received outside of the Self-ID phase of bus initialization.
rcvSelfID	rsc	undef	When one, the receiver will accept incoming self-identification packets. Before setting this bit to one, software must ensure that the self ID buffer pointer register contains a valid address.

5.10 Node identification and status register

This register contains the CSR address for the node on which this chip resides. The 16-bit combination of busNumber and nodeNumber is referred to as the Node ID.

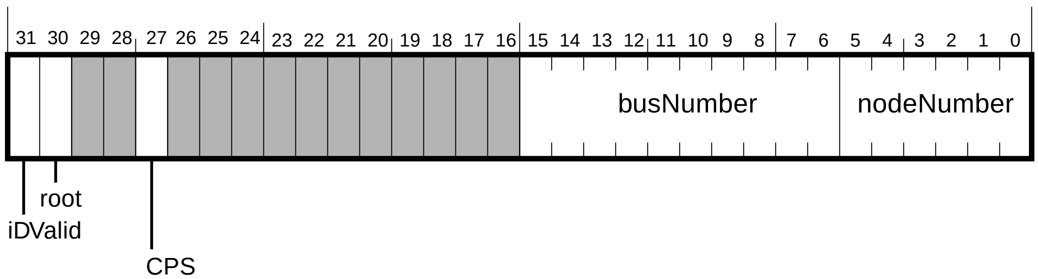


Figure 5-17 — Node ID register

Table 5-17 — Node ID register fields

field name	**rwu**	**reset**	**description**
iDValid	ru	1’b0	This bit indicates whether or not the 1394 Open HCI has a valid node number. It is cleared when the bus reset state is detected and set again when the 1394 Open HCI receives a new node number from the PHY. If iDValid is clear, software should not set ContextControl.*run* for either of the AT DMA contexts.
root	ru	1’b0	This bit is set during the bus reset process if the attached PHY is root.
CPS	ru	1’b0	Set if the PHY is reporting that cable power status is OK (VP 8V).
busNumber	rwu	10’h3FF	This number is used to identify the specific 1394 bus this node belongs to when multiple 1394-compatible busses are connected via a bridge. This field is set to 10’h3FF on a bus reset.
nodeNumber	ru	undef	This number is the physical node number established by the PHY during self-identification. It is automatically set to the value received from the PHY after the self-identification phase. If the PHY sets the nodeNumber to 63, software should not set ContextControl.*run* for either of the AT DMA contexts. As a reminder, links must refrain from acknowledging any packet received with a destination nodeNumber of 63 regardless of the setting of this field.

This register shall be written autonomously and atomically by the Host Controller with the value in PHY register 0 following the self-identification phase of bus initialization. Although IntEvent.*phyRegRcvd* shall not be set when the contents of PHY register 0 are written here, software can use the IntEvent.*selfIDComplete* interrupt to detect that the self-identification phase has completed can then check for a new valid Node ID.

5.11 PHY control register

The PHY control register is used to read or write a PHY register. To read a register, the address of the register is written to the regAddr field along with a 1 in the rdReg bit. When the read request has been sent to the PHY (through the LReq pin), the rdReg bit is cleared to 0. When the PHY returns the register (through a status transfer), the rdDone bit transitions to one and then the IntEvent.*phyRegRcvd* interrupt is set. The address of the register received is placed in the rdAddr field and the contents in the rdData field.

Software shall not issue a read of PHY register 0. The most recently available contents of this register shall be reflected in the NodeID register (section 5.10). The Host Controller shall only write the contents of PHY register 0 into the nodeID register, and never into this register.

To write to a PHY register, the address of the register is written to the regAddr field, the value to write to the wrData field, and a 1 to the wrReg bit. The wrReg bit is cleared when the write request has been transferred to the PHY.

Software shall serialize all PHY register reads and writes. Only after the current PHY register read or write completes may software issue a different PHY register read or write.

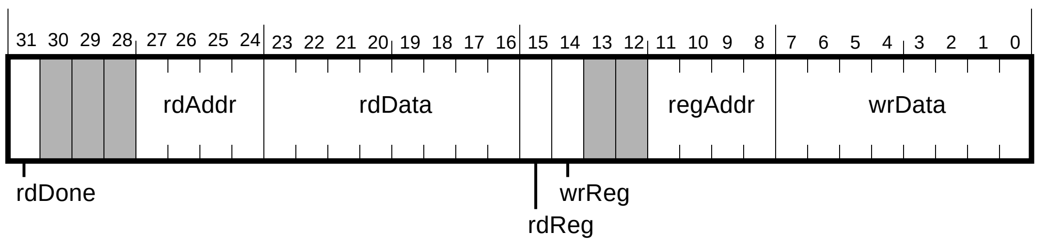


Figure 5-18 — PHY control register

Table 5-18 — PHY control register fields

field name	rwu	reset	description
rdDone	ru	undef	rdDone is cleared to 0 by the Host Controller when either rdReg or wrReg is set to 1. This bit is set to 1 when a register transfer is received from the PHY.
rdAddr	ru	undef	This is the address of the register most recently received from the PHY.
rdData	ru	undef	Contains the data read from the PHY register at rdAddr
rdReg	rwu	1’b0	Set rdReg to initiate a read request to a PHY register. This bit is cleared when the read request has been sent. The wrReg bit must not be set while the rdReg bit is set.
wrReg	rwu	1’b0	Set wrReg to initiate a write request to a PHY register. This bit is cleared when the write request has been sent. The rdReg bit must not be set while the wrReg bit is set.
regAddr	rw	undef	regAddr is the address of the PHY register to be written or read.
wrData	rw	undef	This is the contents to be written to a PHY register. Ignored for a read.

This register shall be written atomically such that all bits are accumulated and written together when rdDone is set

To ensure a consistent interface regardless of the PHY/Link implementation, the register map of P1394A PHYs shall be supported.

5.12 Isochronous Cycle Timer Register

The isochronous cycle timer register is a read/write register that shows the current cycle number and offset. The cycle timer register is split up into three fields. The lower order 12 bits are the cycle offset, the middle 13 bits are the cycle count, and the upper order 7 bits count time in seconds. When the 1394 Open HCI is cycle master, this register is transmitted with the cycle start message. When the 1394 Open HCI is not cycle master, this register is loaded with the data field in each incoming cycle start. In the event that the cycle start message is not received, the fields continue incrementing on their own (when cycleTimerEnable is set in the LinkControl register) to maintain a local time reference.

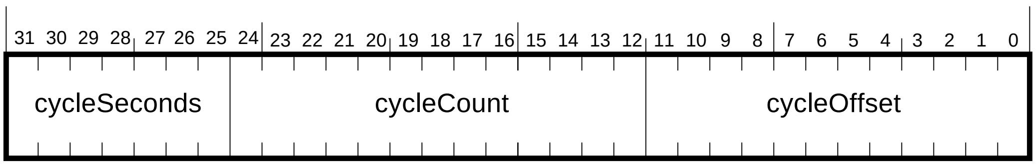


Figure 5-19 — Isochronous cycle timer register

Table 5-19 — Isochronous cycle timer register fields

field name	rwu	reset	description
cycleSeconds	rwu	N/A	This field counts seconds (cycleCount rollovers) modulo 128
cycleCount	rwu	N/A	This field counts cycles (cycleOffset rollovers) modulo 8000.
cycleOffset	rwu	N/A	This field counts 24.576MHz clocks modulo 3072, i.e., 125 µs. If an external 8KHz clock configuration is being used, cycleOffset must be set to 0 at each tick of the external clock. Note that the ability to support an external clock is optional. Implementations which *can support* an external clock are not required to have an external clock.

A host initiated write to the cycleTime register may evoke an IntEvent.*cycleInconsistent* in some implementations.

5.13 Asynchronous Request Filters

The 1394 OpenHCI allows for selective access to host memory and the Asynchronous Receive Request context so that software can maintain host memory integrity. The selective access is provided by two sets of 64-bit registers: PhysRequestFilter and AsynchRequestFilter. These registers allow access to physical memory and the AR Request context on a nodeID basis. The request filters are not applied to quadlet read requests directed at the Config ROM (including the ConfigROM header, BusID, Bus Options, and Global Unique ID registers) nor to accesses directed to the isochronous resource management registers. When the link is enabled, access by any node to the first 1K of CSR config ROM is enabled(see section 5.5.6). The Asynchronous Request Filters *do **not** have any effect* on Asynchronous Response packets.

5.13.1 AsynchronousRequestFilter Registers (set and clear)

When a request is received by the Host Controller from the 1394 bus and that request does not access the first 1K of CSR config ROM on the Host Controller, then the sourceID is used to index into the AsynchronousRequestFilter. If the corresponding bit in the AsynchronousRequestFilter is set to 0, then requests from that device are not enabled; there will be no *ack_* sent, and the requests will be ignored by the Host Controller. If however, the bit is set to 1, the requests are accepted and will be processed according to the address of the request and the setting of the PhysicalRequestFilter register.

Requests to offsets above PhysicalUpperBound (section 5.14), with the exception of offsets handled physically as described in Section 12., are always sent to the Asynchronous Receive Request DMA context. If the AR Request DMA context is not enabled, then the Host Controller will ignore the request.

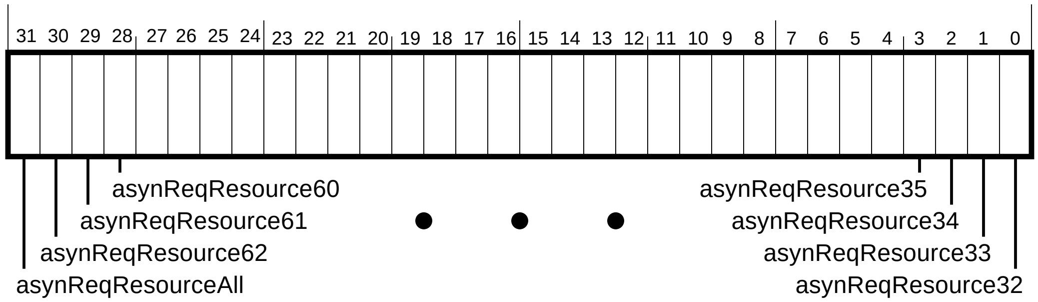


Figure 5-20 — AsynchronousRequestFilterHi (set and clear) register

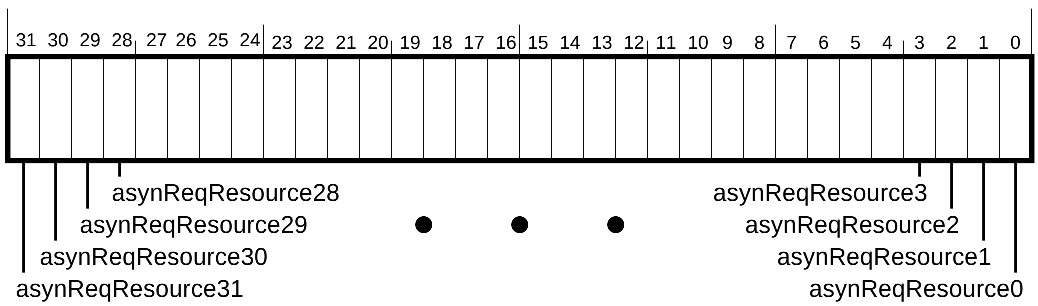


Figure 5-21 — AsynchronousRequestFilterLo (set and clear) register

Table 5-20 — AsynchronousRequestFilter register fields

field name	rscu	reset	description
asynReqResourceN	rscu	1’b0	If set to one for local bus node number N, asynchronous requests received by the Host Controller from that node will be accepted. All asynReqResourceN bits shall be cleared to zero when a bus reset occurs.
asynReqResourceAll	rscu	1’b0	If set to one, all asynchronous requests received by the Host Controller from all bus nodes (including the local bus) will be accepted, and the values of all asynReqResourceN bits shall be ignored. A bus reset does not affect the value of the asynReqResourceAll bit.

The AsynchronousRequestFilter bits are set by writing a one to the corresponding bit in the AsynchronousRequestFilter-HiSet or AsynchronousRequestFilterLoSet address. They are cleared by writing a one to the corresponding bit in the AsynchronousRequestFilterHiClear or AsynchronousRequestFilterLoClear address. If bit “asynReqResourceN” is set, then requests with a sourceID of either {10’h3FF, #n} or {busID, #n} will be accepted. If the asynReqResourceAll bit is set in AsynchronousRequestFilterHi, requests from all bus nodes including those on the local bus are accepted.

Reading the AsynchronousRequestFilter registers returns their current state. All asynReqResourceN bits in the AsynchronousRequestFilter register are cleared to 0 on a 1394 bus reset.

5.13.2 PhysicalRequestFilter Registers (set and clear)

If an asynchronous request is allowed from a node, and the offset is below PhysicalUpperBound (section 5.14) the sourceID of the request is used as an index into the PhysicalRequestFilter. If the corresponding bit in the PhysicalRequestFilter is set to 0, then the request is forwarded to the Asynchronous Receive Request DMA context. If however, the bit is set to 1, then the request is sent to the physical response unit. (Note that within the Physical Range, lock transactions and block transactions with a non-zero extended tcode are always forwarded to the Asynchronous Receive Request DMA context. See Section 12.)

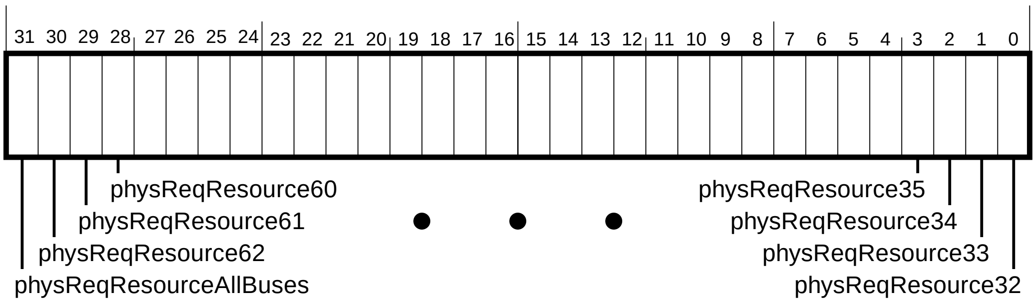


Figure 5-22 — PhysicalRequestFilterHi (set and clear) register

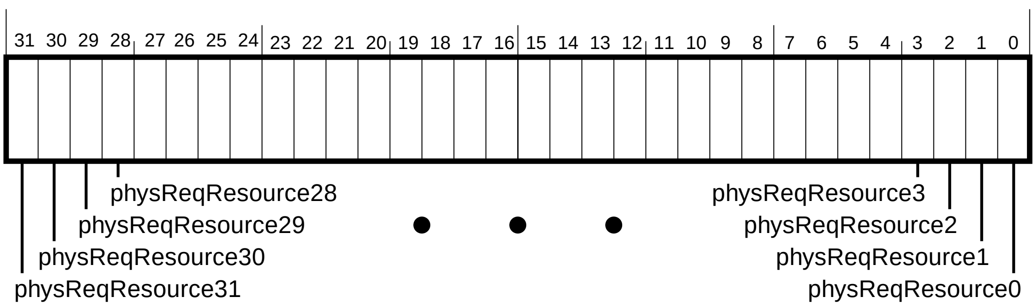


Figure 5-23 — PhysicalRequestFilterLo (set and clear) register

Table 5-21 — PhysicalRequestFilter register fields

field name	rscu	reset	description
physReqResourceN	rscu	1’b0	If set to one for local bus node number N, then asynchronous physical requests received by the Host Controller from that node will be accepted.
physReqResourceAllBuses	rscu	1’b0	If set to one, all asynchronous physical requests received by the Host Controller from non-local bus nodes will be accepted.

The PhysicalRequestFilter bits are set by writing a one to the corresponding bit in the PhysicalRequestFilterHiSet or PhysicalRequestFilterLoSet address. They are cleared by writing a one to the corresponding bit in the PhysicalRequestFilterHiClear or PhysicalRequestFilterLoClear address. If bit “physReqResource*n*” is set, then requests with a sourceID of either {10’h3FF, #n} or {busID, #n} will be accepted. If the physReqResourceAllBuses bit is set in PhysicalRequestFilterHi, physical requests from any device on any other bus are accepted (bus number other than 10’h3FF and busID).

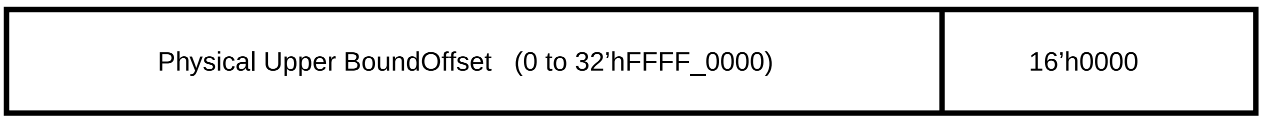
Physical requests that are rejected by the PhysicalRequestFilter are sent to the AR Request DMA context if the AR Request DMA context is enabled. If it is disabled then the Host Controller ignores the requests.

Reading the PhysicalRequestFilter registers returns their current state. All bits in the PhysicalRequestFilter are set to 0 on a 1394 bus reset.

5.14 Physical Upper Bound register (optional)

Asynchronous requests which are candidates to be handled by the physical response unit include requests that have a destination offset which falls within the *physical* range. This range begins at 48’h0 and ends at the offset specified in this register. In general, requests at physUpperBoundOffset or higher will be handled by the Asynchronous Receive Request context. Refer to Chapter 12. for details about Physical Requests.

For use with 64-bit implementations, the Physical Upper Bound register comprises the top 32 bits of a 48-bit offset and provides a mechanism for implementations to specify physical access for offsets above 48’0000_FFFF_FFFF (4GB).



Physical Upper Bound

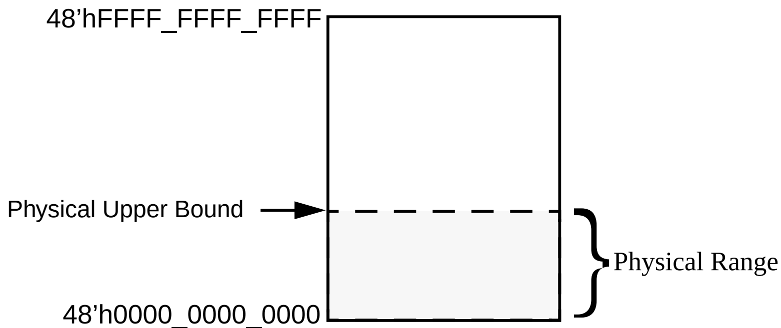


Figure 5-24 — 48-bit Physical Upper Bound

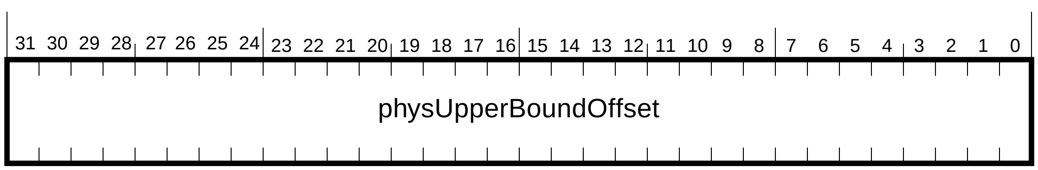


Figure 5-25 — Physical Upper Bound register

Table 5-22 — Physical Upper Bound register fields

field name	rwu	hard reset	soft & bus-reset	description
physUpperBoundOffset	rw *or* r	undef	N/A	Represents the high-order 32 bits of the 48 bit destination offset, with the remaining 16 bits set to 16’h0. Requests to this offset or higher shall be handled by the Asynchronous Receive Request context, with some exceptions as outlined in Chapter 12.. Software shall not set physUpperBoundOffset to a value above 32’hFFFF_0000. If implemented, this shall be a read/write register. If not implemented, this register shall be read-only with a value of 32’h0 and the upper bound of the physical range shall be 48’h0001_0000_0000.

6. Interrupts

The 1394 Open HCI reports two classes of interrupts to the host: DMA interrupts and device interrupts. DMA interrupts are generated when DMA transfers complete (or are aborted). Device interrupts come directly from the remaining 1394 Open HCI logic. For example, one of these interrupts could be sent in response to the asserting edge of cycleStart, a signal which indicates that a new isochronous cycle has started.

The 1394 Open HCI contains two primary 32-bit registers to report and control interrupts: IntEvent and IntMask. Both registers have two addresses: a “Set” address and a “Clear” address. For a write to either register, a “one” bit written to the “Set” address causes the corresponding bit in the register to be set (excluding bits which are read-only), while a “one” bit written to the “Clear” address causes the corresponding bit to be cleared. For both addresses, writing a “zero” bit has no effect on the corresponding bit in the register.

The IntEvent register contains the actual interrupt request bits. Each of these bits corresponds to either a DMA completion event, or a transition on a device interrupt line. The IntMask register is ANDed with the IntEvent register to enable selected bits to generate processor interrupts. Software writes to the IntEventClear register to clear interrupt conditions reported in the IntEvent register.

A processor interrupt is generated when one or more unmasked bits are set in the IntEvent register. Low-level software responds to the interrupt by reading the IntEvent register, then writing the value read to the IntEventClear register. At this point the interrupt request is deasserted (assuming no new interrupt bit has been set). Software can proceed to process the reported interrupts in whatever priority order it chooses, and is free to re-enable interrupts as soon as the IntEventClear register is written.

In addition, the 1394 Open HCI contains four secondary 32-bit registers to report and control interrupts for isochronous transmit and receive contexts. Each register has two addresses: a “Set” address and a “Clear” address.

6.1 IntEvent (set and clear)

This register reflects the state of the various interrupt sources from the 1394 Open HCI. The interrupt bits are set by an asserting edge of the corresponding interrupt signal, or by software by writing a one to the corresponding bit in the IntEventSet address. They are cleared by writing a one to the corresponding bit in the IntEventClear address.

Reading the IntEventSet register returns the current state of the IntEvent register. Reading the IntEventClear register returns the *masked* version of the IntEvent register (*IntEvent & IntMask*).

On a hardware reset or soft reset, the values of all bits in this register are undefined.

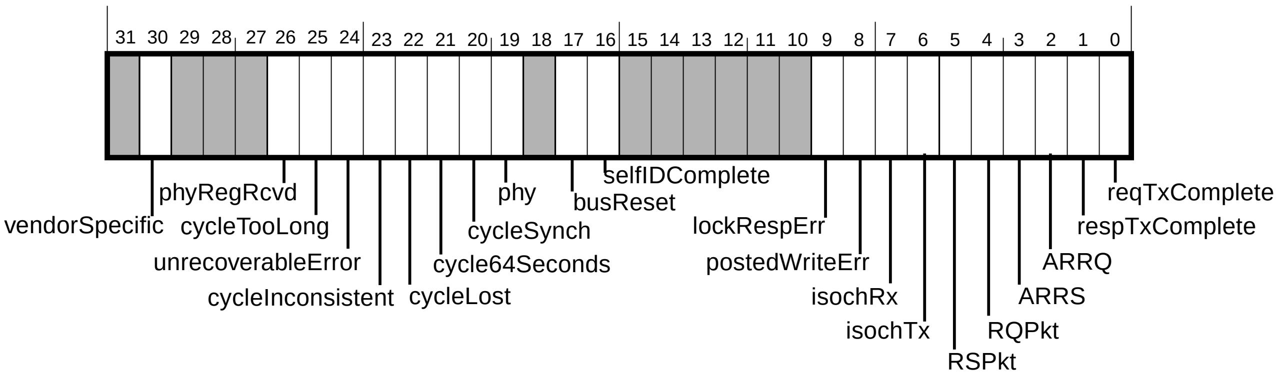


Figure 6-1 — IntEvent register

Table 6-1 — IntEvent register description (Sheet 1 of 2)

Field	Bit #	rscu	Description
reqTxComplete	0	rscu	Asynchronous request transmit DMA interrupt. This bit is conditionally set upon completion of an AT DMA request OUTPUT_LAST* command.
respTxComplete	1	rscu	Asynchronous response transmit DMA interrupt. This bit is conditionally set upon completion of an AT DMA response OUTPUT_LAST* command.
ARRQ	2	rscu	Asynchronous Receive Request DMA interrupt. This bit is conditionally set upon completion of an AR DMA Request context command descriptor.
ARRS	3	rscu	Asynchronous Receive Response DMA interrupt. This bit is conditionally set upon completion of an AR DMA Response context command descriptor.
RQPkt	4	rscu	Indicates that a packet was sent to an asynchronous receive request context buffer and the descriptor’s xferStatus and resCount fields have been updated. This differs from ARRQ above since RQPkt is a per-packet completion indication and ARRQ is a per-command descriptor (buffer) completion indication. AR Request buffers may contain more than one packet.
RSPkt	5	rscu	Indicates that a packet was sent to an asynchronous receive response context buffer and the descriptor’s xferStatus and resCount fields have been updated. This differs from ARRS above since RSPkt is a per-packet completion indication and ARRS is a per-command descriptor (buffer) completion indication. AR Response buffers may contain more than one packet.
isochTx	6	ru	Isochronous Transmit DMA interrupt. Indicates that one or more isochronous transmit contexts have generated an interrupt. This is not a latched event, it is the OR’ing all bits in (isoXmitIntEvent & isoXmitIntMask). The isoXmitIntEvent register indicates which contexts have interrupted. See section 6.3.
isochRx	7	ru	Isochronous Receive DMA interrupt. Indicates that one or more isochronous receive contexts have generated an interrupt. This is not a latched event, it is the OR’ing all bits in (isoRecvIntEvent & isoRecvIntMask). The isoRecvIntEvent register indicates which contexts have interrupted. See section 6.4.
postedWriteErr	8	rscu	Indicates that a host bus error occurred while the Host Controller was trying to write a 1394 write request, which had already been given an ack_complete, into system memory. The 1394 destination offset and sourceID are available in the PostedWriteAddress registers described in section 13.2.8.1.

Table 6-1 — IntEvent register description (Sheet 2 of 2)

Field	Bit #	rscu	Description
lockRespErr	9	rscu	Indicates that the Host Controller attempted to return a lock response for a lock request to a serial bus register described in Section 5.5.1, but did not receive an ack_complete after exhausting all permissible retries.
reserved	10-15		
selfIDcomplete	16	rscu	A selfID packet stream has been received. Will be generated at the end of the bus initialization process if LinkControl.*rcvSelfID* is set. This bit is turned off simultaneously when IntEvent.*busReset* is turned on.
busReset	17	rscu	Indicates that the PHY chip has entered bus reset mode. See section 6.1.1 below for information on when to clear this interrupt.
reserved	18		
phy	19	rscu	Generated when the PHY requests an interrupt through a status transfer.
cycleSynch	20	rscu	Indicates that a new isochronous cycle has started. Set when the low order bit of the internal isochronousCycleTimer.*cycleCount* toggles.
cycle64Seconds	21	rscu	Indicates that the 7th bit of the cycle second counter has changed.
cycleLost	22	rscu	A lost cycle is indicated when no cycle_start packet is sent/received between two successive cycleSynch events.
cycleInconsistent	23	rscu	A cycle start was received that had an isochronous cycleTimer.*seconds* and isochronous cycleTimer.*count* different from the value in the CycleTimer register. Implementations are free to indicate a cycleInconsistent if a host initiated write changes the cycleSeconds or cycleCount fields of the cycleTimer register (section 5.12). For the effect of this condition on isochronous transmit, refer to section 9.5.1 and for isochronous receive refer to section 10.5.1.
unrecoverableError	24	rscu	This event occurs when the Host Controller encounters any error that forces it to stop operations on any or all of its subunits. For example, when a DMA context sets its contextControl.*dead* bit. While unrecoverableError is set, all normal interrupts for the context(s) that caused this interrupt will be blocked from being set.
cycleTooLong	25	rscu	If LinkControl.*cycleMaster* is set, this indicates that an isochronous cycle lasted longer than the allotted time. For implementations with a discrete cycleTooLong timer, hardware is expected to trigger this event no less than 115 µsecs and no more than 120 µsecs after sending a cycle start packet unless a subaction gap or bus reset indication is first observed. LinkControl.*cycleMaster* is cleared by this event.
phyRegRcvd	26	rscu	The 1394 Open HCI has received a PHY register data byte which can be read from the PHY control register (see 5.11).
reserved	27-29		
vendorSpecific	30		Vendor defined.
reserved	31		

6.1.1 busReset

When a bus reset occurs and the busReset interrupt is set to one, the selfIDComplete interrupt is simultaneously cleared to 0. The Host Controller shall prevent software from clearing the busReset interrupt bit during the self-ID phase of bus initialization. Software must take precautions regarding the asynchronous transmit contexts before clearing this interrupt. Refer to section 7.2.3 for further details.

6.2 IntMask (set and clear)

The bits in the IntMask register have the same format as the IntEvent register, with the addition of masterIntEnable (bit 31). A one bit in the IntMask register enables the corresponding IntEvent register bit to generate a processor interrupt. A zero bit in IntMask disables the corresponding IntEvent register bit from generating a processor interrupt. A bit is set in the IntMask register by writing a one to the corresponding bit in the IntMaskSet address and cleared by writing a one to the corresponding bit in the IntMaskClear address.

If masterIntEnable is 0, all interrupts are disabled regardless of the values of all other bits in the IntMask register. The value of masterIntEnable has no effect on the value returned by reading the IntEventClear; even if masterIntEnable is 0, reading IntEventClear will return (IntEvent & IntMask) as described earlier in section 6.1.

On a reset, the IntMask.*masterIntEnable* bit (31) is set to 0 and the value of all other bits is undefined.

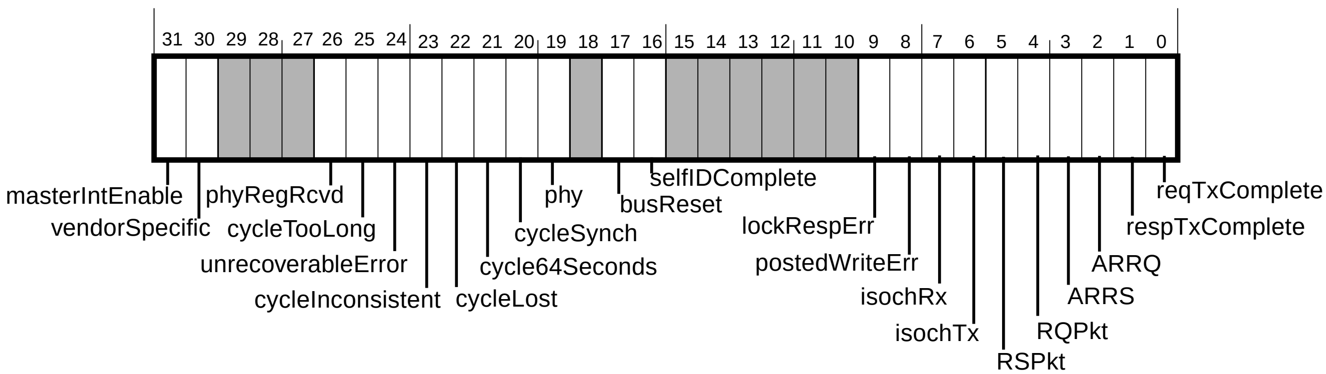


Figure 6-2 — IntMask register

Table 6-2 — IntMask register description

Field	Bit #	rscu	Description
interrupt events for:	0-9	rsc	See Table 6-1.
reserved	10-15		
interrupt events for	16-17	rsc	See Table 6-1.
reserved	18		
interrupt events for	19-26	rsc	See Table 6-1.
reserved	27-29		
vendorSpecific	30		Vendor defined.
masterIntEnable	31	rscu	If set, external interrupts will be generated in accordance with the IntMask register. If clear, no external interrupts will be generated regardless of the IntMask register settings.

6.3 IsochTx interrupt registers

There are two 32-bit registers to report isochronous transmit context interrupts: isoXmitIntEvent and isoXmitIntMask. Both registers have two addresses: a “Set” address and a “Clear” address. For a write to either register, a “one” bit written to the “Set” address causes the corresponding bit in the register to be set, while a “one” bit written to the “Clear” address causes the corresponding bit to be cleared. For all four addresses, writing a “zero” bit has no effect on the corresponding bit in the register.

The isoXmitIntEvent register contains the actual interrupt request bits. Each of these bits corresponds to a DMA completion event for the indicated isochronous transmit context. The isoXmitIntMask register is ANDed with the isoXmitIntEvent register to enable selected bits to generate processor interrupts. If (isoXmitIntMask & isoXmitIntEvent) is not zero, then the IntEvent.*isochTx* bit will be set to one, and if enabled via the IntMask register it will generate a processor interrupt. A software write to the isoXmitIntEventSet register can therefore cause an interrupt (if not otherwise masked). A software write to the isoXmitIntEventClear register will clear interrupt conditions reported in the isoXmitIntEvent register.

Reading the isoXmitIntEventSet register returns the current state of the isoXmitIntEvent register. Reading the isoXmitIntEventClear register returns the *masked* version of the isoXmitIntEvent register (*isoXmitIntEvent & isoXmitIntMask*).

6.3.1 isoXmitIntEvent (set and clear)

This register reflects the interrupt state of the isochronous transmit contexts. An interrupt is generated on behalf of an isochronous transmit context if an OUTPUT_LAST DMA command completes and its *i* bits are set to 2’b11 (interrupt always). Upon determining that the IntEvent.*isochTx* interrupt has occurred, software can check the isoXmitIntEvent register to determine which context(s) caused the interrupt.

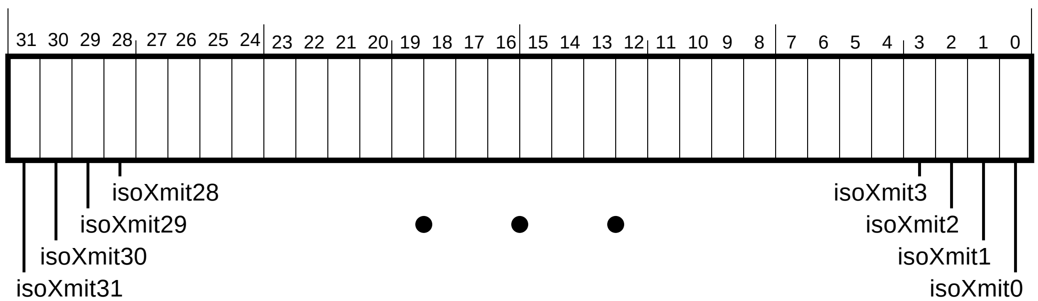


Figure 6-3 — isoXmitIntEvent (set and clear) register

On a hardware reset or soft reset, values of all bits in this register are undefined. Note that in these circumstances the IntMask.*masterIntEnable* is set to zero, therefore masking all interrupts until re-enabled by software.

6.3.2 isoXmitIntMask (set and clear)

The bits in the isoXmitIntMask register have the same format as the isoXmitIntEvent register. Setting a bit in this register enables the corresponding bit in the isoXmitIntMaskSet address and cleared by writing a one to the corresponding bit in the isoXmitIntMaskClear address.

Bits for all unimplemented contexts must read as 0’s. Software can use this register to determine which contexts are supported by writing to it with all 1’s, then reading it back. Contexts with a 1 are implemented, and those with a 0 are not.

On a hardware reset or soft reset, values for all bits in this register are undefined.

6.4 IsochRx interrupt registers

There are two 32-bit registers to report isochronous receive context interrupts: isoRecvIntEvent and isoRecvIntMask. Both registers have two addresses: a “Set” address and a “Clear” address. For a write to either register, a “one” bit written to the “Set” address causes the corresponding bit in the register to be set, while a “one” bit written to the “Clear” address causes the corresponding bit to be cleared. For all four addresses, writing a “zero” bit has no effect on the corresponding bit in the register.

The isoRecvIntEvent register contains the actual interrupt request bits. Each of these bits corresponds to a DMA completion event for the indicated isochronous receive context. The isoRecvIntMask register is ANDed with the isoRecvIntEvent register to enable selected bits to generate processor interrupts. If (isoRecvIntMask & isoRecvIntEvent) is not zero, then the IntEvent.*isochRx* bit will be set to one, and if enabled via the IntMask register it will generate a processor interrupt. A software write to the isoRecvIntEventSet register can therefore cause an interrupt (if not otherwise masked). A software write to the isoRecvIntEventClear register will clear interrupt conditions reported in the isoRecvIntEvent register.

Reading the isoRecvIntEventSet register returns the current state of the isoRecvIntEvent register. Reading the isoRecvIntEventClear register returns the *masked* version of the isoRecvIntEvent register (*isoRecvIntEvent & isoRecvIntMask*).

6.4.1 isoRecvIntEvent (set and clear)

This register reflects the interrupt state of the isochronous receive contexts. An interrupt is generated on behalf of an isochronous receive context if a final command of a DMA descriptor block completes and its *i* bits are set to 2'b11 (interrupt always). Upon determining that the IntEvent.*isochRx* interrupt has occurred, software can check the isoRecvIntEvent register to determine which context(s) caused the interrupt.

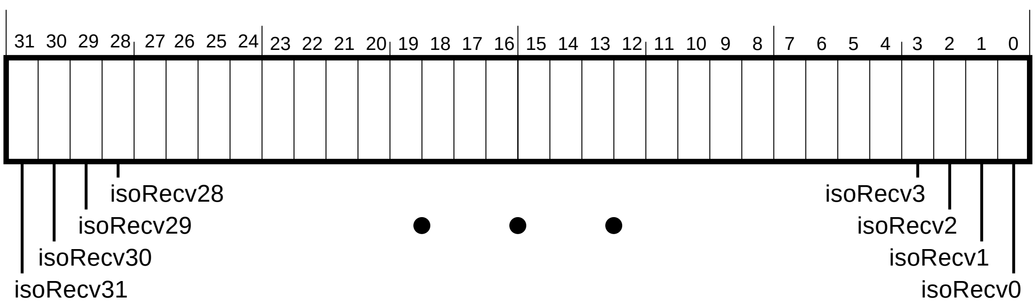


Figure 6-4 — isoRecvIntEvent (set and clear) register

On a hardware reset or soft reset, values of all bits in this register are undefined. Note that in these circumstances the IntMask.*masterIntEnable* is set to zero, therefore masking all interrupts until re-enabled by software.

6.4.2 isoRecvIntMask (set and clear)

The bits in the isoRecvIntMask register have the same format as the isoRecvIntEvent register. Setting a bit in this register enables the corresponding bit in the isoRecvIntMaskSet address and cleared by writing a one to the corresponding bit in the isoRecvIntMaskClear address.

Bits for all unimplemented contexts must read as 0's. Software can use this register to determine which contexts are supported by writing to it with all 1's then reading it back. Contexts with a 1 are implemented, and those with a 0 are not.

On a hardware reset or soft reset, values of all bits in this register are undefined.

7. Asynchronous Transmit DMA

The 1394 OpenHCI divides the transmission of asynchronous packets into three categories: asynchronous requests, asynchronous responses, and physical responses. This chapter describes how to use DMA to transmit asynchronous requests and asynchronous responses. For information regarding physical responses, see section 12., “Physical Requests.”

There is one DMA controller for each transmit context: the Asynchronous Transmit (AT) Request Controller for the AT request context, and the AT Response Controller for the AT response context. Although OpenHCI does not specify how many FIFO’s are required to support the AT DMA controllers, it is required that the re-transmission of request packets never blocks the transmission of response packets.

The AT Request context is used by software to transmit read, write and lock request packets and the AT Response context is used to send response packets to read, write, and lock requests that have earlier been received into the asynchronous receive request context buffers (see section 8., “Asynchronous Receive DMA”).

Each context consists of a context program and two registers. A context program is a list of commands for that context which direct the Host Controller on how to assemble packets for transmission. The DMA controller for that context executes each command, inserting data into the appropriate FIFO and interrupting as requested.

The following sections describe how to set up and manage an AT DMA context program and describe the data formats for the various asynchronous request and response packet types.

7.1 AT DMA Context Programs

Each asynchronous transmit packet, whether a request or response packet, shall be described by a contiguous list of command descriptors referred to as a *descriptor block*. A chain of descriptor blocks is referred to as a context program. There are four different command descriptors that can be used within each descriptor block: OUTPUT_MORE, OUTPUT_MORE-Immediate, OUTPUT_LAST and OUTPUT_LAST-Immediate. In the descriptions that follow, OUTPUT_MORE* refers to both the OUTPUT_MORE and OUTPUT_MORE-Immediate commands, OUTPUT_LAST* refers to both the OUTPUT_LAST and OUTPUT_LAST-Immediate commands and *-Immediate refers to both the OUTPUT_MORE-Immediate and OUTPUT_LAST-Immediate commands.

Each packet shall be specified in one descriptor block. A descriptor block may have either one single OUTPUT_LAST-Immediate descriptor, or may have one OUTPUT_MORE-Immediate descriptor followed by zero to five OUTPUT_MORE descriptors, followed by one OUTPUT_LAST descriptor. This allows software to combine up to seven fragments to specify a single packet. In addition, the first command descriptor in a descriptor block must be one of the *-Immediate commands to <u>transmit the full 1394 packet header</u> for the packet’s tcode type, where *packet header* is defined as all quadlets that appear before the 1394 packet header CRC quadlet and that are required by the respective packet format (defined in section 7.6). Further, a descriptor block for a packet shall not exceed 128 bytes. The OUTPUT_MORE and OUTPUT_LAST command descriptors are 16-bytes in length, and the *-Immediate descriptors are 32-bytes in length. All descriptors must be aligned on a 16-byte boundary.

In the sections below, the format for each command descriptor is shown. The shaded fields are reserved and should be set to 0 by software. Fields with a hardcoded value must be set to that value by software. The values of all other fields are described in each command’s descriptor element summary.

7.1.1 OUTPUT_MORE descriptor

The OUTPUT_MORE command descriptor is used to specify a host memory buffer from which the AT DMA controller will insert bytes into the appropriate transmit FIFO. It has the following format.

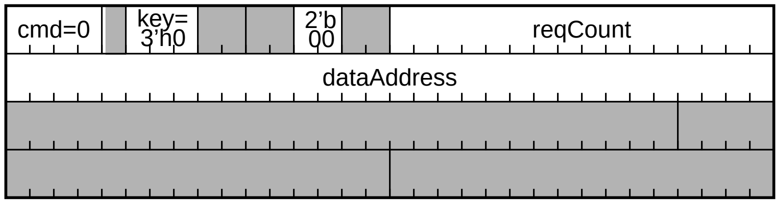


Figure 7-1 — OUTPUT_MORE descriptor format

Table 7-1 — OUTPUT_MORE descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h0 for OUTPUT_MORE.
key	3	Set to 3'h0 for OUTPUT_MORE.
b	2	Branch control. Software must set this field to 2'b00. Values of 2'b11, 2'b10, 2'b01 will result in unspecified behavior.
reqCount	16	Request Count: The number of transmit packet bytes starting at dataAddress.
dataAddress	32	Address of transmit data. dataAddress has no alignment restrictions.

7.1.2 OUTPUT_MORE_Immediate descriptor

The OUTPUT_MORE-Immediate command descriptor is used to specify up to four quadlets of packet header information to be inserted into the appropriate transmit FIFO. It has the following format.

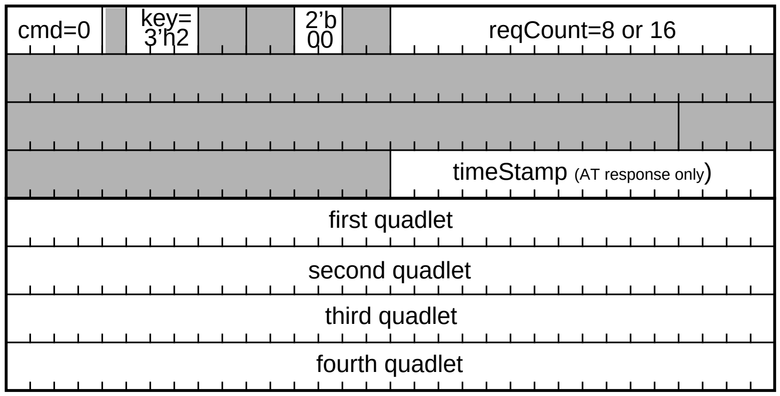


Figure 7-2 — OUTPUT_MORE-Immediate descriptor format

Table 7-2 — OUTPUT_MORE-Immediate descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h0 for OUTPUT_MORE-Immediate
key	3	Set to 3'h2 for OUTPUT_MORE-Immediate.
b	2	Branch control. Software must set this field to 2'b00. Values of 2'b11, 2'b10, 2'b01 will result in unspecified behavior.
reqCount	16	Request Count: The number of transmit packet bytes immediately following the 16th byte of this descriptor. This value must be either 8(two quadlets) or 16(four quadlets). Specifying any other value will result in unspecified behavior. Regardless of the reqCount value, this descriptor is always 32 bytes long.
timeStamp	16	Valid only in the AT response context. This field contains the three low order bits of cycleSeconds and all 13 bits of cycleCount. See section 5.12, "Isochronous Cycle Timer Register" for information about these fields. For AT response packets, timeStamp indicates a time after which the packet should not be transmitted. For further information on the use of this field, see section 7.1.5.3 below.
first, second, third, and fourth quadlets	128	Packet header quadlets to be inserted into the applicable FIFO.

The OUTPUT_MORE-Immediate command shall only be used either to specify the four quadlet 1394 transmit packet header for a block payload or lock packet, or to specify the two quadlet 1394 transmit packet header for an asynchronous stream packet. All OUTPUT_MORE-Immediate command descriptors are 32-bytes in length and are counted as two 16-byte aligned blocks when calculating the Z value.

7.1.3 OUTPUT_LAST descriptor

The OUTPUT_LAST command descriptor is used to specify a host memory buffer from which the AT DMA controller will insert bytes into the appropriate transmit FIFO. This command indicates the end of a packet to the Host Controller. It has the following format.

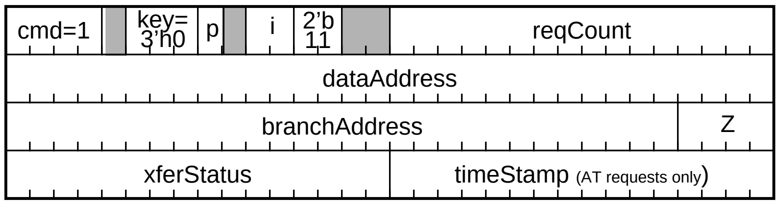


Figure 7-3 — OUTPUT_LAST descriptor format

Table 7-3 — OUTPUT_LAST descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h1 for OUTPUT_LAST.
key	3	Set to 3'h0 for OUTPUT_LAST.
p	1	Ping Timing. A 1 indicates that this is a ping packet. A ping packet is used to discern the round-trip time of transmitting a packet to another node. The timeStamp value written into this descriptor for a ping packet shall be the time from when the last bit of the packet is transmitted from the link to the PHY until either data is received or a subaction gap occurs. For more information on ping timing, see section 7.1.5.3.2. A 0 indicates that this is not a ping packet.
i	2	Interrupt control. Options: 2'b11 - Always interrupt upon command completion. 2'b01 - Interrupt only if did not receive an ack_complete or ack_pending. See table 3-2 for a list of possible ack_ and evt_ values. 2'b00 - Never interrupt. Specifying a value of 2'b10 will result in unspecified behavior.
b	2	Branch control. Software must set this field to 2'b11. Values of 2'b10, 2'b01, and 2'b00 will result in unspecified behavior.
reqCount	16	Request Count: The number of transmit packet bytes described by this descriptor, beginning at dataAddress.
dataAddress	32	Address of transferred data. dataAddress has no alignment restrictions.
branchAddress	28	16-byte aligned address of the next descriptor. A valid host memory address must be provided in this field unless the Z field is 0.
Z	4	This field indicates the number of 16-byte command blocks that comprise the next packet. If this is the last descriptor in the list, the Z value must be 0. Otherwise, valid values are 2 to 8. Note that each *-Immediate command descriptor is counted as two 16-byte blocks and each non-immediate command is counted as one 16-byte block.

Table 7-3 — OUTPUT_LAST descriptor element summary

Element	Bits	Description
xferStatus	16	Written with ContextControl [15:0] after descriptor is processed.
timeStamp	16	For AT request packets that are not ping packets, this field is written by hardware to indicate the transmission time of the packet. This transmission timestamp contains the three low order bits of cycleSeconds and all 13 bits of cycleCount. See section 5.12, “Isochronous Cycle Timer Register” for information about those two fields. For AT request packets that are ping packets, this field is written by hardware to indicate the measured ping duration in units of 49.152 MHz clocks. See section 7.1.5.3.2 for information about this duration value. For AT response packets, timeStamp is not valid (response descriptor blocks use a timestamp in the *-Immediate descriptor). For further information on the use of the timeStamp field, see section 7.1.5.3.

7.1.4 OUTPUT_LAST_Immediate descriptor

The OUTPUT_LAST-Immediate command descriptor is used to specify two to four quadlets of packet header information to be inserted into the appropriate transmit FIFO. This command indicates the end of a packet to the Host Controller. It has the following format.

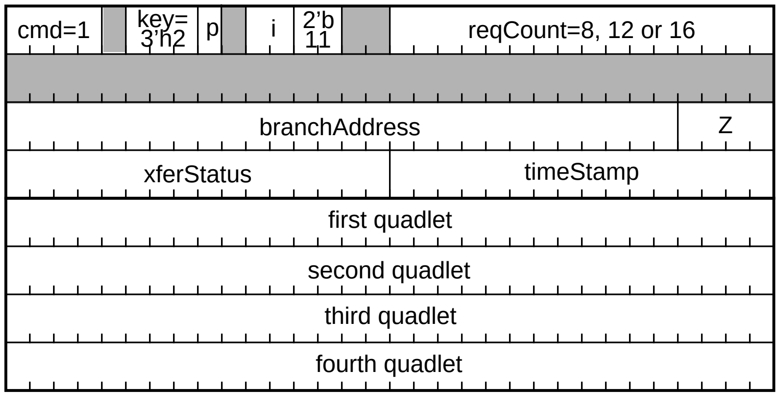


Figure 7-4 — OUTPUT_LAST-Immediate descriptor format

Table 7-4 — OUTPUT_LAST-Immediate descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h1 for OUTPUT_LAST-Immediate.
key	3	Set to 3'h2 for OUTPUT_LAST-Immediate.
p	1	Ping Timing. A 1 indicates that this is a ping packet. A ping packet is used to discern the round-trip time of transmitting a packet to another node. The timeStamp value written into this descriptor for a ping packet shall be the time from when the last bit of the packet is transmitted from the link to the PHY until either data is received or a subaction gap occurs. For more information on ping timing, see section 7.1.5.3.2. A 0 indicates that this is not a ping packet.
i	2	Interrupt control. Options: 2'b11 - Always interrupt upon command completion. 2'b01 - Interrupt only if did not receive an ack_complete or ack_pending. See table 3-2 for a list of possible ack and evt values. 2'b00 - Never interrupt. Specifying a value of 2'b10 will result in unspecified behavior.
b	2	Branch control. Software must set this field to 2'b11. Values of 2'b10, 2'b01, and 2'b00 will result in unspecified behavior.
reqCount	16	Request Count: The number of transmit packet bytes immediately following the 16th byte of this descriptor. Valid values are 8(two quadlets), 12(three quadlets) and 16(four quadlets). Specifying any other values will result in unspecified behavior. Regardless of the reqCount value, this descriptor is always 32 bytes long.
branchAddress	28	16-byte aligned address of the next descriptor. A valid host memory address must be provided in this field unless the Z field is 0.
Z	4	This field indicates the number of 16-byte command blocks that comprise the next packet. If this is the last descriptor in the list, the Z value must be 0. Otherwise, valid values are 2 to 8. Note that each *-Immediate command descriptor is counted as two 16-byte blocks and each non-immediate command is counted as one 16-byte block.
xferStatus	16	Written with ContextControl [15:0] after descriptor is processed.

Table 7-4 — OUTPUT_LAST-Immediate descriptor element summary

Element	Bits	Description
timeStamp	16	For AT request packets that are not ping packets, this field is written by hardware to indicate the transmission time of the packet. This transmission timestamp contains the three low order bits of cycleSeconds and all 13 bits of cycleCount. See section 5.12, “Isochronous Cycle Timer Register” for information about those two fields. For AT request packets that are ping packets, this field is written by hardware to indicate the measured ping duration in units of 49.152 MHz clocks. See section 7.1.5.3.2 for information about this duration value. For AT response packets, this field is written by software to indicate a time after which the packet should not be transmitted. This time is expressed in the same cycleSeconds/cycleCount format as for request packets that are not ping packets. For further information on the use of the timeStamp field, see section 7.1.5.3.
first, second, third, and fourth quadlets	128	Data quadlets to be inserted into the applicable FIFO.

The OUTPUT_LAST-Immediate command will be used to specify information that is protected by the header CRC or for sending a PHY packet. OUTPUT_LAST-Immediate command descriptors are 32-bytes in length regardless of the value of reqCount and are counted as two 16-byte aligned blocks when calculating the Z value.

7.1.5 AT DMA descriptor usage

Fields in the command descriptor are further described below.

7.1.5.1 Command.Z

The Z value is used by the Host Controller to enable several descriptors to be fetched at once, for improved efficiency. Z values must always be encoded correctly. The contiguous descriptors described by a Z value are called a *descriptor block*. The following table summarizes all legal Z values for the Asynchronous Transmit contexts:

Table 7-5 — Z value encoding

Z value	Use
0	Indicates that the current descriptor is the last descriptor in the context program.
1	reserved. (Since all descriptor blocks must start with a *-Immediate command, they are by definition a minimum of two 16-byte blocks in size.)
2-8	Indicates that two to eight 16-byte aligned blocks starting at branchAddress are physically contiguous and specify a single packet. Note that the 32-byte *-Immediate command descriptors must be counted as two 16-byte blocks when calculating the Z value.
9-15	reserved

A single packet that is to be transmitted must be entirely described by one descriptor block. This requirement permits the Host Controller to prefetch all the descriptors for a packet, in order to avoid fetching additional descriptors during a packet transfer. The branch address+Z allows the Host Controller to learn the Z value of the next block. Only the OUTPUT_LAST* descriptor shall specify a branch address+Z for the next packet. BranchAddress+Z values are ignored in all OUTPUT_MORE* descriptors, and should not be specified.

All DMA context programs must use a Z = 0 command to indicate the end of the program. A program which ends in Z=0 can be appended to while the DMA runs, even if the DMA has already reached the end. The mechanism for doing this is described in section 3.2.1.2.

7.1.5.2 Command.xferStatus

Upon the transmission completion of a packet, the 16 least significant bits of the current contents of the DMA Context-Control register are written to the completed packet's OUTPUT_LAST* descriptor's Command.*xferStatus* field. See section 7.2.2 for the contents of this field.

7.1.5.3 Command.timeStamp

The timeStamp field is encoded as follows:

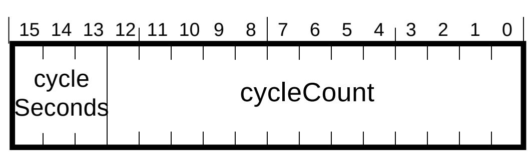


Figure 7-5 — timeStamp format

Table 7-6 — timeStamp description

Field	Bits	Description
cycleSeconds	3	Low order three bits of the seven-bit isochronous cycle timer second count. Possible values are 0 to 7.
cycleCount	13	Full 13 bits of the 13-bit isochronous cycle timer cycle count. Possible values are 0 to 7999.

7.1.5.3.1 timeStamp value for Requests

An asynchronous transmit request packet may initiate a transaction which should complete by a specific time. To permit host software to know when such a transaction began (i.e., when the request was successfully transmitted on the 1394 bus) the Host Controller shall write the timeStamp value in each OUTPUT_LAST* descriptor when the corresponding ack is received. If no ack is received, timeStamp will be written when the ack timeout occurs. TimeStamp shall be written in the same host bus operation in which xferStatus is written.

Note that a transmit request packet may sit in the transmit FIFO for some time before the PHY wins normal arbitration. This delay is usually brief, but could be over 200 cycles (over 25 milliseconds) in the case of a bus with 80% isochronous traffic and 63 nodes each sending maximum-size asynchronous packets as often as possible.

7.1.5.3.2 timeStamp value for Ping Requests

Pinging is used to discern the round-trip time of transmitting a packet to another node. In IEEE 1394-1995 this is done by transmitting a packet to a node and timing how long it takes to receive the corresponding ack. In P1394a, this is done by transmitting a Ping packet to a node and timing how long it takes to receive that node's self-ID packet as a response.

To achieve pinging with OpenHCI, software sets the *p* bit in the packet's OUTPUT_LAST* command descriptor to indicate it is a ping packet. The Host Controller shall transmit the packet and track the timing based on the number of 49.152MHz clocks, and shall place the final result in the descriptor's timeStamp field.

The Ping timer begins counting from zero immediately after the last bit of each transmitted packet is delivered from the link to the PHY. (For controllers that implement the P1394a standardized PHY/Link interface, the timer would start with the first HOLD or IDLE driven by the link after each transmitted packet.) The Ping timer stops counting at the earliest of either data reception or an indication of a subaction gap. (For controllers that implement the P1394a standardized PHY/Link interface, the timer stops with the first of either a RECEIVE indication from the PHY, or a STATUS transfer indicating a subaction gap.)

Aside from the difference in meaning of the timeStamp field when an OUTPUT_LAST has the *p* bit enabled, all other behaviors of the AT Request DMA context remain unchanged for the packet. For example, if an ack_busy* is returned by the destination node, the AT Request DMA shall perform its normal retry behavior. Each retried transfer shall repeat the ping timing, with the last attempt reported to the AT Request DMA command descriptor.

7.1.5.3.3 timeStamp value for Responses

Typically, asynchronous transmit response packets expire at a certain time and should not be transmitted after that time. A timeStamp value can be placed in the first OUTPUT_* descriptor for such packets to indicate the expiration time.

The timeStamp used for asynchronous transmit contains a 3-bit seconds field and a 13-bit cycle number which counts modulo 8000. Before an asynchronous response is put into the transmit FIFO, whether for the initial transmission attempt or for a retry attempt, this timeStamp value is compared to the current cycleTimer. This comparison is used to determine whether or not the packet will be sent or rejected as being too old.

The comparison is broken into two parts. The first compare is done on the seconds field of the timeStamp and the low order three bits of the seconds field in the cycleTimer. The low three bits of cycleTimer.*cycleSeconds* is subtracted from the timeStamp.*cycleSeconds* field using three bit arithmetic. If the most significant bit of the subtraction is 1, then the timeStamp is considered ‘late’ and the packet is rejected. If the most significant bit is 0 but the other two bits are not 0, then the timeStamp is considered to be for some time in the ‘distant’ future and the packet can be sent. If the difference is 0, then the timeStamp and cycleTimer are referring to the same second so the cycle number portion of the timeStamp is compared to the cycle number portion of the cycleTimer to determine if the cycle is early, late or matches. This comparison is done by subtracting the cycleTimer cycle number from the timeStamp cycle number. If the result is negative, then the time for the packet has passed and the packet is rejected. If the difference is positive and the timeout value is positive or zero, then the packet can be sent. This subtraction is signed so a sign bit is assumed to be prepended to both cycle number values.

Table 7-7 — Results of timeStamp.cycleSeconds - cycleTimer.cycleSeconds

	cycleTimer.seconds							
timeStamp.seconds	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	000	111	110	101	100	011	010	001
001	001	000	111	110	101	100	011	010
010	010	001	000	111	110	101	100	011
011	011	010	001	000	111	110	101	100
100	100	011	010	001	000	111	110	101
101	101	100	011	010	001	000	111	110
110	110	101	100	011	010	001	000	111
111	111	110	101	100	011	010	001	000

NOTE: Shaded entries denote ‘late’ values.

For those entries in the table above which are 000, the cycleTimer.*cycleCount* field is subtracted from the timeStamp.*cycleCount* field. If the result is positive or 0, it indicates that the packet can be sent. If the result is negative the packet cannot be sent and the status error code is set to evt_timeout.

Table 7-8 — timeStamp.cycleCount-cycleTime.cycleCount Example 1

timeStamp.cycleCount	cycleTime.cycleCount	difference	action
14’h0FA0	14’h0F9E	14’h0002	send packet
14’h0FA0	14’h0F9F	14’h0001	send packet
14’h0FA0	14’h0FA0	14’h0000	send packet
14’h0FA0	14’h0FA1	14’h3FFF	reject packet

Table 7-9 — timeStamp.cycleCount-cycleTime.cycleCount Example 2

timeStamp.cycleCount	cycleTime.cycleCount	difference	action
14’h1000	14’h0FFE	14’h0002	send packet
14’h1000	14’h0FFF	14’h0001	send packet
14’h1000	14’h1000	14’h0000	send packet
14’h1000	14’h1001	14’h3FFF	reject packet

Table 7-10 — timeStamp.cycleCount-cycleTime.cycleCount Example 3

timeStamp.cycleCount	cycleTime.cycleCount	difference	action
14’h0000	14’h0000	14’h0000	send packet
14’h0000	14’h0001	14’h3FFF	reject packet
...	...	...	...
14’h0000	14’h1000	14’h3000	reject packet
14’h0000	14’h1001	14’h2FFF	reject packet
...	...	...	...
14’h0000	14’h1F3E	14’h20C2	reject packet
14’h0000	14’h1F3F	14’h20C1	reject packet

After a transmit packet has passed the timeStamp check, it may sit in the transmit FIFO for some time before the PHY wins normal arbitration. The Host Controller does not re-examine the timeStamp while the packet waits, even if the descriptor is still active because only part of the packet fits into the FIFO. This delay is usually brief, but could be over 200 cycles (over 25 milliseconds) in the case of a bus with 80% isochronous traffic and 63 nodes each sending maximum-size asynch packets as often as possible.

Software can compute the worst-case FIFO delay based on knowledge of the current node count and the current (or maximum) isochronous load. Software can use this delay to compute an earlier expiration timeStamp to prevent late transmission due to FIFO delay. Using the maximum (not current) isochronous load is advisable, because additional isochronous reservations could be made while the packet is waiting in the transmit FIFO.

Because the Host Controller examines the timeStamp before the packet is loaded into the transmit FIFO, and because the packet may remain in the FIFO for some period until the PHY attached to the Host Controller wins normal arbitration, it is not possible to guarantee that the packet will not be transmitted after it expires. The maximum time the packet waits in the FIFO can be computed by software based on dynamic bus parameters, and this time can be factored into the packet’s expiration timeStamp. (Note, this could be over 200 cycles, in unlikely case where 80% of the bus is isochronous, and 63 nodes are each sending maximum-size asynch packets.)

7.2 AT DMA context registers

Each AT DMA context (request and response) has two registers: CommandPtr and ContextControl. CommandPtr is used by software to tell the Host Controller where the DMA context program begins. ContextControl is used by software to control the context's behavior, and is used by hardware to indicate current status.

7.2.1 CommandPtr

The CommandPtr register specifies the address of the context program which will be executed when a DMA context is started. All descriptors are 16-byte aligned, so the four least-significant bits of any descriptor address must be zero. The four least-significant bits of the CommandPtr register are used to encode a Z value that indicates how many physically contiguous 16-byte blocks of command descriptors are pointed to by descriptorAddress.

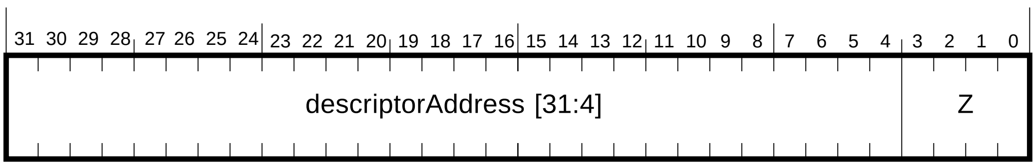


Figure 7-6 — CommandPtr register format

Refer to Section 3.1.2 for a complete description of the CommandPtr register.

7.2.2 ContextControl register (set and clear)

The *ContextControlSet* and *ContextControlClear* registers contain bits that control options, operational state and status for a DMA context. Software can set selected bits by writing ones to the corresponding bits in the *ContextControlSet* register. Software can clear selected bits by writing ones to the corresponding bits in the *ContextControlClear* register. It is not possible for software to set some bits and clear others in an atomic operation. A read from either register will return the same value.

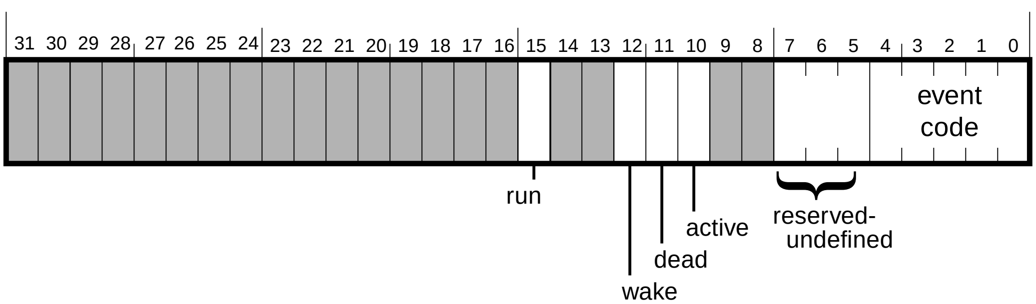


Figure 7-7 — ContextControl (set and clear) register format

Table 7-11 — ContextControl (set and clear) register description

Field	rscu	Description
run	rscu	Refer to section 3.1.1.1 for an explanation of the ContextControl.*run* bit.
wake	rsu	Refer to section 3.1.1.2 for an explanation of the ContextControl.*wake* bit.
dead	ru	Refer to section 3.1.1.4 for an explanation of the ContextControl.*dead* bit.
active	ru	Refer to section 3.1.1.3 for an explanation of the ContextControl.*active* bit.

Table 7-11 — ContextControl (set and clear) register description

Field	rscu	Description
reserved undefined	ru	This field is specified as undefined and may contain any value without impacting the intended processing of this packet.
event code	ru	Following an OUTPUT_LAST* command, the received ack_ code or an “evt_” error code is indicated in this field. Possible values are: ack_complete, ack_pending, ack_busy_X, ack_busy_A, ack_busy_B, ack_data_error, ack_type_error, evt_tcode_err, evt_missing_ack, evt_underrun, evt_descriptor_read, evt_data_read,evt_timeout, evt_flushed and evt_unknown. See Table 3-2, “Packet event codes,” for descriptions and values for these codes.

7.2.2.1 Writing status back to context command descriptors

Upon OUTPUT_LAST* completion, bits 15-0 of the ContextControl register are written to the OUTPUT_LAST* command’s *xferStatus* field. When Command.*xferStatus* is written to memory, the active bit is always one. If software prepared the descriptor’s xferStatus.*active* bit to be zero, this change indicates that the descriptor has been executed, and the xferStatus and timeStamp fields have been updated.

7.2.3 Bus Reset

7.2.3.1 Host Controller Behavior for AT

Upon detection of a bus reset, the Host Controller will cease transmission of asynchronous transmit packets. When this occurs there are two possibilities for AT packets that are left in the FIFO.

- Case 1 is when a bus reset occurs after the packet was transmitted but before an ack was received. For this category, the link side of the Host Controller will return evt_missing_ack.
- Case 2 is when a bus reset occurs after the packet is placed in the FIFO but before it is transmitted. For this category, the link side of the Host Controller may return evt_flushed.

When each context becomes stable (all data transfers have been halted and status writes have been completed), the Host Controller will clear the corresponding ContextControl.*active* bit.

7.2.3.2 Software Guidelines

When a bus reset occurs, the link side will flush the asynchronous transmit FIFO(s) until the IntEvent.*busReset* condition is cleared. Software must make sure however that IntEvent.*busReset* is not cleared until 1) software has cleared the ContextControl.*run* bits for both Asynchronous Transmit contexts, and 2) both Asynchronous Transmit contexts have quiesced and both ContextControl.*active* fields are zero. This is to ensure that all queued asynchronous packets (with potentially stale node numbers) are flushed. Once the contexts are no longer active, software may clear the busReset interrupt condition, and hardware will stop flushing the asynchronous transmit FIFO(s). Before setting ContextControl.*run* for either context following a bus reset, software must ensure that NodeID.*iDValid* is set and that NodeID.*nodeNumber* (section 5.10) does not equal 63.

7.3 Fairness

Packets transmitted via the AT Request queue shall abide by the fairness protocol as supported by the Host Controller (see section 5.8, “FairnessControl register (optional)”). AT response packets shall be transmitted according to the rules for response packets as specified in P1394a.

7.4 AT Retries

The Host Controller will retry busied asynchronous transmit request and response packets based on the configuration of the AT Retries register. For a detailed description of the ATRetries register see section 5.4.

Hardware implementations that support dual-phase retry must ignore the retry code provided by software and must insert a retry code as appropriate with the current state of the retry protocol (retry_1, retry_A or retry_B).

7.5 AT Interrupts

Each asynchronous DMA context has one interrupt indication bit in the intEvent register (section 6.1). For requests, it is the *reqTxComplete* bit and for responses it is the *respTxComplete* bit. This interrupt indication bit will be set to one if a completed OUTPUT_LAST* command has the *i* field set to 2'b11, or if the *i* field is set to 2'b01 and transmission of the packet did not yield an ack_complete or an ack_pending.

7.6 AT Data Formats

There are five basic formats for asynchronous data to be transmitted:

a) no-data packets (used for quadlet read requests and all write responses)
b) quadlet packets (used for quadlet write requests, quadlet read responses and block read requests)
c) block packets (used for lock requests and responses, block write requests and block read responses)
d) PHY packets
e) asynchronous stream packets (tcode 4'hA packets sent during asynchronous period)

All formats are shown below in three sections, asynchronous requests, asynchronous responses, and asynchronous streams.

Note that packets to go out over the 1394 wire are constructed from these Host Controller internal formats, and are not sent in the exact order as shown in the formats below. For example, destinationID is transmitted in the first quadlet, and source ID is automatically provided and transmitted in the second quadlet.

7.6.1 Asynchronous Transmit Requests

7.6.1.1 No-data transmit

The no-data request transmit format is shown below. The first quadlet contains packet control information. The second and third quadlets contain 16-bit destination ID and the 48-bit quadlet-aligned destination offset. Note that this packet requires only three quadlets. Therefore when transmitted via an OUTPUT_LAST-Immediate descriptor, the descriptor's fourth quadlet is unused.

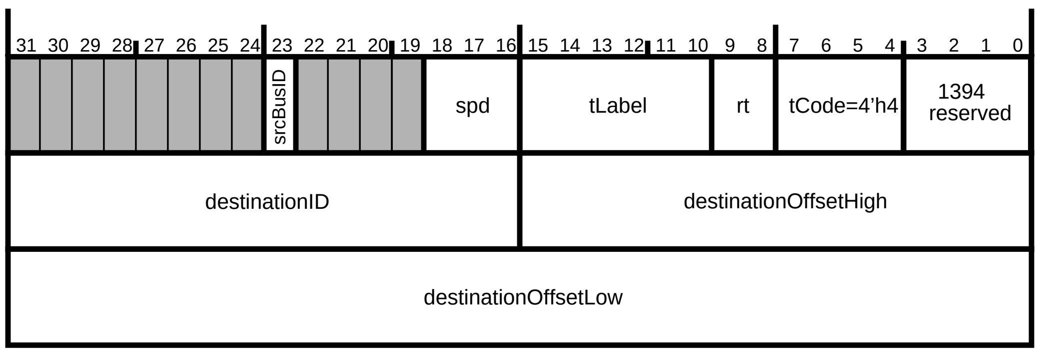


Figure 7-8 — Quadlet read request transmit format

Table 7-12 — Quadlet read request transmit fields

field name	bits	description
srcBusID	1	Source bus ID selector. If clear, the high order 10 bits of the source_ID field of the transmitted packet will be 10'h3FF. If set, the high order 10 bits of the source_ID field of the transmitted packet will be Node_ID.*busNumber* (see section 5.10).
spd	3	This field indicates the speed at which this packet is to be transmitted. 3'b000 = 100 Mbits/sec, 3'b001 = 200 Mbits/sec, and 3'b010 = 400 Mbits/sec, other values are reserved.
tLabel	6	This field is the transaction label, which is used to pair up a response packet with its corresponding request packet.
rt	2	The retry code for this packet. Software should set rt to retry_X (2'b01). Hardware may elect to ignore the software provided retry code and substitute an rt as appropriate for the implemented retry mechanism. I.e., hardware implementing single phase retry can use either the software provided rt or provide the equivalent 2'b01 constant, and hardware implementing dual phase retry should provide the proper retry_1, retry_A or retry_B code upon transmission.
tCode	4	The transaction code for this packet.
1394 reserved		Required by IEEE 1394-1995 to be all zeros. OpenHCI will pass these bits along as-is and will not verify or modify them.
destinationID	16	This is the concatenation of the 10-bit bus number and the 6-bit node number for the destination of this packet.
destinationOffsetHigh, destinationOffsetLow	16 32	The concatenation of these two fields addresses a quadlet in the destination node's address space. This address must be quadlet-aligned (modulo 4).

7.6.1.2 Quadlet transmit

The quadlet request transmit formats are shown below. The first quadlet contains packet control information. The second and third quadlets contain 16-bit destination ID and the 48-bit, quadlet-aligned destination offset. For write quadlet requests the fourth quadlet is the quadlet data.

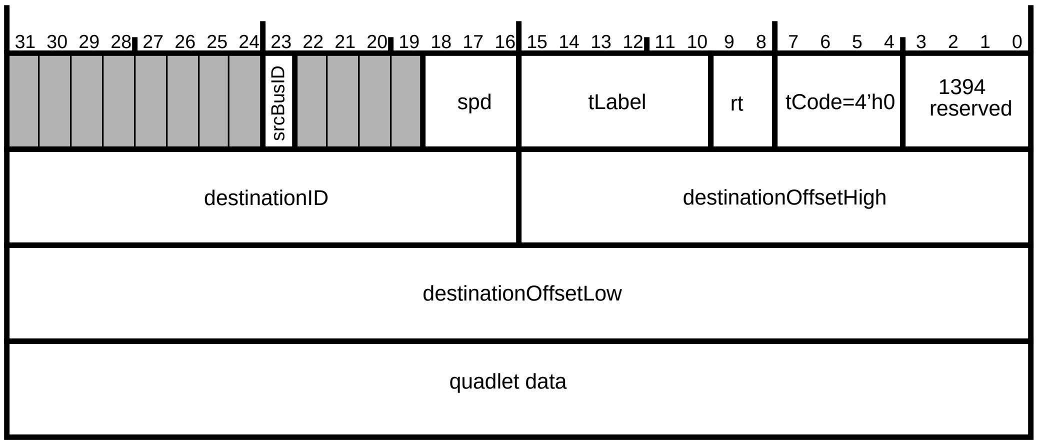


Figure 7-9 — Quadlet write request transmit format

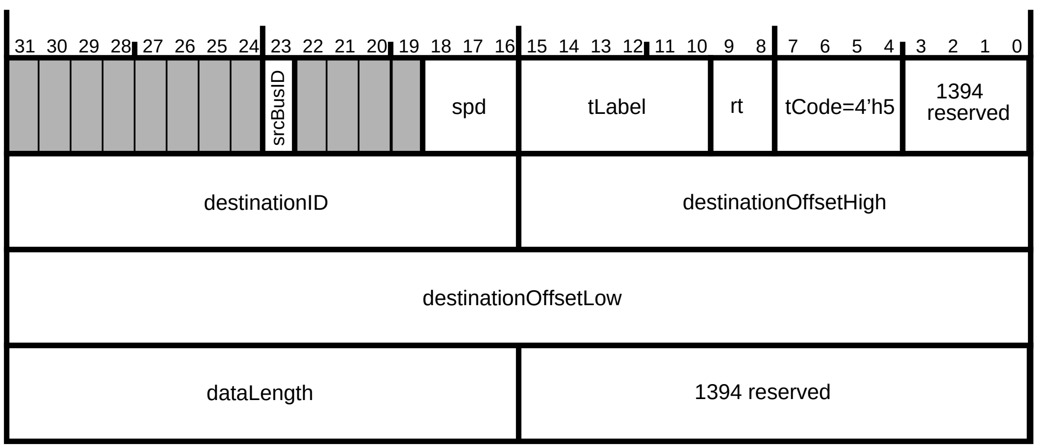


Figure 7-10 — Block read request transmit format

Table 7-13 — Quadlet transmit fields

field name	bits	description
srcBusID, spd, tLabel, rt, tCode, 1394 reserved, destinationID, destinationOffsetHigh, destinationOffsetLow		See Table 7-12.
quadlet data	32	For quadlet write requests this field holds the data to be transferred.
dataLength	16	The number of bytes requested in a block read request.

7.6.1.3 Block transmit

The block request transmit formats are shown below. The first quadlet contains packet control information. The second and third quadlets contain the 16-bit destination node ID and the 48-bit destination offset. The fourth quadlet contains the length of the data field and the extended transaction code (all zeros except for lock transactions). The block data, if any, follows the extended code.

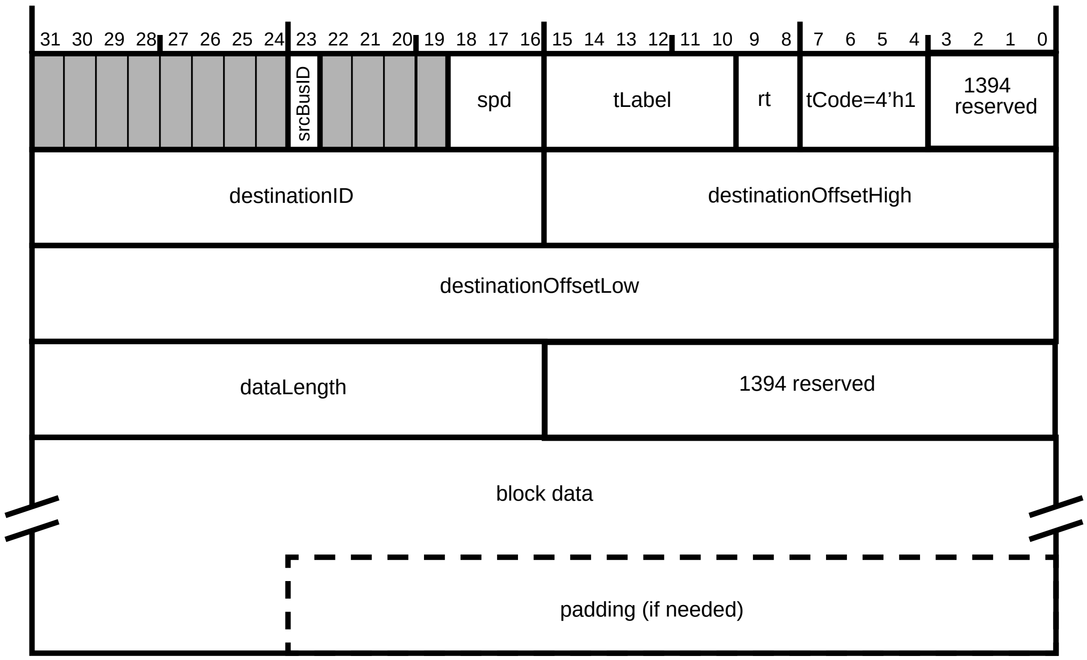


Figure 7-11 — Write request transmit format

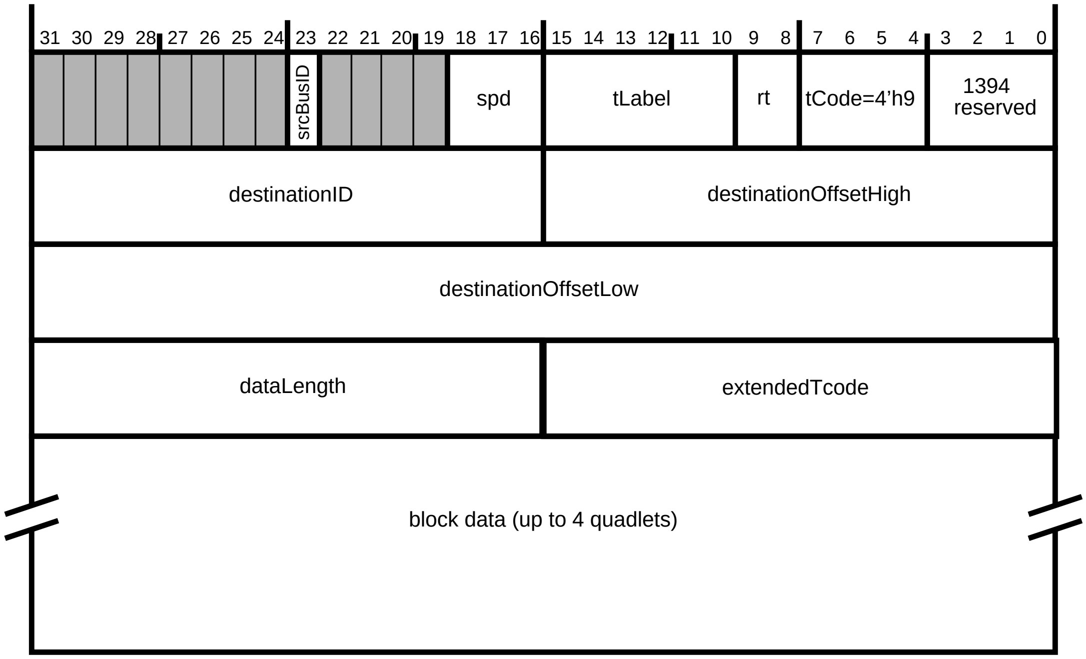


Figure 7-12 — Lock request transmit format

Table 7-14 — Block transmit fields

field name	bits	description
srcBusID, spd, tLabel, rt, tCode, 1394 reserved, destinationID, destinationOffsetHigh, destinationOffsetLow		See Table 7-12.
dataLength	16	The number of bytes of data to be transmitted in this packet.
extendedTcode	16	If the tCode indicates a lock transaction, this specifies the actual lock action to be performed with the data in this packet.
block data		The data to be sent. If dataLength==0, no data should be written into the FIFO for this field. Regardless of the destination or source alignment of the data, the first byte of the block must appear in the leftmost byte of the first quadlet.
padding		If the dataLength mod 4 is not zero, then zero-value bytes are added onto the end of the packet to guarantee that a whole number of quadlets is sent.

7.6.1.4 PHY packet transmit

The PHY packet transmit format is shown below. The first quadlet contains packet control information. Software should set spd to S100 (3'b000) for compliance with 1394-1995 and P1394a. The remaining two quadlets contain data that is transmitted without any formatting on the bus. No CRC is appended to the packet, nor is any data in the first quadlet sent. This packet is used to send a PHY configuration, Link-on, and P1394a Ping packets.

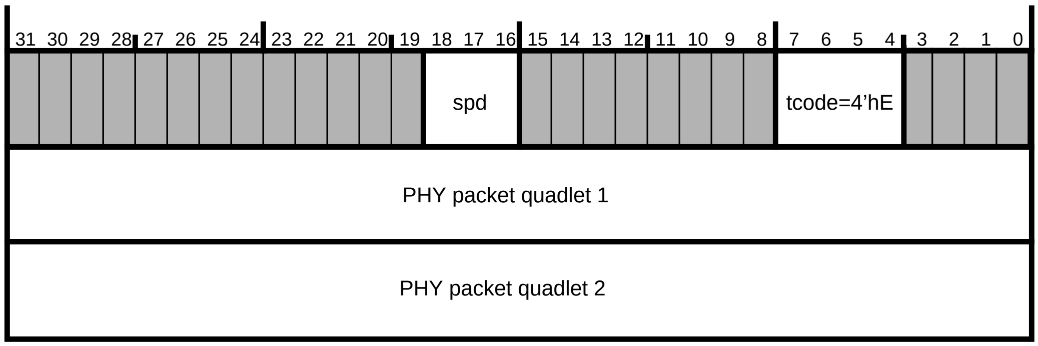


Figure 7-13 — PHY packet transmit format

7.6.2 Asynchronous Transmit Responses

7.6.2.1 No-data transmit

The no-data transmit format is shown below. The first quadlet contains packet control information. The second and third quadlets contain 16-bit destination ID and the response code. Note that this packet requires only three quadlets. Therefore when transmitted via an OUTPUT_LAST-Immediate descriptor, the descriptor's fourth quadlet is unused.

write response transmit format

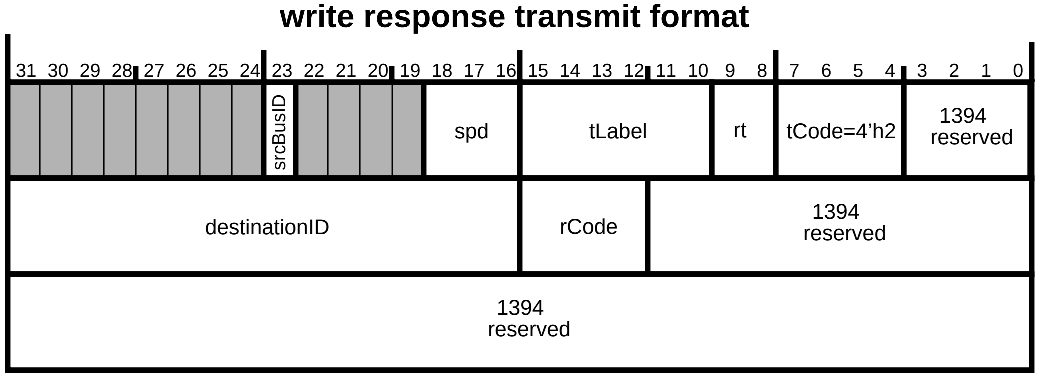


Figure 7-14 — Write response transmit format

Table 7-15 — Write response transmit fields

field name	bits	description
srcBusID	1	Source bus ID selector. If clear, the high order 10 bits of the source_ID field of the transmitted packet will be 10'h3FF. If set, the high order 10 bits of the source_ID field of the transmitted packet will be Node_ID.*busNumber* (see section 5.10).
spd	3	This field indicates the speed at which this packet is to be transmitted. 3'b000 = 100 Mbits/sec, 3'b001 = 200 Mbits/sec, and 3'b010 = 400 Mbits/sec, other values are reserved.
tLabel	6	This field is the transaction label, which is used to pair up a response packet with its corresponding request packet.
rt	2	The retry code for this packet. Software should set rt to retry_X (2'b01). Hardware may elect to ignore the software provided retry code and substitute an rt as appropriate for the implemented retry mechanism. I.e., hardware implementing single phase retry can use either the software provided rt or provide the equivalent 2'b01 constant, and hardware implementing dual phase retry should provide the proper retry_1, retry_A or retry_B code upon transmission.
tCode	4	The transaction code for this packet.
1394 reserved		Required by IEEE 1394-1995 to be all zeros. OpenHCI will pass these bits along as-is and will not verify them or modify them.
destinationID	16	This is the concatenation of the 10-bit bus number and the 6-bit node number for the destination of this packet.
rCode	4	Response code for this response packet.

7.6.2.2 Quadlet transmit

The quadlet read response transmit format is shown below. The first quadlet contains packet control information. The second and third quadlets contain 16-bit destination ID and the 4-bit response code. The fourth quadlet is the quadlet data for read responses.

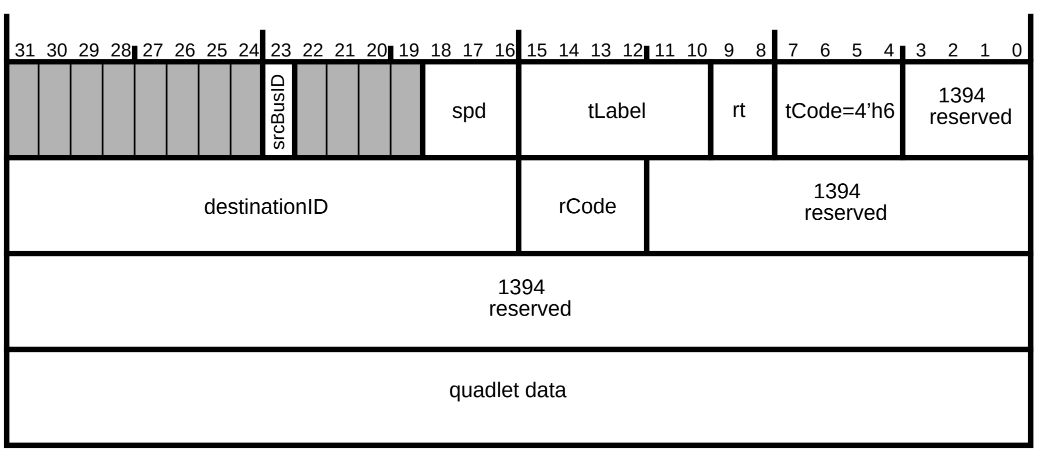


Figure 7-15 — Quadlet read response transmit format

Table 7-16 — Quadlet transmit fields

field name	bits	description
srcBusID, spd, tLabel, rt, tCode, 1394 reserved, destinationID, rCode		See Table 7-15.
quadlet data	32	For quadlet read responses, this field holds the data to be transferred.

7.6.2.3 Block transmit

The block response transmit formats are shown below. The first quadlet contains packet control information. The second and third quadlets contain the 16-bit destination node ID and the response code and reserved data. The fourth quadlet contains the length of the data field and the extended transaction code (all zeros except for lock transactions). The block data, if any, follows the extended code.

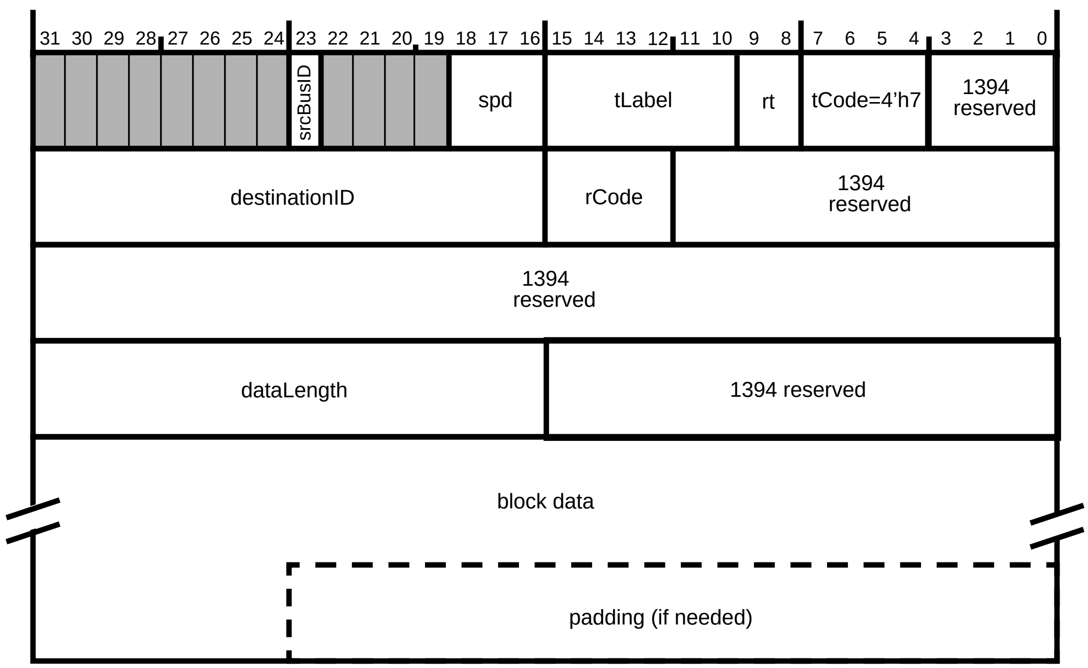


Figure 7-16 — Block read response transmit format

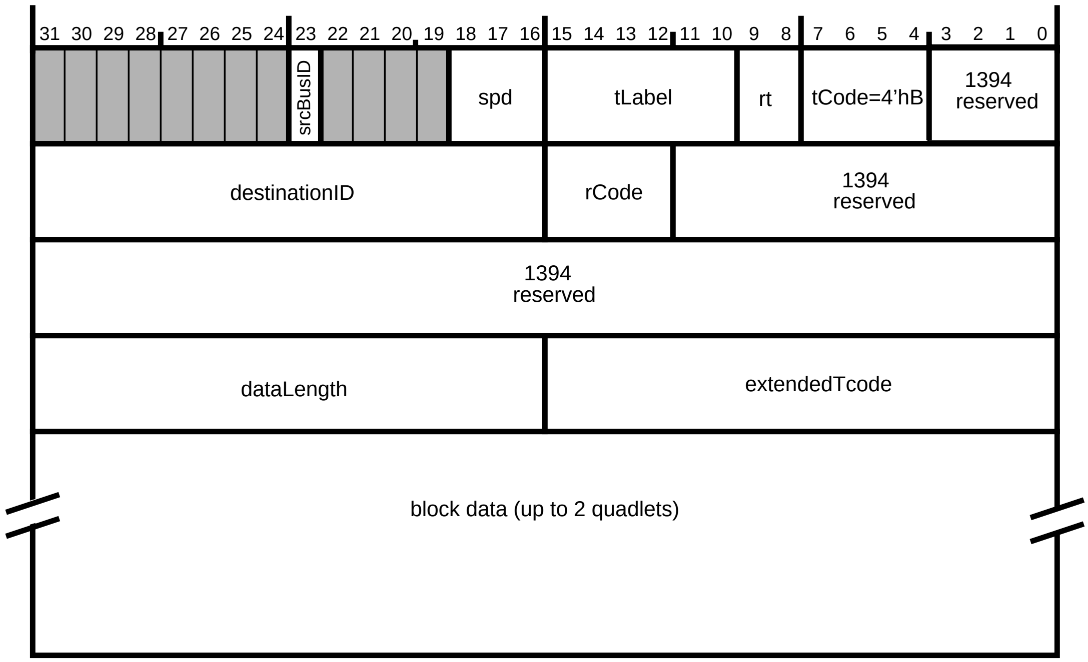


Figure 7-17 — Lock response transmit format

Table 7-17 — Block transmit fields

field name	bits	description
srcBusID, spd, tLabel, rt, tCode, 1394 reserved, destinationID, rCode		See Table 7-15.
dataLength	16	The number of bytes of data to be transmitted in this packet.
extendedTcode	16	If the tCode indicates a lock transaction, this specifies the actual lock action to be performed with the data in this packet.
block data		The data to be sent. Regardless of the destination or source alignment of the data, the first byte of the block must appear in the leftmost byte of the first quadlet.
padding		If the dataLength mod 4 is not zero, then zero-value bytes are added onto the end of the packet to guarantee that a whole number of quadlets is sent.

7.6.3 Asynchronous Transmit Streams

An asynchronous stream packet is a packet in the format of an isochronous packet (e.g., using tcode = 4'hA) that is transmitted during the asynchronous period. It is transmitted via the Asynchronous Transmit Request context and as such, it is governed by the same fairness rules as other asynchronous packets. This packet format consists of two header quadlets (as specified in either the OUTPUT_MORE-Immediate or OUTPUT_LAST-Immediate descriptor) and an optional data payload. The data payload in host memory is not required be aligned on a quadlet boundary. Padding is added by the Host Controller if needed. The format is as follows.

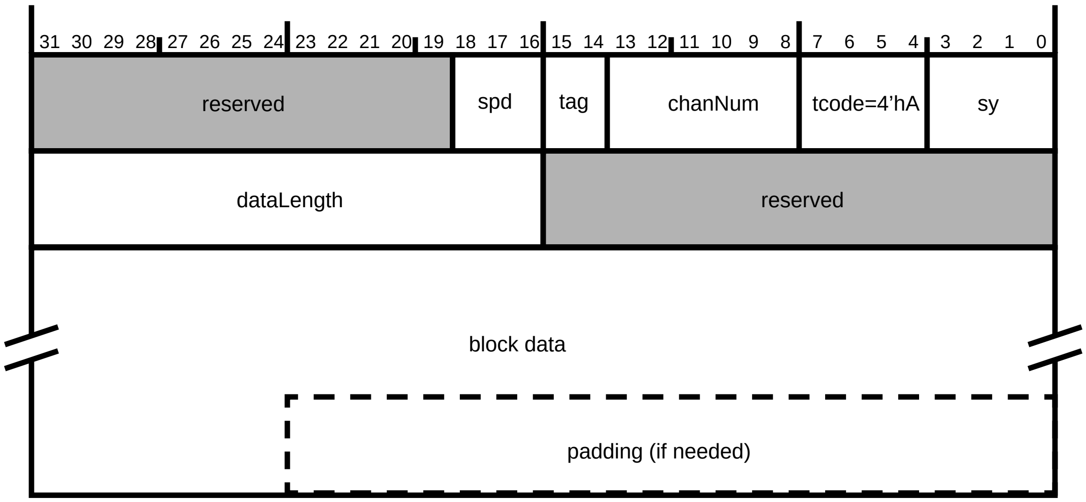


Figure 7-18 — Asynchronous stream packet format

Table 7-18 — Asynchronous stream packet fields

field name	bits	description
spd	3	This field indicates the speed at which this packet is to be transmitted. 3'b000 = 100 Mbits/sec, 3'b001 = 200 Mbits/sec, and 3'b010 = 400 Mbits/sec, other values are reserved.
tag	2	The data format of the isochronous data (see IEEE 1394 specification)
chanNum	6	The channel number this data is associated with.
tcode	4	The transaction code for this packet.
sy	4	Transaction layer specific synchronization bits.
dataLength	16	Indicates the number of bytes in this packet.
block data		The data to be sent with this packet. The first byte of data must appear in the leftmost byte of the first quadlet. The last quadlet should be padded with zeroes, if necessary.
padding		If the dataLength mod 4 is not zero, then zero-value bytes are added onto the end of the packet to guarantee that a whole number of quadlets is sent.

Note that packets to go out over the 1394 wire are constructed from this Host Controller internal format, and are not sent in the exact order as shown above. For example, spd, shown in the first quadlet, is not transmitted at all as part of the asynchronous stream packet header.

8. Asynchronous Receive DMA

The Asynchronous Receive DMA controller performs the function of accepting packets for which there is no explicit destination. This includes all packets which are accepted by the link module, but are not handled by any other receive DMA function. However this does not include cycle start packets. There are two asynchronous receive (AR) contexts, an AR Request context and an AR Response context. Each context uses a DMA context program to move such packets into memory to be interpreted by the host processor software.

Since the collection of packets that must be handled by the AR contexts may be of widely varying lengths, each context operates in *buffer-fill* mode in which multiple packets may be concatenated into the supplied buffers. Software is responsible for parsing through these buffers and taking the appropriate action required for a packet, and hardware is required to make these buffers parsable.

This chapter describes the AR context program components, how the AR contexts are managed and how the Asynchronous Receive controller operates. For information regarding receive FIFO implementation, refer to Section 3.3.

8.1 AR DMA Context Programs

The Asynchronous Receive DMA controller consists of two contexts for handling all asynchronous packets not handled by the physical DMA controller. A context program is a list of DMA descriptors used to identify buffers in host memory into which the Host Controller places received asynchronous packets.

The DMA descriptors are 16-bytes in length and must be aligned on a 16-byte boundary. There is one type of command descriptor used in an AR context program: INPUT_MORE.

8.1.1 INPUT_MORE descriptor

The INPUT_MORE command descriptor is used to specify a host memory buffer into which the AR controller will place the received asynchronous packets from the Host Controller receive FIFO. It has the following format.

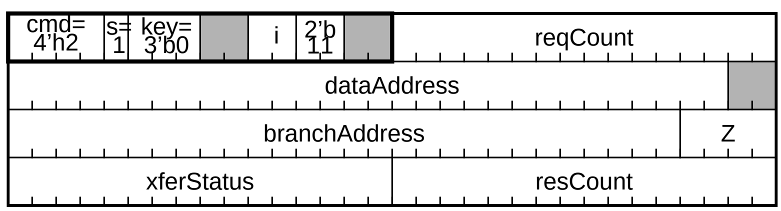


Figure 8-1 — INPUT_MORE descriptor format

Table 8-1 — INPUT_MORE descriptor element summary

Element	Bits	Description
cmd	4	Software must set this field in all AR command descriptors to 4’h2 for INPUT_MORE, and hardware may assume that all AR descriptors are INPUT_MORE commands. This indicates to the AR controller that this descriptor contains a buffer address for storing received asynchronous packets.
s	1	Status control. Software must set this field to 1. Hardware always writes status regardless of the setting of this bit.
key	3	This field must be set to 3’b0.

Table 8-1 — INPUT_MORE descriptor element summary

Element	Bits	Description
i	2	Interrupt control. Valid values are 2'b11 to generate an IntEvent.*ARRQ* or IntEvent.*ARRS* interrupt when the descriptor is completed (see section 6.1), or 2'b00 for no interrupt. Behavior is unspecified if set to 2'b01 or 2'b10. Note that in addition to the per-descriptor (buffer) interrupts, interrupts can also be generated on a per-packet basis for each complete packet received using the IntEvent.*RQPkt* and IntEvent.*RSPkt* interrupts described in section 6.1. These per-packet interrupts are not affected by the setting of the *i* bit in an INPUT_MORE descriptor.
b	2	Branch control. Software must set this field to 2'b11. Values of 2'b10, 2'b01, and 2'b00 will result in unspecified behavior.
reqCount	16	Request count: The size in bytes of the input buffer pointed to by dataAddress. ReqCount must be a multiple of 4 (representing a whole number of quadlets).
dataAddress	32	Host memory address of receive buffer. This address must be aligned on a quadlet boundary.
branchAddress	28	16-byte aligned address of the next descriptor. A valid address must be provided in this field unless the Z field is 0.
Z	4	Z may be set to 0 or 1. If this is the last descriptor in the context program, Z must be set to 0, otherwise it must be set to 1.
xferStatus	16	Written with ContextControl [15:0] whenever resCount is updated.
resCount	16	Residual count: while this descriptor is in-use by the Host Controller, resCount is updated each time a packet is written to the receive buffer to indicate the number of bytes (out of a max of reqCount) which have not been filled with received data. For further information on resCount see section 8.4.2, "AR DMA Controller processing."

Note that the Command.*resCount* and Command.*xferStatus* fields are updated in an indivisible operation.

8.1.2 AR DMA descriptor usage

An asynchronous receive context program consists of one or more INPUT_MORE command descriptors. Each descriptor, other than the final one, must provide a branchAddress with a Z value of 1 for the next block. The final command descriptor must have a Z value of 0 to indicate the end of the context program. Section 3.2.1.2 describes a safe method by which additional INPUT_MORE command descriptors may be appended to an active DMA program, regardless of whether or not the AR DMA has reached the final command descriptor.

Software may only modify a (non-completed) descriptor that may have been prefetched if a) the descriptor's current Z value is 0, and b) only the branchAddress and Z fields of the descriptor are modified.

8.2 bufferFill mode

Received asynchronous packets can be either solicited responses or unsolicited requests. Since software must be prepared to handle several packets of variable size, the Asynchronous Receive DMA contexts operate in bufferFill mode. In bufferFill mode, all received packets are concatenated into a contiguous stream of data. This data is then metered out into buffers described by a DMA context program, filling each buffer completely. As each packet is put into a buffer, the descriptor's resCount is updated to reflect the number of remaining bytes available in the buffer. Packets may straddle multiple buffers in this mode (see packet 2 in the illustration below). In addition to the overall concept of bufferFill mode, there are several nuances for Asynchronous receive which are described in detail in section 8.4.2.

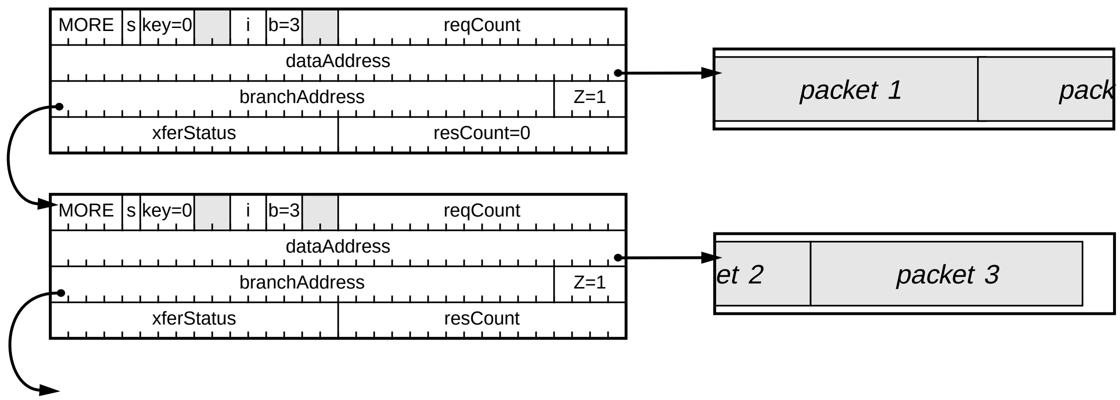


Figure 8-2 — bufferFill receive mode

8.3 Asynchronous Receive Context Registers

The AR request context and AR response context each have a CommandPtr register and a ContextControl register. CommandPtr is used by software to tell the Host Controller where the DMA context program begins. ContextControl is used by software to control the context's behavior, and is used by hardware to indicate current status.

8.3.1 AR DMA CommandPtr register

The CommandPtr register specifies the address of the context program which will be executed when a DMA context is started. All descriptors are 16-byte aligned, so the four least-significant bits of any descriptor address must be zero. The least-significant bit of the CommandPtr register is used to encode a Z value. For each AR context (Request and Receive) Z may be either 1 to indicate that descriptorAddress points to a valid command descriptor, or 0 to indicate that there are no descriptors in the context program.

Refer to section 3.1.2 for a full description of the CommandPtr register.

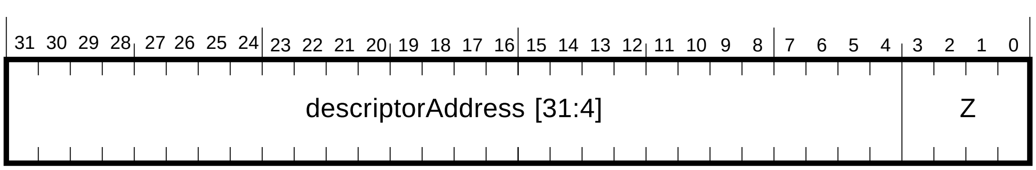


Figure 8-3 — CommandPtr register format

8.3.2 AR ContextControl register (set and clear)

The *ContextControlSet* and *ContextControlClear* registers contain bits that control options, operational state, and status for a DMA context. Software can set selected bits by writing ones to the corresponding bits in the *ContextControlSet* register. Software can clear selected bits by writing ones to the corresponding bits in the *ContextControlClear* register. It is not possible for software to set some bits and clear others in an atomic operation. A read from either register will return the same value and is referred to as the *ContextControlStatus* register.

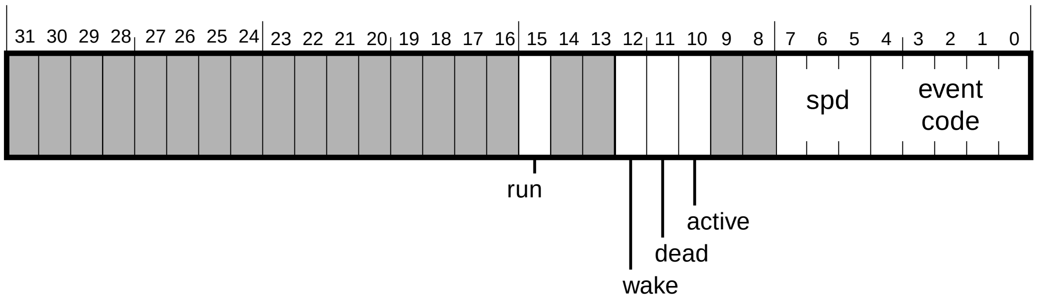


Figure 8-4 — AR ContextControl (set and clear) register format

Table 8-2 — AR ContextControl (set and clear) register description

Field	RSC	Description
run	rscu	Refer to section 3.1.1.1 for an explanation of the ContextControl.*run* bit.
wake	rsu	Refer to section 3.1.1.2 for an explanation of the ContextControl.*wake* bit.
dead	ru	Refer to section 3.1.1.4 for an explanation of the ContextControl.*dead* bit.
active	ru	Refer to section 3.1.1.3 for an explanation of the ContextControl.*active* bit.
spd	ru	This field indicates the speed at which the last packet was received by this context. 3’b000 = 100 Mbits/sec, 3’b001 = 200 Mbits/sec and 3’b010 = 400 Mbits/sec. All other values are reserved. Software should not attempt to interpret the contents of this field while the ContextControl.*active* or ContextControl.*wake* bits are set.
event code	ru	The packet ack_ code or an “evt_” error code is indicated in this field. Possible values are: ack_complete, ack_pending, ack_type_error, evt_descriptor_read, evt_data_write, evt_bus_reset, evt_unknown, and evt_no_status. See Table 3-2, “Packet event codes,” for descriptions and values for these codes.

8.4 AR DMA Controller

8.4.1 Asynchronous Filter Registers

Software can control from which nodes it will receive *request* packets by utilizing the asynchronous filter registers. There are two registers, one for filtering out all requests from a specified set of nodes (AsynchronousRequestFilter register) and one for filtering out physical requests from a specified set of nodes (PhysicalRequestFilter register). The settings in both registers have a direct impact on how the AR Request context is used, e.g., disabling only physical receives from a node will cause all request packets from that node to be routed to the AR Request context buffer(s). The usage and interrelationship between these registers is fully described in section 5.13, “Asynchronous Request Filters.” Asynchronous *response* packets are never filtered.

8.4.2 AR DMA Controller processing

The AR DMA controller writes the entire packet, as described in the Asynchronous Receive Data Formats section, into memory for software to process. This includes the packet header and packet reception status. Data chaining across context commands is supported.

For the AR request context, command.*reqCount* should always be set to at least the maximum possible packet length for an asynchronous packet as specified in the max_rec field of the bus_info_block, plus five quadlets for the header and trailer (2^(max_rec+1) + 20 bytes). This means a single packet can cross at most one buffer boundary. This requirement also makes it easier for the Host Controller implementation to combine asynchronous receive FIFO's (see section 3.3).

When the host software transmits an asynchronous request, it must first ensure that there is enough buffer space allocated in the AR response context's context program to receive the response packet including headers and timestamp. Failure to preallocate this space may result in the hardware discarding responses that arrive when the AR response context is out of descriptors even though ack_complete may have been sent to the source node.

Since the AR request context and AR response context buffers must always be parseable by software there are three essential requirements.

a) The Host Controller must write a packet into a buffer(s) by first writing the asynchronous packet header, followed by the packet data, followed by a packet trailer.
b) Requests or responses with data-length errors, CRC errors, FIFO overrun errors or buffer overrun errors must not be presented to the software. Although the host memory buffers may have been written in anticipation of a good packet, the xferStatus and resCount will not be updated. This in effect “backs out” the packet.
c) After each packet is written into the buffer(s), hardware must update the resCount for the INPUT_MORE descriptor(s) for the buffer(s), to accurately reflect the number of unused bytes remaining.

Software must initialize resCount to the value of reqCount. Upon the first packet arrival into a buffer, the Host Controller must write the appropriate residual count, based on (resCount - (packetHeaderLen + dataLength + statusquadlet)). Note that neither the header CRC nor data CRC quadlets are inserted into the buffer.

As depicted in figure 8-2 on page 89, it is possible for a received packet to straddle multiple buffers. For the AR Request context, the buffer size requirements (mentioned above) ensure that a packet can only straddle two buffers. However, the AR Response context does not have a buffer size requirement and therefore AR response packets may straddle more than two buffers. To ensure that the receive buffers for a context remain parsable, hardware must follow the procedure shown below. (First buffer refers to the buffer receiving the first byte of the packet or packet header, and final buffer refers to the buffer receiving the last byte of the packet or packet trailer.)

1) After filling to the end of a buffer with a partial packet, advance to the next descriptor block and obtain the next buffer (dataAddress), retaining all state for the first buffer as well as for the new buffer.
2) Continue writing packet bytes into the new buffer. If the end of the buffer is reached, advance to the next buffer without updating xferStatus and without retaining state for it or any other interim buffers. Write the remaining packet bytes into the final buffer (for the packet).
3) If there is no error: 1) write the trailer quadlet into the final buffer, 2) update xferStatus and resCount into the **final** buffer's descriptor, and 3) update xferStatus and resCount into the first buffer's descriptor (where xferStatus is the current value of ContextControl[15:0]). At that point the first buffer's state is no longer needed.
4) If there *is* an error, then the packet must be ‘backed-out’ by reverting back to the previous state of the first buffer (as saved earlier). XferStatus and resCount are not updated for either descriptor.

By following these steps, the AR context buffers remain intact and can be parsed. Since interim buffers (those containing an inner portion of one packet) for the AR Response context will not have their status updated, software must only use resCount values when the corresponding xferStatus indicates the active bit is set to one. It follows from this that if the xferStatus.*active* bit is set in a descriptor, then all prior descriptors have been filled.

8.4.2.1 AR DMA Packet Trailer

The trailer quadlet written by the Host Controller at the end of each packet has the following format.

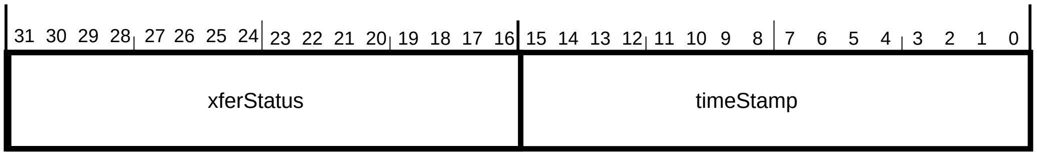


Figure 8-5 — AR DMA packet trailer format

Table 8-3 — AR DMA trailer fields

field name	bits	description
xferStatus	16	Written with ContextControl[15:0].
timeStamp	16	The low order 3 bits of cycleTimer.*cycleSeconds* and the full 13 bits of cycleTimer.*cycleCount* at some time during receipt of the packet.

8.4.2.2 Error Handling

Packets resulting in an ack_data_error will, in effect, not go into an AR DMA buffer. Since an ack_data_error condition is not known until all data (plus data CRC) has arrived, many "corrupted" data bytes may have been moved into an AR DMA buffer by the time the error situation is discovered. In this circumstance, hardware is required to halt its writing of the packet into the AR DMA buffer without updating the resCount field. By not advancing the residual count location, it will appear as though the packet never was written into the AR DMA buffer at all.

Similarly, if a bus reset occurs after a packet has been received but before the ack is sent, the packet may be "backed-out" of the buffer(s) as described for ack_data_error above.

8.4.2.3 Bus Reset Packet

To assist software in determining which asynchronous request packets arrived before and after a bus reset, necessary since node numbers may have changed, the Host Controller inserts a synthesized PHY packet into the AR DMA Request Context buffer (if active) as soon as a bus reset condition is detected. This packet has the following format.

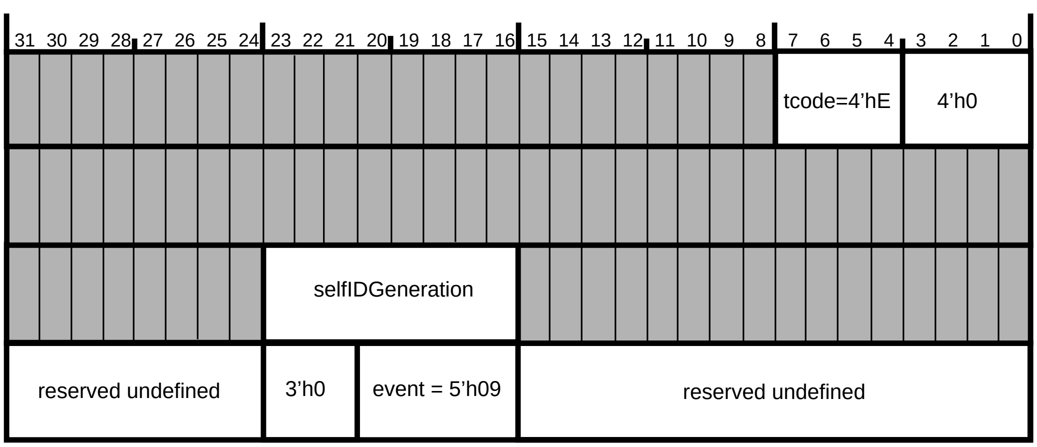


Figure 8-6 — AR Request Context Bus Reset packet format

Table 8-4 — AR Request Context Bus Reset packet description

Field	**bits**	a) Description
tcode	4	Set to 4’hE to indicate a PHY packet.
selfIDGeneration	8	The selfIDCount.*selfIDGeneration* value at the time this packet is created.
reserved undefined	8 + 16	This field is specified as undefined and may contain any value without impacting the intended processing of this packet.
eventCode	5	A value of 5’h09 (evt_bus_reset) identifies this as a synthesized bus_reset packet.

Software can distinguish the bus-reset packet from authentic PHY packets by the value of eventCode which is set to evt_bus_reset. Software can further interpret and coordinate received asynchronous packets across multiple bus resets by using the selfIDGeneration number provided in the bus-reset packet. Since the bus-reset packet is fabricated when a bus reset is initially detected, the selfIDGeneration number is for the new (not previous) generation and will be the same as the selfIDGeneration number in the SelfIDCount register as well as in the selfID buffer.

If more than one bus reset has occurred without any intervening packets, then only the “last” one is required to result in a synthesized bus-reset packet.

If the input FIFO is full when a bus reset occurs, the link side of the FIFO must later insert the bus-reset packet when space becomes available. If the AR DMA request context does not have enough buffer space for the bus-reset packet, the packet shall be synthesized once buffer space becomes available.

The bus reset interrupt (IntEvent.*busReset*) is independent of the time when this packet goes from the FIFO into a host buffer. This interrupt shall occur as soon as possible after a bus reset has been detected. The bus-reset packet is no different from any other packet going into the AR Request buffer in that IntEvent.*RQPkt* will be generated like it would for other packets.

8.5 PHY Packets

PHY packets will be received by asynchronous receive DMA if LinkControl.*rcvPhyPkt* is 1, and will be received by the AR Request context. PHY packets in the AR Request context will include the phy packet’s “logical inverse” quadlet which must be verified by software to be the logical inverse of the previous quadlet. The format of this packet is shown in section 8.7.1.4.

A packet is treated as a PHY packet if it is two quadlets and fails the CRC check. This includes any Self-ID packet that arrives outside of the Self-ID phase of bus initialization.

8.6 Asynchronous Receive Interrupts

There are two interrupts for each context (request and response) that software can use to gauge the usage of the receive buffers. If software needs to be informed of the arrival of each packet being sent to the context buffers, it can use the RQPkt or RSPkt interrupts in the IntEvent register (see section 6.1). If software needs to be informed of the completion of a buffer, it can set the context command.*i* field to 2’b11, which will trigger either the ARRQ or ARRS interrupt in the IntEvent register.

8.7 Asynchronous Receive Data Formats

The Host Controller shall only receive packets which have tcodes that are defined by an approved IEEE 1394 standard. Packets with undefined tcodes will be dropped.

There are four basic formats for asynchronous data to be received:

a) no-data packets (used for quadlet read requests and all write responses)
b) quadlet packets (used for quadlet write requests, quadlet read responses, and block read requests)
c) block packets (used for lock requests and responses, block write requests, and block read responses)
d) PHY packets

The names and descriptions of the fields in the received data are given in table 8-5.

Table 8-5 — Asynch receive fields

field name	bits	description
destinationID	16	This field is the concatenation of busNumber (or all ones for "local bus") and nodeNumber (or all ones for broadcast) for this node.
tLabel	6	This field is the transaction label, which is used to pair up a response packet with its corresponding request packet.
rt	2	The retry code for this packet. 00=retry1, 01=retryX, 10=retryA, 11=retryB
tCode	4	The transaction code for this packet.
1394 reserved		Required by IEEE 1394-1995 to be all zeros. OpenHCI will pass these bits along as received and will not verify or modify them.
sourceID	16	This is the node ID (bus number + node number) of the sender of this packet.
destinationOffsetHigh, destinationOffsetLow	16 32	The concatenation of these two fields addresses a quadlet in this node's address space. This address must be quadlet-aligned (modulo 4).
rCode	4	Response code for response packets.
quadlet data	32	For quadlet write requests and quadlet read responses, this field holds the data received.
dataLength	16	The number of bytes of data to be received in a block packet.
extendedTcode	16	If the tCode indicates a lock transaction, this specifies the actual lock action to be performed with the data in this packet.
block data		The data received. Regardless of the destination or source alignment of the data, the first byte of the block will appear in the leftmost byte of the first quadlet.
padding		If the dataLength mod 4 is not zero, then bytes have been added onto the end of the packet by the transmitting node to guarantee that a whole number of quadlets is received.
xferStatus	16	Written with ContextControl[15:0].
timeStamp	16	The low order 3 bits of cycleTimer.*cycleSeconds* and the full 13 bits of cycleTimer.*cycleCount* at some time during receipt of the packet.

8.7.1 Asynchronous Receive Requests

8.7.1.1 No-data receive

The no-data receive format is shown below. The first quadlet contains the destination node ID and the rest of the packet header. The second and third quadlets contain 16-bit source ID and the 48-bit, quadlet-aligned destination offset. The last quadlet contains packet reception status.

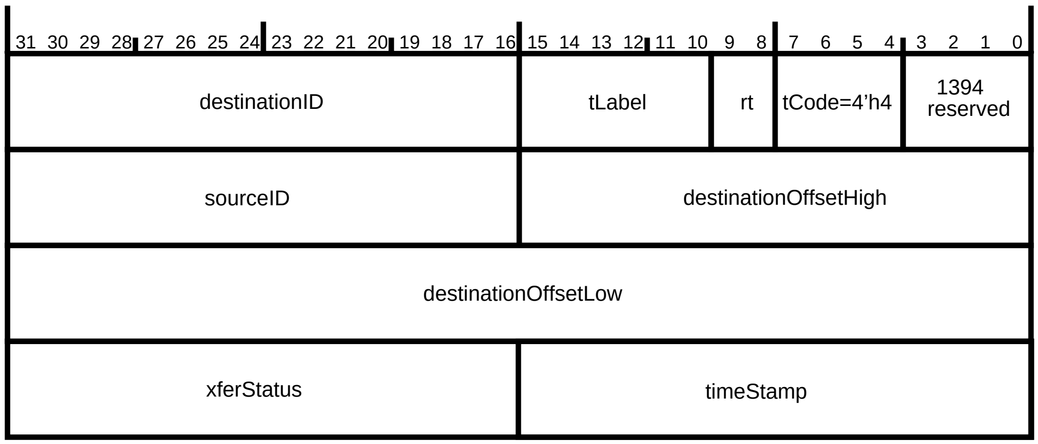


Figure 8-7 — Quadlet read request receive format

8.7.1.2 Quadlet Receive

The quadlet receive formats are shown below. The first quadlet contains the destination node ID and the rest of the packet header. The second and third quadlets contain 16-bit source ID and the 48-bit, quadlet-aligned destination offset. The fourth quadlet is the quadlet data for write quadlet requests, and is the data length and reserved for block read requests. The last quadlet contains packet reception status.

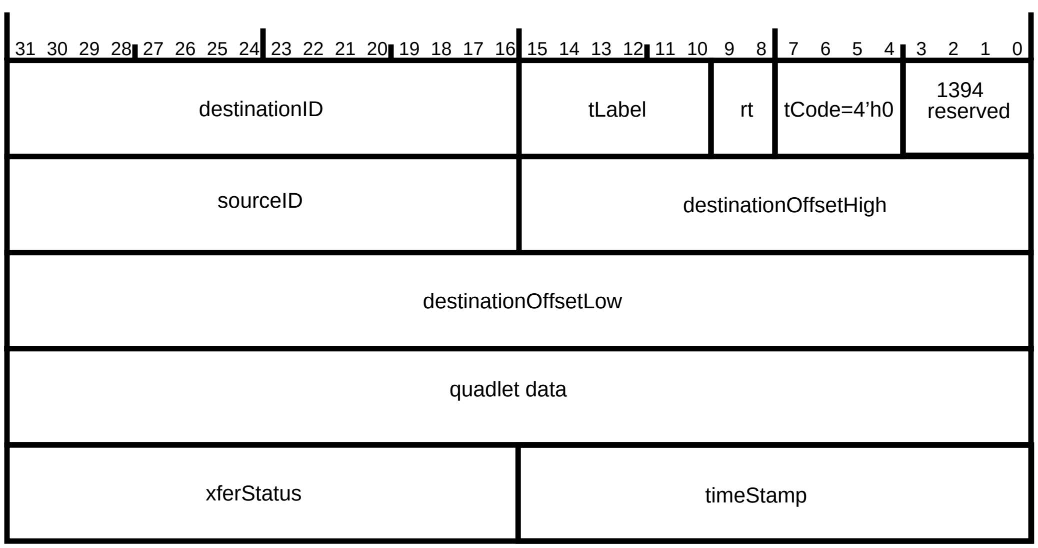


Figure 8-8 — Quadlet write request receive format

31–16	15–10	9–8	7–4	3–0
destinationID	tLabel	rt	tCode=4'h5	1394 reserved
sourceID	destinationOffsetHigh			
destinationOffsetLow				
dataLength	1394 reserved			
xferStatus	timeStamp			

Figure 8-9 — Block read request receive format

8.7.1.3 Block receive

The block receive formats are shown below. The first quadlet contains the destination node ID and the rest of the packet header. The second and third quadlets contain the 16-bit source ID and the 48-bit destination offset. The fourth quadlet contains the length of the data field and the extended transaction code (all zeros except for lock transactions). The block data, if any, follows the extended Tcode. The last quadlet contains packet reception status.

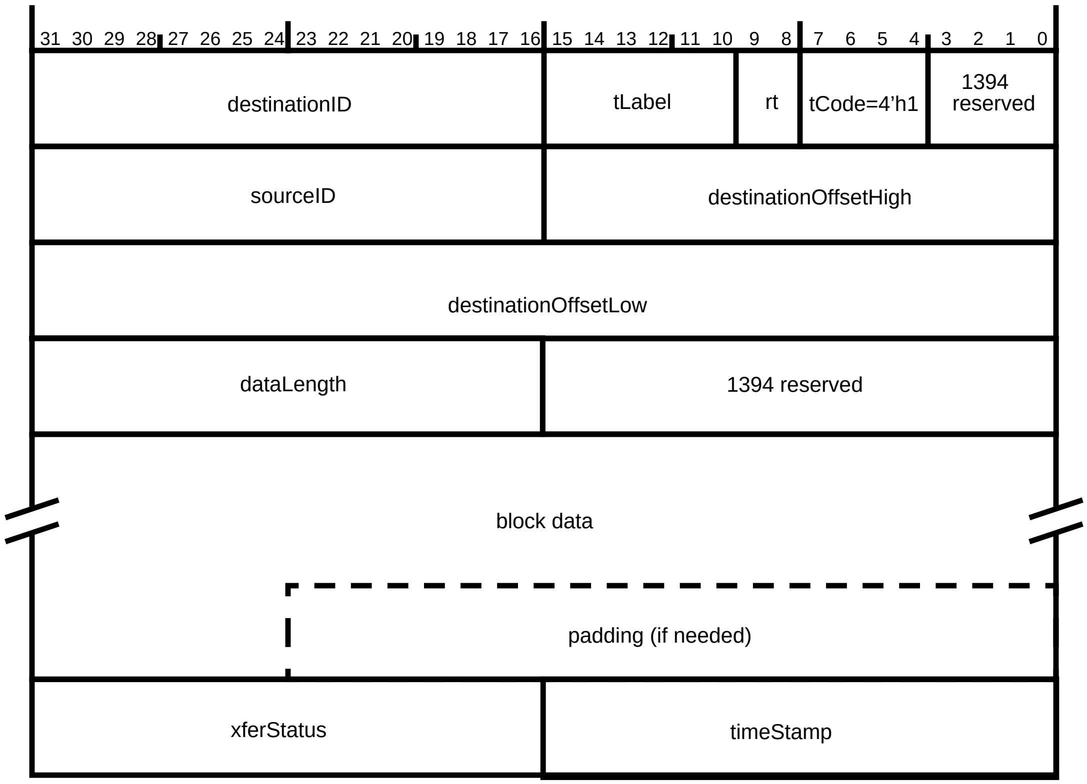


Figure 8-10 — Block write request receive format

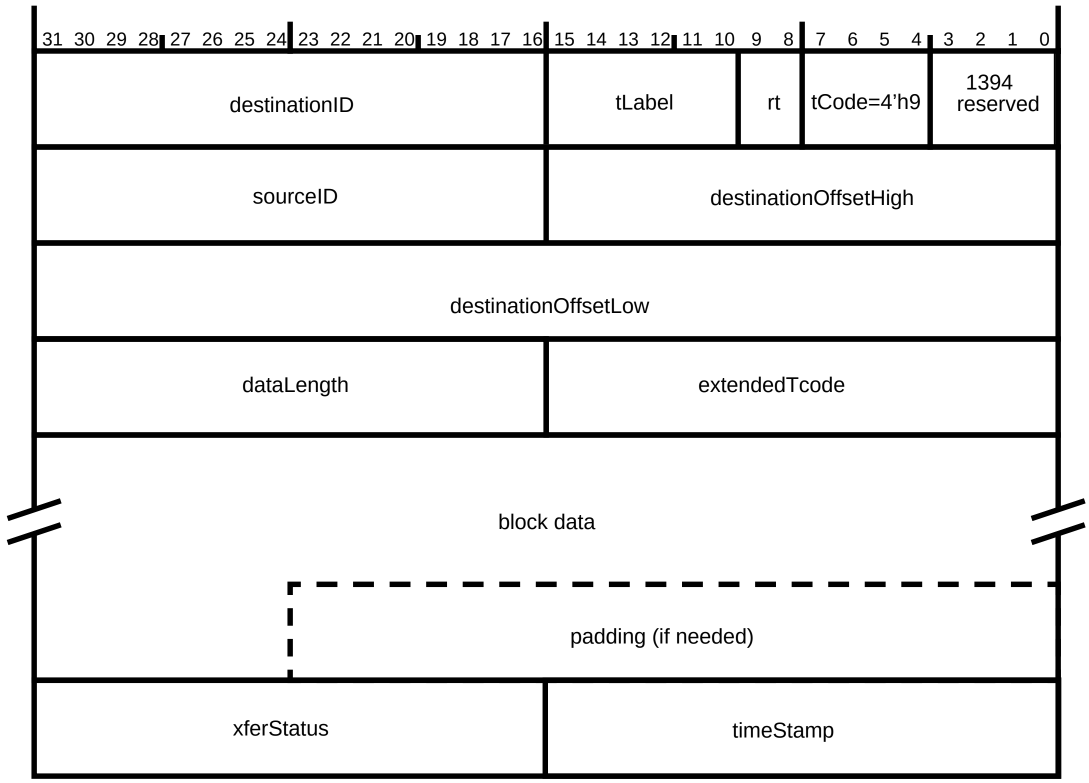


Figure 8-11 — Lock request receive format

8.7.1.4 PHY packet receive

The PHY packet receive format is shown below. The first quadlet contains a synthesized packet header with a tCode of 4'hE. The second quadlet contains the PHY quadlet and the third quadlet contains the inverse of the previous quadlet. Software is required to verify the integrity of the second quadlet by checking it against the third quadlet. The final (fourth) quadlet contains the packet trailer. The value of xferStatus.*event* shall be evt_no_status for PHY packets.

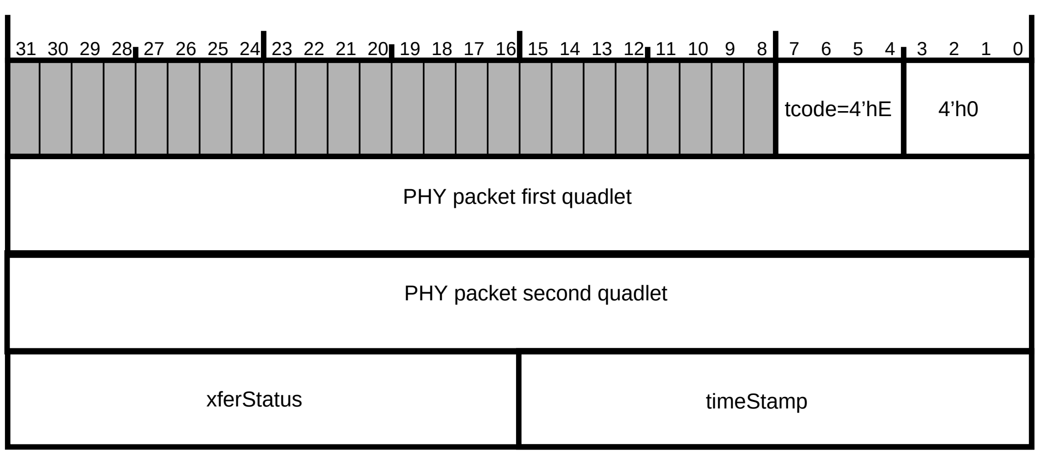


Figure 8-12 — PHY packet receive format

8.7.2 Asynchronous Receive Responses

8.7.2.1 No-data receive

The no-data receive format is shown below. The first quadlet contains the destination node ID and the rest of the packet header. The second and third quadlets contain 16-bit source ID and the response code. The last quadlet contains packet reception status.

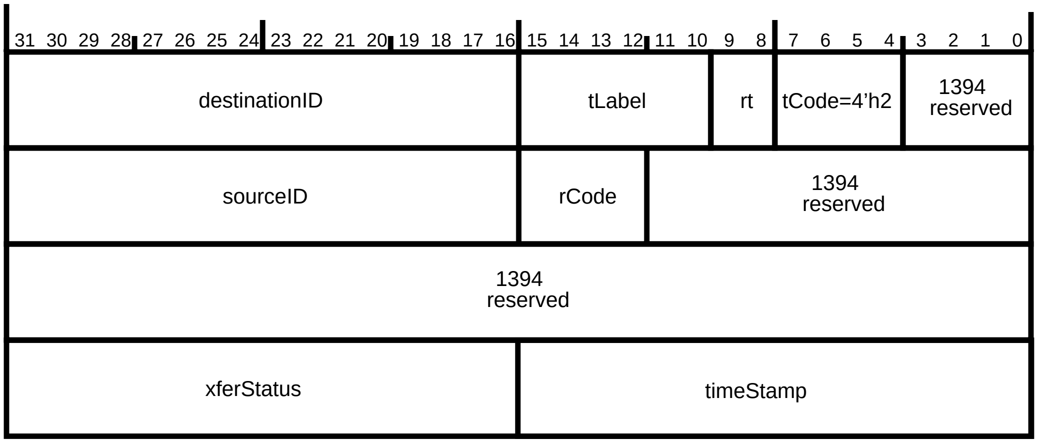


Figure 8-13 — Write response receive format

8.7.2.2 Quadlet Receive

The quadlet receive format is shown below. The first quadlet contains the destination node ID and the rest of the packet header. The second and third quadlets contain 16-bit source ID and the response code. The fourth quadlet is the quadlet data for read responses. The last quadlet contains packet reception status.

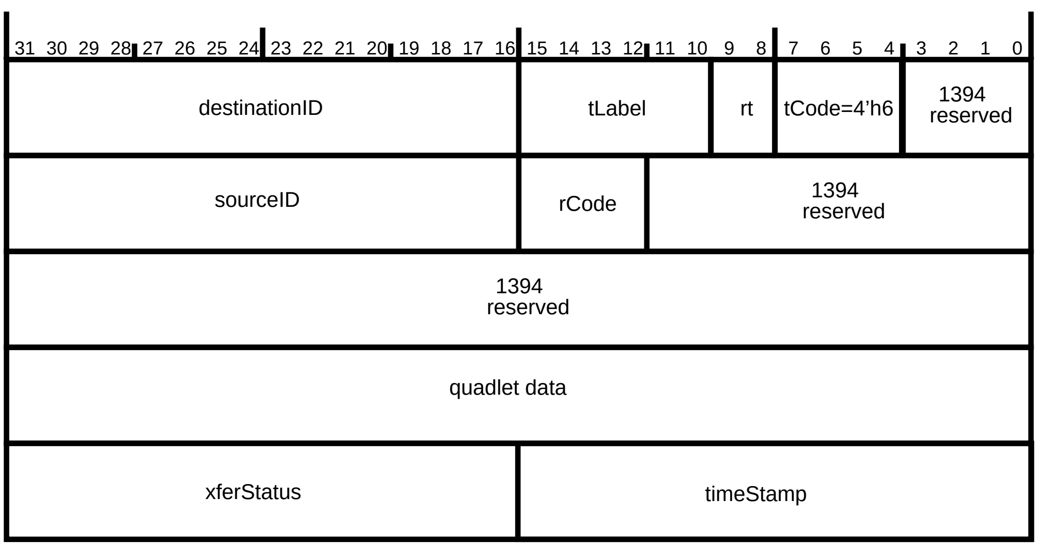


Figure 8-14 — Quadlet read response receive format

8.7.2.3 Block receive

The block receive formats are shown below. The first quadlet contains the destination node ID and the rest of the packet header. The second and third quadlets contain the 16-bit source ID and the response code and reserved data. The fourth quadlet contains the length of the data field and the extended transaction code (all zeros except for lock transactions). The block data, if any, follows the extended Tcode. The last quadlet contains packet reception status.

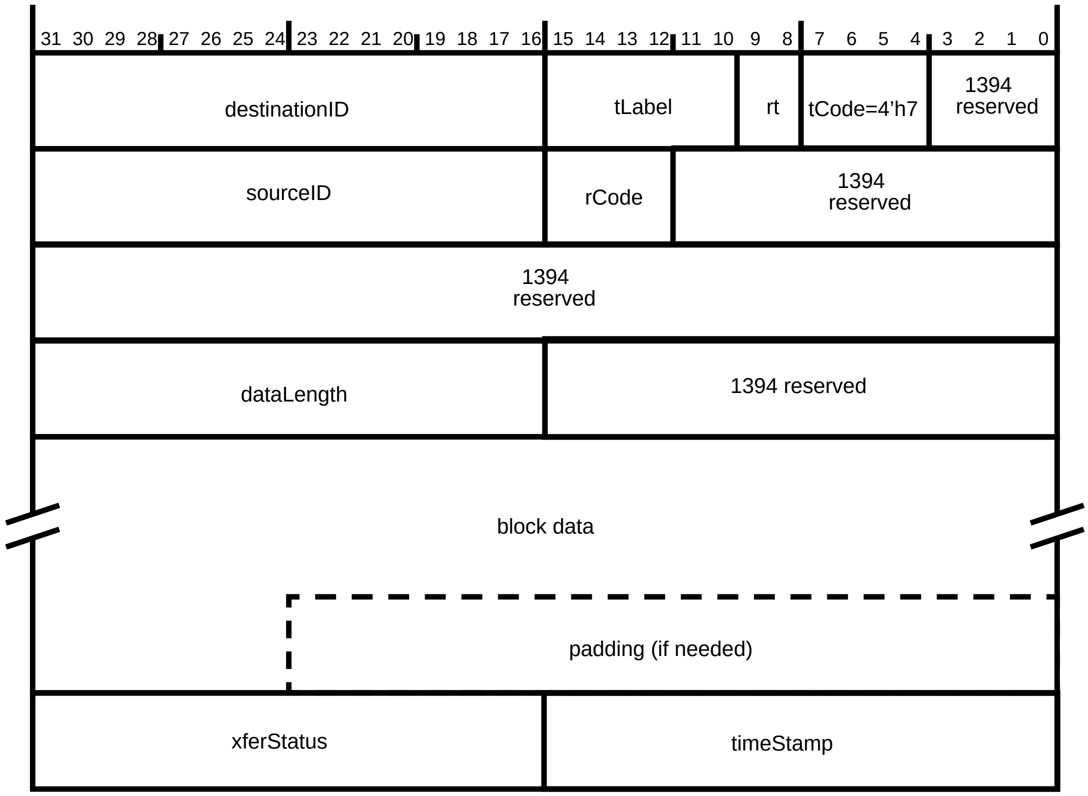


Figure 8-15 — Block read response receive format

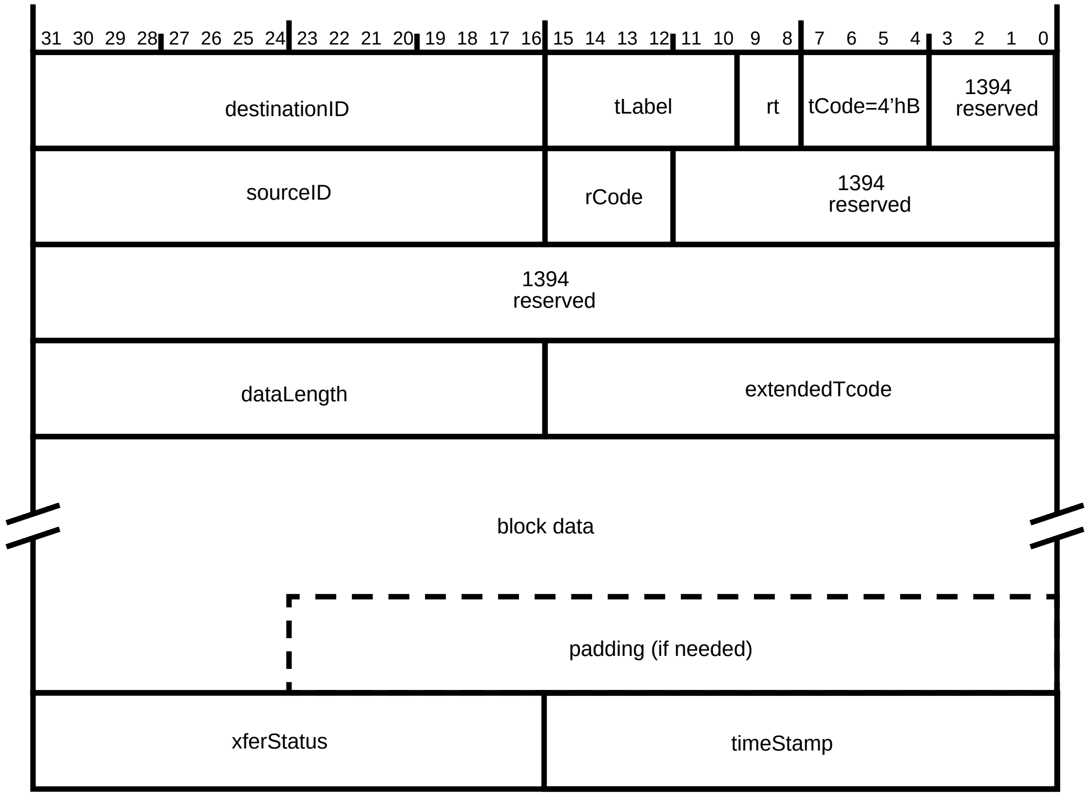


Figure 8-16 — Lock response receive format

9. Isochronous Transmit DMA

The Isochronous Transmit DMA (IT DMA) controller has a required minimum of four and an implementation maximum of 32 isochronous transmit contexts. Each context is controlled by a DMA context program. Each IT DMA context will transmit data for a single isochronous channel.

9.1 IT DMA Context Programs

For isochronous transmit DMA, a context program is a list of DMA command descriptors used to identify buffers in host memory from which the Host Controller transmits packets onto the 1394 bus. The descriptors are 16- and 32-bytes in length and must be aligned on a 16-byte boundary. There are five IT DMA command descriptors: OUTPUT_MORE, OUTPUT_MORE-Immediate, OUTPUT_LAST, OUTPUT_LAST-Immediate and STORE_VALUE.

9.1.1 IT DMA command descriptor overview

There are two components to a 1394 isochronous packet, the packet header and the packet data, and there are many ways in which software may need to organize this information in host memory. To accommodate the variety of packet organization, there are four IT DMA descriptor commands used to instruct the Host Controller on how to assemble the packets, and one descriptor command for writing a quadlet into host memory for software tracking purposes.

If a packet has two or more data fragments an OUTPUT_MORE-Immediate and possibly some OUTPUT_MORE commands are used. The OUTPUT_MORE-Immediate command is used to specify the packet header, and each OUTPUT_MORE command allows for the specification of one packet fragment.

To indicate the end of a packet, either the OUTPUT_LAST or OUTPUT_LAST-Immediate command must be used. The OUTPUT_LAST command allows for the specification of one data fragment, and the OUTPUT_LAST-Immediate is used to specify a packet solely consisting of an isochronous packet header. Unlike the OUTPUT_MORE commands, the OUTPUT_LAST commands indicate to the Host Controller that there is no more data to send for a packet.

The STORE_VALUE command descriptor provides a mechanism for software to monitor progress on a context without using interrupts. This command will write a quadlet to a specified host memory location.

9.1.2 OUTPUT_MORE descriptor

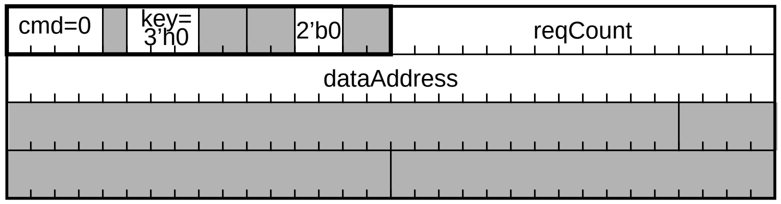


Figure 9-1 — OUTPUT_MORE command descriptor format

Table 9-1 — OUTPUT_MORE descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h0 for OUTPUT_MORE. Identifies one data fragment used to build the packet.
key	3	This field must be set to 3'h0.
b	2	Branch control. Must be set to 2'b00. Behavior is unspecified if set to 2'b01, 2'b10 or 2'b11.
reqCount	16	Request count. The size of the specified buffer in bytes pointed to by dataAddress.
dataAddress	32	Address of transmit buffer. dataAddress has no alignment restrictions.

The OUTPUT_MORE descriptor is used to specify one data fragment for the packet. It shall not be used for specifying the packet header, and must be preceded by an OUTPUT_MORE-Immediate or another OUTPUT_MORE.

9.1.3 OUTPUT_MORE-Immediate descriptor

Figure 9-2 — OUTPUT_MORE-Immediate descriptor format

Table 9-2 — OUTPUT_MORE-Immediate descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h0 for OUTPUT_MORE-Immediate.
key	3	This field must be set to 3'h2.
b	2	Branch control. Must be set to 2'b00. Behavior is unspecified if set to 2'b01, 2'b10 or 2'b11.
reqCount	16	Must be set to 8 to accommodate the IT packet header. Using any other value yields unspecified results.
skipAddress	28	16-byte aligned address of the next descriptor to be used if a missed cycle is detected. Used only within the first command descriptor in a descriptor block. The first command must either have a valid skipAddress, or must set the Z field to 0.
Z	4	Used to indicate the number of descriptors needed for the *skip* descriptor block. Z may be a value from 0 to 8. A zero indicates there is no skipAddress, and the DMA for this context stops. A value of 1 to 8 indicates that there are 1 to 8 descriptors used in the skip packet.
first quadlet second quadlet	32 32	Quadlets to be inserted into the isochronous transmit FIFO for the isochronous packet header (see section 9.6).

The OUTPUT_MORE-Immediate descriptor shall be used, and shall only be used, to specify the isochronous header for a non-zero data length packet. This is an efficient way for software to provide the packet header information since the data is built into the descriptor and does not need to be fetched from a separate memory buffer.

OUTPUT_MORE-Immediate command descriptors are 32 bytes in length regardless of the value of reqCount, and are counted as two 16-byte aligned blocks when calculating the Z value.

9.1.4 OUTPUT_LAST descriptor

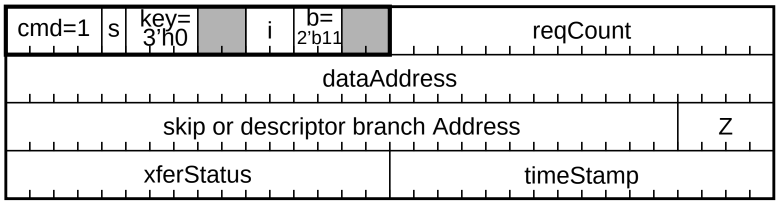


Figure 9-3 — OUTPUT_LAST command descriptor format

Table 9-3 — OUTPUT_LAST descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h1 for OUTPUT_LAST. Each command identifies one data fragment used to build the packet. OUTPUT_LAST is used to signify the end of the isochronous packet to be transmitted.
s	1	Status control. If set to one, xferStatus and timeStamp will be updated upon descriptor completion. If set to zero, neither field is updated.
key	3	This field must be set to 3'h0.
i	2	Interrupt control. Valid values are 2'b11 to generate an IsochTx interrupt when the descriptor is completed (see section 6.1), or 2'b00 for no interrupt. Behavior is unspecified if set to 2'b01 or 2'b10.
b	2	Branch control. This field must be set to 2'b11 to branch to the location specified in the branchAddress field. Behavior is unspecified for all other values.
reqCount	16	Request count: The size of the buffer in bytes pointed to by dataAddress.
dataAddress	32	Address of transmit buffer. dataAddress has no alignment restrictions.
branchAddress	28	16-byte aligned address of the next descriptor. Used only within OUTPUT_LAST* commands.
skipAddress		16-byte aligned address of the next descriptor to be used if a missed cycle is detected. Used only within the first command descriptor in a descriptor block. OUTPUT_LAST may only be the first descriptor in a descriptor block when reqCount is 0.
Z	4	Used in OUTPUT_LAST to indicate the number of descriptors needed in the *next* descriptor block. Z may be a value from 0 to 8. A zero indicates this is the last descriptor in the list for this IT DMA context. A value of 1 to 8 indicates that there are 1 to 8 descriptors used in the next descriptor block.
xferStatus	16	Written with ContextControl [15:0] after the descriptor is processed if s = 1.
timeStamp	16	Contains the three low order bits of cycleSeconds and all 13 bits of cycleCount, and is written when xferStatus is written. TimeStamp indicates the cycle for which the IT DMA controller queued the transmission of this packet. See section section 5.12, "Isochronous Cycle Timer Register," for information about cycle* fields.

The OUTPUT_LAST descriptor is used to indicate the end of a packet. If reqCount is non-zero, this specifies the last data fragment for the packet. It shall not be used for specifying the packet header.

An OUTPUT_LAST with reqCount=0 is used to indicate that no packet is to be sent for the current cycle. The IT DMA controller will advance the context to the next descriptor block (branchAddress) for the next cycle. An OUTPUT_LAST with a reqCount=0 shall not be preceded by any OUTPUT_MORE* descriptors in the descriptor block.

9.1.5 OUTPUT_LAST-Immediate descriptor

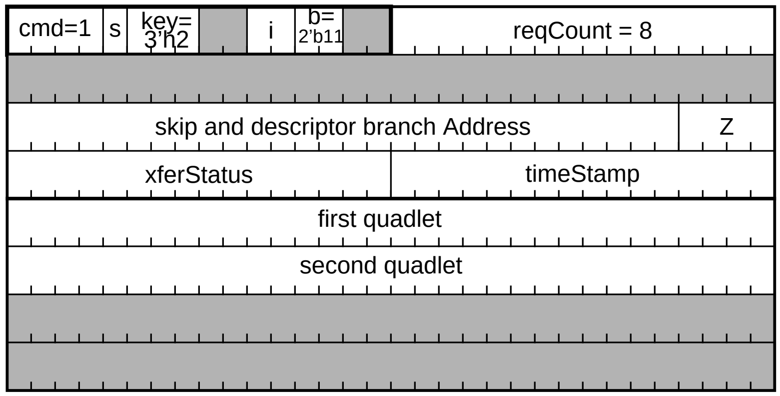


Figure 9-4 — OUTPUT_LAST-Immediate command descriptor format

Table 9-4 — OUTPUT_LAST-Immediate descriptor element summary

Element	Bits	Description
cmd, s		Same as in Table 9-3.
key	3	This field must be set to 3'h2.
i, b		Same as in Table 9-3.
reqCount	16	Must be set to 16'h0008 to accommodate the IT packet header. Using any other value yields unspecified results.
branchAddress	28	16-byte aligned address of the next descriptor. Used only within OUTPUT_LAST* commands.
skipAddress		16-byte aligned address of the next descriptor to be used if a missed cycle is detected. Used only within the first command descriptor in a descriptor block.
Z, xferStatus, timeStamp		Same as in Table 9-3.
quadlets	32*4	The first and second quadlets are used to specify the 2 quadlets required for the isochronous packet header. (See section 9.6).

The OUTPUT_LAST-Immediate descriptor must be used, and must only be used, to specify the isochronous header for a packet with zero data bytes. OUTPUT_LAST-Immediate command descriptors are 32-bytes in length regardless of the value of reqCount and are counted as two 16-byte aligned blocks when calculating the Z value.

9.1.6 STORE_VALUE descriptor

The STORE_VALUE command descriptor instructs the Host Controller to write a specified 32-bit value to a specified host memory location. If used, STORE_VALUE must be the first command descriptor in a descriptor block, and only one is permitted per descriptor block. STORE_VALUE must not be the only descriptor in a descriptor block and shall be followed by one or more OUTPUT_* descriptors. It has the following format.

Figure 9-5 — STORE_VALUE descriptor

Table 9-5 — STORE_VALUE descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h8 for STORE_VALUE.
key	3	This field must be set to 3'h6.
storeDoublet	16	16-bit value to be stored into the quadlet aligned dataAddress upon execution of this command. StoreDoublet is written as a 32 bit value, where bits 31:16 are 0's and bits 15:0 contain the storeDoublet value provided in the descriptor.
dataAddress	32	Quadlet aligned host memory address into which storeDoublet (padded to 32) bits is written.
skipAddress	28	16-byte aligned address of the next descriptor to be used if a missed cycle is detected. The skipAddress must be valid or the Z field must be 0. If the skip address is used, the store action specified by this descriptor will *not* be executed.
Z	4	Used to indicate the number of descriptors needed for the *skip* descriptor block. Z may be a value from 0 to 8. A zero indicates there is no skipAddress, and the DMA for this context stops. A value of 1 to 8 indicates that there are 1 to 8 descriptors used in the skip packet.

The STORE_VALUE command provides a mechanism for software to monitor a context's progress independent of using interrupts. For example a running IT context program could perform a STORE_VALUE periodically into a memory host location where software would look to determine the latest IT DMA context progress.

9.1.7 IT DMA descriptor usage

The Z value is used by the Host Controller to enable several descriptors to be fetched at once, for improved efficiency. Z values must always be encoded correctly. The contiguous descriptors described by a Z value are called a *descriptor block*. The following table summarizes all legal Z values:

Table 9-6 — Z value encoding

Z value	Use
0	Indicates that the current descriptor is the last descriptor in the context program.
1-8	Indicates that starting at descriptorAddress, there are one to eight 16-byte aligned physically contiguous descriptors and descriptor components.
9-15	reserved

Each isochronous transmit descriptor block for a packet shall be specified with the command descriptors according to the following rules:

- A maximum of 8 command descriptors may be used.
- Only one STORE_VALUE may be used, and it must be the first descriptor in a descriptor block.
- If STORE_VALUE is used, it shall be followed by at least one OUTPUT_* descriptor, and the Z value for the descriptor block shall be between 2-8 inclusively.
- If the packet dataLength is not zero, one OUTPUT_MORE-Immediate must be used, followed by zero to five OUTPUT_MORE's, followed by one OUTPUT_LAST.
- If the packet dataLength is zero, one OUTPUT_LAST-Immediate must be used.
- If no packet is to be sent during a cycle, one OUTPUT_LAST with reqCount=0 must be used and shall not be preceded by any other OUTPUT_* descriptor.

The isochronous packet header must be specified using a *-Immediate command. The OUTPUT_LAST* command must have a branch control value of 2'b11. All other commands must have a branch control value of 2'b00. Depending on the aggregate number of bytes being transmitted for one descriptor block, hardware may assist with padding. If the sum of all reqCounts modulo 4 is 0, then padding is not necessary. If the sum of all reqCounts module 4 is not 0, then hardware will insert padding up to a quadlet boundary.

To indicate the end of the context program, all IT DMA context programs must use an OUTPUT_LAST or OUTPUT_LAST-Immediate command with a branch (b) value of 2'b11 (branch always) and a Z value of 0 to indicate the end of the program. A program which ends can be appended to while the DMA runs, even if the DMA has already reached the last descriptor.

The first command in an isochronous packet descriptor block must have a skipAddress which points to the descriptor to branch to if this packet cannot be transmitted (typically due to a lost cycle). The value of the Command.*b* field in that descriptor does not affect a skip branch.

The use of many OUTPUT_MORE* commands to describe a single packet will generally cause extra fetch latencies, as the Host Controller fetches payload buffers from different parts of memory. These latencies may differ for each Host Controller implementation, bus, and host memory architecture. Software is expected to construct IT DMA context programs with a sufficiently low number of OUTPUT_MORE* commands so that the Host Controller can satisfy application-specific latency requirements.

ITDMA context programs must contain exactly one descriptor block to be processed per cycle. Each descriptor block must be identified with an accurate Z value, both when the program is started, and on each branch within the program. Each descriptor block must end with an unconditional branch to the next descriptor block, even if the next block follows immediately in consecutive memory. (The branch enables the ITDMA to learn the Z value for the next descriptor block). Each descriptor block must begin with a command that contains a branch to the skipAddress (also with a Z code).

Some applications of isochronous transfer do not transfer a packet on every isochronous cycle. Therefore the ITDMA will sometimes not transmit a packet for one or more channels. Within a context program, a non-transmit cycle is indicated by a descriptor block whose only transfer command is an OUTPUT_LAST with a reqCount of zero. (This is not a zero-length packet, which would be sent with an OUTPUT_LAST-Immediate.)

9.2 IT Context Registers

Each isochronous transmit context consists of two registers: CommandPtr and IT ContextControl. CommandPtr is used by software to tell the IT DMA controller where the DMA context program begins. IT ContextControl is used by software to control the context's behavior, and is used by hardware to indicate current status.

9.2.1 CommandPtr

The CommandPtr register specifies the address of the context program which will be executed when a DMA context is started. All descriptors are 16-byte aligned, so the four least-significant bits of any descriptor address must be zero. The four least-significant bits of the CommandPtr register are used to encode a Z value that indicates how many physically contiguous descriptors are pointed to by descriptorAddress.

Refer to section 3.1.2 for a full description of the CommandPtr register.

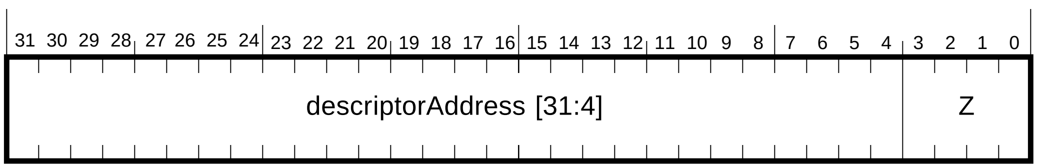


Figure 9-6 — CommandPtr register format

9.2.2 IT ContextControl Register

The IT *ContextControl* set and clear registers contains bits that control options, operational state, and status for the isochronous transmit DMA contexts. Software can set selected bits by writing ones to the corresponding bits in the *ContextControlSet* register. Software can clear selected bits by writing ones to the corresponding bits in the *ContextControlClear* register. It is not possible for software to set some bits and clear others in an atomic operation. A read from either register will return the same value.

The context control register used for isochronous transmit DMA contexts is shown below. In addition to the standard ContextControl fields as described in section 3.1.1, it includes a mechanism for starting transmit at a specified cycle time.

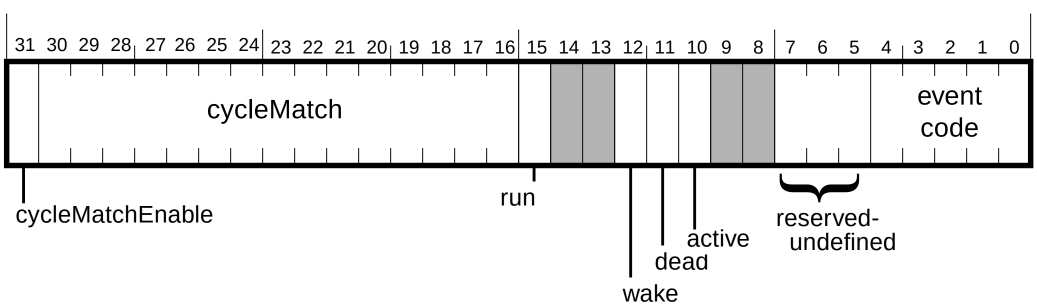


Figure 9-7 — IT DMA ContextControl (set and clear) register format

Table 9-7 — IT DMA ContextControl (set and clear) register description

field	rscu	reset	description
cycleMatchEnable	rscu	undef	When set to one, processing will occur such that the packet described by the context's first descriptor block will be transmitted in the cycle whose number is specified in the cycleMatch field of this register. The 15-bit cycleMatch field must match the low order two bits of cycleSeconds and the 13-bit cycleCount field in the cycle start packet that is sent or received immediately before isochronous transmission begins. Since the IT DMA controller may work ahead, the processing of the first descriptor block may begin slightly in advance of the actual cycle in which the first packet is transmitted. The effects of this bit however are impacted by the values of other bits in this register and are explained below this table. Once the context has become active, hardware clears the cycleMatchEnable bit.
cycleMatch	rsc	undef	Contains a 15-bit value, corresponding to the low order two bits of the bus CycleTime.*cycleSeconds* and the 13-bit CycleTime.*cycleCount* field. If ContextControl.*cycleMatchEnable* is set, then this IT DMA context will become enabled for transmits when the low order two bits of the bus CycleTime.*cycleSeconds* and CycleTime.*cycleCount* value equals the cycleMatch value.
run	rscu	1'b0	Refer to section 3.1.1.1 and the description following this table for an explanation of the ContextControl.*run* bit.
wake	rsu	undef	Refer to section 3.1.1.2 for an explanation of the ContextControl.*wake* bit.
dead	ru	1'b0	Refer to section 3.1.1.4 for an explanation of the ContextControl.*dead* bit.
active	ru	1'b0	Refer to section 3.1.1.3 for an explanation of the ContextControl.*active* bit.
reserved undefined	ru	undef	This field is specified as undefined and may contain any value without impacting the intended processing of this packet.
event code	ru	undef	Following an OUTPUT_LAST* command, the error code is indicated in this field. Possible values are: ack_complete, evt_underrun, evt_descriptor_read, evt_data_read, evt_tcode_err and evt_unknown. See Table 3-2, "Packet event codes," for descriptions and values for these codes.

The cycleMatch field is used to start an IT DMA context program on a specified cycle. Software enables matching by setting the cycleMatchEnable bit. When the low order two bits of the bus CycleTime.*cycleSeconds* and CycleTime.*cycleCount* value matches the cycleMatch value, hardware clears the cycleMatchEnable bit to 0, sets the ContextControl.*active* bit to 1, and begins executing descriptor blocks for the context. The transition of an IT DMA context to the active state from the not-active state is dependent upon the values of the run and cycleMatchEnable bits.

- If run transitions to 1 when cycleMatchEnable is 0, then the context will become active (active = 1).
- If both run and cycleMatchEnable are set to 1, then the context will become active when the low order two bits of the bus CycleTime.*cycleSeconds* and 13-bit CycleTime.*cycleCount* values match the 15-bit cycleMatch value.
- If both run and cycleMatchEnable are set to 1, and cycleMatchEnable is subsequently cleared, the context becomes active.
- If both run and active are 1 (the context is active), and then cycleMatchEnable is set to 1, this will result in unspecified behavior.

Due to software latencies, software attempts to manage the startup of a context too close to the current time may not be effective.

In addition, the usability of cycleMatchEnable for IT contexts will be impacted by the cycleInconsistent interrupt. Refer to Section 9.5.1 for more information.

9.3 Isochronous transmit DMA controller

The following sections describe how software manages the multiple isochronous transmit DMA contexts. Each context has a CommandPtr pointing to the current DMA descriptor. For every cycle start packet that the Host Controller receives or sends, the IT DMA controller can transmit exactly one descriptor block describing exactly one packet from each DMA context that is in the ContextControl.*run* state.

9.3.1 IT DMA Processing

Each IT DMA context command pointer corresponds to a list of packets to be sent on successive 1394 cycles. Generally, each list represents a single isochronous channel. Isochronous channel numbers are not tied to any internal indexing scheme utilized by the Host Controller to track all implemented IT DMA contexts. Each IT DMA context program pointed to by each CommandPtr will specify the entire isochronous packet header, including the isochronous channel number, for each packet that is transmitted. The entire ITDMA is summarized in the following figure:

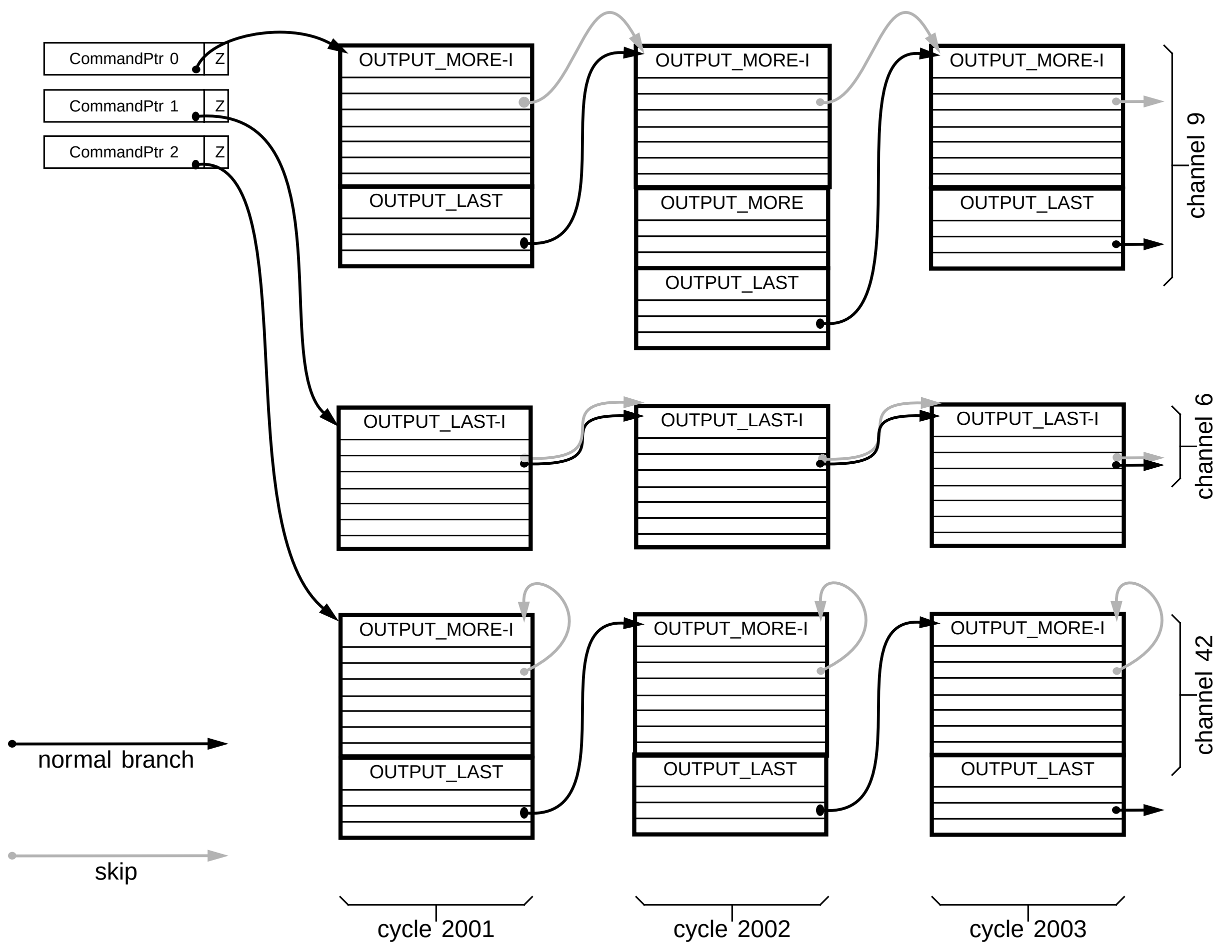


Figure 9-8 — ITDMA summary

In the example, three channels are being transmitted. Three cycles of transmit are shown. Context 0 is sending on isochronous channel 9, using an OUTPUT_MORE-Immediate to send each packet header and an OUTPUT_LAST for each payload. In cycle 2002 the payload spans a page boundary, so channel 9 uses an extra OUTPUT_MORE. Channel 9 will skip to the next packet if any cycle is lost. Context 1 is sending on isochronous channel 6, with zero length packets and only headers. Because channel 6 uses a single descriptor per packet, the skip branch is equal to the normal next packet branch. Context 2 is sending on isochronous channel 42, with each skip branch pointing to itself. If a cycle is lost, channels 6 and 9 will advance to the next packet, while channel 42 will fall behind by one packet, without skipping any packets.

For every cycle, the IT DMA controller shall process each running context in order, from the lowest numbered context through the highest numbered context. For each cycle, the IT DMA controller will complete one descriptor block for each active IT DMA context. Once a packet has been transferred into the transmit FIFO, the packet is considered sent even though it may not have been transmitted yet on the 1394 wire.

If there is a disruption while the IT DMA controller is processing a context, such as a bus reset or the loss of the isochronous phase, the IT DMA controller is required to continue through its list of active contexts taking the skip branch address for each of the remaining contexts.

9.3.2 Prefetching IT Packets

The Host Controller is permitted to work up to two cycles ahead of the current cycle time. The result is that it's possible for data for a 1394 cycle to be put into the FIFO long before it is sent on the bus. This in effect creates a time decoupling of the host side (input) of the FIFO from the link side (output) of the FIFO.

Since the host side and the link side are not time synchronized, the host side may have its own cycle timer. This keeps track of the cycle number for which data is being put into the FIFO. It is *not* the same cycle timer that the link side uses. When the Host Controller is initialized, the timers are set to the same value and then the host side can start putting things into the FIFO. Whenever the difference between the host side cycle time and the link side cycle time is less than two, the host can start putting packets into the FIFO.

By working up to two cycles ahead it's possible for two 1394 cycles worth of packets to be in the FIFO at the same time. To convey to the link side where the 1394 cycle boundary is between the packets, the host side puts a delimiter into the FIFO each time processing is completed for all contexts for a cycle. When a cycle start appears on the 1394 bus, the link starts taking packets out of the FIFO and sends the data on the bus until the link reaches the delimiter.

9.3.3 Isochronous Transmit Cycle Loss

The IT DMA controller can send multiple packets (multiple isochronous channels) in each isochronous cycle. Because isochronous cycles can be lost, the IT DMA is organized so that one cycle's worth of packets can be skipped, if necessary, to catch up. The loss of an isochronous cycle is usually uncommon, and typically results from a bus reset.

If isochronous cycles were lost, and no corrective action was taken, the transmitter would gradually fall behind, sending each packet some number of cycles after the transmission time intended by software.

In order to permit the transmitter to avoid falling behind, each packet in an IT DMA context program contains a *skip branch address*. Any time the IT DMA wants to correct for a cycle loss, it will follow this branch instead of transmitting the packet. For each cycle's worth of packets (descriptor blocks), the IT DMA will either put all of the packets into the FIFO and advance to the next descriptor block pointed to by branchAddress or will not put any packets into the FIFO and will advance to the next descriptor block pointed to by skipAddress. SkipAddress is used for any condition in which the IT DMA cannot acquire the bus to transmit all packets for a cycle within that cycle.

Software can use the skip branch in at least four ways. 1) Branching to the next packet will cause the IT DMA to skip packets to recover from cycle loss. 2) Branching to the same packet will cause the IT DMA to fall behind (on that channel only) without skipping any packets due to cycle loss. 3) Branching to an alternate context program can allow the generation of an interrupt, and the possible early completion of transmission. 4) Stopping the IT DMA context program due to cycle loss. Software can use the third and fourth methods to cease transmission on cycle loss in the application-specific case that the receiver cannot tolerate either late or lost packets.

Because the Host Controller will generally load isochronous transmit packets into a FIFO in advance of transmission, some packets may be considered complete when cycle loss is detected, even though they have not yet left the transmit FIFO. In this situation, the Host Controller will hold those packets in the FIFO until they can be transmitted, and will then complete the transmission of each context packet that had been intended to go out in the same cycle. The Host Controller will then apply the skip branching on the packets for the next cycle (the first cycle for which no transmission has been performed). If a context in the ITDMA is arranged to skip packets on cycle loss, the packet skipped will be the one scheduled for the cycle following the cycle that was lost. If the Host Controller preloads more than one cycle's worth of packets, the skip may be delayed by a similar number of cycles, so that the transmit FIFO can empty normally, without being flushed.

The illustration below shows how each of these cases works. In this example, the ITDMA attempts to keep two cycles ahead of the bus. In other words, it tries to have two complete cycles in the transmit FIFO (if they will fit) whenever possible. Context A illustrates case 1 (above), where the skip branch is chosen so that packets are skipped. Note that because of the FIFO preload, the two packets skipped on Context A (A_4 and A_5) follow a delayed packet (A_3) that was already in the FIFO. While it might have been possible to skip only one packet if the FIFO was flushed, it would be much harder for the Host Controller to have packet A_5 ready in time to send it on cycle 6. Context B illustrates case 2, where packets are not skipped. While context A loses two packets, context B instead falls two cycles behind. Context C illustrates case 3, where transmission ends in response to a detected cycle loss. Packets C_2 and C_3 were already in the FIFO, so they are transmitted, followed by the end-of-program packet C_x. The descriptor block for packet C_x loops to itself in case additional cycles are lost before C_x is sent. This loop guarantees that C_x will be sent before the program ends. Context D illustrates case 4, where transmission ends in response to a detected cycle loss without an end-of-program packet. The skip address indicates the end of list (Z=0) and no more packets are loaded into the FIFO upon detection of cycle loss.

In these examples, the packets that are "in the FIFO" assume an infinitely large transmit FIFO. The Host Controller will transmit packets as shown, even if they are too big to actually fit into the FIFO.

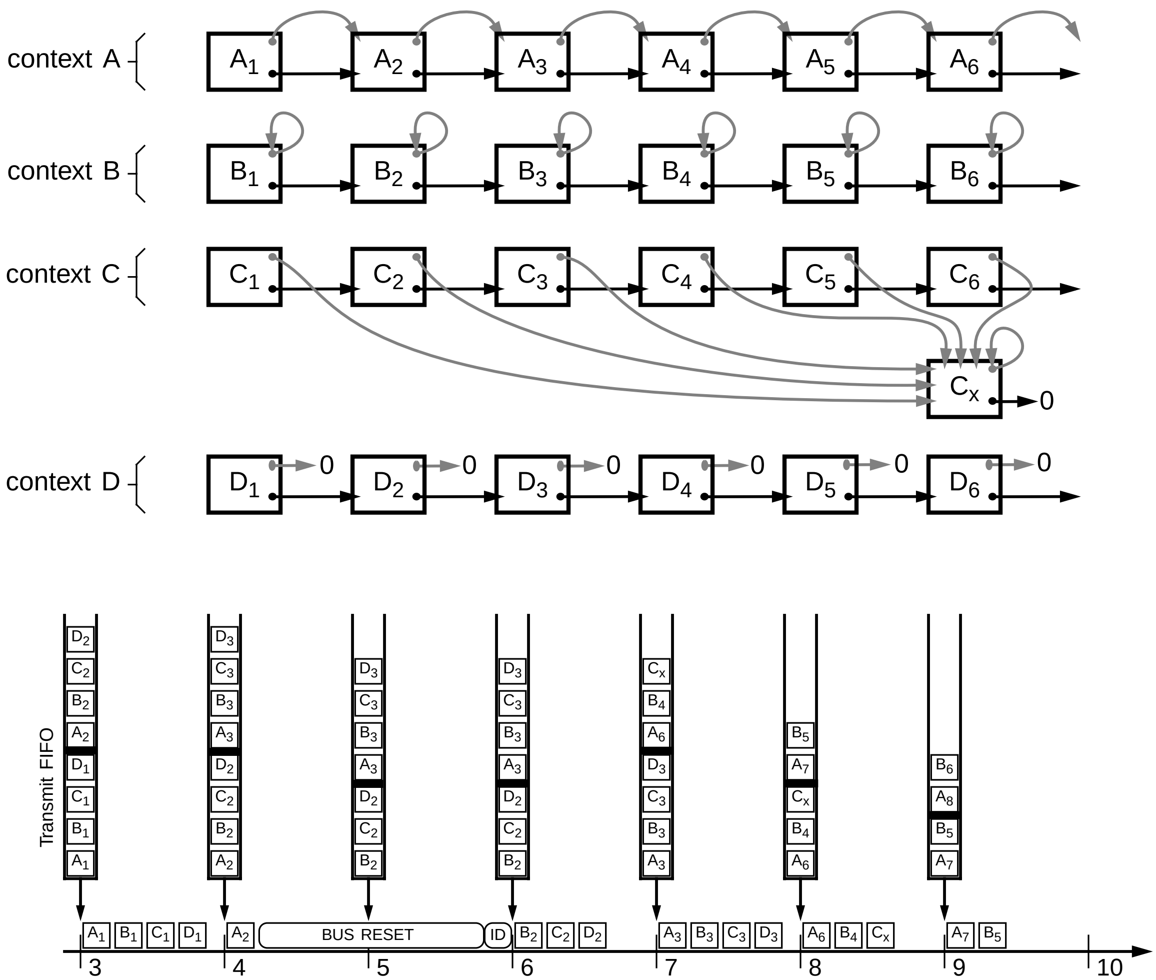


Figure 9-9 — Isochronous transmit cycle loss example

If a cycle loss is detected while the IT DMA is mid packet, that context's descriptor block will not branch to the skipAddress, but will advance to the next descriptor block.

9.3.4 FIFO Underrun

If there is a FIFO underrun such that the isochronous period ends before all active contexts have been processed for that cycle, then the following shall occur:

- The packet that underran is lost.
- The context with the underrun
    1) records evt_underrun or evt_data_read in the event code of the OUTPUT_LAST_* as appropriate (refer to section 13.2.3), and
    2) advances to the branchAddress descriptor block.
- If there are contexts remaining to be processed for the now lost cycle, they continue to be processed normally and then advance to the next descriptor block pointed to by branchAddress.

- If there were contexts processed subsequent to the underrun, then all contexts will follow the skip branch during the next cycle.
- If there were *no* contexts to be processed after the context that underran, then processing for the next cycle continues as normal.

Through these steps, the Host Controller ensures that either all contexts skip or no contexts skip for a given cycle.

9.3.5 Determining the number of implemented IT DMA contexts

The number of supported isochronous transmit DMA contexts will vary for 1394 OpenHCI implementations from a minimum of four to a maximum of 32. Software can determine the number of supported IT DMA contexts by writing 32'hFFFF_FFFF to isoXmitIntMask register (see section 6.3.1), and then reading it back. Bits returned as 1's indicate supported contexts, and bits returned as 0's indicate unsupported/unimplemented contexts.

9.4 Appending to an IT DMA Context Program

As described in Section 3.2.1.2, "Appending to Running List," software may freely append to a context program without knowledge of where the controller is in processing the list of descriptor blocks. Unlike other DMA contexts, the IT DMA contexts can have two pointers that may require updating in the known last descriptor block; the skipAddress and the branchAddress. When an IT context has reached the end of its context program and active is 0, setting wake will result in using the descriptor (*not* descriptor block) which had Z=0 and will use the provided address, be it a skip or branch, for retrieving the next descriptor block.

9.5 IT Interrupts

Each of the possible 32 isochronous transmit contexts can generate an interrupt, so each IT context has a bit in the isoXmitIntEvent register. Software can enable interrupts on a per-context basis by setting the corresponding isoXmitMask bit to one.

To efficiently handle interrupts which could conceivably be generated from 32 different contexts in close proximity to one another, there is a single bit for all IT DMA contexts in the Host Controller IntEvent register. This bit signifies that at least one but potentially several IT DMA contexts attempted to generate an interrupt. Software can read the isoXmitInt-Event register to find out which context(s) are involved. For more information on the isoXmitIntEvent register, see section 6.3.1.

9.5.1 cycleInconsistent Interrupt

When the IntEvent.*cycleInconsistent* condition occurs (table 6-1), the IT DMA controller shall continue processing running contexts normally, with the exception that contexts with the ContextControl.*cycleMatchEnable* bit set will remain inactive and cycleMatch processing shall be, in effect, disabled. To re-enable cycleMatch processing, software must first stop the IT contexts for which cycleMatch is enabled (by clearing ContextControl.*run* to 0 and waiting for ContextControl.*active* to go to 0), then must clear the IntEvent.*cycleInconsistent* interrupt. The stopped IT contexts may then be started, but software should not schedule any transmits to occur for these contexts for at least two cycles immediately following the clearing of the interrupt condition.

9.5.2 busReset Interrupt

Bus reset does not affect isochronous transmit.

9.6 IT Data Format

An isochronous transmit packet consists of two header quadlets (as specified in either the OUTPUT_MORE-Immediate or OUTPUT_LAST-Immediate descriptor) and an optional data payload. The data payload in host memory is not required be aligned on a quadlet boundary. Padding is added by the Host Controller if needed. The format is as follows.

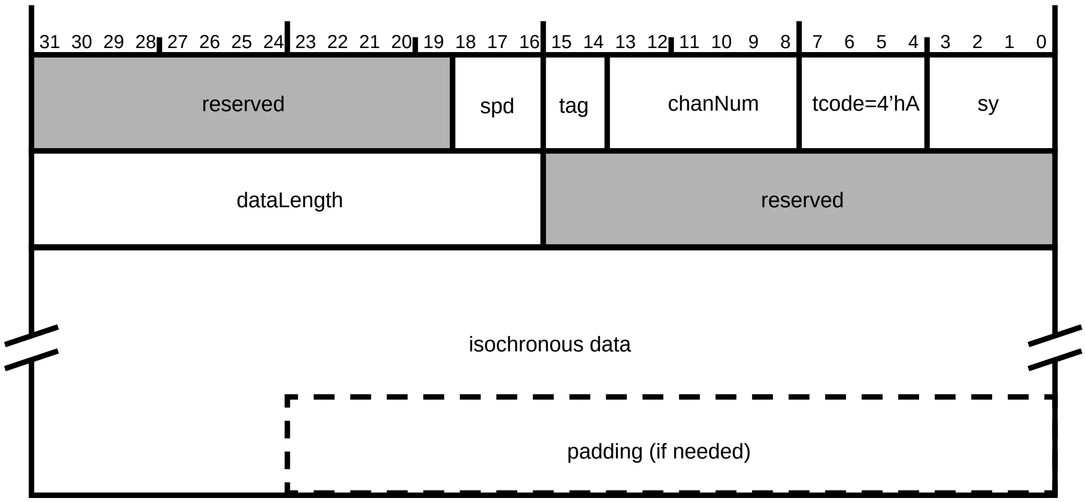


Figure 9-10 — Isochronous transmit format

Table 9-8 — Isochronous transmit fields

field name	bits	description
spd	3	This field indicates the speed at which this packet is to be transmitted. 3'b000 = 100 Mbits/sec, 3'b001 = 200 Mbits/sec, and 3'b010 = 400 Mbits/sec, other values are reserved.
tag	2	The data format of the isochronous data (see IEEE 1394 specification)
chanNum	6	The channel number this data is associated with.
tcode	4	The transaction code for this packet.
sy	4	Transaction layer specific synchronization bits.
dataLength	16	Indicates the number of bytes in this packet.
isochronous data		The data to be sent with this packet. The first byte of data must appear in the leftmost byte of the first quadlet of this field. The last quadlet should be padded with zeroes, if necessary.
padding		If the dataLength mod 4 is not zero, then zero-value bytes are added onto the end of the packet to guarantee that a whole number of quadlets is sent.

Note that packets to go out over the 1394 wire are constructed from this Host Controller internal format, and are not sent in the exact order as shown above. For example, spd, shown in the first quadlet, is not transmitted at all as part of the isochronous packet header.

10. Isochronous Receive DMA

The Isochronous Receive DMA (IR DMA) controller has a required minimum of four and an implementation maximum of 32 isochronous receive DMA contexts. Each context is controlled by a DMA context program. One single IR DMA context can receive packets from multiple isochronous channels, and the remaining DMA contexts can each receive packets from a single isochronous channel. IR DMA contexts can either receive exactly one packet per buffer, or they can concatenate packets into a stream that completely fills each of a series of buffers. Packets may be received with or without isochronous packet headers and timeStamps.

10.1 IR DMA Context Programs

For isochronous receive DMA, a context program is a list of DMA descriptors used to identify buffers in host memory into which the Host Controller places received isochronous packets. The descriptors are 16 bytes in length and must be aligned on a 16 byte boundary. There are two kinds of descriptor commands available: INPUT_MORE and INPUT_LAST.

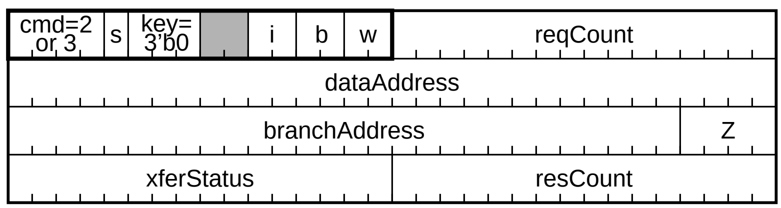


Figure 10-1 — INPUT_MORE/INPUT_LAST descriptor format

Table 10-1 — INPUT_MORE/INPUT_LAST descriptor element summary

Element	Bits	Description
cmd	4	Set to 4'h2 for INPUT_MORE, or set to 4'h3 for INPUT_LAST. INPUT_MORE is required for receiving packets in buffer-fill mode (see section 10.2.1), and may also be used in packet-per-buffer mode. INPUT_LAST is required for receiving packets in packet-per-buffer mode (see section 10.2.2), and must be the final descriptor in a descriptor block. It is not permitted in buffer-fill mode.
s	1	Used with packet-per-buffer mode only (see section 10.2.2). If set to one, xferStatus and resCount will be updated upon descriptor completion. If set to zero, neither field is updated. Assumed to be one for buffer-fill mode.
key	3	This field must be set to 3'b0.
i	2	Interrupt control. Valid values are 2'b11 to generate an IsochRx interrupt when the descriptor is completed (see section 6.1), or 2'b00 for no interrupt. Behavior is unspecified for 2'b01 and 2'b10. In packet-per-buffer mode (see section 10.2.2), software must set *i* to 0 in INPUT_MORE descriptors and may be ignored by hardware.
b	2	Branch control. Valid values are 2'b11 to branch to branchAddress, and 2'b00 not to branch. Behavior is unspecified for 2'b01 and 2'b10. For buffer-fill mode (see section 10.2.1), this field must always be set to 2'b11. For packet-per-buffer mode (see section 10.2.2), this field must be 2'b00 for INPUT_MORE commands and 2'b11 for INPUT_LAST commands.

Table 10-1 — INPUT_MORE/INPUT_LAST descriptor element summary

Element	Bits	Description
w	2	Wait control. Valid values are 2'b11 to wait for a packet with a sync field which matches the sync specified in the context's IRContextMatch register (see section 10.3), or 2'b00 not to wait. For packet-per-buffer mode, 2'b11 can only be used in the first descriptor of a descriptor block. For buffer-fill mode a w of 2'b11 affects all packets received into the buffer - the wait condition will apply the sync match requirement to *each* packet to be received into the indicated buffer and not just to the first packet. Therefore, if needed it is recommended that w only be set to 2'b11 for the very first descriptor only in a buffer-fill context. Note that all packets are filtered on the IRContextMatch tag values regardless of the value of this (w) field. Behavior is unspecified for 2'b01 and 2'b10.
reqCount	16	Request count: The size of the input buffer in bytes.
dataAddress	32	Address of receive buffer. Any receive buffer which will contain one or more packet headers must have a quadlet aligned dataAddress. Buffers to contain data only and no headers may have a byte aligned dataAddress.
branchAddress	28	16-byte aligned address of the next descriptor. This field is not used for INPUT_MORE commands in packet-per-buffer mode.
Z	4	For buffer-fill mode (see section 10.2.1), Z must be either 1 to indicate the branchAddress is a valid address for the next INPUT_MORE, or 0 to indicate this descriptor is the end of the context program. For packet-per-buffer mode (see section 10.2.2), if the command is INPUT_LAST, Z may be a value from 1 to 8 to indicate the number of descriptors in the next descriptor block, or 0 to indicate the end of the context program. If the command is INPUT_MORE, then Z is not used.
xferStatus	16	Composed of 16-bits from ContextControl[15:0]. For buffer-fill mode, xferStatus is written when resCount is updated. For packet-per-buffer mode, xferStatus is written after the descriptor is processed if s = 1.
resCount	16	Residual count: The number of bytes remaining in the dataAddress buffer (out of a maximum of reqCount). Written if in packet-per-buffer mode and s = 1, or each time a packet is received in buffer-fill mode. For further details on when resCount is updated in buffer-fill mode, see section 10.2.1.

The Z value is used by the Host Controller to fetch multiple command descriptors at once, for improved efficiency. Z values must always be encoded correctly. The contiguous descriptors described by a Z value are called a *descriptor block*. The following table summarizes all legal Z values:

Table 10-2 — Z value encoding

Z value	Use
0	Indicates that the current descriptor is the last descriptor in the context program.
1-8	Indicates that 1 to 8 descriptors starting at descriptorAddress are physically contiguous.
9-15	reserved

To indicate the end of the context program, all IR DMA context programs must indicate the end of the program by using a command descriptor with a *b* value of 2'b11 (branch always) and a *Z* value of 0. A context program can be appended to while the DMA runs, even if the DMA has already reached the last descriptor. section 3.2.1.2 describes how to append to a context program.

When an IR DMA context is running and/or active, software shall not modify any command descriptors within the context program with the exception of the last command descriptor (the one descriptor in a program with *b*=2'b11 and *Z*=4'h0). The last command descriptor may only be modified according to the steps described in section 3.2.1.2.

10.2 Receive Modes

The Host Controller can write isochronous receive packets into host memory buffers in one of two ways. It can place them using either buffer-fill mode or packet-per-buffer mode.

10.2.1 Buffer Fill Mode

In bufferFill mode, all received packets are concatenated into a contiguous stream of data. This data is then metered out into buffers described by a DMA context program, filling each buffer completely. Packets may straddle multiple buffers in this mode (see packet 2 in the illustration below).

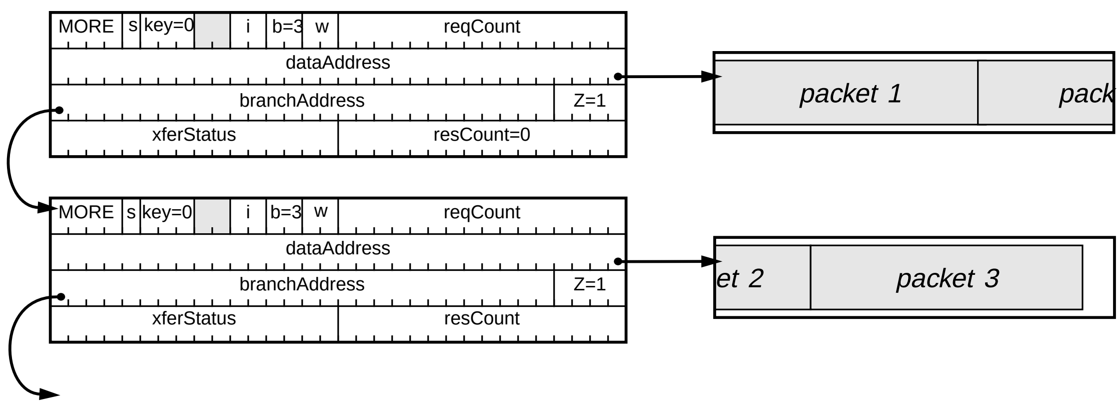


Figure 10-2 — IR Buffer Fill Mode

A context program for an isochronous receive context in buffer-fill mode consists of a list of independent INPUT_MORE descriptors, each branching to the next descriptor in the list. Since each descriptor must always branch to the subsequent one, the *b* field must always be set to 2'b11 to indicate a branch. If a buffer-fill mode INPUT_MORE descriptor is not the last descriptor in the list, its Z value must be set to 1 to instruct the Host Controller to fetch the next single descriptor. If it is the last one in the list, Z must be set to 0. Also, to ensure an accurate *resCount* value software must initialize resCount to the value of reqCount.

As depicted above, it is possible for a received packet to straddle multiple buffers. To ensure that the receive buffers for a context remain parsable, hardware must follow the following procedure.

1) After filling to the end of a buffer with a partial packet, advance to the next descriptor block and obtain the next buffer (dataAddress), retaining all state for the first buffer as well as for the new buffer.
2) Continue writing packet bytes into the subsequent buffer(s). If the end of a buffer is reached, advance to the next buffer without updating status and without retaining state for any of the interim buffers. Write the remaining packet bytes into the final packet buffer.
3) If there is no data error: a) conditionally write the trailer quadlet into the last buffer, b) update xferStatus and resCount into the **final** buffer's descriptor, and c) update xferStatus and resCount into the **first** buffer's descriptor. At that point the previous state of the first buffer is no longer needed and the first buffer's descriptor is completed.
4) If there *is* an error, then the packet must be 'backed-out' by reverting back to the previous state (as saved earlier). XferStatus and resCount are <u>not updated</u> for either descriptor.

By following these steps, the IR context buffers remain intact and can be parsed. Since interim buffers (those containing an inner portion of one packet) will not have their status updated, software must only use resCount values when the corresponding xferStatus indicates the active bit is set to one. It follows from this that if the xferStatus.*active* bit is set in a descriptor, then all prior descriptors have been filled.

For information on the effect of a host bus error on an IR DMA context in buffer-fill mode, refer to section 13.2.6.

10.2.2 Packet-per-Buffer Mode

In packet-per-buffer mode, each received packet is placed in the buffer(s) described by one descriptor block. Any leftover bytes are discarded, and packets never straddle multiple descriptor blocks. Both INPUT_MORE and INPUT_LAST are allowed in packet-per-buffer mode. Each INPUT_LAST marks the end of a packet, though the final byte may have been used up in a previous INPUT_MORE (see packet 2 in the illustration below). Each packet starts in an INPUT_* command that follows an INPUT_LAST.

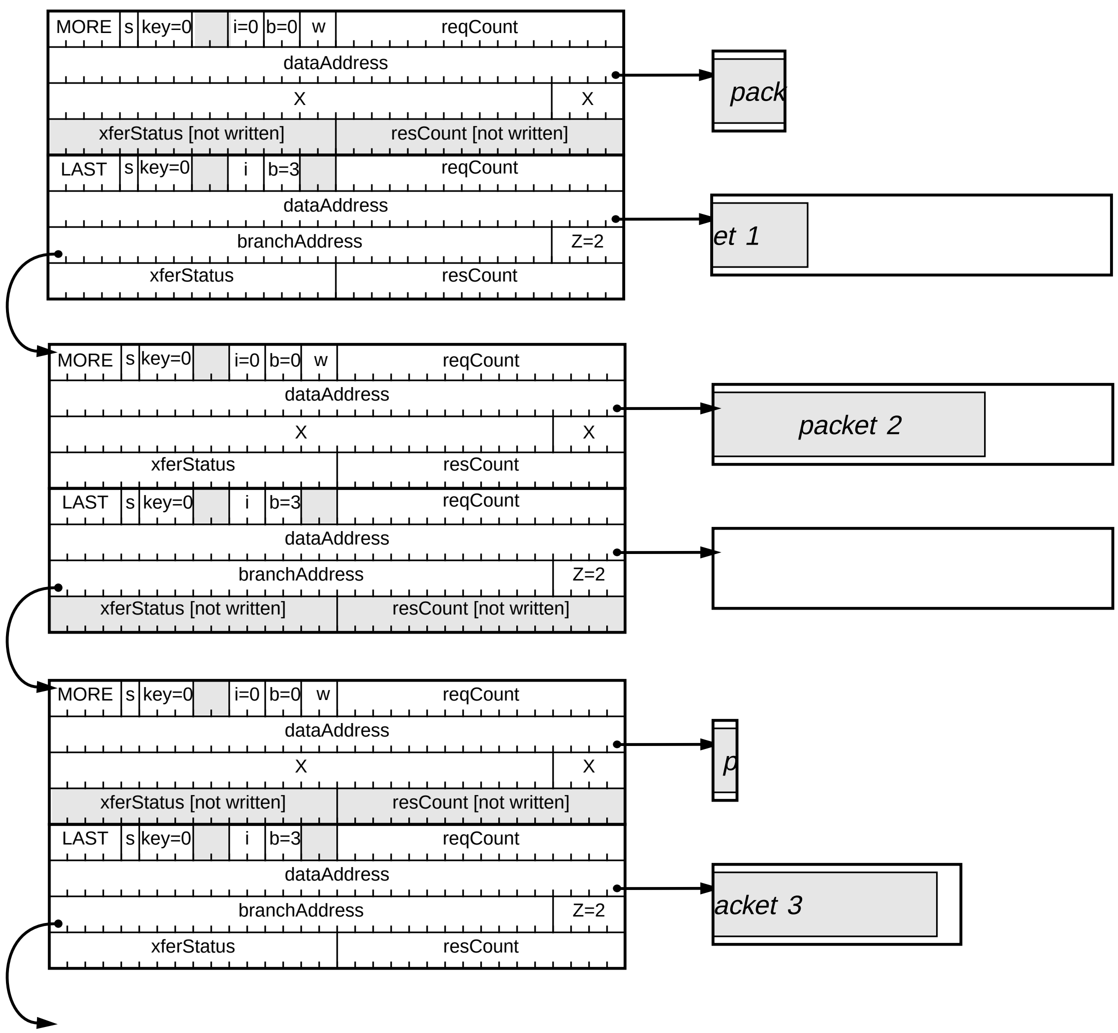


Figure 10-3 — packet-per-buffer receive mode

A context program for an isochronous receive context in packet-per-buffer mode consists of a series of descriptor blocks. Each descriptor block will receive one packet and must contain a contiguous set of 0 to 7 INPUT_MORE descriptors, followed by one INPUT_LAST descriptor. This requirement permits the Host Controller to prefetch all the descriptors for a packet, in order to avoid fetching additional descriptors during a packet transfer. INPUT_MORE descriptors must have the *b* field set to 2’b00 (never branch). INPUT_LAST descriptors must have the *b* field set to 2’b11 (always branch), and must either have a valid address in branchAddress with a Z value of 1 to 8, or must have a Z value of 0 to indicate it’s the last descriptor in the context program.

For information on the effect of a host bus error on an IR DMA context in packet-per-buffer mode, refer to section 13.2.6.

10.2.2.1 Command.xferStatus and Command.resCount updates

In packet-per-buffer mode, when s=1 the xferStatus and resCount fields are updated only in the descriptor for the buffer which receives the last byte of the packet. ResCount is only valid in a descriptor if the xferStatus field has the ContextControl.*run* bit set. To obtain accurate values for xferStatus, it is recommended that software initialize xferStatus to zero (evt_no_status).

In figure 10-3 above, there are 3 shaded xferStatus quadlets. The shaded quadlets are status fields that were never updated, and the unshaded status quadlets reflect status fields that were updated. In the top descriptor block, the xferStatus quadlet in the first descriptor was not written because packet 1 did not complete in the first descriptor’s buffer. In the middle descriptor block, the first descriptor was big enough to hold packet 2 completely. Since the first descriptor’s buffer received the last byte of packet 2, the first descriptor’s status was written, and the second descriptor’s status is ignored. Although the OUTPUT_LAST’s status is ignored in this example, its *i* bit is used to determine whether or not an interrupt is triggered for this descriptor block.

If a descriptor block describes buffer space that cannot fit an entire packet (including header if isochHeader mode is enabled), then the overflow bytes are discarded. When this occurs, xferStatus.ack will be set to evt_long_packet.

10.3 IR Context Registers

Each isochronous receive context consists of three registers: CommandPtr, IRContextControl, and IRContextMatch. CommandPtr is used by software to tell the IR DMA controller where the DMA context program begins. IRContextControl is used by software to control the context’s behavior, and is used by hardware to indicate current status. IRContext-Match is used to start on a specified cycle number and to filter received packets based on their tag bits and possibly sync bits. This section describes each register in detail.

10.3.1 CommandPtr

The CommandPtr register specifies the address of the context program which will be executed when a DMA context is started. All descriptors are 16-byte aligned, so the four least-significant bits of any descriptor address must be zero. The four least-significant bits of the CommandPtr register are used to encode a Z value that indicates how many physically contiguous descriptors are pointed to by descriptorAddress. In buffer-fill mode, Z will be either one or zero. In packet-per-buffer mode, Z will be from zero to eight.

Refer to section 3.1.2 for a full description of the CommandPtr register.

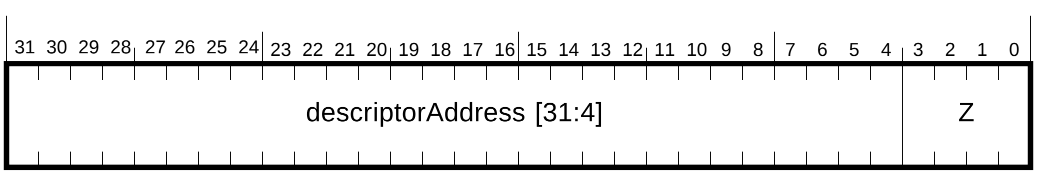


Figure 10-4 — CommandPtr register format

10.3.2 IRContextControl register (set and clear)

The *IRContextControl* register contains bits that control options, operational state, and status for the isochronous receive DMA contexts. Software can set selected bits by writing ones to the corresponding bits in the *ContextControlSet* register. Software can clear selected bits by writing ones to the corresponding bits in the *ContextControlClear* register. It is not possible for software to set some bits and clear others in an atomic operation. A read from either register will return the same value.

The context control register used for isochronous receive DMA contexts is shown below. It includes several fields which permit software to filter packets based on various combinations of fields within the isochronous packet header.

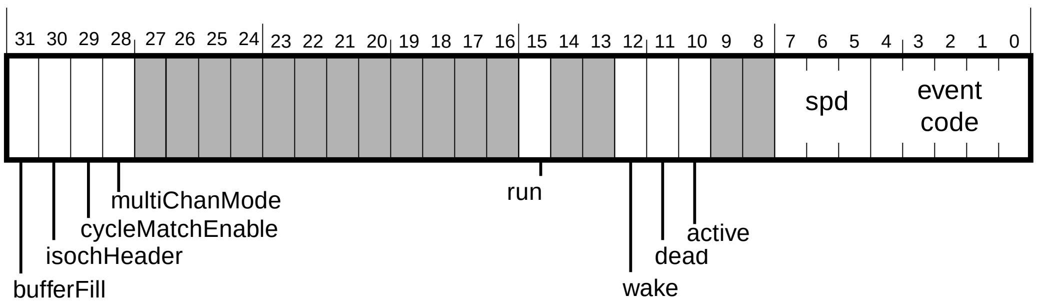


Figure 10-5 — IR DMA ContextControl (set and clear) register format

Table 10-3 — IR DMA ContextControl (set and clear) register description

field	rscu	reset	description
bufferFill	rsc	undef	When set to one, received packets are placed back-to-back to completely fill each receive buffer (specified by an INPUT_MORE command). When clear, each received packet is placed in a single buffer (described by zero to seven INPUT_MORE commands followed by an INPUT_LAST command). If the multiChanMode bit is set to one, this bit must also be set to one. The value of bufferFill must not be changed while *active* or *run* are set to one.
isochHeader	rsc	undef	When set to one, received isochronous packets will include the complete 4-byte isochronous packet header seen by the link layer. The end of the packet will be marked with a xferStatus (bits 15:0 of this register) in the first doublet, and a 16-bit timeStamp indicating the time of the most recently received (or sent) cycleStart packet. When clear, the packet header is stripped off of received isochronous packets. The packet header, if received, immediately precedes the packet payload. Details are shown in section 10.6. The value of isochHeader must not be changed while *active* or *run* are set to one.
cycleMatchEnable	rscu	undef	In general, when set to one, the context will begin running only when the 15-bit cycleMatch field in the contextMatch register matches the two bits of the bus CycleTime.*cycleSeconds* and 13-bit CycleTime.*cycleCount* values. The effects of this bit however are impacted by the values of other bits in this register and are explained below. Once the context has become active, hardware clears the cycleMatchEnable bit. The value of cycleMatchEnable must not be changed while *active* or *run* are set to one.

Table 10-3 — IR DMA ContextControl (set and clear) register description

field	rscu	reset	description
multiChanMode	rsc	undef	When set to one, the corresponding isochronous receive DMA context will receive packets for all isochronous channels enabled in the IRChannelMaskHi and IRChannelMaskLo registers (see section 10.4.1.1). The isochronous channel number specified in the IRDMA context match register is ignored. When set to zero, the IRDMA context will receive packets for that single channel. Only one IRDMA context may use the IRChannelMask registers. If more than one IRDMA context control register has the multiChanMode bit set, results are undefined. See section 10.4.3 for more information. The value of multiChanMode must not be changed while *active* or *run* are set to one
run	rscu	1'b0	Refer to section 3.1.1.1 and the description following this table for an explanation of the ContextControl.*run* bit.
wake	rsu	undef	Refer to section 3.1.1.2 for an explanation of the ContextControl.*wake* bit.
dead	ru	1'b0	Refer to section 3.1.1.4 for an explanation of the ContextControl.*dead* bit.
active	ru	1'b0	Refer to section 3.1.1.3 for an explanation of the ContextControl.*active* bit.
spd	ru	undef	This field indicates the speed at which the packet was received. 3'b000 = 100 Mbits/sec, 3'b001 = 200 Mbits/sec and 3'b010 = 400 Mbits/sec. All other values are reserved.
event code	ru	undef	For bufferFill mode, possible values are: ack_complete, evt_descriptor_read, evt_data_write and evt_unknown. Packets with data errors (either dataLength mismatches or dataCRC errors) and packets for which a FIFO overrun occurred are 'backed-out' as described in section 10.2.1. For packet-per-buffer mode, possible values are: ack_complete, ack_data_error, evt_long_packet, evt_overrun, evt_descriptor_read, evt_data_write and evt_unknown. See Table 3-2, "Packet event codes," for descriptions and values for these codes.

The cycleMatchEnable bit is used to start an IR DMA context program on a specified cycle. When the cycleStart packet's low order two bits of cycleSeconds and 13-bit cycleCount values match the 15-bit cycleMatch value (in the IR contextMatch register), hardware sets the cycleMatchEnable bit to 0, sets the ContextControl.*active* bit to 1, and begins executing descriptor blocks for the context. The transition of an IR DMA context to the active state, from the not-active state is dependent upon the values of the run and cycleMatchEnable bits.

- If run transitions to 1 when cycleMatchEnable is 0, then the context will become active (active = 1).
- If both run and cycleMatchEnable are set to 1, then the context will become active when the cycleStart packet's low order two bits of cycleSeconds and 13-bit cycleCount values match the 15-bit cycleMatch value indicated in the IR contextMatch register.
- If both run and cycleMatchEnable are set to 1, and cycleMatchEnable is subsequently cleared, the context becomes active.
- If both run and active are 1 (the context is active), and then cycleMatchEnable is set to 1, this will result in unspecified behavior.

10.3.3 Isochronous receive contextMatch register

The IR ContextMatch register is used to start a context running on a specified cycle number, to filter incoming isochronous packets based on tag values and to wait for packets with a specified sync value. All packets are checked for a matching tag value, and a compare on sync is only performed when the descriptor's *w* field is set to 2'b11. See section 10.1 for proper usage of the *w* field. This register should only be written when ContextControl.*active* is 0, otherwise unspecified behavior will result.

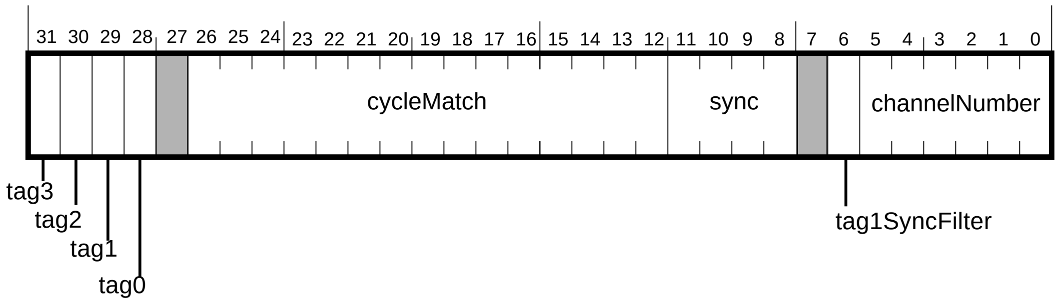


Figure 10-6 — IR DMA ContextMatch register format

Table 10-4 — IR DMA ContextMatch register description

field	rwu	reset	description
tag3	rw	undef	If set, this context will match on isochronous receive packets with a tag field of 2'b11.
tag2	rw	undef	If set, this context will match on isochronous receive packets with a tag field of 2'b10.
tag1	rw	undef	If set, this context will match on isochronous receive packets with a tag field of 2'b01.
tag0	rw	undef	If set, this context will match on isochronous receive packets with a tag field of 2'b00.
cycleMatch	rw	undef	Contains a 15-bit value, corresponding to the low order two bits of cycleSeconds and the 13-bit cycleCount field in the cycleStart packet. If ContextControl.*cycleMatchEnable* is set, then this IR DMA context will become enabled for receives when the two low order bits of the bus cycleTime.*cycleSeconds* and cycleTime.*cycleCount* values equal the cycleMatch value.
sync	rw	undef	This field contains the 4 bit field which is compared to the sync field of each isochronous packet for this channel when the command descriptor's *w* field is set to 2'b11.
tag1SyncFilter	rw	undef	If set and the contextMatch.*tag1* bit is set, then packets with tag 2'b01 shall only be accepted into the context if the two most-significant bits of the packet's sync field are 2'b00. Packets with tag values other than 2'b01 shall be filtered according to the tag0, tag2 and tag3 bits above with no additional restrictions. If clear, this context will match on isochronous receive packets as specified in the tag0-3 bits above with no additional restrictions.
channelNumber	rw	undef	This six bit field indicates the isochronous channel number for which this IR DMA context will accept packets.

At least one tag bit must be set to 1, otherwise no received packets will match and the context will, in effect, wait forever.

10.4 Isochronous receive DMA controller

The following sections describe how software manages the multiple isochronous receive DMA contexts. Each context has a CommandPtr pointing to the initial DMA descriptor, a ContextControl register, and a contextMatch register to start the context based on a cycle number and to filter packets. The IR DMA controller has one set of IRMultiChanMask registers used to specify a set of isochronous channels for the single isochronous context in multiChanMode.

10.4.1 Isochronous receive multi-channel support

Any IR DMA context can receive packets from multiple isochronous channels per cycle by enabling ContextControl.*multiChanMode* and using the IRMultiChanMask registers. There is a single set of IRMultiChanMask registers available in the IR DMA controller, and only **one** IR DMA context may be using them at any given time as determined by the setting of ContextControl.*multiChanMode* bit (see section section 10.3.2).

A context to be enabled for multiChanMode, <u>must also</u> be enabled for bufferFill and isochHeader modes. If multiChanMode is enabled without bufferFill and isochHeader, the resulting behavior is undefined.

If an IR DMA context is in multi-channel mode, therefore using the IRMultiChanMask registers, the isochronous channel field in the IR DMA context Match register (section 10.3.3) is ignored.

10.4.1.1 IRMultiChanMask registers (set and clear)

An isochronous channel mask is used to enable packet receives from up to 64 specified isochronous data channels. Software enables receives for any number of isoch channels by writing ones to the corresponding bits in the IRMultiChanMaskHiSet and IRMultiChanMaskLoSet addresses. To disable receives for any isoch channels, software writes ones to the corresponding bits in the IRMultiChanMaskHiClear and IRMultiChanMaskLoClear addresses.

A read of each IRChanMask register shows which channels are enabled; a one for enabled, a zero for disabled. The IRMultiChanMask registers are not changed by a bus reset. The state of these registers is undefined following a hard reset or soft reset.

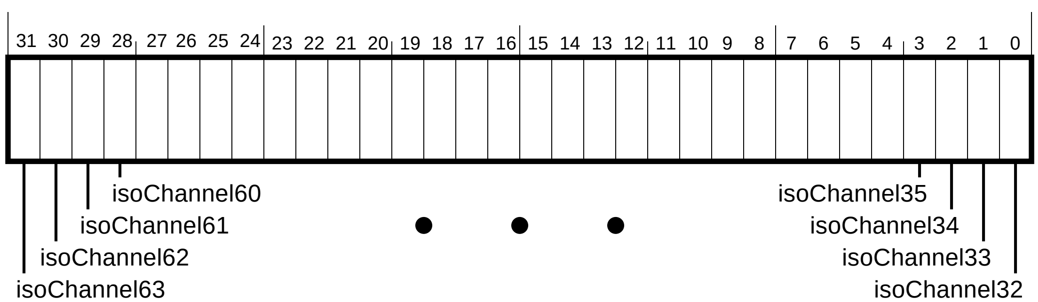


Figure 10-7 — IRMultiChanMaskHi (set and clear) register

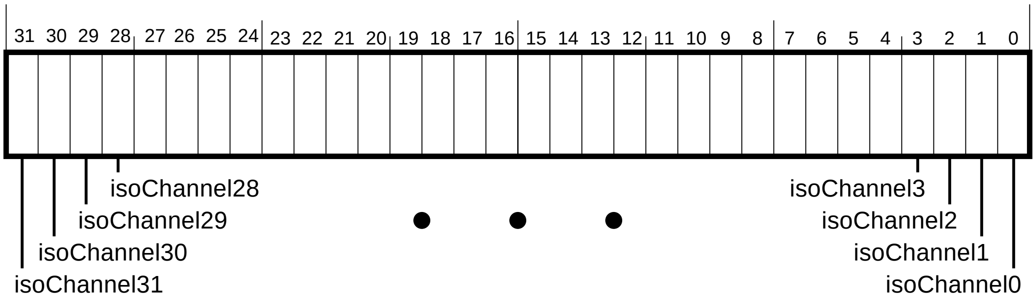


Figure 10-8 — IRMultiChanMaskLo (set and clear) register

10.4.2 Isochronous receive single-channel support

Each isochronous receive DMA context can receive one packet per cycle from one isochronous data channel. Data chaining across DMA context commands is supported when the ContextControl.*bufferFill* bit is set.

To configure a context to receive packets from an isochronous channel, write the channel number into the contextMatch register's channelNumber field.

To start a context on a particular cycle, write the starting cycle time into the ContextMatch register, and enable the ContextControl.*cycleMatchEnable* and ContextControl.*run* bits. When the low order two bits of the bus CycleTime.*cycleSeconds* and CycleTime.*cycleCount* values equal the ContextMatch.*cycleMatch* value, the IR DMA controller will clear the ContextControl.*cycleMatchEnable* bit and the context will begin receiving packets. (see sections 10.3.2 and 10.3.3).

To wait for a packet with specified sync value in the isochronous packet header, set the desired configuration in the sync field of the ContextMatch register and set the DMA command descriptor's *w* (wait) field to 2'b11. When the IR DMA controller detects a *w* field of 2'b11, it waits until a packet arrives matching the specified sync and directs it to the buffer identified in the waiting descriptor's dataAddress field. Packets with the specified channel number and tag bits but which do not match the specified sync are discarded.

When an IR DMA context is stopped either because it reached the end of the context program or because the run bit is cleared, some packets following the intended stop point may have already entered the receive FIFO. These packets will be discarded when they reach the bottom of the FIFO, unless another IR DMA context is able to receive them.

10.4.3 Duplicate channels

If more than one IR DMA context specifies receives for packets from the same isochronous channel, the context destination for that channel's packets is undefined.

If more than one IR DMA context has the ContextControl.*multiChanMode* bit set, then the context destination for IRmultiChanMask packets is undefined.

If an isochronous channel is specified both in a single channel context and in the multiChannel context, then the packet will be routed to the multiChannel context and the single channel context shall remain active.

10.4.4 Determining the number of implemented IR DMA contexts

The number of supported isochronous receive DMA contexts will vary for 1394 OpenHCI implementations from a minimum of four to a maximum of 32. Software can determine the number of supported IR DMA contexts by writing 32'hFFFF_FFFF to the isoRecvtIntMask register (see section 6.4.1), and then reading it back. Bits returned as 1's indicate supported contexts, and bits returned as 0's indicate unsupported/unimplemented contexts.

10.5 IR Interrupts

Each of the possible 32 isochronous receive contexts can generate an interrupt, therefore each IR DMA context has a bit in the isoRecvIntEvent register. Software can enable interrupts on a per-context basis by setting the corresponding isoRecvIntMask bit to one.

To efficiently handle interrupts which could conceivably be generated from 32 different contexts in close proximity to one another, there is a single bit for all IR DMA contexts in the Host Controller IntEvent register. This bit signifies that at least one but potentially several IR DMA contexts attempted to generate an interrupt. Software can read the isoRecvIntEvent register to find out which context(s) are involved. For more information on the isoRecvIntEvent register, see section 6.4.

10.5.1 cycleInconsistent Interrupt

When the IntEvent.*cycleInconsistent* condition occurs (table 6-1), the IR DMA controller shall continue processing running contexts normally, with the exception that contexts with the ContextControl.*cycleMatchEnable* bit set will remain inactive and cycleMatch processing shall be, in effect, disabled. To re-enable cycleMatch processing, software must first stop the IR contexts for which cycleMatch is enabled (by clearing ContextControl.*run* to 0 and waiting for ContextControl.*active* to go to 0), then must clear the IntEvent.*cycleInconsistent* interrupt. The stopped IR contexts may then be started.

10.5.2 busReset Interrupt

Bus reset does not affect isochronous receive.

10.6 IR Data Formats

The Host Controller shall only receive packets which have tcodes that are defined by an approved IEEE 1394 standard. Packets with undefined tcodes will be dropped.

There are four formats for isochronous receive packets depending upon the setting of the ContextControl.*isochHeader* and ContextControl.*bufferFill* bits (see section 10.3). If the ContextControl.*isochHeader* bit is zero, then only the isochronous data without any padding, header quadlet or timestamp quadlet is put in the buffer.

Table 10-5 — Isochronous receive fields

field name	bits	description
dataLength	16	Indicates the number of bytes in this packet.
tag	2	The data format of the isochronous data (see IEEE 1394 specification)
chanNum	6	The channel number this data is associated with.
tcode	4	The transaction code as received for this packet.
sy	4	Transaction layer specific synchronization bits.

Table 10-5 — Isochronous receive fields

field name	bits	description
isochronous data		The data received with this packet. The first byte of data must appear in the leftmost byte of the first quadlet of this field. The last quadlet should be padded with zeroes, if necessary.
padding		If the dataLength mod 4 is not zero, then zero-value bytes have been added onto the end of the packet to guarantee that a whole number of quadlets was sent. In three formats, the pad bytes are stripped off the packet.
xferStatus	16	Contains bits [15:0] from the ContextControl register.
timeStamp	16	The time at which this packet was received into the link, specified by the three low order bits of cycleSeconds, and the full 13-bits of cycleCount from the most recently received (or sent) cycle start packet.

10.6.1 bufferFill mode formats

10.6.1.1 IR with header/trailer

The format of an isochronous receive packet when ContextControl.*bufferFill*=1 and ContextControl.*isochHeader*=1 is shown below.

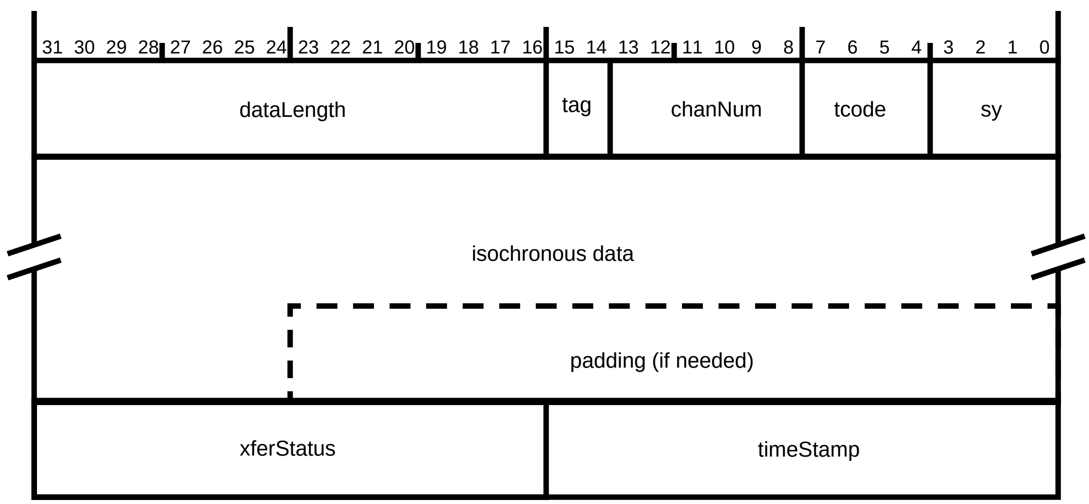


Figure 10-9 — Receive isochronous format in bufferFill mode with header/trailer

10.6.1.2 IR without header/trailer

The format of the isochronous receive packet when ContextControl.*bufferFill*=1 and ContextControl.*isochHeader*=0 is shown below.

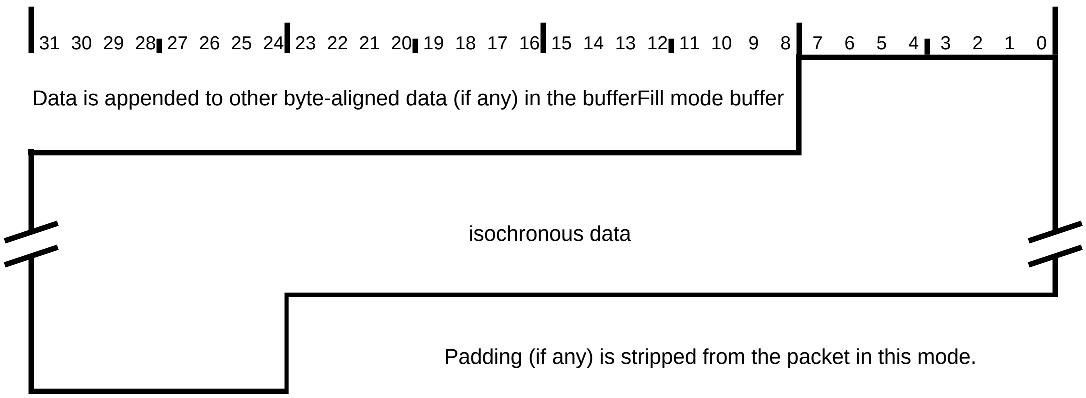


Figure 10-10 — Receive isochronous format in bufferFill mode without header/trailer

10.6.2 packet-per-buffer mode formats

10.6.2.1 IR with header/trailer

The format of an isochronous receive packet when ContextControl.*bufferFill*=0 and ContextControl.*isochHeader*=1 is shown below. Note that although xferStatus may be written as a side-effect of writing timeStamp, xferStatus does not contain valid or otherwise useful values.

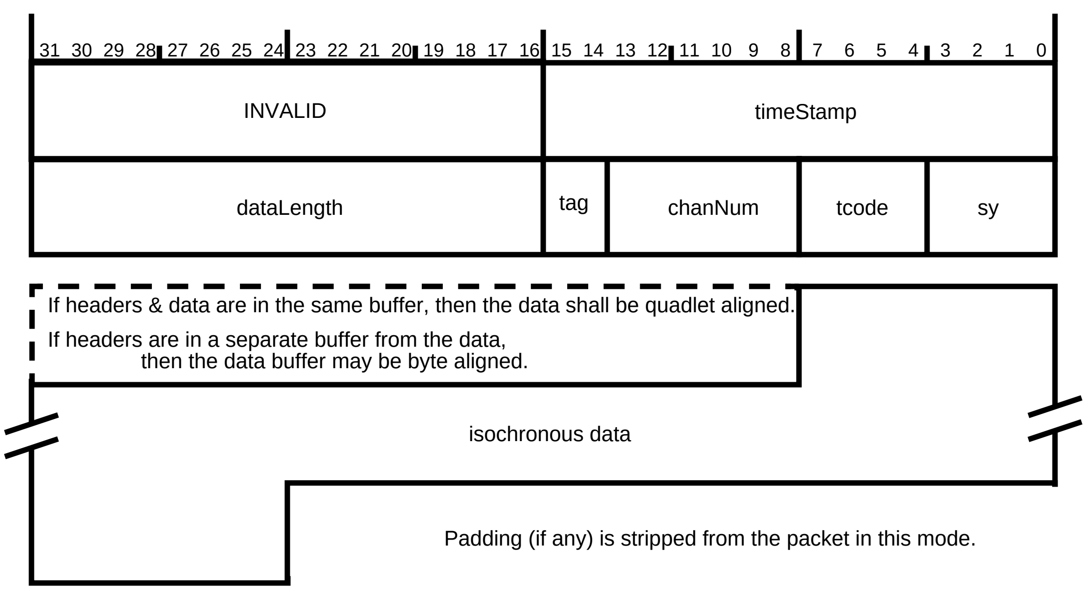


Figure 10-11 — Receive isochronous format in packet-per-buffer mode with header/trailer

10.6.3 IR without header/trailer

The format of the isochronous receive packet when ContextControl.*bufferFill*=0 and ContextControl.*isochHeader*=0 is shown below.

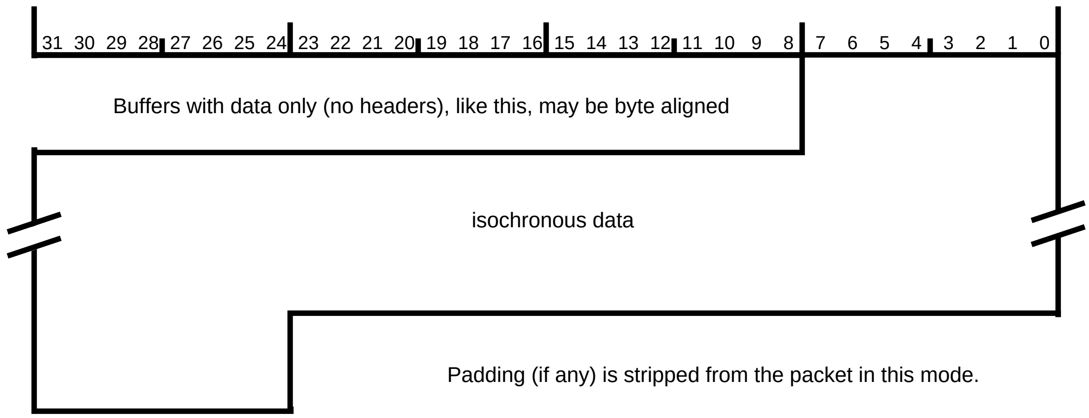


Figure 10-12 — Receive isochronous format in packet-per-buffer mode without header/trailer

11. Self ID Receive

The purpose of the SelfID DMA controller is to receive self ID packets during the bus initialization process. The self ID packets are received using a special pair of DMA registers, the Self ID Buffer Pointer register and the Self ID Count register.

11.1 Self ID Buffer Pointer Register

The Self ID Buffer Pointer register points to the buffer the SelfID packets will be DMA'ed into during bus initialization.

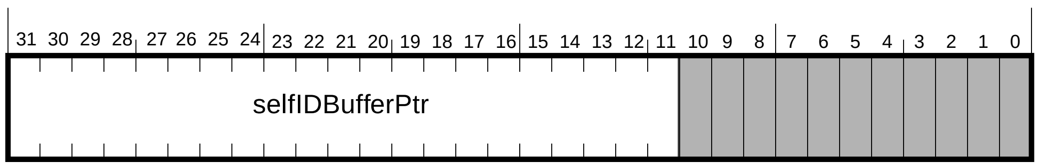


Figure 11-1 — Self ID Buffer Pointer register

Table 11-1 — Self ID Buffer Pointer register

field name	rwu	reset	description
selfIDBufferPtr	rw	undef	Contains the 2K-byte aligned base address of the buffer in host memory where received self-ID packets are stored. The contents of this field are undefined after a chip reset.

11.2 Self ID Count Register

This register keeps a count of the number of times the bus self ID process has occurred, flags self ID packet errors and keeps a count of the amount of self ID data in the Self ID buffer.

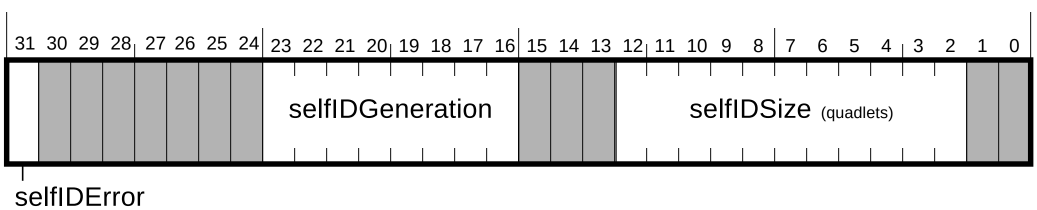


Figure 11-2 — Self ID Count register

Table 11-2 — Self ID Count register

field name	rwu	reset	description
selfIDError	ru	undef	When this bit is one, an error was detected during the most recent self ID packet reception. The contents of the self ID buffer are undefined. This bit is cleared after a self ID reception in which no errors are detected. Note that an error can be a hardware error or a host bus write error.
selfIDGeneration	ru	undef	The value in this field increments each time a bus reset is detected. This field rolls over to 0 after reaching 255.
selfIDSize	ru	undef	This field indicates the number of quadlets that have been written into the selfID buffer for the current selfIDGeneration. This includes the header quadlet and the selfID data. This field is cleared to zero as soon as a bus reset is detected.

The self ID stream can be (63 devices) * (4 packets/device) * (8 bytes/packet) = 2016 bytes. If a bus reset is received part way through a self ID sequence, the old data will be overwritten. To keep things straight, the generation counter is written into memory as the first quadlet of the stream. For a consistent stream, software reads the generation counter in memory, then the stream, then the SelfIDCount register. If the generation counter in the register matches the one in memory, then the self ID stream is consistent.

If the selfIDError flag is set, then there was either a hardware error in receiving the last self ID sequence or a host bus error while writing to the host buffer, so the self ID data is not trustworthy. Any self ID data received after the error is flushed. If all 2048 bytes are received, the selfIDSize field is set to 9’h7FF and the selfIDError flag is set. (This is only possible if >64 nodes are on the bus... a gross error condition.)

Whenever a bus reset occurs, the Host Controller clears the selfIDSize field to zero, at the same time the bus reset interrupt is triggered. This allows software responding to a bus reset to know that self IDs have not yet been received.

The Host Controller does not verify the integrity of the self-ID packets and software is responsible for performing this function (i.e., using the logical inverse quadlet).

11.3 Self-ID receive

The self-ID receive format is shown below. The first quadlet contains the time stamp and the self ID generation number (see section 11.2 “Self ID Count Register”). The remaining quadlets contain data that is received from the time a bus reset ends to the first subaction gap. This is the concatenation of all the self-ID packets received. Note that the bit-inverted check quadlets are included in the FIFO and must be checked by the application.

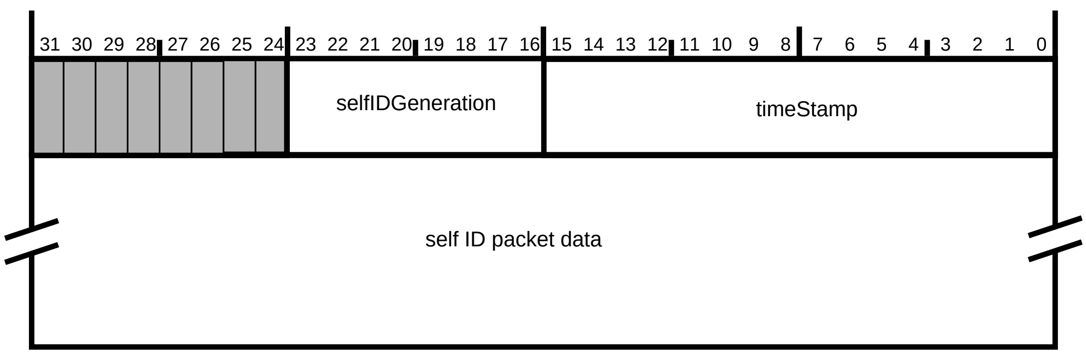


Figure 11-3 — Self-ID receive format

Table 11-3 — Self-ID receive fields

field name	description
selfIDGeneration	See table 11-2.
timeStamp	The three low order bits from cycleTimer.*cycleSeconds*, and the full 13-bits of cycleTimer.*cycleCount* at the time this status quadlet was generated.
self ID packet data	The data received during the selfID process of the bus initialization phase. Note that each selfID packet includes the data quadlet and inverted quadlet.

11.4 Enabling the SelfID DMA

The RcvSelfID bit in the LinkControl register (see section 5.9, “LinkControl registers (set and clear),”) allows the receiver to accept incoming self-identification packets. Before setting this bit, software must ensure that the self ID buffer pointer register contains a valid address.

11.5 Interrupt Considerations for SelfID DMA

IntEvent.*SelfIDcomplete* (section 6.1) is set when the selfID phase of bus initialization completes.

11.6 SelfIDs Received Outside of Bus Initialization

SelfID packets received outside of the bus initialization self-ID phase are routed to the AR DMA Request context and use the PHY packet receive format.

12. Physical Requests

When a block or quadlet read request or a block or quadlet write request is received, the 1394 Open HCI chip handles the operation automatically without involving software if the offset address in the request packet header meets a specific set of criteria listed below. Requests that do not meet these criteria are directed to the AR DMA Request context unless otherwise specified. Host Controller registers which are written via physical access to the Host Controller will yield unspecified results.

The 1394 Open HCI checks to see if the offset address in the request packet header is one of the following.

a) If the offset falls within the *physical range*, then the offset address is used as the memory address for the block or quadlet transaction. Physical range is defined by offsets inclusively between a lower bound of 48’h0 and an upper bound of either the PhysicalUpperBound offset minus one (section 5.14), or 48’h0000_FFFF_FFFF if the PhysicalUpperBound register is not implemented. If the high order 16-bits of the offset address is 16’h0000 and PhysicalUpperBound is not implemented, then the lower 32 bits of the offset address are used as the memory address for the block or quadlet transaction.

   Lock transactions and block transactions with a non-zero extended tcode are not supported in this address space, instead they are diverted to the AR DMA Request context. For read requests, the information needed to formulate the response packet is passed to the Physical Response Unit. Requests are only accepted if the source node ID of the request has a corresponding bit in the Asynchronous Request Filter registers and Physical Request Filter registers(section 5.13).

b) If the offset address selects one of the following addresses, the physical request unit will directly handle quadlet compare-swaps and quadlet reads. Other requests shall be sent an ack_type_error. (See section 5.5.1.)
   1) BUS_MANAGER_ID (48’hFFFFF000021C). Local register is BusManagerID.
   2) BANDWIDTH_AVAILABLE (48’hFFFFF0000220). Local register is BandwidthAvailable.
   3) CHANNELS_AVAILABLE_HI (48’hFFFFF0000224). Local register is ChannelsAvailableHi.
   4) CHANNELS_AVAILABLE_LO (48’hFFFFF0000228). Local register is ChannelsAvailableLo.

c) If the offset address is one of the following addresses, the Physical Request controller will directly handle quadlet reads. Other requests shall be sent an ack_type_error.
   1) Config ROM header (1st quadlet of the Config ROM) (48’hFFFFF0000400). Local register is ConfigROMheader (section 5.5.2).
   2) Bus ID (1st quadlet of the Bus_Info_Block) (48’hFFFFF0000404). Local register is BusID (section 5.5.3).
   3) Bus options (2nd quadlet of the Bus_Info_Block) (48’hFFFFF0000408). Local register is BusOptions (section 5.5.4).
   4) Global unique ID (3rd and 4th quadlets of the Bus_Info_Block) (48’hFFFFF000040C and 48’hFFFFF0000410). Local registers are GlobalIDHi and GlobalIDLo (section 5.5.5).
   5) Configuration ROM (48’hFFFFF0000414 to 48’hFFFFF00007FF). Mapped by the ConfigROMmapping register to a 1K byte block of system memory (section 5.5.6)

For information about ack codes for write requests, see section 3.3.2.

12.1 Filtering Physical Requests

Software can control from which nodes it will receive packets by utilizing the asynchronous filter registers. There are two registers, one for filtering out all requests from a specified set of nodes (AsynchronousRequestFilter register) and one for filtering out physical requests from a specified set of nodes (PhysicalRequestFilter register). The settings in both registers have a direct impact on how the AR DMA Request context is used, e.g., disabling only physical receives from a node will cause all request packets from that node to be routed to the AR DMA Request context. The usage and interrelationship between these registers is fully described in section 5.13, “Asynchronous Request Filters.”

12.2 Posted Writes

For write requests which are handled by the Physical Request controller, the Host Controller may send an ack_complete before the data is actually written to system memory. These writes are referred to as *posted writes*. Since posted writes impact the Physical Request controller and the Asynchronous Receive Request DMA context, further information about posted writes is located in section 3.3.3, “Posted Writes.” Information on host bus error handling of posted writes is provided in section 13.2.8, “Posted Write Error.”

12.3 Physical Responses

The response packet generated for a physical read, non-posted write, and lock request shall contain the transaction label as it appeared in the request, the destination_ID as provided in the request’s source_ID, and shall be transmitted at the speed at which the request was received. The source bus ID in the response packet shall be equal to the destination bus ID from the original request; note that this is not necessarily the same as the contents of the busNumber field in the Open HCI Node ID register.

Unlike AR Response packets, physical responses do not track a SPLIT_TIMEOUT expiration time.

12.4 Physical Response Retries

There is a separate nibble-wide MaxPhysRespRetries field in the ATRetries Register (see section 5.4) that tells the Physical Response Unit how many times to attempt to retry the transmit operation for the response packet when an ack_busy* or ack_data_error is received from the target node. If the retry count expires, the packet is dropped and software is *not* notified.

12.5 Interrupt Considerations for Physical Requests

Physical read request handling does not cause an interrupt to be generated under any circumstances. Physical write requests will generate an interrupt when posted write processing yields an error. Lock requests to the serial bus registers will generate an interrupt when the Host Controller is unable to deliver a lock response packet.

12.6 Bus Reset

On a bus reset, all pending physical requests (those for which ack_pending was sent) shall be discarded. Following a bus reset, only physical requests to the autonomous CSR resources (see section 5.5) can be handled immediately. Other physical requests may be processed after software initializes the filter registers (section 5.13).

13. Host Bus Errors

OpenHCI has three primary goals when dealing with host bus error conditions:

1) continue transmission and/or reception on all contexts not involved in the error;
2) provide information to software which is sufficient to allow recovery from the error when possible;
3) provide a means of error recovery on a context other than a general chip reset.

13.1 Causes of Host Bus Errors

Host bus errors can generally be classified as one of the following:

1) addressing error (e.g., non-existent memory location)
2) operation error (e.g., attempt to write to read-only memory)
3) data transfer error (e.g., parity or unrecoverable ECC) and
4) time out (e.g., reply on split transaction bus was not received in time).

Each of these errors can occur at three identifiable stages in the processing of a descriptor:

1) descriptor fetch,
2) data transfer (read or write), and
3) an optional descriptor status update.

In general, the nature of the bus error is not as significant as the stage of descriptor processing in which is occurs. For example, the difference between an addressing error and a data parity error is not significant to the error processing.

13.2 Host Controller Actions When Host Bus Error Occurs

When a host bus error occurs, the Host Controller performs a defined set of actions for all context types. Additionally, there are a set of actions that are performed that are dependent on the context type. The following sections outline these actions.

13.2.1 Descriptor Read Error

When an error occurs during the reading of a descriptor or descriptor block, the behavior of the Host Controller is the same regardless of the context type. The Host Controller will set ContextControl.*dead* and ContextControl.*event* will be set to evt_descriptor_read to indicate that the descriptor fetch failed. The unrecoverable error IntEvent is generated and the context's IntEvent is not set. Additionally, CommandPtr will be set to point to a descriptor within the descriptor block in which the error occurred. Since the descriptor could not be read, its xferStatus and resCount will not be written with current values, and software must refer to ContextControl.*event* for the status.

13.2.2 xferStatus Write Error

For any type of context, when the Host Controller encounters an error writing the status to a descriptor, it sets ContextControl.*dead*. The values that would have been written to xferStatus of a descriptor are retained in ContextControl for inspection by system software. The unrecoverable error IntEvent is generated and the context's IntEvent is not set regardless of the setting of the interrupt (I) field in the descriptor. Additionally, CommandPtr will be set to point to a descriptor within the descriptor block in which the error occurred.

13.2.3 Transmit Data Read Error

For asynchronous request transmit, asynchronous response transmit and isochronous transmit the Host Controller handles system data read errors in a similar manner. The Host Controller will not stop processing for the context. Instead, the event code in the status of the OUTPUT_LAST* descriptor is set to indicate that there was an error and the nature of the error. The indicated errors are evt_data_read or evt_underrun. If the error occurs before a packet's header is placed in the output FIFO, the Host Controller can immediately abort the packet transfer, optionally set the descriptor status to evt_data_read or evt_underrun and move on to the next descriptor block. If, however, the error occurs after the header has been placed in the output FIFO, the Host Controller will stop placing data in the output FIFO. This will cause the Host Controller to send a packet with a length that does not agree with the data_length field of the header. If the Host Controller receives an ack_data_error from the addressed node, then the Host Controller will substitute evt_data_read or evt_underrun as appropriate. If the device returns anything other than ack_data_error, then the Host Controller will store that value in the status for the packet. It should be noted that this means that if the addressed node returns an ack_pending on a block write, the error indication will be lost.

If the packet was a broadcast write, an isochronous packet, or an asynchronous stream packet, no ack code is received from any node. In this case, the Host Controller assumes that ack_data_error was received and proceeds as outlined above.

Note: Underruns which occur due to host bus latency shall not be construed to be host bus data errors, and as a result such asynchronous request and response packets may be retried as described in section 5.4.

13.2.4 Isochronous Transmit Data Write Error

A data write error can occur when the Host Controller attempts to write to the address indicated in a STORE_VALUE descriptor. This error is handled like a data read error with the exception that the event code is set to evt_data_write. The Host Controller may not begin placing the packet associated with a STORE_VALUE into the output FIFO until the STORE_VALUE operation is complete. This is to prevent the possibility of having multiple errors that cannot be properly reported to system software.

13.2.5 Asynchronous Receive DMA Data Write Error

When host bus error occurs while the Host Controller is attempting to write to either the request or response buffer, the Host Controller will set the corresponding ContextControl.*dead* and set ContextControl.*event* to evt_data_write. The unrecoverable error IntEvent is generated and the context's IntEvent is not set regardless of the setting of the interrupt (I) field in the descriptor. CommandPtr.*descriptorAddress* will point to the descriptor that contained the buffer descriptor for the memory address at which the error occurred. Any data in the input FIFO for the context is discarded.

13.2.6 Isochronous Receive Data Write Error

If a data write error occurs for a context that is in packet-per-buffer mode, the Host Controller will set ContextControl.*event* to evt_data_write or evt_overrun and conditionally update xferStatus of the descriptor in which the error occurred. Any remaining data in the input FIFO for the packet is discarded. The resCount value in a descriptor that has an error will not necessarily reflect the correct number of data bytes successfully written to memory. If a FIFO overrun occurs for a context that is in buffer-fill mode, the packet is treated as if a data length error had occurred and is 'backed out' of the receive buffer (xferStatus and resCount not updated) and the remainder of the packet is discarded from the input FIFO.

If a host bus error occurs for a context in buffer-fill mode the Host Controller will set ContextControl.*dead* and set ContextControl.*event* to evt_data_write. The unrecoverable error IntEvent is generated and the context's IntEvent is not set regardless of the setting of the interrupt (I) field in the descriptor. CommandPtr.*descriptorAddress* will point to the descriptor that contained the buffer descriptor for the memory address at which the error occurred. Any data in the input FIFO for the context is discarded.

13.2.7 Physical Read Error

When an external node does a physical access and the Host Controller's read of system memory fails, the Host Controller will return an error indication to the requester either by forming a response containing a response code of resp_data_error or by purposely truncating the response packet which forces a data_length mismatch at the requester. If the device replies with ack_busy or ack_data_error the host should retry the packet. If the error was caused by a FIFO underrun, the Host Controller will retry with the same response. If, however, the error was a host bus error, the response packet will be changed to resp_data_error.

13.2.8 Posted Write Error

Whether to be handled by the Physical Request controller or by the Asychronous Receive Request context, write requests to certain address ranges (see chapter 12., "Physical Requests,") may be acked with ack_complete before the data is actually written to system memory. Since the sending node has been notified that the action is complete, when the Host Controller cannot complete a *posted write* operation due to a host bus error the system must be notified so that software can recover.

If an error occurs in writing the posted data packet, then the Host Controller sets the IntEvent.*PostedWriteErr* bit to indicate that an error has occurred and the write remains pending. Software can then read the source node ID and offset address from PostedWriteAddressLo and PostedWriteAddressHi and then clear IntEvent.*PostedWriteErr*. When software clears IntEvent.*PostedWriteErr*, that write is no longer pending.

A Host Controller implementation is allowed to support any number of posted writes. However, for each posted write, there must be an error reporting register to hold the source node ID and offset address should that posted write fail.

If the Host Controller has as many pending physical writes as it has reporting registers additional physical writes may not be posted. Instead the Host Controller will need to return ack_pending and only return a complete indication when the write is actually done.

Although the Host Controller may allow several pending writes, error reporting is through a single pair of software visible registers. If multiple posted write failures have occurred, software will access them one at a time through the PostedWriteAddress registers. When software clears IntEvent.*PostedWriteErr*, this is a signal to the Host Controller that software has completed reading of the current contents of PostedWriteAddressLo/Hi and that the Host Controller can report another error by again setting IntEvent.*PostedWriteErr* and presenting a new set of values when software reads PostedWriteAddressLo/Hi.

13.2.8.1 PostedWriteAddress Register

If IntEvent.*postedWriteErr* is set, then these registers contain the 48 bits of the 1394 destination offset of the write request that resulted in a host bus error.

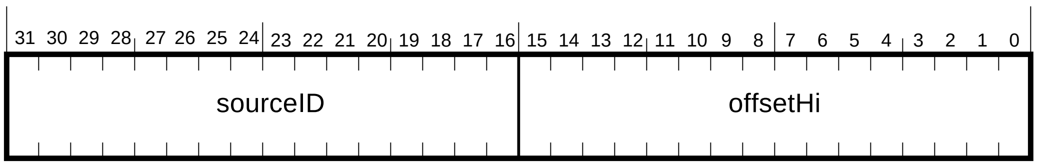


Figure 13-1 — PostedWriteAddressHi register

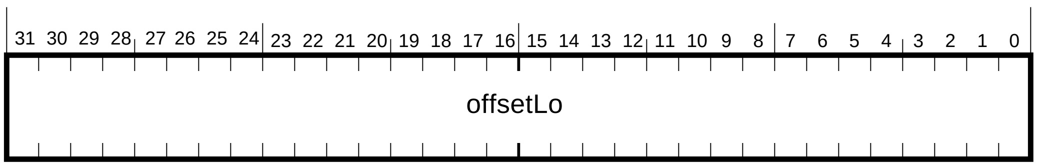


Figure 13-2 — PostedWriteAddressLo register

Table 13-1 — PostedWriteAddress register description

field name	rwu	reset	description
sourceID	ru	undef	The busNumber and nodeNumber of the node that issued the write request that failed.
offsetHi	ru	undef	The upper 16-bits of the 1394 destination offset of the write request that failed.
offsetLo	ru	undef	The low 32-bits of the 1394 destination offset of the write request that failed.

The PostedWriteAddress register is a 64-bit register which indicates the bus and node numbers (source ID) of the node that issued the write that failed, and the address that node attempted to access. The IntEvent.*PostedWriteErr* bit allows hardware to generate an interrupt when a write fails.

The PostedWriteAddress registers point to a queue in the Host Controller. This queue is accessed by software through the PostedWriteAddress registers. When a posted write fails, its address and node's source ID are placed in this queue, and the interrupt is generated. In addition, that packet is removed from the FIFO. By removing the packet from the FIFO, the Host Controller is not blocked from performing future transactions on the 1394 and host buses.

When software reads from these registers, that entry is removed from the queue, the next address and source ID are placed at the head of the queue, and another interrupt is generated. When the queue is empty, the Host Controller stops generating interrupts.

In order to guarantee the accuracy of the Posted Write error registers, software must perform the following algorithm when the posted write error interrupt is encountered:

1) Read the PostedWriteAddressHi register
2) Read the PostedWriteAddressLo register
3) Clear the IntEvent.*PostedWriteError* bit.

This will guarantee that software receives all information it requires about the first posted write, allowing another interrupt to be generated for future posted writes, and simplifies the Host Controller hardware. The Host Controller does not have to monitor that all three events occur before it moves to the next item in the queue. It may consider the information read once it sees the IntEvent.*PostedWriteError* bit cleared to 0.

13.2.8.2 Queue Rules

The Host Controller is only allowed to post as many writes as its posted write error queue is deep. For example, if the Host Controller has a queue depth of two, it shall only return “ack_complete” on two physical writes. All other physical writes must return either “ack_pending” or “ack_busy” event codes. Only when a previous posted write is successfully transferred into host memory, or when a posted write that resulted in an error is removed from the queue through the method described above by software, is the Host Controller allowed to accept more posted writes.

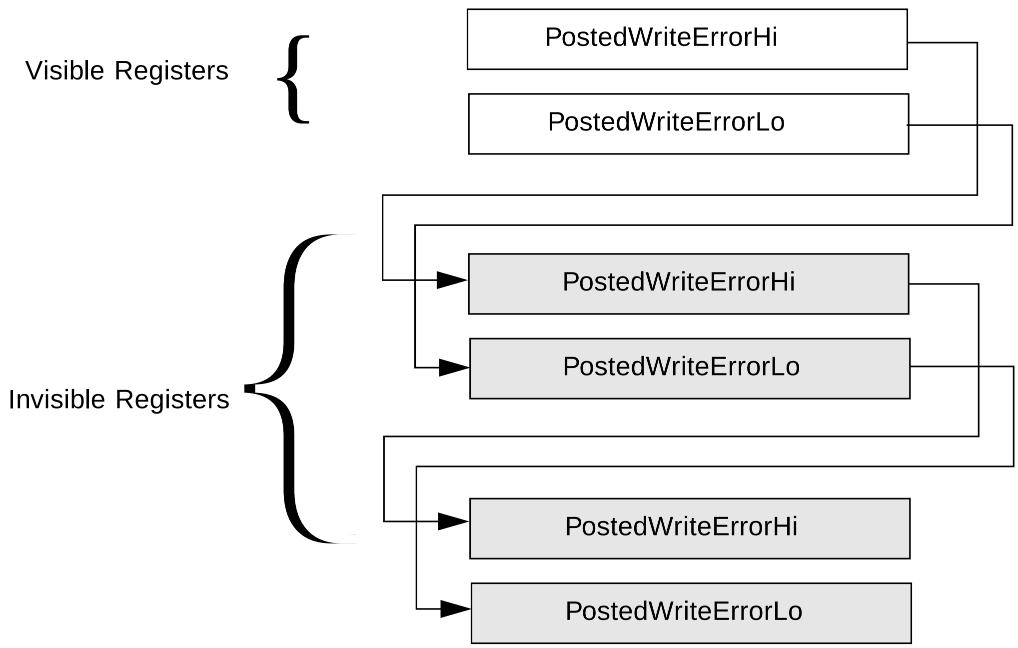


Figure 13-3 — Posted Write Error Queue

An example queue is shown in Figure 13-3. In this case, the queue is three entries deep, so this particular Host Controller can accept three posted writes.

Note that the Host Controller is not required to implement the posted write functionality at all. Software may enable posted writes, but the Host Controller will never accept posted writes. It will therefore never report a posted write error, and does not need to implement this queue.

However, posted writes represent a performance gain to the overall 1394 system. By accepting posted writes, the Host Controller and 1394 nodes are able to transfer data without excessive overhead on the 1394 bus. The 1394 Open HCI does not mandate that a certain level of posting be required, allowing individual hardware implementations to determine the posting depth based upon system needs.

Annex A. PCI Interface

A.1 PCI Configuration Space

OpenHCI's may be on any number of buses, this appendix only discusses their designs with PCI bus. This section describes the PCI requirements for IEEE 1394 Open Host Controller Interface compliant devices implemented using the PCI bus (abbreviated as OHC's herein). Only the registers and functions unique to a PCI-based OHC (basically, PCI configuration registers) are described in this appendix. OpenHCI compliant 1394 controllers must adhere to the requirements given in the PCI Local Bus Specification, Revision 2.1.

Typically, the PCI registers and expansion ROM are only accessed during boot-up and PCI device initialization. They are not typically accessed during runtime by device drivers. The PCI configuration registers, taken in total, are called the PCI configuration space. The PCI configuration space for OpenHCI is header type 0. Header type 8'h00 is the format for the device's configuration header region which is the first 16 dwords of PCI configuration space. Operational registers are memory mapped into PCI memory address space and pointed to by Base_Adr_0 register in the PCI configuration space. The operational registers are described in the body of this specification. PCI configuration space is not directly memory or I/O mapped - it's access is system dependent. Software reset issued through an OpenHCI control register does not affect the contents of the PCI configuration space.

A.2 Busmastering Requirements

The 1394 OpenHCI controller requires a bursting capable busmaster ability on the PCI bus. If the busmaster bit in the command register transitions from 1 to zero (see section A.3.1), the PCI logic supporting the OpenHCI controller logic must kill all DMA contexts.

A.3 PCI Configuration Space for 1394 OpenHCI With PCI Interface

Figure A-1 shows the PCI configuration space for a 1394 OpenHCI controller designed for PCI attachment. The format of this configuration space must be compliant with *PCI Local Bus Specification, Revision 2.1* (PCI Special Interest Group, 1995). Any registers not pointed to by the Base_Adr_0 (OHCI registers) pointer are vendor specific. Vendor specific registers must not be required for correct operation of the 1394 OpenHCI controller with a 1394 OpenHCI device driver.

Figure A-1 — PCI Configuration Space

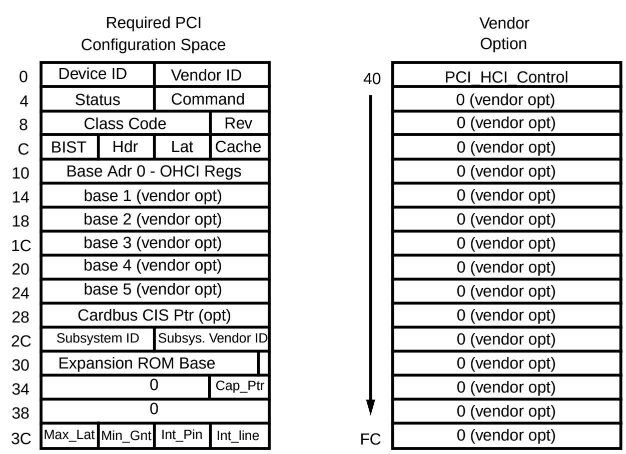
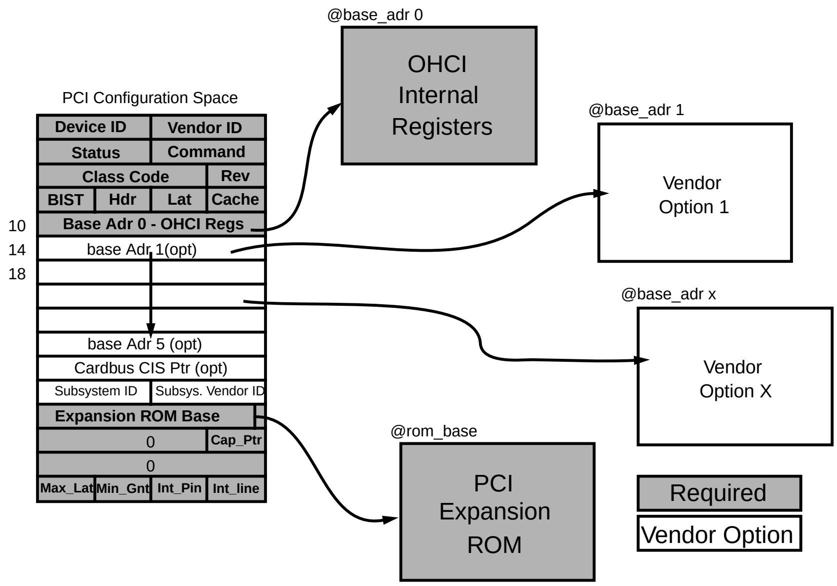


Figure A-2 shows the resources pointed to by the various Base_Adr registers and the Expansion ROM Base Address register.

Figure A-2 — Pointers to OHCI Resources in PCI Configuration Space



A.3.1 COMMAND Register

This register provides coarse control over the device's ability to generate and respond to PCI cycles. For the 1394 OpenHCI it is required that the Host Controller support both PCI bus-mastering and memory-mapping of all operational registers into the memory address space of the PC host. Consequently, the fields **MA** and **BM** should always be set to 1'b1 during device configuration.

Once the Host Controller starts processing DMA descriptor lists, the action of resetting either field **MA** or **BM** to 1'b0 will halt all PCI operations from the 1394 OHC. (Do this carefully). If the field **MA** is reset to 1'b0, the Host Controller can no longer respond to any software command addressed to it and interrupt generation is halted.

Table A-1 — COMMAND Register

Field	Bits	Read/ Write	Description
	0	rw	Refer to PCI Local Bus Specification, Revision 2.1, for definition
Memory Space	1	rw	**MEMORY SPACE** Set to 1'b1 so that the OpenHCI controller can respond to PCI memory cycles
BusMaster	2	rw	**BUS MASTER** Set to 1'b1 so that the OpenHCI controller can act as a bus-master
	3-5	rw	**Refer to PCI Specification, Revision 2.1, for definition**
Parity Error Response	6	rw	**Parity Error Response** Set to 1'b1 if error detection on the PCI bus is desired.
	7	rw	Refer to PCI Specification, Revision 2.1, for definition

A.3.2 STATUS Register

This register tracks the status of PCI bus-related events.

Table A-2 — STATUS Register

Field	Bits	Read/ Write	Description
	3-0	r	**Reserved.**
	4	r	**CAPABILITIES LIST** See *PCI Power Management Specification 0.99a*. May be 0 for motherboard-only OHCI controllers such as those integrated into a south bridge. The default value of this bit is 1.
-	15-5	-	**See** the *PCI Local Bus Specification, Revision 2.1*.

A.3.3 CLASS_CODE Register

This register identifies the basic function of the device, and a specific programming interface code for an 1394 OpenHCI-compliant Host Controller.

Table A-3 — CLASS_CODE Register

Field	Bits	Read/ Write	Description
PI	7-0	r	**PROGRAMMING INTERFACE** A constant value of 8’h10 Identifies the device being a 1394 OpenHCI Host Controller.
SC	15-8	r	**SUB CLASS** A constant value of 8’h00 Identifies the device being of IEEE 1394.
BC	23-16	r	**BASE CLASS** A constant value of 8’h0C Identifies the device being a serial bus controller.

A.3.4 Revision_ID Register

The Revision ID must contain the vendor’s revision level of their OpenHCI silicon. It is required that each new revision of silicon receive a new revision ID.

A.3.5 Base_Adr_0 Register

The Base_Adr_0 register specifies the base address of a contiguous memory space in the PCI memory space of the host. This memory space is assigned to the operational registers defined in this specification. All of the operational registers described in this document are directly mapped into the first 2 kilobyte of this memory space. Vendor unique registers are not allowed within the first 2 KB of this memory space.

Those hardware registers that are used to implement vendor specific features are not covered by this 1394 OpenHCI Specification. Additional vendor unique address spaces may be allocated by adding additional base address registers beginning at offset h14 in PCI configuration space.

Table A-4 — Base_Adr_0 Register

Field	Bits	Read/ Write	Description
IND	0	r	**MEMORY SPACE INDICATOR** A constant value of 1'b0 Indicates that the operational registers of the device are mapped into memory space of the main memory of the PC host system
TP	2-1	r	This bit must be programmed consistent with the *PCI Local Bus Specification, Revision 2.1*
PM	3	r	**PREFETCH MEMORY** A constant value of 1'b0 Indicates that there is no support for "prefetchable memory"
	X-4	rw	Default value of 0 and is read only. 10 <= X. Represents a minimum of 2-KB addressing space for the OpenHCI's operational registers.
OHCI_REG_PTR	31-(X+1)	rw	**OHCI Register Pointer** Specifies the upper bits of the 32-bit starting base address. This represents a minimum of 2-KB addressing space for the OpenHCI's operational registers. X > 10. If X is 11 the addressing space is 2KB, if 12 it's 4KB etc... On x86 systems which will be booting from a 1394 device, the BIOS may need to map this address range into the option ROM area below 1M. Requesting large blocks of address space using the register may result in a non-optimal system configuration.

A.3.6 CAP_PTR Register (opt)

This register is a pointer to a linked list of additional capabilities.

Table A-5 — CAP_PTR Register

Field	Bits	Read/ Write	Description
	7-0	r	**CAP_PTR** The CAP_PTR provides an offset into the function's PCI configuration space for the location of the first item in the Capabilities Linked List. The CAP_PTR offset is dWord aligned so the two least significant bits are always "8'h00." See the *PCI Power Management Specification 0.99a* for more details. This field only has meaning if bit 4 in the Status register is set.

A.4 PCI_HCI_Control Register

This register has 1394 OpenHCI specific control bits. Vendor options are not allowed in this register. It is reserved for OpenHCI use only.

Table A-6 — PCI_HCI_Control Register

Field	Bits	Read/ Write	Description
PCI_Global_Swap	0	rw	**PCI Global Swap Bit** When this bit is b1, all quadlets read from and written to the PCI interface are byte swapped. PCI addresses, such as expansion ROM and PCI config registers, are unaffected by this bit (they are not byte swapped under any circumstances). The hardware reset value of this bit is b0. This bit is not required for motherboard implementations.
	31-1	rw	These are reserved bits. They must be written as zeros and read as zeros.

A.5 PCI Expansion ROM for 1394 OpenHCI

1394 Open Host Controller's used on add-in adapters will clearly require PCI expansion ROMs that provide BIOS, Open Firmware, etc. to boot and configure the card. If this ROM is non-writable and soldered to the card (not socketed), it is also permitted that the serial ROM image which the Open Host Controller autoloads at boot up can be included in this expansion ROM (saving the cost of a serial ROM). If this is done, the serial ROM image must be loaded into the 1394 Open Host Controller by hardware state machine without software intervention or control. It cannot be modifiable by software or 1394 devices under any circumstances.

A.6 PCI Bus Errors

Any PCI bus error encountered must be reported to the OpenHCI operational logic for error handling. The nature of the error response is context dependent and discussed in the body of the document. No distinction is made between the various PCI bus errors. Basically, only one all encompassing error signal is provided to the operational logic by the PCI specific interface logic. It is the responsibility of the implementer to insure that PCI bus errors are reported in a timely fashion, consistent with their overall OpenHCI implementation, that insures that the errors are associated with the engine, context, etc. that the error should be posted to.

When the "Parity Error Response" bit in the Command Register in PCI Configuration Space is enabled (see section A.3.1), the PCI interface logic in the OpenHCI must assert PERR# in accordance with the *PCI Local Bus Specification, Revision 2.1* when data with bad parity is received by the 1394 OpenHCI controller.

Annex B. Summary of Register Reset Values (Informative)

The table below is a summary of all register reset values described in this document and is provided for convenience. In the event of a discrepancy between values shown in this table and the normative part of this document, the normative part of this document shall be considered correct.

All registers are shown below in address order. Refer to section 4.2, “Register Map,” for the complete list. Fields for each register are shown along with their values following a hardware reset, a software reset and a bus reset. Refer to section 2.1.2.3 for interpretation of reset values notation. All values for bus reset are N/A unless otherwise specified.

Table B-1 — Register Reset Summary

Register Fields	RESET			See clause(s)
	Hardware	Software	Bus	
Version				5.2
GUID_ROM	N/A			
version	N/A			
revision	N/A			
GUID_ROM				5.3
addrReset	undef			
rdStart	1’b0			
rdData	undef			
ATRetries				5.4
secondLimit	3’h0			
cycleLimit	13’h0			
maxPhysRespRetries	undef			
maxATRespRetries	undef			
maxATReqRetries	undef			
Bus Management CSR registers				5.5.1
BUS_MANAGER_ID	6’3F	undef	6’3F	
BANDWIDTH_AVAILABLE	13’h1333	undef	13’h1333	
CHANNELS_AVAILABLE_HI	32’h FFFF_FFFF	undef	32’h FFFF_FFFF	
CHANNELS_AVAILABLE_LO	32’h FFFF_FFFF	undef	32’h FFFF_FFFF	
CSRReadData	undef			5.5.1
CSRCompareData	undef			5.5.1
CSRControl				5.5.1
csrDone	1’b1			
csrSel	undef			
ConfigROMhdr				5.5.2
info_length	8’h00	N/A		
crc_length	8’h00	N/A		
rom_crc_value	16’h0000			

Table B-1 — Register Reset Summary

Register Fields	RESET			See clause(s)
	Hardware	Software	Bus	
BusID	N/A			5.5.3
BusOptions				5.5.4
irmc	undef			
cmc	undef			
isc	undef			
bmc	undef			
pmc	undef			
cyc_clk_acc	undef			
max_rec	max implemented	N/A		
g	undef			
link_spd	max link speed	undef		
GUIDHi				5.5.5
node_vendor_ID	24’b0	N/A		
chip_ID_hi	8’b0	N/A		
GUIDLo				5.5.5
chip_ID_lo	32’b0	N/A		
ConfigROMmap				5.5.6
configROMaddr	undef			
PostedWriteAddressLo				13.2.8.1
offsetLo	undef			
PostedWriteAddressHi				13.2.8.1
sourceID	undef			
offsetHi	undef			
VendorID				5.6
VendorUnique	N/A			
VendorCompanyID	N/A			
HCControl				5.7
noByteSwapData	undef			
programPhyEnable	** see table 5-12	N/A		
aPhyEnhanceEnable	** see table 5-12	N/A		
LPS	1’b0			
postedWriteEnable	undef			
linkEnable	1’b0			
softReset	**see table 5-12			

Table B-1 — Register Reset Summary

Register Fields	RESET			See clause(s)
	Hardware	Software	Bus	
SelfIDBuffer				11.1
selfIDBufferPtr	undef			
SelfIDCount				11.2
selfIDError	undef		*	
selfIDGeneration	undef		*	
selfIDSize	undef		9’b0 -> *	
IRMultiChanMaskHi **IRMultiChanMaskLo**				10.4.1.1
isoChannel*N*	undef			
IntEvent				6.1
selfIDcomplete	undef		1’b0	
busReset	undef		1’b1	
all other bits	undef			
IntMask				6.2
masterIntEnable	1’b0			
all other bits	undef			
IsoXmitIntEvent				6.3.1
isoXmit*N*	undef			
IsoXmitIntMask				6.3.2
isoXmit*N*	undef			
IsoRecvIntEvent				6.4.1
isoRecv*N*	undef			
IsoRecvIntMask				6.4.2
isoRecv*N*	undef			
FairnessControl				5.8
pri_req	undef	N/A		
LinkControl				5.9
cycleSource	undef			
cycleMaster	undef			
cycleTimerEnable	undef			
rcvPhyPkt	undef			
rcvSelfID	undef			

Table B-1 — Register Reset Summary

Register Fields	RESET Hardware	RESET Software	RESET Bus	See clause(s)
NodeID				5.10
iDValid	1’b0		1’b0 -> 1’b1	
root	1’b0		1’b1 (conditional)	
CPS	1’b0			
busNumber	10’h3FF		10’h3FF	
nodeNumber	undef		from phy	
PhyControl				5.11
rdDone	undef			
rdAddr	undef			
rdData	undef			
rdReg	1’b0			
wrReg	1’b0			
regAddr	undef			
wrData	undef			
Isochronous Cycle Timer				5.12
cycleSeconds	N/A			
cycleCount	N/A			
cycleOffset	N/A			
AsynchronousRequestFilterHi AsynchronousRequestFilterLo				5.13.1
asynReqResourceN	1’b0		1’b0	
asynReqResourceAll	1’b0			
PhysicalRequestFilterHi PhysicalRequestFilterLo				5.13.2
physReqResourceN	1’b0		1’b0	
physReqResourceAllBuses	1’b0		1’b0	
PhysicalUpperBound				5.14
physUpperBoundOffset	undef	N/A		
CommandPtr				3.1.2, 7.2.1, 8.3.1, 9.2.1, 10.3.1
descriptorAddress	undef			
Z	undef			

Table B-1 — Register Reset Summary

Register Fields	RESET			See clause(s)
	Hardware	Software	Bus	
AT Request ContextControl **AT Response ContextControl**				3.1, 7.2.2, 7.2.3
run	1'b0			
wake	undef			
dead	1'b0			
active	1'b0		1'b0	
event code	undef			
AR Request ContextControl **AR Response ContextControl**				3.1, 8.3.2
run	1'b0			
wake	undef			
dead	1'b0			
active	1'b0			
spd	undef			
event code	undef			
IT ContextControl				3.1, 9.2.2
cycleMatchEnable	undef			
cycleMatch	undef			
run	1'b0			
wake	undef			
dead	1'b0			
active	1'b0			
event code	undef			
IR ContextControl				3.1, 10.3.2
bufferFill	undef			
isochHeader	undef			
cycleMatchEnable	undef			
multiChanMode	undef			
run	1'b0			
wake	undef			
dead	1'b0			
active	1'b0			
spd	undef			
event code	undef			

Table B-1 — Register Reset Summary

Register Fields	RESET			See clause(s)
	Hardware	Software	Bus	
IR ContextMatch				10.3.3
tag3	undef			
tag2	undef			
tag1	undef			
tag0	undef			
cycleMatch	undef			
sync	undef			
tag1SyncFilter	undef			
channelNumber	undef			

Annex C. Summary of Bus Reset Behavior (Informative)

This section is a summary of Open HCI bus reset behavior. In the event of a discrepancy between information presented here and in the normative part of this document, the normative part of this document shall be considered correct.

C.1 Overview

Following a bus reset, node ID’s for nodes on the bus may have changed from the values they had been prior to the bus reset. Since asynchronous packets include a source and destination node ID, it is imperative that packets with *stale* node ID’s do not go out on the 1394 bus. Isochronous packets do not include any node ID information and therefore must be allowed to continue un-interrupted after a bus reset. To accomplish this behavior, several things must happen in real-time by the Open Host Controller when a bus reset occurs. The following sections describe bus reset behavior for each DMA type.

C.2 Asynchronous Transmit: Request & Response

While the bus reset interrupt, IntEvent.*busReset*, is active, the Host Controller will inhibit AT Request and AT Response transmits and flush all packets from the AT Request & AT Response FIFO(s). The host software must wait until both AT contexts are inactive (ContextControl.*active* == 0) before clearing the bus reset interrupt. Refer to sections 7.2.3.1 and 7.2.3.2 for more information.

C.3 Asynchronous Receive: Request & Response

Since all nodes are required to only transmit asynchronous packets that have node ID’s as they were assigned in the most recent bus reset/ Self ID process, AR Requests and AR Responses continue to be processed normally by the hardware. To assist software in determining which Request packets arrived before and after the bus reset, the Host Controller inserts a fabricated *bus reset packet* in the appropriate location in the receive queue. This way, packets which arrive in the receive buffer after the bus reset packet can be interpreted using the current node ID assignments.

Also upon detection of a bus reset the Host Controller will clear all bits in the Asynchronous Filter registers *except* for the Asynchronous Request Filter HI.*asynReqResourceAll* bit. If this bit is also 0, receipt of all asynchronous requests which do not reference the first 1K of CSR config ROM will be prevented and software is responsible for subsequently enabling the Asynchronous Filter registers as appropriate.

Refer to section 8.4.2.3 for information on the bus reset packet, and section 5.13 for information on the asynchronous filter registers.

C.4 Isochronous Transmit

A bus reset does not affect the transmission of isochronous packets, which continue being transmitted for their assigned channels. It is software’s responsibility to perform the necessary isochronous resource re-allocation and make any communication to the talker’s and/or receivers’ control registers.

C.5 Isochronous Receive

A bus reset does not affect the receipt of isochronous packets, which continue being received for their assigned channels. It is software’s responsibility to perform the necessary isochronous resource re-allocation and communicate as required to the talkers and/or receivers.

C.6 Self ID Receive

The receipt of self ID packets is part of the bus reset process. When a bus reset occurs, and the IntEvent.busReset bit is set, the IntEvent.*selfIDComplete* interrupt is cleared. Once the Self ID phase of bus initialization has completed the IntEvent.*selfIDComplete* is set to inform software that bus initialization self ID packets have been received. See Chapter 11.0 for further information.

C.7 Physical Requests/Responses

C.7.1 Physical Response

The Host Controller will flush all Physical Asynchronous Transmit Response packets from all asynchronous transmit FIFO's. The Physical AT Response engine will resume processing incoming requests which arrive following the bus reset.

C.7.2 Physical Requests

Posted write requests, that is, write requests for which ack_complete was sent but which have not yet been processed, will be processed normally.

All split transaction AR Requests are flushed until a bus reset boundary is detected. After the bus reset boundary, normal physical receive transactions are resumed.

In response to a bus reset, Host Controller clears the Physical Request Filter registers and physical handling of requests outside the first 1K of CSR config ROM is disabled. Software is responsible for subsequently enabling the Physical Request Filter registers as appropriate. See section 5.13.2 for further information.

C.8 Control Registers

In response to a bus reset, the NodeID.*iDValid* bit is cleared indicating that the Host Controller does not yet have a valid node ID, and therefore software must not enable asynchronous transmits. When the self ID phase of bus initialization has completed and the new Node ID has been determined, the PHY returns status which initializes NodeID.*nodeNumber* and the Host Controller sets NodeID.*iDValid* at which point software may restart asynchronous transmit.

A bus reset will also cause the Host Controller's Isochronous Resource Management registers to be reset. Refer to section 5.5.1 for further information.

Annex D. IT DMA Supplement (Informative)

The OpenHCI Isochronous Transmit DMA (IT DMA) is documented in Chapter 9.0. This Annex provides supplementary explanation and example, to aid in understanding the IT DMA. It is intended that this Annex will agree completely with Chapter 9.0. If there is any disagreement, this Annex is faulty, and the information in Chapter 9.0 overrides this Annex.

D.1 IT DMA Behavior

The flowcharts given in the next two sections illustrate the behavior of the IT DMA as documented in Chapter 9.0. These flowcharts are provided in order to help the reader visualize the end result of IT DMA operation, through a set of events that could occur within the IT DMA. These flowcharts do not specify the IT DMA algorithm, although they should yield the same output as that specified by Chapter 9.0. Furthermore, these flowcharts do not dictate an implementation strategy. The variables such as *M* and *N* do not necessarily correspond to OpenHCI registers. The presence of a task on the “Link side” flowchart or the “DMA side” flowchart does not mandate that the associated logic be implemented in any particular part of OpenHCI. Such distinctions also do not imply anything about clock domains, signal routing, or other implementation-specific aspects of an OpenHCI product.

D.2 IT DMA Flowchart Summary

The output of the IT DMA is illustrated in this Annex using two flowcharts. One flowchart represents activity that is likely to take place within the DMA engines of a particular OpenHCI. The other flowchart represents activity that is likely to take place in the Link (or “Link Core”) portion of a particular OpenHCI. These two flowcharts execute simultaneously, with no interdependencies other than those shown by the shared variables, and other shared state such as the local cycle timer or the cycle start value most recently received or sent. Note also that neither flowchart contains an exit or a stop condition. It is intended that both flowcharts begin execution at the same instant, and then remain in operation forever. In practice, the flowcharts might be restarted after a full chip reset, or other similar OpenHCI event.

The flowcharts do not attempt to capture every possible error condition, such as a dead condition in the IT DMA. Only the states required for ordinary IT DMA processing are shown, and the level of detail varies somewhat. In this sense, cycle loss and cycle match are considered normal IT DMA events. Bus resets are not specifically identified, but those that cause cycle loss will be handled by the flowchart algorithm.

Because the flowcharts do not mandate implementation details, they also do not necessarily show the most optimal way of implementing the IT DMA. For example, the detection of a cycle loss could possibly be performed with less delay, potentially giving the IT DMA more time to recover, thus improving the FIFO readiness for following cycles, and reducing the chance of further cycle losses. The presentation of these example flowcharts does not preclude a more efficient implementation, within the behavior specified in Chapter 9.0.

D.3 DMA-side IT DMA flowchart

The following flowchart shows logic for processing the DMA component of the IT DMA in a manner that (when coupled with the Link side shown below) agrees with that specified in Chapter 9.0.

The DMA-side flowchart has two major components. The top half consists of a loop that synchronizes the activity of the DMA side to the correct cycle number. This loop implements a two-cycle workahead. If the FIFO were arbitrarily large, this algorithm would always keep two cycles worth of packets in the FIFO, in addition to the packets for any cycle currently being transmitted. The bottom half consists of a loop for each of the IT DMA contexts. This loop processes one cycles worth of packets, either loading them all into the FIFO, or performing skip processing for all of them.

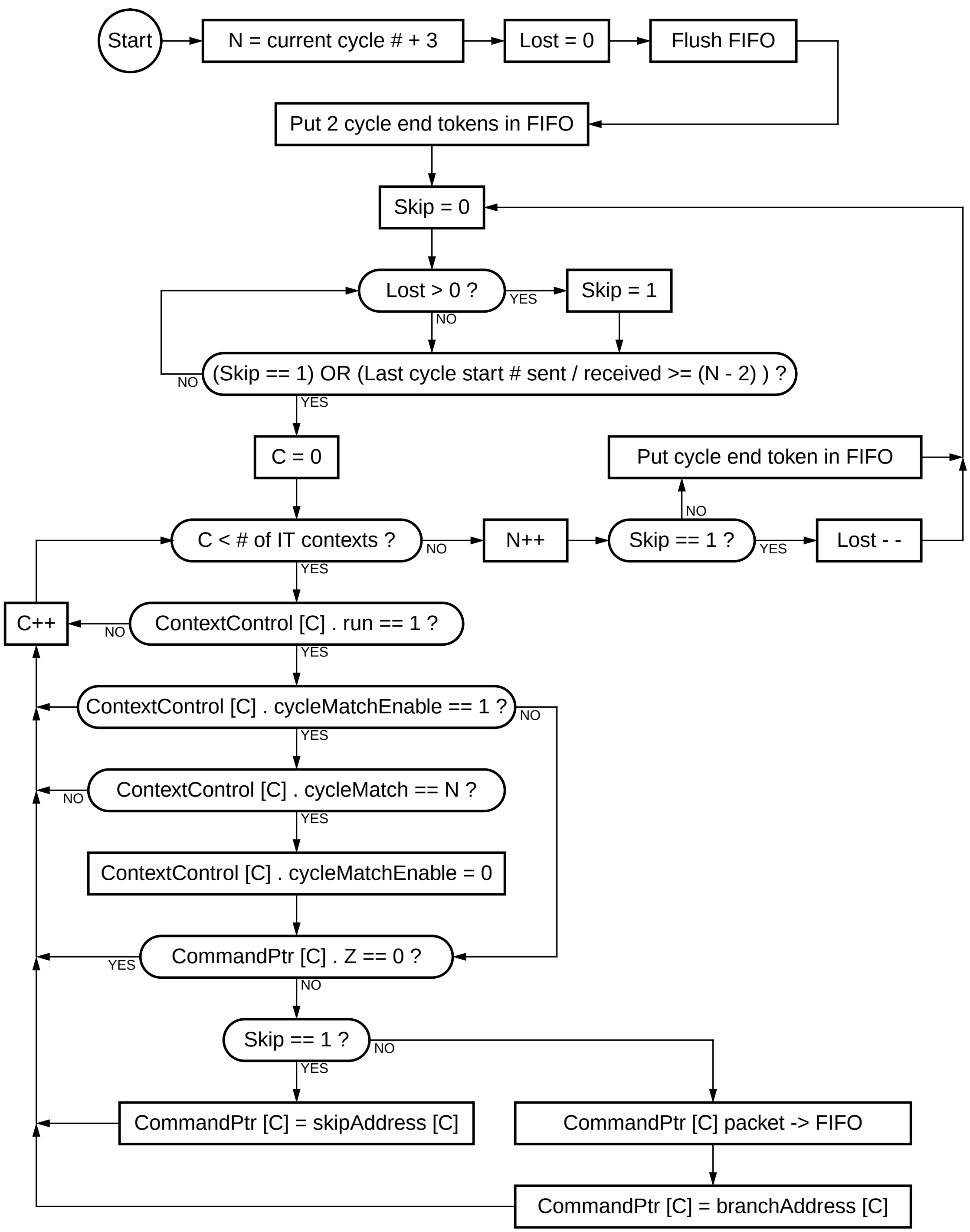


Figure D-1 — IT DMA DMA-Side Flowchart

A key point in understanding the DMA side flowchart is that neither the top loop nor the bottom loop necessarily corresponds to a single cycle of real time (although, on average, they do). For example, the top loop tries to coordinate two-cycle workahead. In most systems, the FIFO is likely to be too small for full two-cycle workahead. In fact, if the FIFO is smaller than the largest packet, there will be times when the workahead is zero cycles. The top loop acts as a gate - in the rare case that the DMA really achieves two cycles of workahead, the top loop will idle the DMA until there is more work to do. Similarly, the bottom loop may correspond to more than one cycle of real time. If, in the middle of transmitting a cycle, a cycle loss occurs, the bottom loop does not exit. It will continue to attempt to transmit the remaining packets for the original cycle, and will not exit until it does. This behavior agrees with Chapter 9.0, in that packets are never flushed to compensate for a cycle loss. Any packet already in the FIFO, or even potentially in the FIFO, will be transmitted (eventually).

D.3.1 DMA-side top half

The top half of the DMA-side flowchart regulates the IT DMA workahead, if any. The flowchart illustrated will attempt to maintain a two-cycle workahead. To do this, the algorithm communicates with the Link side in three ways. First, both sides share access to the local cycle timer and the most recent cycle start packet. Second, both sides share a variable called Lost, which is a count of the number of lost cycles that have not yet been handled. Finally, the two sides communicate through the IT FIFO. The DMA side places packets into the FIFO, and the Link side removes them. The DMA side also places end-of-cycle tokens in the FIFO, which are removed by the Link side. Many implementations are likely to also use an end-of-packet token. This flowchart does not show such tokens, and it does not prohibit them.

Because the DMA side wants to work two cycles ahead, when it first starts running it must hold off the Link side, so that it can try to put two cycles worth of packets in the FIFO. The DMA side immediately places two end-of-cycle tokens into the FIFO. The Link side will consume one end-of-cycle token per cycle, as detailed below, so these two tokens will hold off the Link side for two cycles, while the DMA side tries to work ahead.

The DMA side keeps a private variable N, to indicate the cycle number for which it wants to load packets into the FIFO. If the DMA side were always able to maintain two-cycle workahead, N would usually be two greater than the current cycle number. More likely, N will vary between zero and two greater than the current cycle number, depending on how much of the desired two-cycle workahead can actually fit into the FIFO. Because the flowchart is entered in the midst of some cycle, and it is too late to perform any IT DMA for that cycle, N is initialized to the current cycle number, plus three.

The DMA side also has a private variable called Skip. This variable is changed only between entries to the bottom-half loop, and it controls whether the bottom-half loop will attempt to transmit a cycles worth of packets, or apply skip processing to a cycles worth of packets.

The top-half loop acts as a gate to the bottom-half loop. The bottom-half can be entered for two reasons. First, the top-half can determine that the workahead is less than two cycles, because the last cycle start number sent or received is greater than or equal to N minus two. Second, the top-half will immediately enter the bottom half if it learns that there is a lost cycle to be handled. This condition is indicated by the shared variable Lost being greater than zero. When this is the case, the DMA side will enter the bottom half loop regardless of the current cycle number, so that skip processing can begin as soon as possible. Because cycles cannot be lost more often than once per cycle, it is not possible for the DMA side to achieve excess workahead due to immediately entering the bottom-half loop whenever Lost is greater than zero.

D.3.2 DMA-side bottom half

The bottom-half loop begins by initializing a private variable C to zero. The variable C will count the IT DMA context index currently being processed. For each context, cycle match processing is applied, if needed, regardless of whether or not a cycle loss has caused cycle skip processing. This causes the cycle match mechanism to correctly start a context even if the desired starting cycle is lost. In such a case, the first packet of that context will be subjected to cycle skip processing, rather than being loaded into the FIFO. Within the bottom-half loop, each active context (including one just activated due to cycle match) will either load one packet into the FIFO, or receive skip processing. [Nit: an empty cycle might not load anything into the FIFO.]

When a packet is loaded into the FIFO, the DMA side flowchart will remain in the block “packet -> FIFO” as long as necessary to complete loading the packet into the FIFO. If the packet is larger than the FIFO, but two-cycle workahead had been achieved prior to this packet, the DMA side might remain in this block for about two whole cycles. During this time, the workahead drops from two to zero, and when the end of the packet is finally loaded into the FIFO, the DMA will immediately begin work on the next packet (same or next cycle).

When skip processing is applied, the DMA side merely replaces a context’s command pointer with the skip address of the descriptor pointed to by the current value of the command pointer.

At the end of the bottom-half loop, the private variable N is incremented, to indicate that one more cycle has been processed. If the cycle’s packets were loaded into the FIFO normally, an end-of-cycle token is placed in the FIFO. However, if skip processing was applied, no packets were loaded into the FIFO, and no end-of-cycle token is placed in the FIFO. As described below, the Link side consumes an end-of-cycle token only for cycles that are not lost, so no token is required when skip processing is applied.

If skip processing was applied, the DMA side atomically decrements the shared variable Lost, to indicate that one lost cycle has been handled.

D.4 Link-side IT DMA flowchart

The following flowchart shows logic for processing the Link-side component of the IT DMA in a manner that (when coupled with the DMA side shown above) agrees with that specified in Chapter 9.0.

Like the DMA side flowchart, the Link side flowchart keeps a private variable M to indicate what cycle number it wants to work on next. Because the Link side begins work simultaneously with the DMA side, there will already be a cycle in progress for which it is too late to possibly do any IT DMA work. So, the Link side initializes M to the current cycle number plus one.

Like the DMA side, the Link side flowchart has a top half and a bottom half. The top half watches the cycle number, and tries to keep transmission synchronized with the cycle timer. The bottom half transmits packets from the FIFO. Unlike the DMA side, the Link side flowchart can move between the top and bottom halves several times during a single cycle’s worth of packets. However, in the absence of cycle loss, the top and bottom halves each run once per cycle.

D.4.1 Link-side top half

The top-half has two roles. First, it watches for the cycle start event that indicates that isochronous transmission can begin. When this happens, it sends control to the bottom half. Second, the top half detects cycle losses that occur outside of the isochronous period. If, while waiting for a cycle start, the top half determines that a cycle loss has occurred, it will communicate this to the DMA side, and then wait to begin work on the following cycle.

In normal operation, the top half waits until cycle M occurs, due to the transmission or reception of the cycle start packet for cycle M. After processing cycle M, or if cycle M is lost, the top half increments M and then begins waiting for the next cycle. While waiting for cycle M, the top half tries to detect cycle loss. The detection algorithm is simple: If the cycle timer rolls over twice, without the receipt or transmission of a cycle start packet, then cycle loss has occurred. There are various ways to more quickly determine that a cycle has been lost, such as the observance of a subaction gap on the bus after the cycle timer has rolled over once. Such strategies, if compatible with Chapter 9.0, may be valuable optimizations, but they are not illustrated here.

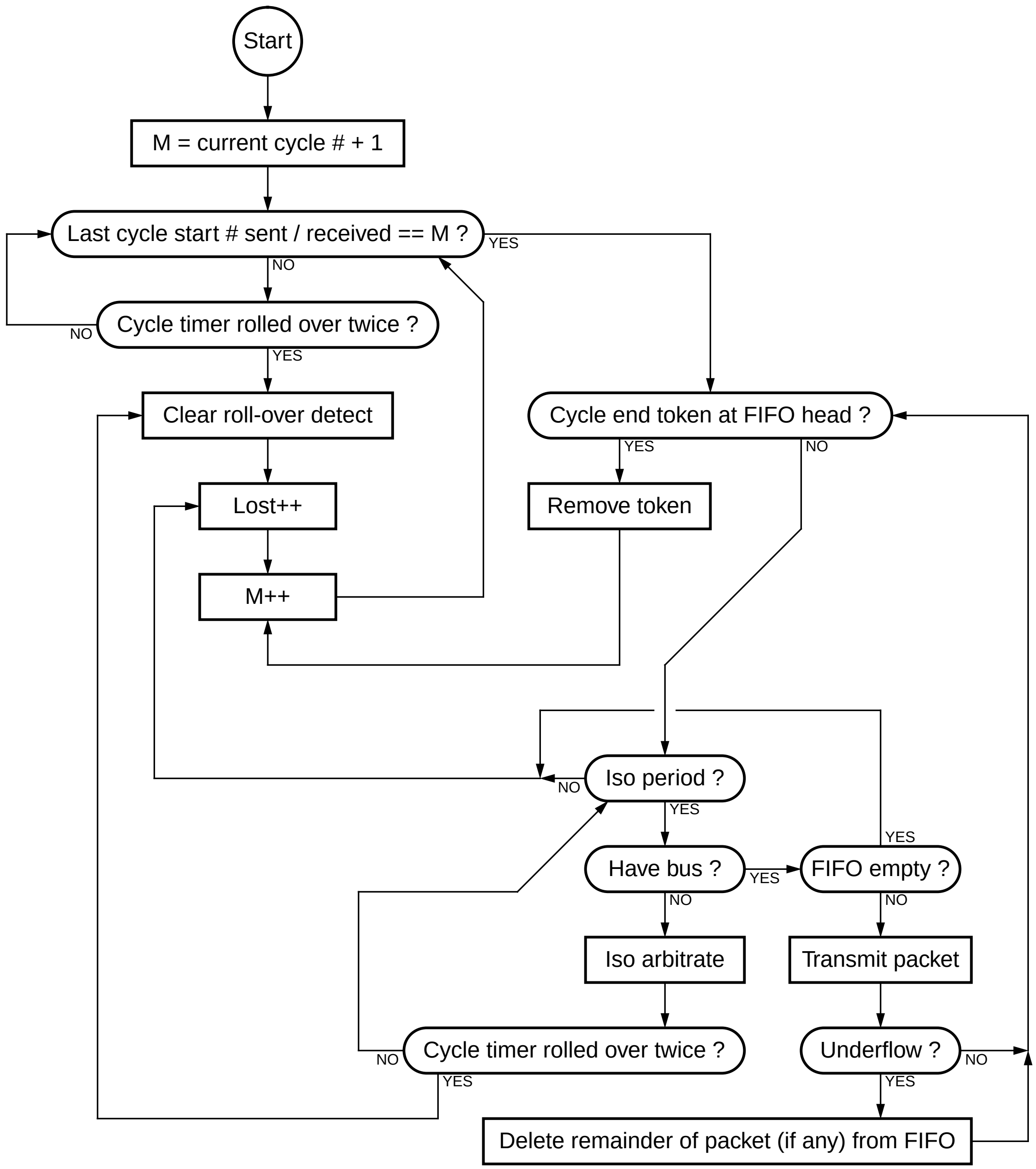


Figure D-2 — IT DMA Link-Side Flowchart

D.4.2 Link-side bottom half

The bottom half of the Link-side flowchart attempts to remove packets from the FIFO and transmit them on the 1394 bus. The bottom half will process at most one cycle's worth of packets. However, if cycle loss occurs during the bottom half, it will indicate this to the DMA side and then return to the top half. The remainder (if any) of the cycle that was being transmitted will be transmitted by a future visit to the bottom half.

The bottom half begins by checking for an end-of-cycle token on the output of the FIFO. If this token is present, then the bottom half has finished work on transmitting one (possibly empty) cycle. The token is removed, M is incremented, and the top half now waits for the next cycle.

If the bottom of the FIFO does not contain an end-of-cycle token, then the bottom half of the Link side flowchart will attempt to transmit packets on the 1394 bus until it does reach an end-of-cycle token. When attempting to transmit packets, the bottom half first checks to see if the 1394 bus is in an isochronous period. When the bottom half is first entered, due to the sending or reception of cycle start packet M, the bus should always be in an isochronous period. However, after some time in the bottom half, the isochronous period may have ended due to a cycle loss. The bottom half checks this before each packet, and if it finds that the bus is not in an isochronous period, it indicates a cycle loss and exits to the top half.

If the bottom half has a packet to transmit, and the 1394 bus is in an isochronous period, the bottom half will then attempt to arbitrate for the 1394 bus. In most silicon implementations, arbitration may have begun earlier, but for the purpose of this flowchart, this is the point at which arbitration actually matters, so it is shown here. Note that if we have already sent at least one packet in the bottom half, then we should already have won arbitration at this point.

If we have not yet won arbitration, the bottom half will loop tightly until we do win arbitration, or a cycle loss is detected. If the cycle timer rolls over twice while we attempt to arbitrate, or if we receive any other indication that the isochronous period has ended, then we indicate a cycle loss and exit the bottom half. As with the top half, there may be ways to optimize the detection of a cycle loss, in order to more rapidly signal the DMA side that recovery is required. These methods are not illustrated here, but as long as they comply with Chapter 9.0, they are not precluded.

If the bottom half does win arbitration, it must then immediately transmit an isochronous packet. Until this time (while arbitrating) it did not matter if the FIFO was empty (due to the DMA having fallen behind). In such a case, the DMA may have caught up and loaded something into the FIFO, in which case transmission can proceed. However, if the FIFO is empty after arbitration is won, then a cycle loss is indicated.

After winning arbitration without detecting a cycle loss and with some data in the FIFO, the bottom half can then begin transmitting a packet on the bus. This process continues until a single packet has been transmitted. If, during transmission, the FIFO underflows, the Link side will clean up the FIFO by eating any leftover parts of the packet that underflowed (but not any following packets). If an end-of-cycle token does not follow immediately, then a cycle loss will be indicated. However, an underflow on the last packet of a cycle does not cause a cycle loss (although the packet itself may be lost).

Finally, after transmitting a packet, with or without underflow, the bottom half checks to see if the cycle has been completed, by looking for an end-of-cycle token at the bottom of the FIFO. If the cycle is complete, the bottom half increments M and returns to the top half. If the cycle is not complete, the bottom half will attempt to transmit the next packet for the current cycle. In this case, if an underflow occurred and the bus was lost, a cycle loss will then be indicated, and the transmission of the next packet will be delayed until the following cycle, as specified in Chapter 9.0.

Annex E. Sample IT DMA Controller Implementation (Informative)

The OpenHCI IT DMA controller is documented in Chapter 9.0. This Annex describes a sample *implementation* of the IT DMA controller. It is intended to faithfully implement the behaviors specified in Chapter 9.0. If there is any disagreement the information in Chapter 9.0 overrides this Annex.

The basic idea behind this IT DMA implementation is that the DMA side keeps track of how far “ahead” or “behind” it is from the link side. When the *ahead_ctr* is positive the DMA side is working ahead of the link. When the *ahead_ctr* is negative the DMA side is catching up. The DMA side *cycle_count* is calculated by adding the *ahead_ctr* value to a version of the link side *cycle_count* that has been exported to the DMA side. This allows the IT DMA controller to work reliably after a cycle inconsistent event. CycleInconsistent events do not affect contexts that don’t care about the cycle number. There is no need to shutdown all contexts when a cycleInconsistent condition is detected. Software only needs to stop/reconfigure/restart contexts that care about the cycle number.

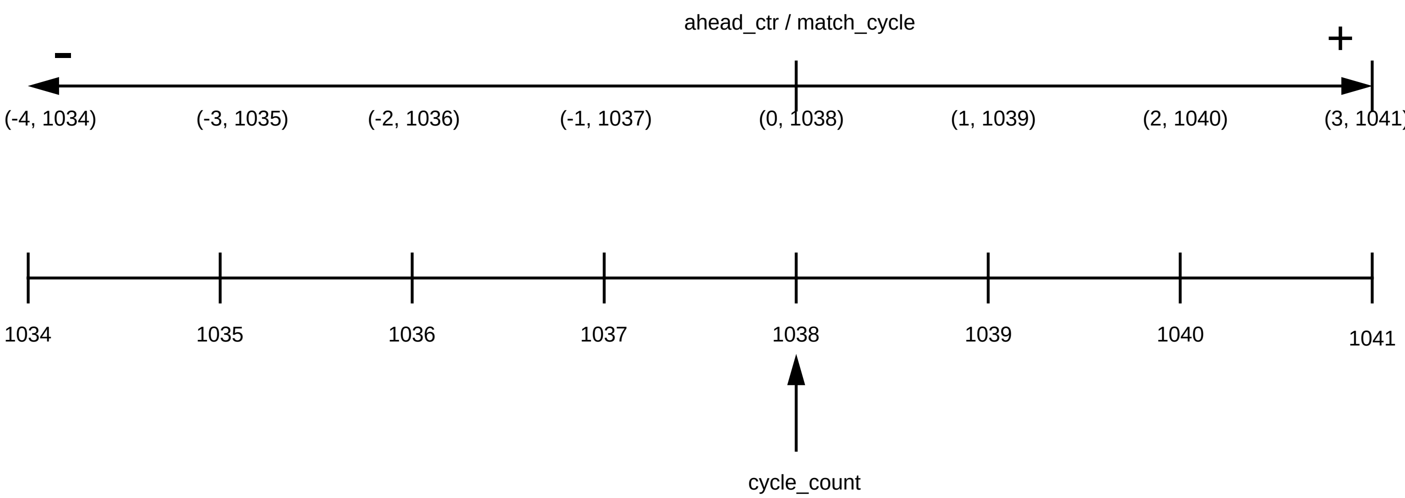


Figure E-1 — DMA Cycle Matching Continuum

This IT DMA controller implementation also maintains a lost counter (*lost_ctr*) that indicates the number of cycle to skip and the logic needed to calculate a current cycle count value for cycle matching purposes.

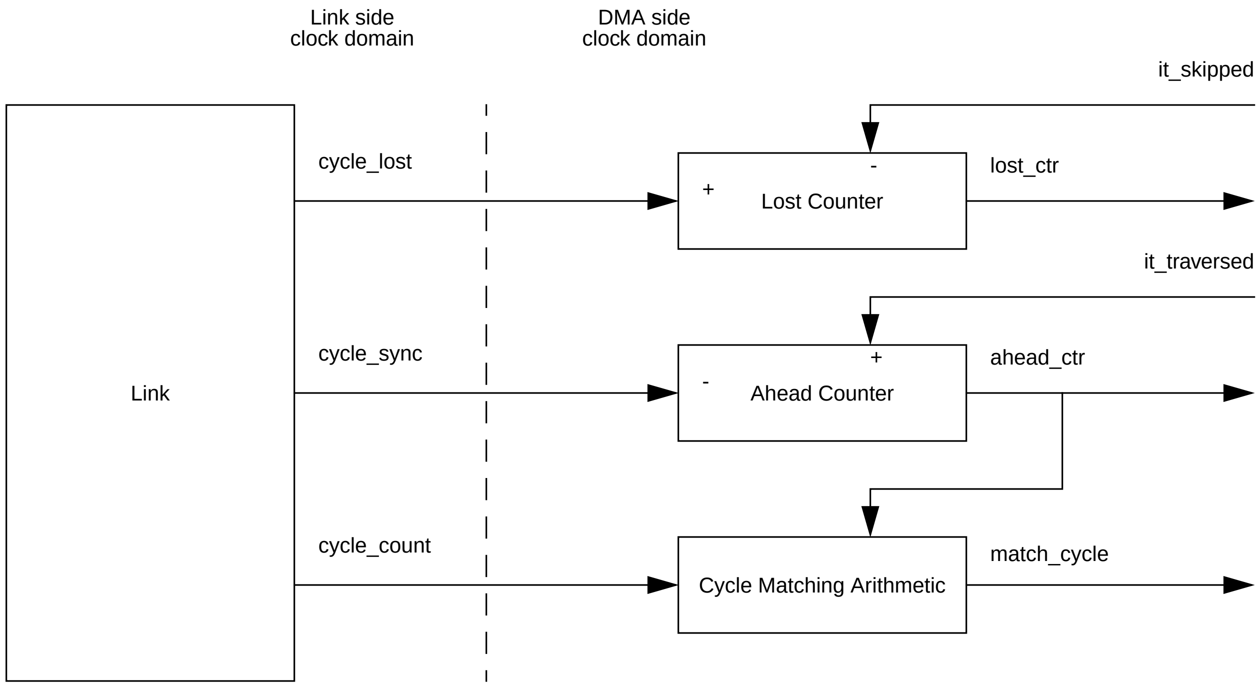


Figure E-2 — IT DMA Controller counters and cycle matching logic

The following pseudo-code is included to describe how the counters can be implemented.

```
always @(posedge dma_clk or negedge reset_z)
   if(reset_z)
      ahead_ctr <= #1 0;
   else if(it_traverse_done && !cycle_sync && (ahead_ctr != AHEAD_MAX))
      ahead_ctr <= #1 ahead_ctr + 1;
   else if(!it_traverse_done && cycle_sync && (ahead_ctr != AHEAD_MIN))
      ahead_ctr <= #1 ahead_ctr - 1;
always @(posedge dma_clk or negedge reset_z)
   if(reset_z)
      lost_ctr <= #1 0;
   else if(!it_skipped && cycle_lost && (lost_ctr != LOST_MAX))
      lost_ctr <= #1 lost_ctr + 1;
   else if(it_skipped && !cycle_lost && (lost_ctr != LOST_MIN))
      lost_ctr <= #1 lost_ctr - 1;


// signed arithmetic assumed here
match_cycle = (cycle_count + ahead_ctr) % 8000;
it_skipped = it_traverse_done && skipping_this_cycle
```

At start-up time, the IT DMA controller “primes the pump” by writing two “isochronous end” tokens into the isochronous transmit FIFO. This causes the *ahead_ctr* to begin with a value of 2. When the following *cycle_sync* event is received from the link-side the *ahead_ctr* is decremented. The IT DMA controller attempts to service the IT contexts when *ahead_ctr* is less than 2 or the *lost_ctr* is greater than 0. So the IT DMA controller will service the IT contexts and then write an isochronous end token (when not skipping) into the FIFO, causing the *ahead_ctr* to increment back to 2. The IT DMA controller is then stalled until the next *cycle_sync* or *cycle_lost* event.

The IT DMA controller uses a calculated cycle count value, *match_cycle*, for matching purposes. It compares the cycleMatch value to the link's cycle_count plus the *ahead_ctr* value (modulo 8000). Some care must be taken to synchronize the updates to the *ahead_ctr* with the changes to the *cycle_count*. This is actually not too difficult since the *cycle_sync* event pulse originates from the link, too. The Host Controller designer just needs to be careful about balancing the synchronization of the *cycle_count* and *cycle_sync* signals. The *cycle_lost* signal needs to be synchronized, too; but it isn't critical that it be balanced with the others. The pseudo-code shown above assumes the *cycle_lost* is translated into single clock cycle pulse on the *dma_clk*.

If the DMA side is unable to service the IT contexts for a span of several 1394 cycles the *ahead_ctr* will continue to decrement and become a negative number. At the same time the link side will generate *cycle_lost* events and the *lost_ctr* will increment. When the DMA side is able to continue it will iteratively traverse the IT contexts performing skip processing until *lost_ctr* equals 0. It can then start stuffing packets into the isochronous transmit FIFO until *ahead_ctr* equals 2.

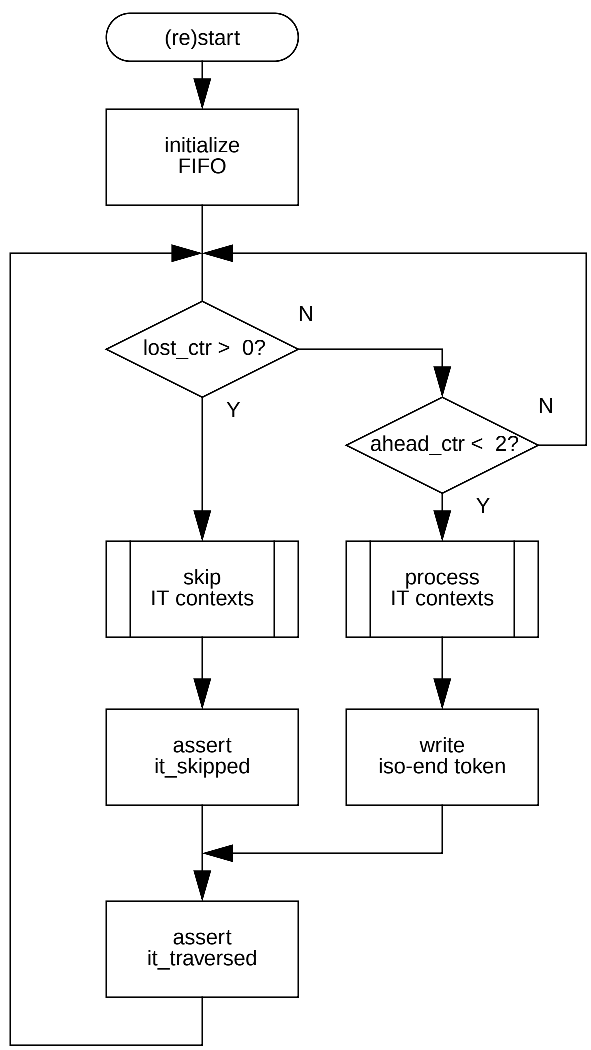


Figure E-3 — IT DMA Flowchart

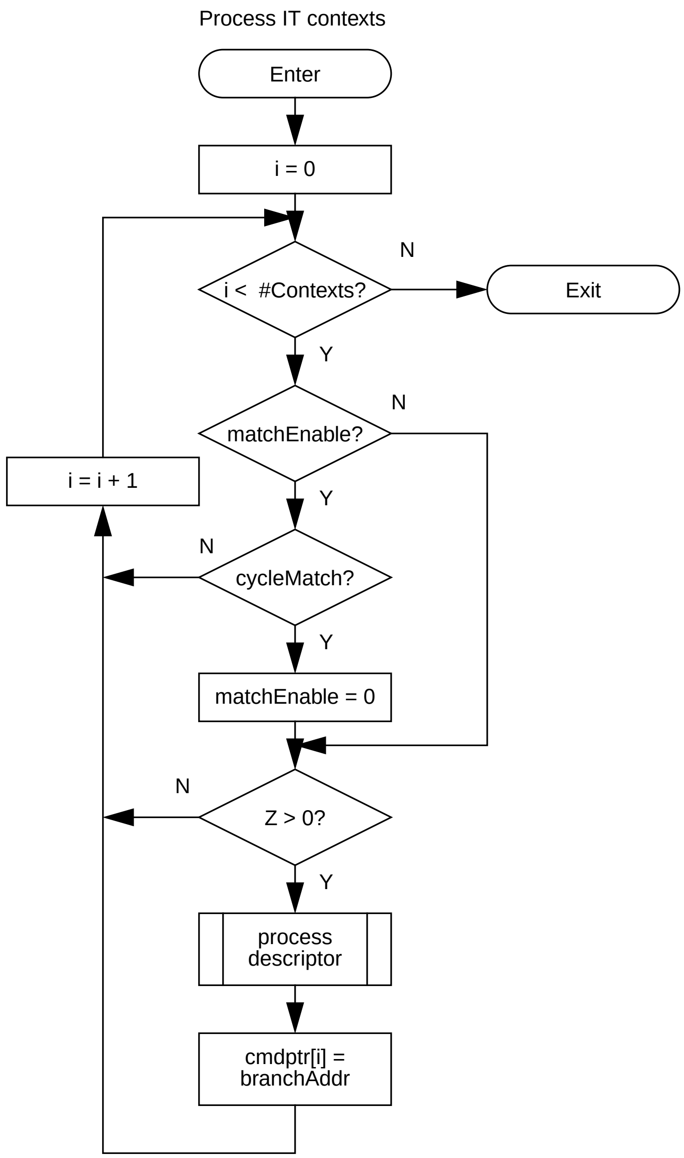


Figure E-4 — Process IT Contexts Flowchart

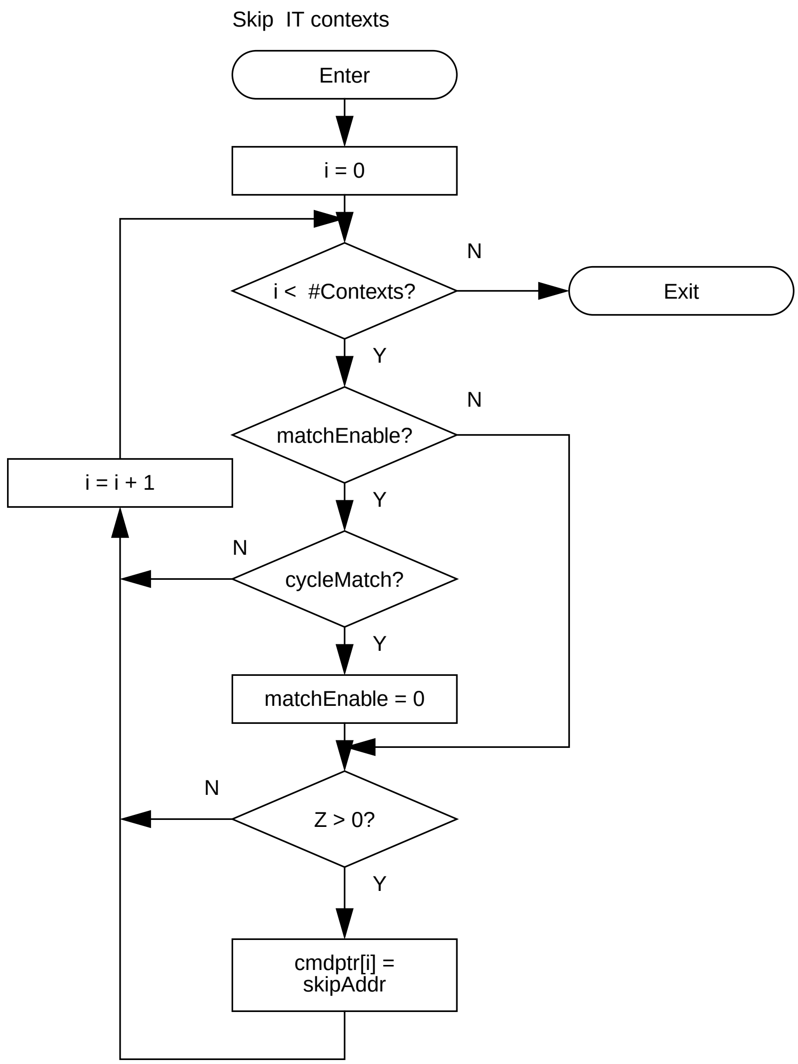


Figure E-5 — Skip IT Contexts Flowchart