PCI Local Bus Specification
Revision 3.0

February 3, 2004

REVISION	REVISION HISTORY	DATE
1.0	Original issue.	6/22/92
2.0	Incorporated connector and add-in card specification.	4/30/93
2.1	Incorporated clarifications and added 66 MHz chapter.	6/1/95
2.2	Incorporated ECNs and improved readability.	12/18/98
2.3	Incorporated ECNs, errata, and deleted 5 volt only keyed add-in cards.	3/29/02
3.0	Incorporated ECNs, errata, and removed support for the 5.0 volt keyed system board connector. Moved the Expansion ROM description to the PCI Firmware Specification.	2/3/04

PCI-SIG disclaims all warranties and liability for the use of this document and the information contained herein and assumes no responsibility for any errors that may appear in this document, nor does PCI-SIG make a commitment to update the information contained herein.

Contact the PCI-SIG office to obtain the latest revision of the specification.

Questions regarding this PCI specification or membership in PCI-SIG may be forwarded to:

PCI-SIG
5440 SW Westgate Drive
Suite 217
Portland, Oregon 97221
Phone: 503-291-2569
Fax: 503-297-1090
e-mail administration@pcisig.com
http://www.pcisig.com

DISCLAIMER

Contents

Figures

Tables

Preface

Specification

This document contains the formal specifications of the protocol, electrical, and mechanical features of the *PCI Local Bus Specification, Revision 3.0*, as the production version effective February 3, 2004. The *PCI Local Bus Specification, Revision 2.3,* issued March 29, 2002, is not superseded by this specification. For compliance information and schedules, refer to the PCI-SIG home page at http://www.pcisig.com.

Following publication of the *PCI Local Bus Specification, Revision* 3.0, there may be future approved errata and/or approved changes to the specification prior to the issuance of another formal revision. To assure designs meet the latest level requirements, designers of PCI devices must refer to the PCI-SIG home page at http://www.pcisig.com for any approved changes.

Incorporation of Engineering Change Notices (ECNs)

The following ECNs have been incorporated into this production version of the specification:

ECN	Description
System Board Connector	Removed support for the 5 volt keyed system board PCI connector
MSI-X	Enhanced MSI interrupt support

Document Conventions

The following name and usage conventions are used in this document:

asserted, deasserted	The terms *asserted* and *deasserted* refer to the globally visible state of the signal on the clock edge, not to signal transitions.
edge, clock edge	The terms *edge* and *clock edge* refer to the rising edge of the clock. On the rising edge of the clock is the only time signals have any significance on the PCI bus.
#	A # symbol at the end of a signal name indicates that the signal's asserted state occurs when it is at a low voltage. The absence of a # symbol indicates that the signal is asserted at a high voltage.
reserved	The contents or undefined states or information are not defined at this time. Using any reserved area in the PCI specification is not permitted. All areas of the PCI specification can only be changed according to the by-laws of the PCI-SIG. Any use of the reserved areas of the PCI specification will result in a product that is not PCI-compliant. The functionality of any such product cannot be guaranteed in this or any future revision of the PCI specification.
signal names	Signal names are indicated with this font.
signal range	A signal name followed by a range enclosed in brackets, for example AD[31::00], represents a range of logically related signals. The first number in the range indicates the most significant bit (msb) and the last number indicates the least significant bit (lsb).
implementation notes	Implementation notes are not part of this specification and are included for clarification and illustration only.

1

1. Introduction

1.1. Specification Contents

The PCI Local Bus is a high performance 32-bit or 64-bit bus with multiplexed address and data lines. The bus is intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in cards, and processor/memory systems.

The *PCI Local Bus Specification, Rev. 3.0,* includes the protocol, electrical, mechanical, and configuration specification for PCI Local Bus components and add-in cards. The electrical definition provides for the 3.3V signaling environment.

The *PCI Local Bus Specification* defines the PCI hardware environment. Contact the PCI SIG for information on the other PCI Specifications. For information on how to join the PCI SIG or to obtain these documents, refer to Section 1.6.

1.2. Motivation

When the *PCI Local Bus Specification* was originally developed in 1992, graphics-oriented operating systems such as Windows and OS/2 had created a data bottleneck between the processor and its display peripherals in standard PC I/O architectures. Moving peripheral functions with high bandwidth requirements closer to the system's processor bus can eliminate this bottleneck. Substantial performance gains are seen with graphical user interfaces (GUIs) and other high bandwidth functions (i.e., full motion video, SCSI, LANs, etc.) when a "local bus" design is used.

PCI successfully met these demands of the industry and is now the most widely accepted and implemented expansion standard in the world.

1.3. PCI Local Bus Applications

The PCI Local Bus has been defined with the primary goal of establishing an industry standard, high performance local bus architecture that offers low cost and allows differentiation. While the primary focus is on enabling new price-performance points in today's systems, it is important that a new standard also accommodates future system requirements and be applicable across multiple platforms and architectures. Figure 1-1 shows the multiple dimensions of the PCI Local Bus.

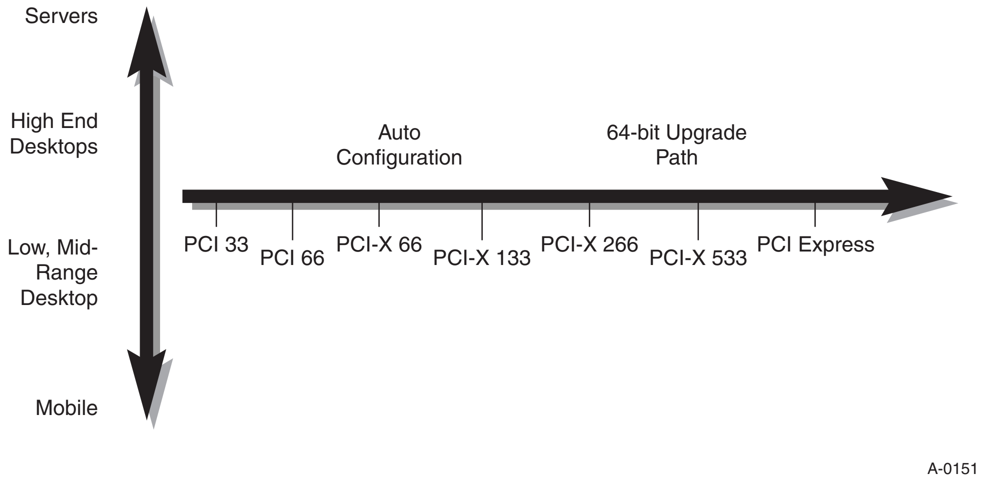


Figure 1-1: PCI Local Bus Applications

While the initial focus of local bus applications was on low to high end desktop systems, the PCI Local Bus also comprehends the requirements from mobile applications up through servers. The PCI Local Bus specifies 3.3 volt signaling requirements.

The PCI component and add-in card interface is processor independent, enabling an efficient transition to future processor generations and use with multiple processor architectures. Processor independence allows the PCI Local Bus to be optimized for I/O functions, enables concurrent operation of the local bus with the processor/memory subsystem, and accommodates multiple high performance peripherals in addition to graphics (motion video, LAN, SCSI, FDDI, hard disk drives, etc.). Movement to enhanced video and multimedia displays (i.e., HDTV and 3D graphics) and other high bandwidth I/O will continue to increase local bus bandwidth requirements. A transparent 64-bit extension of the 32-bit data and address buses is defined, doubling the bus bandwidth and offering forward and backward compatibility of 32-bit and 64-bit PCI Local Bus peripherals. A forward and backward compatible PCI-X specification (see the *PCI-X Addendum to the PCI Local Bus Specification*) is also defined, increasing the bandwidth capabilities of the 33 MHz definition by a factor of four.

The PCI Local Bus standard offers additional benefits to the users of PCI based systems. Configuration registers are specified for PCI components and add-in cards. A system with embedded auto configuration software offers true ease-of-use for the system user by automatically configuring PCI add-in cards at power on.

1.4. PCI Local Bus Overview

The block diagram (Figure 1-2) shows a typical PCI Local Bus system architecture. This example is not intended to imply any specific architectural limits. In this example, the processor/cache/memory subsystem is connected to PCI through a *PCI bridge*. This bridge provides a low latency path through which the processor may directly access PCI devices mapped anywhere in the memory or I/O address spaces. It also provides a high bandwidth path allowing PCI masters direct access to main memory. The bridge may include optional functions such as arbitration and hot plugging. The amount of data buffering a bridge includes is implementation specific.

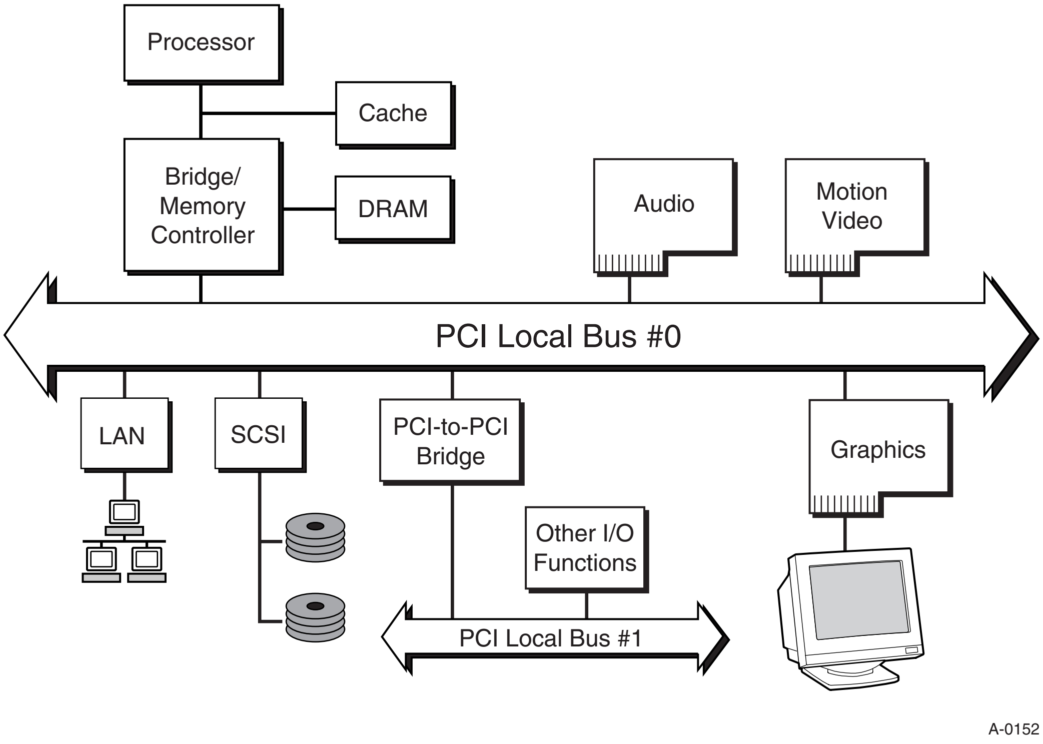


Figure 1-2: PCI System Block Diagram

Typical PCI Local Bus implementations will support up to four add-in card connectors, although expansion capability is not required. PCI add-in cards use an edge connector and system boards that allow a female connector to be mounted parallel to other system board connectors. The transition to the 3.3V signaling environment is complete with this specification. Support of universally keyed add-in cards (which can be plugged into either a 3.3V or 5V keyed system board connector) is provided for backward compatibility with the many systems that supported 5V keyed system board connectors.

Four types of PCI add-in cards are defined: long, short, Low Profile, and variable height short length. Systems are not required to support all add-in card types. The long add-in cards include an extender to support the end of the add-in card.

1.5. PCI Local Bus Features and Benefits

The PCI Local Bus was specified to establish a high performance local bus standard for several generations of products. The PCI specification provides a selection of features that can achieve multiple price-performance points and can enable functions that allow differentiation at the system and component level. Features are categorized by benefit as follows:

High Performance

- Transparent upgrade from 32-bit data path at 33 MHz (132 MB/s peak) to 64-bit data path at 33 MHz (264 MB/s peak), from 32-bit data path at 66 MHz (264 MB/s peak) to 64-bit data path at 66 MHz (532 MB/s peak), and from 32-bit data path at 133 MHz (532 MB/s peak) to 64-bit data path at 133 MHz (1064 MB/s peak).
- Variable length linear and cacheline wrap mode bursting for both read and writes improves write dependent graphics performance.
- Low latency random accesses (60-ns write access latency for 33 MHz PCI to 30-ns for 133 MHz PCI-X to slave registers from master parked on bus).
- Capable of full concurrency with processor/memory subsystem.
- Synchronous bus with operation up to 33 MHz, 66 MHz, or 133 MHz.
- Hidden (overlapped) central arbitration.

Low Cost

- Optimized for direct silicon (component) interconnection; i.e., no glue logic. Electrical/driver (i.e., total load) and frequency specifications are met with standard ASIC technologies and other typical processes.
- Multiplexed architecture reduces pin count (47 signals for target; 49 for master) and package size of PCI components or provides for additional functions to be built into a particular package size.

Ease of Use

- Enables full auto configuration support of PCI Local Bus add-in cards and components. PCI devices contain registers with the device information required for configuration.

Longevity

- Processor independent. Supports multiple families of processors as well as future generations of processors (by bridges or by direct integration).
- Support for 64-bit addressing.
- The 3.3-volt system signaling environment is specified with the universal add-in card providing backward compatibility to 5-volt signaling systems.

Interoperability/ Reliability

- Small form factor add-in cards.
- Present signals allow power supplies to be optimized for the expected system usage by monitoring add-in cards that could surpass the maximum power budgeted by the system.
- Over 2000 hours of electrical SPICE simulation with hardware model validation.
- Forward and backward compatibility of 32-bit and 64-bit add-in cards and components.
- Forward and backward compatibility with PCI 33 MHz, PCI 66 MHz, PCI-X 66 MHz, and PCI-X 133 MHz add-in cards and components.
- Increased reliability and interoperability of add-in cards by comprehending the loading and frequency requirements of the local bus at the component level, eliminating buffers and glue logic.

Flexibility

- Full multi-master capability allowing any PCI master peer-to-peer access to any PCI master/target.

Data Integrity

- Provides parity on both data and address and allows implementation of robust client platforms.

Software Compatibility

- PCI components can be fully compatible with existing driver and applications software. Device drivers can be portable across various classes of platforms.

1.6. Administration

This document is maintained by PCI-SIG. PCI-SIG, an incorporated non-profit organization of members of the microcomputer industry, was established to monitor and enhance the development of the PCI Local Bus in three ways. PCI-SIG is chartered to:

☐ Maintain the forward compatibility of all PCI Local Bus revisions or addenda.

☐ Maintain the PCI Local Bus specification as a simple, easy to implement, stable technology in the spirit of its design.

☐ Contribute to the establishment of the PCI Local Bus as an industry wide standard and to the technical longevity of the PCI Local Bus architecture.

PCI-SIG membership is available to all applicants within the microcomputer industry. Benefits of membership include:

☐ Ability to submit specification revisions and addendum proposals

☐ Participation in specification revisions and addendum proposals

☐ Automatically receive revisions and addenda

☐ Voting rights to determine the Board of Directors membership

☐ Vendor ID number assignment

☐ PCI technical support

☐ PCI support documentation and materials

☐ Participation in PCI-SIG sponsored trade show suites and events, conferences, and other PCI Local Bus promotional activities

☐ Participation in the compliance program including participation at the "PCI Compliance Workshops" and the opportunity to be included in the "PCI Integrator's List"

An annual PCI-SIG membership costs US$3,000. This membership fee supports the activities of PCI-SIG including the compliance program, PCI-SIG administration, and vendor ID issuing and administration.

For information on how to become a PCI-SIG member or on obtaining PCI Local Bus documentation, please contact:

PCI-SIG
5440 SW Westgate Drive
Suite 217
Portland, Oregon 97221
Phone: 503-291-2569
Fax: 503-297-1090
e-mail administration@pcisig.com
http://www.pcisig.com

2. Signal Definition

The PCI interface requires a minimum[1] of 47 pins for a target-only device and 49 pins for a master to handle data and addressing, interface control, arbitration, and system functions. Figure 2-1 shows the pins in functional groups, with required pins on the left side and optional pins on the right side. The direction indication on signals in Figure 2-1 assumes a combination master/target device.

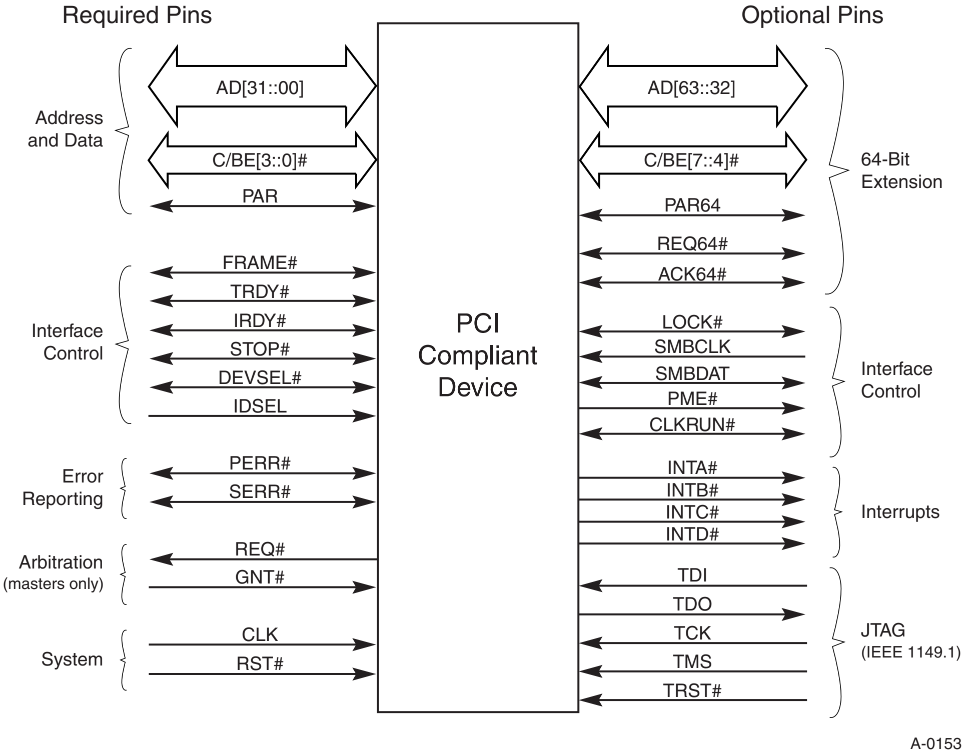


Figure 2-1: PCI Pin List

---

[1] The minimum number of pins for a system board-only device is 45 for a target-only and 47 for a master (PERR# and SERR# are optional for system board-only applications). Systems must support all signals defined for the connector. This includes individual REQ# and GNT# signals for each connector. The PRSNT[1::2]# pins are not device signals and, therefore, are not included in Figure 2-1, but are required to be connected on add-in cards.

2.1. Signal Type Definition

The following signal type definitions are from the view point of all devices other than the arbiter or central resource. For the arbiter, REQ# is an input, GNT# is an output, and other PCI signals for the arbiter have the same direction as a master or target. The central resource is a "logical" device where all system type functions are located (refer to Section 2.4 for more details).

in	*Input* is a standard input-only signal.
out	*Totem Pole Output* is a standard active driver.
t/s	*Tri-State* is a bi-directional, tri-state input/output pin.
s/t/s	*Sustained Tri-State* is an active low tri-state signal owned and driven by one and only one agent at a time. The agent that drives an s/t/s pin low must drive it high for at least one clock before letting it float. A new agent cannot start driving a s/t/s signal any sooner than one clock after the previous owner tri-states it. A pull-up is required to sustain the inactive state until another agent drives it and must be provided by the central resource.
o/d	*Open Drain* allows multiple devices to share as a wire-OR. A pull-up is required to sustain the inactive state until another agent drives it and must be provided by the central resource.

2.2. Pin Functional Groups

The PCI pin definitions are organized in the functional groups shown in Figure 2-1. A # symbol at the end of a signal name indicates that the asserted state occurs when the signal is at a low voltage. When the # symbol is absent, the signal is asserted at a high voltage. The signaling method used on each pin is shown following the signal name.

2.2.1. System Pins

CLK	in	*Clock* provides timing for all transactions on PCI and is an input to every PCI device. All other PCI signals, except RST#, INTA#, INTB#, INTC#, INTD#, PME#, and CLKRUN# are sampled on the rising edge of CLK and all other timing parameters are defined with respect to this edge. PCI operates up to 33 MHz with a minimum frequency of 0 Hz (refer to Chapter 4), 66 MHz with a minimum frequency of 33 MHz (refer to Chapter 7), or 133 MHz with a minimum of 50 MHz (refer to the *PCI-X Addendum to the PCI Local Bus Specification*).
RST#	in	*Reset* is used to bring PCI-specific registers, sequencers, and signals to a consistent state. What effect RST# has on a device beyond the PCI sequencer is beyond the scope of this specification, except for reset states of required PCI configuration registers. A device that can wake the system while in a powered down bus state has additional requirements related to RST#. Refer to the *PCI Power Management Interface Specification* for details. Anytime RST# is asserted, all PCI output signals must be driven to their benign state. In general, this means they must be asynchronously tri-stated. REQ# and GNT# must both be tri-stated (they cannot be driven low or high during reset). To prevent AD, C/BE#, and PAR signals from floating during reset, the central resource may drive these lines during reset (bus parking) but only to a logic low level; they may not be driven high. Refer to Section 3.8.1 for special requirements for AD[63::32], C/BE[7::4]#, and PAR64 when they are not connected (as in a 64-bit add-in card installed in a 32-bit connector). RST# may be asynchronous to CLK when asserted or deasserted. Although asynchronous, deassertion is guaranteed to be a clean, bounce-free edge. Except for configuration accesses, only devices that are required to boot the system will respond after reset.

2.2.2. Address and Data Pins

AD[31::00]	t/s	*Address* and *Data* are multiplexed on the same PCI pins. A bus transaction consists of an address[2] phase followed by one or more data phases. PCI supports both read and write bursts. The address phase is the first clock cycle in which FRAME# is asserted. During the address phase, AD[31::00] contain a physical address (32 bits). For I/O, this is a byte address; for configuration and memory, it is a DWORD address. During data phases, AD[07::00] contain the least significant byte (lsb) and AD[31::24] contain the most significant byte (msb). Write data is stable and valid when IRDY# is asserted; read data is stable and valid when TRDY# is asserted. Data is transferred during those clocks where both IRDY# and TRDY# are asserted.
C/BE[3::0]#	t/s	*Bus Command* and *Byte Enables* are multiplexed on the same PCI pins. During the address phase of a transaction, C/BE[3::0]# define the bus command (refer to Section 3.1 for bus command definitions). During the data phase, C/BE[3::0]# are used as Byte Enables. The Byte Enables are valid for the entire data phase and determine which byte lanes carry meaningful data. C/BE[0]# applies to byte 0 (lsb) and C/BE[3]# applies to byte 3 (msb).
PAR	t/s	*Parity* is even[3] parity across AD[31::00] and C/BE[3::0]#. Parity generation is required by all PCI agents. PAR is stable and valid one clock after each address phase. For data phases, PAR is stable and valid one clock after either IRDY# is asserted on a write transaction or TRDY# is asserted on a read transaction. Once PAR is valid, it remains valid until one clock after the completion of the current data phase. (PAR has the same timing as AD[31::00], but it is delayed by one clock.) The master drives PAR for address and write data phases; the target drives PAR for read data phases.

---

[2] The DAC uses two address phases to transfer a 64-bit address.

[3] The number of "1"s on AD[31::00], C/BE[3::0]#, and PAR equals an even number.

2.2.3. Interface Control Pins

FRAME#	s/t/s	*Cycle Frame* is driven by the current master to indicate the beginning and duration of an access. FRAME# is asserted to indicate a bus transaction is beginning. While FRAME# is asserted, data transfers continue. When FRAME# is deasserted, the transaction is in the final data phase or has completed.
IRDY#	s/t/s	*Initiator Ready* indicates the initiating agent's (bus master's) ability to complete the current data phase of the transaction. IRDY# is used in conjunction with TRDY#. A data phase is completed on any clock both IRDY# and TRDY# are asserted. During a write, IRDY# indicates that valid data is present on AD[31::00]. During a read, it indicates the master is prepared to accept data. Wait cycles are inserted until both IRDY# and TRDY# are asserted together.
TRDY#	s/t/s	*Target Ready* indicates the target agent's (selected device's) ability to complete the current data phase of the transaction. TRDY# is used in conjunction with IRDY#. A data phase is completed on any clock both TRDY# and IRDY# are asserted. During a read, TRDY# indicates that valid data is present on AD[31::00]. During a write, it indicates the target is prepared to accept data. Wait cycles are inserted until both IRDY# and TRDY# are asserted together.
STOP#	s/t/s	*Stop* indicates the current target is requesting the master to stop the current transaction.
LOCK#	s/t/s	*Lock* indicates an atomic operation to a bridge that may require multiple transactions to complete. When LOCK# is asserted, non-exclusive transactions may proceed to a bridge that is not currently locked. A grant to start a transaction on PCI does not guarantee control of LOCK#. Control of LOCK# is obtained under its own protocol in conjunction with GNT#. It is possible for different agents to use PCI while a single master retains ownership of LOCK#. Locked transactions may be initiated only by host bridges, PCI-to-PCI bridges, and expansion bus bridges. Refer to Appendix F for details on the requirements of LOCK#.

IMPLEMENTATION NOTE

Restricted LOCK# Usage

The use of **LOCK#** by a host bridge is permitted but strongly discouraged. A non-bridge device that uses **LOCK#** is not compliant with this specification. The use of **LOCK#** may have significant negative impacts on bus performance.

- ☐ PCI-to-PCI Bridges must not accept any new requests while they are in a locked condition except from the owner of **LOCK#** (see Section F.1).
- ☐ Arbiters are permitted to grant exclusive access of the bus to the agent that owns **LOCK#** (see Section F.1).

These two characteristics of **LOCK#** may result in data overruns for audio, streaming video, and communications devices (plus other less real time sensitive devices). The goal is to drive the use of **LOCK#** to zero and then delete it from all PCI specifications.

IDSEL	in	*Initialization Device Select* is used as a chip select during configuration read and write transactions.
DEVSEL#	s/t/s	*Device Select*, when actively driven, indicates the driving device has decoded its address as the target of the current access. As an input, **DEVSEL#** indicates whether any device on the bus has been selected.

2.2.4. Arbitration Pins (Bus Masters Only)

REQ#	t/s	*Request* indicates to the arbiter that this agent desires use of the bus. This is a point-to-point signal. Every master has its own REQ# which must be tri-stated while RST# is asserted.
GNT#	t/s	*Grant* indicates to the agent that access to the bus has been granted. This is a point-to-point signal. Every master has its own GNT# which must be ignored while RST# is asserted.

While RST# is asserted, the arbiter must ignore all REQ#[4] lines since they are tri-stated and do not contain a valid request. The arbiter can only perform arbitration after RST# is deasserted. A master must ignore its GNT# while RST# is asserted. REQ# and GNT# are tri-state signals due to power sequencing requirements in the case where the bus arbiter is powered by a different supply voltage than the bus master device.

2.2.5. Error Reporting Pins

The error reporting pins are required[5] by all devices and may be asserted when enabled:

PERR#	s/t/s	*Parity Error* is only for the reporting of data parity errors during all PCI transactions except a Special Cycle. The PERR# pin is sustained tri-state and must be driven active by the agent receiving data (when enabled) two clocks following the data when a data parity error is detected. The minimum duration of PERR# is one clock for each data phase that a data parity error is detected. (If sequential data phases each have a data parity error, the PERR# signal will be asserted for more than a single clock.) PERR# must be driven high for one clock before being tri-stated as with all sustained tri-state signals. Refer to Section 3.7.4.1 for more details.

---

[4] REQ# is an input to the arbiter, and GNT# is an output.

[5] Some system board devices are granted exceptions (refer to Section 3.7.2 for details).

SERR#	o/d	*System Error* is for reporting address parity errors, data parity errors on the Special Cycle command, or any other system error where the result will be catastrophic. If an agent does not want a non-maskable interrupt (NMI) to be generated, a different reporting mechanism is required. SERR# is pure open drain and is actively driven for a single PCI clock by the agent reporting the error. The assertion of SERR# is synchronous to the clock and meets the setup and hold times of all bused signals. However, the restoring of SERR# to the deasserted state is accomplished by a weak pull-up (same value as used for s/t/s) which is provided by the central resource not by the signaling agent. This pull-up may take two to three clock periods to fully restore SERR#. The agent that reports SERR# to the operating system does so anytime SERR# is asserted.

2.2.6. Interrupt Pins (Optional)

Interrupts on PCI are optional and defined as "level sensitive," asserted low (negative true), using open drain output drivers. The assertion and deassertion of INTx# is asynchronous to CLK. A device asserts its INTx# line when requesting attention from its device driver unless the device is enabled to use message signaled interrupts (MSI) (refer to Section 6.8 for more information). Once the INTx# signal is asserted, it remains asserted until the device driver clears the pending request. When the request is cleared, the device deasserts its INTx# signal. PCI defines one interrupt line for a single function device and up to four interrupt lines for a *multi-function*[6] device or connector. For a single function device, only INTA# may be used while the other three interrupt lines have no meaning.

INTA#	o/d	*Interrupt A* is used to request an interrupt.
INTB#	o/d	*Interrupt B* is used to request an interrupt and only has meaning on a multi-function device.
INTC#	o/d	*Interrupt C* is used to request an interrupt and only has meaning on a multi-function device.
INTD#	o/d	*Interrupt D* is used to request an interrupt and only has meaning on a multi-function device.

---

[6] When several independent functions are integrated into a single device, it will be referred to as a multi-function device. Each function on a multi-function device has its own configuration space.

Any function on a multi-function device can be connected to any of the INTx# lines. The Interrupt Pin register (refer to Section 6.2.4 for details) defines which INTx# line the function uses to request an interrupt. If a device implements a single INTx# line, it is called INTA#; if it implements two lines, they are called INTA# and INTB#; and so forth. For a multi-function device, all functions may use the same INTx# line or each may have its own (up to a maximum of four functions) or any combination thereof. A single function can never generate an interrupt request on more than one INTx# line.

The system vendor is free to combine the various INTx# signals from the PCI connector(s) in any way to connect them to the interrupt controller. They may be wire-ORed or electronically switched under program control, or any combination thereof. The system designer must insure that each INTx# signal from each connector is connected to an input on the interrupt controller. This means the device driver may not make any assumptions about interrupt sharing. All PCI device drivers must be able to share an interrupt (chaining) with any other logical device including devices in the same multi-function package.

IMPLEMENTATION NOTE

Interrupt Routing

How interrupts are routed on the system board is system specific. However, the following example may be used when another option is not required and the interrupt controller has four open interrupt request lines available. Since most devices are single function and, therefore, can only use INTA# on the device, this mechanism distributes the interrupts evenly among the interrupt controller's input pins.

INTA# of Device Number 0 is connected to IRQW on the system board. (Device Number has no significance regarding being located on the system board or in a connector.) INTA# of Device Number 1 is connected to IRQX on the system board. INTA# of Device Number 2 is connected to IRQY on the system board. INTA# of Device Number 3 is connected to IRQZ on the system board. The table below describes how each agent's INTx# lines are connected to the system board interrupt lines. The following equation can be used to determine to which INTx# signal on the system board a given device's INTx# line(s) is connected.

MB = (D + I) MOD 4

MB = System board Interrupt (IRQW = 0, IRQX = 1, IRQY = 2, and IRQZ = 3)

D = Device Number

I = Interrupt Number (INTA# = 0, INTB# = 1, INTC# = 2, and INTD# = 3)

Device Number on System Board	Interrupt Pin on Device	Interrupt Pin on System Board
0, 4, 8, 12, 16, 20, 24, 28	INTA#	IRQW
	INTB#	IRQX
	INTC#	IRQY
	INTD#	IRQZ
1, 5, 9, 13, 17, 21, 25, 29	INTA#	IRQX
	INTB#	IRQY
	INTC#	IRQZ
	INTD#	IRQW
2, 6, 10, 14, 18, 22, 26, 30	INTA#	IRQY
	INTB#	IRQZ
	INTC#	IRQW
	INTD#	IRQX
3, 7, 11, 15, 19, 23, 27, 31	INTA#	IRQZ
	INTB#	IRQW
	INTC#	IRQX
	INTD#	IRQY

2.2.7. Additional Signals

PRSNT[1::2]#	in	The *Present* signals are not signals for a device, but are provided by an add-in card. The Present signals indicate to the system board whether an add-in card is physically present in the slot and, if one is present, the total power requirements of the add-in card. These signals are required for add-in cards but are optional for system boards. Refer to Section 4.4.1. for more details.

IMPLEMENTATION NOTE

PRSNT# Pins

At a minimum, the add-in card must ground one of the two PRSNT[1::2]# pins to indicate to the system board that an add-in card is physically in the connector. The signal level of PRSNT1# and PRSNT2# inform the system board of the power requirements of the add-in card. The add-in card may simply tie PRSNT1# and/or PRSNT2# to ground to signal the appropriate power requirements of the add-in card. (Refer to Section 4.4.1 for details.) The system board provides pull-ups on these signals to indicate when no add-in card is currently present.

CLKRUN#	in, o/d, s/t/s	*Clock running* is an optional signal used as an input for a device to determine the status of CLK and an open drain output used by the device to request starting or speeding up CLK. CLKRUN# is a sustained tri-state signal used by the central resource to request permission to stop or slow CLK. The central resource is responsible for maintaining CLKRUN# in the asserted state when CLK is running and deasserts CLKRUN# to request permission to stop or slow CLK. The central resource must provide the pull-up for CLKRUN#.

IMPLEMENTATION NOTE

CLKRUN#

CLKRUN# is an optional signal used in the PCI mobile environment and not defined for the connector. Details of the CLKRUN# protocol and other mobile design considerations are discussed in the *PCI Mobile Design Guide*.

M66EN	in	The *66MHZ_ENABLE* pin indicates to a device whether the bus segment is operating at 66 or 33 MHz. Refer to Section 7.5.1 for details of this signal's operation.
PME#	o/d	The *Power Management Event* signal is an optional signal that can be used by a device to request a change in the device or system power state. The assertion and deassertion of PME# is asynchronous to CLK. This signal has additional electrical requirements over and above standard open drain signals that allow it to be shared between devices which are powered off and those which are powered on. In general, this signal is bused between all PCI connectors in a system, although certain implementations may choose to pass separate buffered copies of the signal to the system logic. Devices must be enabled by software before asserting this signal. Once asserted, the device must continue to drive the signal low until software explicitly clears the condition in the device. The use of this pin is specified in the *PCI Bus Power Management Interface Specification*. The system vendor must provide a pull-up on this signal, if it allows the signal to be used. System vendors that do not use this signal are not required to bus it between connectors or provide pull-ups on those pins.
3.3Vaux	in	An optional 3.3 volt auxiliary power source delivers power to the PCI add-in card for generation of power management events when the main power to the card has been turned off by software. The use of this pin is specified in the *PCI Bus Power Management Interface Specification*. A system or add-in card that does not support PCI bus power management must treat the 3.3Vaux pin as reserved.

IMPLEMENTATION NOTE

PME# and 3.3Vaux

PME# and 3.3Vaux are optional signals defined by the *PCI Bus Power Management Interface Specification*. Details of these signals can be found in that document.

2.2.8. 64-Bit Bus Extension Pins (Optional)

The 64-bit extension pins are collectively optional. That is, if the 64-bit extension is used, all the pins in this section are required.

AD[63::32]	t/s	*Address* and *Data* are multiplexed on the same pins and provide 32 additional bits. During an address phase (when using the DAC command and when REQ64# is asserted), the upper 32-bits of a 64-bit address are transferred; otherwise, these bits are *reserved*[7] but are stable and indeterminate. During a data phase, an additional 32-bits of data are transferred when a 64-bit transaction has been negotiated by the assertion of REQ64# and ACK64#.
C/BE[7::4]#	t/s	*Bus Command* and *Byte Enables* are multiplexed on the same pins. During an address phase (when using the DAC command and when REQ64# is asserted), the actual bus command is transferred on C/BE[7::4]#; otherwise, these bits are reserved and indeterminate. During a data phase, C/BE[7::4]# are Byte Enables indicating which byte lanes carry meaningful data when a 64-bit transaction has been negotiated by the assertion of REQ64# and ACK64#. C/BE[4]# applies to byte 4 and C/BE[7]# applies to byte 7.
REQ64#	s/t/s	*Request 64-bit Transfer*, when asserted by the current bus master, indicates it desires to transfer data using 64 bits. REQ64# also has the same timing as FRAME#. REQ64# also has meaning at the end of reset as described in Section 3.8.1.
ACK64#	s/t/s	*Acknowledge 64-bit Transfer*, when actively driven by the device that has positively decoded its address as the target of the current access, indicates the target is willing to transfer data using 64 bits. ACK64# has the same timing as DEVSEL#.

---

[7] Reserved means reserved for future use by the PCI SIG Board of Directors. Reserved bits must not be used by any device.

PAR64	t/s	*Parity Upper DWORD* is the even[8] parity bit that protects AD[63::32] and C/BE[7::4]#. PAR64 must be valid one clock after each address phase on any transaction in which REQ64# is asserted. PAR64 is stable and valid for 64-bit data phases one clock after either IRDY# is asserted on a write transaction or TRDY# is asserted on a read transaction. (PAR64 has the same timing as AD[63::32] but delayed by one clock.) The master drives PAR64 for address and write data phases; the target drives PAR64 for read data phases.

2.2.9. JTAG/Boundary Scan Pins (Optional)

The IEEE Standard 1149.1, *Test Access Port and Boundary Scan Architecture,* is included as an optional interface for PCI devices. IEEE Standard 1149.1 specifies the rules and permissions for designing an 1149.1-compliant IC. Inclusion of a Test Access Port (TAP) on a device allows boundary scan to be used for testing of the device and the add-in card on which it is installed. The TAP is comprised of four pins (optionally five) that are used to interface serially with a TAP controller within the PCI device.

TCK	in	*Test Clock* is used to clock state information and test data into and out of the device during operation of the TAP.
TDI	in	*Test Data Input* is used to serially shift test data and test instructions into the device during TAP operation.
TDO	out	*Test Output* is used to serially shift test data and test instructions out of the device during TAP operation.
TMS	in	*Test Mode Select* is used to control the state of the TAP controller in the device.
TRST#	in	*Test Reset* provides an asynchronous initialization of the TAP controller. This signal is optional in IEEE Standard 1149.1.

These TAP pins operate in the same electrical environment (3.3V or 5V) as the I/O buffers of the device's PCI interface. The drive strength of the TDO pin is not required to be the same as standard PCI bus pins. TDO drive strength should be specified in the device's data sheet.

The system vendor is responsible for the design and operation of the 1149.1 serial chains ("rings") required in the system. The signals are supplementary to the PCI bus and are not operated in a multi-drop fashion. Typically, an 1149.1 ring is created by connecting one device's TDO pin to another device's TDI pin to create a serial chain of devices. In this application, the IC's receive the same TCK, TMS, and optional TRST# signals. The entire 1149.1 ring (or rings) is (are) connected either to a system board test connector for test purposes or to a resident 1149.1 Controller IC.

---

[8] The number of “1”s on AD[63::32], C/BE[7::4]#, and PAR64 equals an even number.

The PCI specification supports add-in cards with a connector that includes 1149.1 Boundary Scan signals. Methods of connecting and using the 1149.1 test rings in a system with add-in cards include:

- Only use the 1149.1 ring on the add-in card during manufacturing test of the add-in card. In this case, the 1149.1 ring on the system board would not be connected to the 1149.1 signals for the add-in cards. The system board would be tested by itself during manufacturing.
- For each add-in card connector in a system, create a separate 1149.1 ring on the system board. For example, with two add-in card connectors there would be three 1149.1 rings on the system board.
- Utilize an IC that allows for hierarchical 1149.1 multi-drop addressability. These IC's would be able to handle the multiple 1149.1 rings and allow multi-drop addressability and operation.

Add-in cards that do not support the IEEE Standard 1149.1 interface must hardwire the add-in card's TDI pin to its TDO pin.

2.2.10. System Management Bus Interface Pins (Optional)

The SMBus interface pins are collectively optional. If the optional management features described in Section 8 are implemented, SMBCLK and SMBDAT are both required.

SMBCLK	o/d	Optional SMBus interface clock signal. This pin is reserved for the optional support of SMBCLK.
SMBDAT	o/d	Optional SMBus interface data signal. This pin is reserved for the optional support of SMBDAT.

2.3. Sideband Signals

PCI provides all basic transfer mechanisms expected of a general purpose, multi-master I/O bus. However, it does not preclude the opportunity for product specific function/performance enhancements via *sideband signals*. A sideband signal is loosely defined as any signal not part of the PCI specification that connects two or more PCI compliant agents and has meaning only to these agents. Sideband signals are permitted for two or more devices to communicate some aspect of their device specific state in order to improve the overall effectiveness of PCI utilization or system operation.

No pins are allowed in the PCI connector for sideband signals. Therefore, sideband signals must be limited to the system board environment. Furthermore, sideband signals may never violate the specified protocol on defined PCI signals or cause the specified protocol to be violated.

2.4. Central Resource Functions

Throughout this specification, the term *central resource* is used to describe bus support functions supplied by the host system, typically in a PCI compliant bridge or standard chipset. These functions include, but are not limited to, the following:

- ☐ Central Arbitration. (REQ# is an input and GNT# is an output.)
- ☐ Required signal pull-ups as described in Section 4.3.3 and "keepers" as described in Section 3.8.1.
- ☐ Subtractive Decode. Only one agent on a PCI bus can use subtractive decode and would typically be a bridge to a standard expansion bus (refer to Section 3.6.1).
- ☐ Convert processor transaction into a configuration transaction.
- ☐ Generation of the individual IDSEL signals to each device for system configuration.
- ☐ Driving REQ64# during reset.

3. Bus Operation

3.1. Bus Commands

Bus commands indicate to the target the type of transaction the master is requesting. Bus commands are encoded on the C/BE[3::0]# lines during the address phase.

3.1.1. Command Definition

PCI bus command encodings and types are listed below, followed by a brief description of each. Note: The command encodings are as viewed on the bus where a "1" indicates a high voltage and "0" is a low voltage. Byte enables are asserted when "0".

C/BE[3::0]#	Command Type
0000	Interrupt Acknowledge
0001	Special Cycle
0010	I/O Read
0011	I/O Write
0100	Reserved
0101	Reserved
0110	Memory Read
0111	Memory Write
1000	Reserved
1001	Reserved
1010	Configuration Read
1011	Configuration Write
1100	Memory Read Multiple
1101	Dual Address Cycle
1110	Memory Read Line
1111	Memory Write and Invalidate

The *Interrupt Acknowledge* command is a read implicitly addressed to the system interrupt controller. The address bits are logical don't cares during the address phase and the byte enables indicate the size of the vector to be returned.

The *Special Cycle* command provides a simple message broadcast mechanism on PCI. It is designed to be used as an alternative to physical signals when sideband communication is necessary. This mechanism is fully described in Section 3.6.2.

The *I/O Read* command is used to read data from an agent mapped in I/O Address Space. AD[31::00] provide a byte address. All 32 bits must be decoded. The byte enables indicate the size of the transfer and must be consistent with the byte address.

The *I/O Write* command is used to write data to an agent mapped in I/O Address Space. All 32 bits must be decoded. The byte enables indicate the size of the transfer and must be consistent with the byte address.

Reserved command encodings are reserved for future use. PCI targets must not alias reserved commands with other commands. Targets must not respond to reserved encodings. If a reserved encoding is used on the interface, the access typically will be terminated with Master-Abort.

The *Memory Read* command is used to read data from an agent mapped in the Memory Address Space. The target is free to do an anticipatory read for this command only if it can guarantee that such a read will have no side effects. Furthermore, the target must ensure the coherency (which includes ordering) of any data retained in temporary buffers after this PCI transaction is completed. Such buffers must be invalidated before any synchronization events (e.g., updating an I/O status register or memory flag) are passed through this access path.

The *Memory Write* command is used to write data to an agent mapped in the Memory Address Space. When the target returns "ready," it has assumed responsibility for the coherency (which includes ordering) of the subject data. This can be done either by implementing this command in a fully synchronous manner, or by insuring any software transparent posting buffer will be flushed before synchronization events (e.g., updating an I/O status register or memory flag) are passed through this access path. This implies that the master is free to create a synchronization event immediately after using this command.

The *Configuration Read* command is used to read the Configuration Space of each agent. An agent is selected during a configuration access when its IDSEL signal is asserted and AD[1::0] are 00. During the address phase of a configuration transaction, AD[7::2] address one of the 64 DWORD registers (where byte enables address the byte(s) within each DWORD) in Configuration Space of each device and AD[31::11] are logical don't cares to the selected agent (refer to Section 3.2.2.3). AD[10::08] indicate which device of a multi-function agent is being addressed.

The *Configuration Write* command is used to transfer data to the Configuration Space of each agent. Addressing for configuration write transactions is the same as for configuration read transactions.

The *Memory Read Multiple* command is semantically identical to the Memory Read command except that it additionally indicates that the master may intend to fetch more than one cacheline before disconnecting. The memory controller continues pipelining memory requests as long as FRAME# is asserted. This command is intended to be used with bulk sequential data transfers where the memory system (and the requesting master) might gain some performance advantage by sequentially reading ahead one or more additional cacheline(s) when a software transparent buffer is available for temporary storage.

The *Dual Address Cycle (DAC)* command is used to transfer a 64-bit address to devices that support 64-bit addressing when the address is not in the low 4-GB address space. Targets that support only 32-bit addresses must treat this command as reserved and not respond to the current transaction in any way.

The *Memory Read Line* command is semantically identical to the Memory Read command except that it additionally indicates that the master intends to fetch a complete cacheline. This command is intended to be used with bulk sequential data transfers where the memory system (and the requesting master) might gain some performance advantage by reading up to a cacheline boundary in response to the request rather than a single memory cycle. As with the Memory Read command, prefetched buffers must be invalidated before any synchronization events are passed through this access path.

The *Memory Write and Invalidate* command is semantically identical to the Memory Write command except that it additionally guarantees a minimum transfer of one complete cacheline; i.e., the master intends to write all bytes within the addressed cacheline in a single PCI transaction unless interrupted by the target. Note: All byte enables must be asserted during each data phase for this command. The master may allow the transaction to cross a cacheline boundary only if it intends to transfer the entire next line also. This command requires implementation of a configuration register in the master indicating the cacheline size (refer to Section 6.2.4 for more information) and may only be used with Linear Burst Ordering (refer to Section 3.2.2.2). It allows a memory performance optimization by invalidating a "dirty" line in a write-back cache without requiring the actual write-back cycle thus shortening access time.

3.1.2. Command Usage Rules

All PCI devices (except host bus bridges) are required to respond as a target to configuration (read and write) commands. All other commands are optional.

A master may implement the optional commands as needed. A target may also implement the optional commands as needed, but if it implements basic memory commands, it must support all the memory commands, including Memory Write and Invalidate, Memory Read Line, and Memory Read Multiple. If not fully implemented, these performance optimizing commands must be aliased to the basic memory commands. For example, a target may not implement the Memory Read Line command; however, it must accept the request (if the address is decoded for a memory access) and treat it as a Memory Read command. Similarly, a target may not implement the Memory Write and Invalidate command, but must accept the request (if the address is decoded for a memory access) and treat it as a Memory Write command.

For block data transfers to/from system memory, Memory Write and Invalidate, Memory Read Line, and Memory Read Multiple are the recommended commands for masters capable of supporting them. The Memory Read or Memory Write commands can be used if for some reason the master is not capable of using the performance optimizing commands. For masters using the memory read commands, any length access will work for all commands; however, the preferred use is shown below.

While Memory Write and Invalidate is the only command that requires implementation of the Cacheline Size register, it is strongly suggested the memory read commands use it as well. A bridge that prefetches is responsible for any latent data not consumed by the master.

Memory command recommendations vary depending on the characteristics of the memory location and the amount of data being read. Memory locations are characterized as either *prefetchable* or *non-prefetchable*. Prefetchable memory has the following characteristics:

- ☐ There are no side effects of a read operation. The read operation cannot be destructive to either the data or any other state information. For example, a FIFO that advances to the next data when read would not be prefetchable. Similarly, a location that cleared a status bit when read would not be prefetchable.
- ☐ When read, the device is required to return all bytes regardless of the byte enables (four or eight depending upon the width of the data transfer (refer to Section 3.8.1)).
- ☐ Bridges are permitted to merge writes into this range (refer to Section 3.2.6).

All other memory is considered to be non-prefetchable.

The preferred use of the read commands is:

Memory Read	When reading data in an address range that has side effects (not prefetchable) or reading a single DWORD
Memory Read Line	Reading more than a DWORD up to the next cacheline boundary in a prefetchable address space
Memory Read Multiple	Reading a block which crosses a cacheline boundary (stay one cacheline ahead of the master if possible) of data in a prefetchable address range

The target should treat the read commands the same even though they do not address the first DWORD of the cacheline. For example, a target that is addressed at DWORD 1 (instead of DWORD 0) should only prefetch to the end of the current cacheline. If the Cacheline Size register is not implemented, then the master should assume a cacheline size of either 16 or 32 bytes and use the read commands recommended above. (This assumes linear burst ordering.)

IMPLEMENTATION NOTE

Using Read Commands

Different read commands will have different affects on system performance because host bridges and PCI-to-PCI bridges must treat the commands differently. When the Memory Read command is used, a bridge will generally obtain only the data the master requested and no more since a side effect may exist. The bridge cannot read more because it does not know which bytes are required for the next data phase. That information is not available until the current data phase completes. However, for Memory Read Line and Memory Read Multiple, the master guarantees that the address range is prefetchable, and, therefore, the bridge can obtain more data than the master actually requested. This process increases system performance when the bridge can prefetch and the master requires more than a single DWORD. (Refer to the *PCI-PCI Bridge Architecture Specification* for additional details and special cases.)

As an example, suppose a master needed to read three DWORDs from a target on the other side of a PCI-to-PCI bridge. If the master used the Memory Read command, the bridge could not begin reading the second DWORD from the target because it does not have the next set of byte enables and, therefore, will terminate the transaction after a single data transfer. If, however, the master used the Memory Read Line command, the bridge would be free to burst data from the target through the end of the cacheline allowing the data to flow to the master more quickly.

The Memory Read Multiple command allows bridges to prefetch data farther ahead of the master, thereby increasing the chances that a burst transfer can be sustained.

It is highly recommended that the Cacheline Size register be implemented to ensure correct use of the read commands. The Cacheline Size register must be implemented when using the optional Cacheline Wrap mode burst ordering.

Using the correct read command gives optimal performance. If, however, not all read commands are implemented, then choose the ones which work the best most of the time. For example, if the large majority of accesses by the master read entire cachelines and only a small number of accesses read more than a cacheline, it would be reasonable for the device to only use the Memory Read Line command for both types of accesses.

A bridge that prefetches is responsible for any latent data not consumed by the master. The simplest way for the bridge to correctly handle latent data is to simply mark it invalid at the end of the current transaction.

IMPLEMENTATION NOTE

Stale-Data Problems Caused By Not Discarding Prefetch Data

Suppose a CPU has two buffers in adjacent main memory locations. The CPU prepares a message for a bus master in the first buffer and then signals the bus master to pick up the message. When the bus master reads its message, a bridge between the bus master and main memory prefetches subsequent addresses including the second buffer location.

Some time later, the CPU prepares a second message using the second buffer in main memory and signals the bus master to come and get it. If the intervening bridge has not flushed the balance of the previous prefetch, then when the master attempts to read the second buffer the bridge may deliver stale data.

Similarly, if a device were to poll a memory location behind a bridge, the device would never observe a new value of the location if the bridge did not flush the buffer after each time the device read it.

3.2. PCI Protocol Fundamentals

The basic bus transfer mechanism on PCI is a burst. A burst is composed of an address phase and one or more data phases. PCI supports bursts in both Memory and I/O Address Spaces.

All signals are sampled on the rising edge of the clock.[9] Each signal has a setup and hold aperture with respect to the rising clock edge in which transitions are not allowed. Outside this aperture, signal values or transitions have no significance. This aperture occurs only on "qualified" rising clock edges for AD[31::00], AD[63::32], PAR[10], PAR64, and IDSEL signals[11] and on every rising clock edge for LOCK#, IRDY#, TRDY#, FRAME#, DEVSEL#, STOP#, REQ#, GNT#, REQ64#, ACK64#, SERR# (on the falling edge of SERR# only), and PERR#. C/BE[3::0]#, and C/BE[7::4]# (as bus commands) are qualified on the clock edge that FRAME# is first asserted. C/BE[3::0]# and C/BE[7::4]# (as byte enables) are qualified on each rising clock edge following the completion of an address phase or data phase and remain valid the entire data phase. RST#, INTA#, INTB#, INTC#, and INTD# are not qualified nor synchronous.

---

[9] The only exceptions are RST#, INTA#, INTB#, INTC#, and INTD# which are discussed in Sections 2.2.1 and 2.2.6.

[10] PAR and PAR64 are treated like an AD line delayed by one clock.

[11] The notion of qualifying IDSEL signals is fully defined in Section 3.6.3.

3.2.1. Basic Transfer Control

The fundamentals of all PCI data transfers are controlled with three signals (see Figure 3-5).

FRAME#	is driven by the master to indicate the beginning and end of a transaction.
IRDY#	is driven by the master to indicate that it is ready to transfer data.
TRDY#	is driven by the target to indicate that it is ready to transfer data.

The interface is in the Idle state when both FRAME# and IRDY# are deasserted. The first clock edge on which FRAME# is asserted is the *address phase*, and the address and bus command code are transferred on that clock edge. The next[12] clock edge begins the first of one or more *data phases* during which data is transferred between master and target on each clock edge for which both IRDY# and TRDY# are asserted. Wait cycles may be inserted in a data phase by either the master or the target when IRDY# or TRDY# is deasserted.

The source of the data is required to assert its *x*RDY# signal unconditionally when data is valid (IRDY# on a write transaction, TRDY# on a read transaction). The receiving agent may delay the assertion of its *x*RDY# when it is not ready to accept data. When delaying the assertion of its *x*RDY#, the target and master must meet the latency requirements specified in Sections 3.5.1.1 and 3.5.2. In all cases, data is only transferred when IRDY# and TRDY# are both asserted on the same rising clock edge.

Once a master has asserted IRDY#, it cannot change IRDY# or FRAME# until the current data phase completes regardless of the state of TRDY#. Once a target has asserted TRDY# or STOP#, it cannot change DEVSEL#, TRDY#, or STOP# until the current data phase completes. Neither the master nor the target can change its mind once it has committed to the current data transfer until the current data phase completes. (A data phase completes when IRDY# and [TRDY# or STOP#] are asserted.) Data may or may not transfer depending on the state of TRDY#.

At such time as the master intends to complete only one more data transfer (which could be immediately after the address phase), FRAME# is deasserted and IRDY# is asserted indicating the master is ready. After the target indicates that it is ready to complete the final data transfer (TRDY# is asserted), the interface returns to the Idle state with both FRAME# and IRDY# deasserted.

---

[12] The address phase consists of two clocks when the command is the Dual Address Cycle (DAC).

3.2.2. Addressing

PCI defines three physical address spaces. The *Memory* and *I/O Address Spaces* are customary. The *Configuration Address Space* has been defined to support PCI hardware configuration. Accesses to this space are further described in Section 3.2.2.3.

PCI targets (except host bus bridges) are required to implement Base Address register(s) to request a range of addresses which can be used to provide access to internal registers or functions (refer to Chapter 6 for more details). The configuration software uses the Base Address register to determine how much space a device requires in a given address space and then assigns (if possible) where in that space the device will reside.

IMPLEMENTATION NOTE

Device Address Space

It is highly recommended, that a device request (via Base Address register(s)) that its internal registers be mapped into Memory Space and not I/O Space. Although the use of I/O Space is allowed, I/O Space is limited and highly fragmented in PC systems and will become more difficult to allocate in the future. Requesting Memory Space instead of I/O Space allows a device to be used in a system that does not support I/O Space. A device may map its internal register into both Memory Space and optionally I/O Space by using two Base Address registers, one for I/O and the other for Memory. The system configuration software will allocate (if possible) space to each Base Address register. When the agent's device driver is called, it determines which address space is to be used to access the device. If the preferred access mechanism is I/O Space and the I/O Base Address register was initialized, then the driver would access the device using I/O bus transactions to the I/O Address Space assigned. Otherwise, the device driver would be required to use memory accesses to the address space defined by the Memory Base Address register. Note: Both Base Address registers provide access to the same registers internally.

When a transaction is initiated on the interface, each potential target compares the address with its Base Address register(s) to determine if it is the target of the current transaction. If it is the target, the device asserts DEVSEL# to claim the access. For more details about DEVSEL# generation, refer to Section 3.2.2.3.2. How a target completes address decode in each address space is discussed in the following sections.

3.2.2.1. I/O Space Decoding

In the I/O Address Space, all 32 AD lines are used to provide a full byte address. The master that initiates an I/O transaction is required to ensure that AD[1::0] indicate the least significant valid byte for the transaction.

The byte enables indicate the size of the transfer and the affected bytes within the DWORD and must be consistent with AD[1::0]. Table 3-1 lists the valid combinations for AD[1::0] and the byte enables for the initial data phase.

Table 3-1: Byte Enables and AD[1::0] Encodings

AD[1::0]	Starting Byte	Valid BE#[3:0] Combinations
00	Byte 0	xxx0 or 1111
01	Byte 1	xx01 or 1111
10	Byte 2	x011 or 1111
11	Byte 3	0111 or 1111

Note: If BE#[3::0] = 1111, AD[1::0] can have any value.

A function may restrict what type of access(es) it supports in I/O Space. For example, a device may restrict its driver to only access the function using byte, word, or DWORD operations and is free to terminate all other accesses with Target-Abort. How a device uses AD[1::0] and BE#[3::0] to determine which accesses violate its addressing restrictions is implementation specific.

A device (other than an expansion bus bridge) that claims legacy I/O addresses whenever its I/O Space enable bit is set (i.e., without the use of Base Address Registers) is referred to as a legacy I/O device. Legacy I/O devices are discussed in Appendix G.

3.2.2.2. Memory Space Decoding

In the Memory Address Space, the AD[31::02] bus provides a DWORD aligned address. AD[1::0] are not part of the address decode. However, AD[1::0] indicate the order in which the master is requesting the data to be transferred.

Table 3-2 lists the burst ordering requested by the master during Memory commands as indicated on AD[1::0].

Table 3-2: Burst Ordering Encoding

AD1	AD0	Burst Order
0	0	Linear Incrementing
0	1	Reserved (disconnect after first data phase)[13]
1	0	Cacheline Wrap mode
1	1	Reserved (disconnect after first data phase)

All targets are required to check AD[1::0] during a memory command transaction and either provide the requested burst order or terminate the transaction with Disconnect in one of two ways. The target can use Disconnect With Data during the initial data phase or Disconnect Without Data for the second data phase. With either termination, only a single data phase transfers data. The target is not allowed to terminate the transaction with Retry solely because it does not support a specific burst order. If the target does not support the burst order requested by the master, the target must complete one data phase and then terminate the request with Disconnect. This ensures that the transaction will complete (albeit slowly, since each request will complete as a single data phase transaction). If a target supports bursting on the bus, the target must support the linear burst ordering. Support for cacheline wrap is optional.

In linear burst order mode, the address is assumed to increment by one DWORD (four bytes) for 32-bit transactions and two DWORDs (eight bytes) for 64-bit transactions after each data phase until the transaction is terminated (an exception is described in Section 3.9). Transactions using the Memory Write and Invalidate command can only use the linear incrementing burst mode.

---

[13] This encoded value is reserved and cannot be assigned any "new" meaning for new designs. New designs (master or targets) cannot use this encoding. Note that in an earlier version of this specification, this encoding had meaning and there are masters that generate it and some targets may allow the transaction to continue past the initial data phase.

A cacheline wrap burst may begin at any address offset within the cacheline. The length of a cacheline is defined by the Cacheline Size register (refer to Section 6.2.4) in Configuration Space which is initialized by configuration software. The access proceeds by incrementing one DWORD address (two DWORDS for a 64-bit data transaction) until the end of the cacheline has been reached, and then wraps to the beginning of the same cacheline. It continues until the rest of the line has been transferred. For example, an access where the cacheline size is 16 bytes (four DWORDs) and the transaction addresses DWORD 08h, the sequence for a 32-bit transaction would be:

> First data phase is to DWORD 08h
>
> Second data phase is to DWORD 0Ch (which is the end of the current cacheline)
>
> Third data phase is to DWORD 00h (which is the beginning of the addressed cacheline)
>
> Last data phase is to DWORD 04h (which completes access to the entire cacheline)

If the burst continues once a complete cacheline has been accessed, the burst continues at the same DWORD offset of the next cacheline. Continuing the burst of the previous example would be:

> Fifth data phase is to DWORD 18h
>
> Sixth data phase is to DWORD 1Ch (which is the end of the second cacheline)
>
> Seventh data phase is to DWORD 10h (which is the beginning of the second cacheline)
>
> Last data phase is to DWORD 14h (which completes access to the second cacheline)

If a target does not implement the Cacheline Size register, the target must signal Disconnect with or after the completion of the first data phase if Cacheline Wrap or a reserved mode is used.

If a master starts with one burst ordering, it cannot change the burst ordering until the current transaction ends, since the burst ordering information is provided on AD[1::0] during the address phase.

A device may restrict what access granularity it supports in Memory Space. For example, a device may restrict its driver to only access the device using byte, word, or DWORD operations and is free to terminate all other accesses with Target-Abort.

3.2.2.3. *Configuration Space Decoding*

Every device, other than host bus bridges, must implement Configuration Address Space. Host bus bridges may optionally implement Configuration Address Space. In the Configuration Address Space, each function is assigned a unique 256-byte space that is accessed differently than I/O or Memory Address Spaces. Configuration registers are described in Chapter 6. The following sections describe:

☐ Configuration commands (Type 0 and Type 1)

☐ Software generation of configuration commands

☐ Software generation of Special Cycles

☐ Selection of a device's Configuration Space

☐ System generation of IDSEL

3.2.2.3.1. Configuration Commands (Type 0 and Type 1)

Because of electrical loading issues, the number of devices that can be supported on a given bus segment is limited. To allow systems to be built beyond a single bus segment, PCI-to-PCI bridges are defined. A PCI-to-PCI bridge requires a mechanism to know how and when to forward configuration accesses to devices that reside behind the bridge.

To support hierarchical PCI buses, two types of configuration transactions are used. They have the formats illustrated in Figure 3-1, which shows the interpretation of AD lines during the address phase of a configuration transaction.

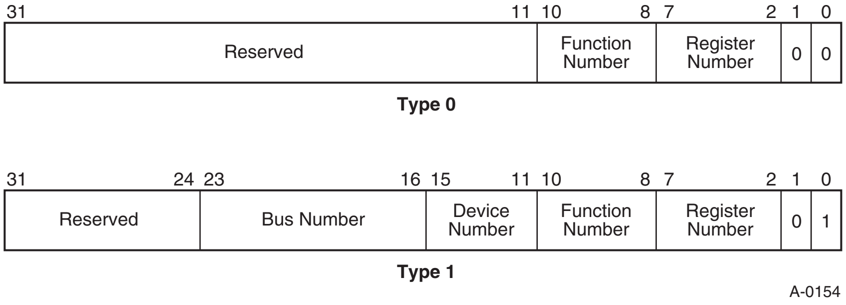


Figure 3-1: Address Phase Formats of Configuration Transactions

Type 1 and Type 0 configuration transactions are differentiated by the values on AD[1::0]. A Type 0 configuration transaction (when AD[1::0] = "00") is used to select a device on the bus where the transaction is being run. A Type 1 configuration transaction (when AD[1::0] = "01") is used to pass a configuration request to another bus segment.

The *Register Number* and *Function Number* fields have the same meaning for both configuration types, and *Device Number* and *Bus Number* are used only in Type 1 transactions. Targets must ignore reserved fields.

Register Number	is an encoded value used to select a DWORD in the Configuration Space of the intended target.
Function Number	is an encoded value used to select one of eight possible functions on a multifunction device.
Device Number	is an encoded value used to select one of 32 devices on a given bus. (Refer to Section 3.2.2.3.5 for limitations on the number of devices supported.)
Bus Number	is an encoded value used to select 1 of 256 buses in a system.

Bridges (both host and PCI-to-PCI) that need to generate a Type 0 configuration transaction use the Device Number to select which IDSEL to assert. The Function Number is provided on AD[10::08]. The Register Number is provided on AD[7::2]. AD[1::0] must be "00" for a Type 0 configuration transaction.

A Type 0 configuration transaction is not propagated beyond the local PCI bus and must be claimed by a local device or terminated with Master-Abort.

If the target of a configuration transaction resides on another bus (not the local bus), a Type 1 configuration transaction must be used. All targets except PCI-to-PCI bridges ignore Type 1 configuration transactions. PCI-to-PCI bridges decode the Bus Number field to determine if the destination bus of the configuration transaction resides behind the bridge. If the Bus Number is not for a bus behind the bridge, the transaction is ignored. The bridge claims the transaction if the transaction is to a bus behind the bridge. If the Bus Number is not to the secondary bus of the bridge, the transaction is simply passed through unchanged. If the Bus Number matches the secondary bus number, the bridge converts the transaction into a Type 0 configuration transaction. The bridge changes AD[1::0] to "00" and passes AD[10::02] through unchanged. The Device Number is decoded to select one of 32 devices on the local bus. The bridge asserts the correct IDSEL and initiates a Type 0 configuration transaction. Note: PCI-to-PCI bridges can also forward configuration transactions upstream (refer to the *PCI-to-PCI Bridge Architecture Specification* for more information).

A standard expansion bus bridge must not forward a configuration transaction to an expansion bus.

3.2.2.3.2. Software Generation of Configuration Transactions

Systems must provide a mechanism that allows software to generate PCI configuration transactions. This mechanism is typically located in the host bridge. For PC-AT compatible systems, the mechanism[14] for generating configuration transactions is defined and specified in this section. A device driver should use the API provided by the operating system to access the Configuration Space of its device and not directly by way of the hardware mechanism. For other system architectures, the method of generating configuration transactions is not defined in this specification.

Two DWORD I/O locations are used to generate configuration transactions for PC-AT compatible systems. The first DWORD location (CF8h) references a read/write register that is named CONFIG_ADDRESS. The second DWORD address (CFCh) references a read/write register named CONFIG_DATA. The CONFIG_ADDRESS register is 32 bits with the format shown in Figure 3-2. Bit 31 is an enable flag for determining when accesses to CONFIG_DATA are to be translated to configuration transactions on the PCI bus. Bits 30 to 24 are reserved, read-only, and must return 0's when read. Bits 23 through 16 choose a specific PCI bus in the system. Bits 15 through 11 choose a specific device on the bus. Bits 10 through 8 choose a specific function in a device (if the device supports multiple functions). Bits 7 through 2 choose a DWORD in the device's Configuration Space. Bits 1 and 0 are read-only and must return 0's when read.

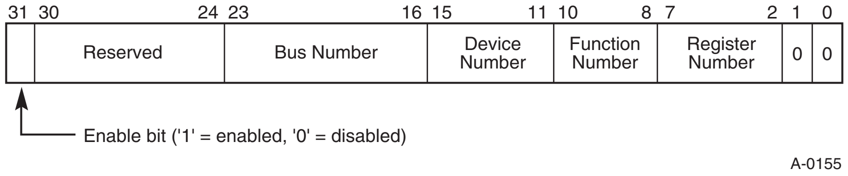


Figure 3-2: Layout of CONFIG_ADDRESS Register

Anytime a host bridge sees a full DWORD I/O write from the host to CONFIG_ADDRESS, the bridge must latch the data into its CONFIG_ADDRESS register. On full DWORD I/O reads to CONFIG_ADDRESS, the bridge must return the data in CONFIG_ADDRESS. Any other types of accesses to this address (non-DWORD) have no effect on CONFIG_ADDRESS and are executed as normal I/O transactions on the PCI bus. Therefore, the only I/O Space consumed by this register is a DWORD at the given address. I/O devices that share the same address but use BYTE or WORD registers are not affected because their transactions will pass through the host bridge unchanged.

When a host bridge sees an I/O access that falls inside the DWORD beginning at CONFIG_DATA address, it checks the Enable bit and the Bus Number in the CONFIG_ADDRESS register. If the Enable bit is set and the Bus Number matches the bridge's Bus Number or any Bus Number behind the bridge, a configuration cycle translation must be done.

---

[14] In versions 2.0 and 2.1 of this specification, two mechanisms were defined. However, only one mechanism (Configuration Mechanism #1) was allowed for new designs and the other (Configuration Mechanism #2) was included for reference.

There are two types of translation that take place. The first, Type 0, is a translation where the device being addressed is on the PCI bus connected to the host bridge. The second, Type 1, occurs when the device is on another bus somewhere behind this bridge.

For Type 0 translations (see Figure 3-3), the host bridge does a decode of the Device Number field to assert the appropriate IDSEL line[15] and performs a configuration transaction on the PCI bus where AD[1::0] = "00". Bits 10 - 8 of CONFIG_ADDRESS are copied to AD[10::8] on the PCI bus as an encoded value which is used by components that contain multiple functions. AD[7::2] are also copied from the CONFIG_ADDRESS register. Figure 3-3 shows the translation from the CONFIG_ADDRESS register to AD lines on the PCI bus.

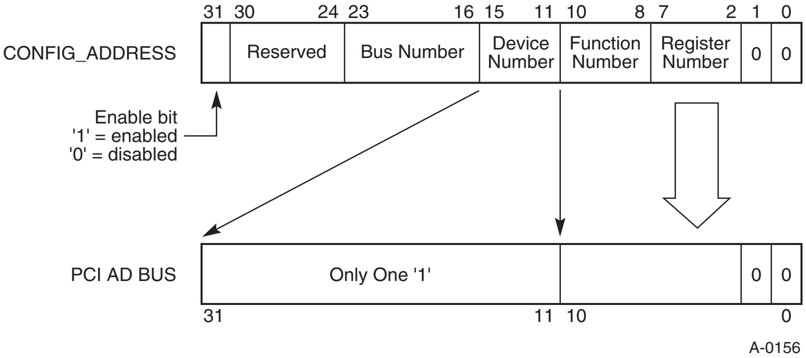


Figure 3-3: Host Bridge Translation for Type 0 Configuration Transactions Address Phase

For Type 1 translations, the host bridge directly copies the contents of the CONFIG_ADDRESS register (excluding bits 31 and 0) onto the PCI AD lines during the address phase of a configuration transaction making sure that AD[1::0] is "01".

In both Type 0 and Type 1 translations, byte enables for the data transfers must be directly copied from the processor bus.

---

[15] If the Device Number field selects an IDSEL line that the bridge does not implement, the bridge must complete the processor access normally, dropping the data on writes and returning all ones on reads. The bridge may optionally implement this requirement by performing a Type 0 configuration access with no IDSEL asserted. This will terminate with Master-Abort which drops write data and returns all ones on reads.

IMPLEMENTATION NOTE

Bus Number Registers and Peer Host Bridges

For host bridges that do not support peer host buses, translating configuration accesses into configuration transactions is simple. If the Bus Number in the CONFIG_ADDRESS register is zero, a Type 0 configuration translation is used. If the Bus Number in the CONFIG_ADDRESS register is non-zero, a Type 1 configuration translation is used.

For host bridges that support peer host buses, one peer bridge typically is designated to always acknowledge accesses to the CONFIG_ADDRESS register. Other peer bridges would snoop the data written to this register. Accesses to the CONFIG_DATA register are typically handshaken by the bridge doing the configuration translation.

Host bridges that support peer host buses require two Configuration Space registers whose contents are used to determine when the bridge does configuration transaction translation. One register (Bus Number) specifies the bus number of the PCI bus directly behind the bridge, and the other register (Subordinate Bus Number) specifies the number of the last hierarchical bus behind the bridge. (A PCI-to-PCI bridge requires an additional register, which is its Primary Bus Number.) System configuration software is responsible for initializing these registers to appropriate values. The host bridge determines the configuration translation type (1 or 0) based on the value of the bus number in the CONFIG_ADDRESS register. If the Bus Number in the CONFIG_ADDRESS register matches the Bus Number register, a Type 0 configuration transaction is used. If the Bus Number in CONFIG_ADDRESS is greater than the Bus Number register and less than or equal to the Subordinate Bus Number register, a Type 1 configuration transaction is used. If the Bus Number in CONFIG_ADDRESS is less than the Bus Number register or greater than the Subordinate Bus Number register, the configuration transaction is addressing a bus that is not implemented or is behind some other host bridge and is ignored.

3.2.2.3.3. Software Generation of Special Cycles

This section defines how a host bridge in a PC-AT compatible systems may optionally implement the configuration mechanism for accessing Configuration Space to allow software to generate a transaction that uses a Special Cycle command. Host bridges are not required to provide a mechanism for allowing software to generate a transaction using a Special Cycle command.

When the CONFIG_ADDRESS register is written with a value such that the Bus Number matches the bridge's bus number, the Device Number is all 1's, the Function Number is all 1's, and the Register Number has a value of zero, then the bridge is primed to generate a transaction using a Special Cycle command the next time the CONFIG_DATA register is written. When the CONFIG_DATA register is written, the bridge generates a transaction that uses a Special Cycle command encoding (rather than Configuration Write command) on the C/BE[3::0]# pins during the address phase and drives the data from the I/O write onto AD[31::00] during the first data phase. After CONFIG_ADDRESS has been set up this way, reads to CONFIG_DATA have undefined results. In one possible implementation, the bridge can treat it as a normal configuration operation (i.e., generate a Type 0 configuration transaction on the PCI bus). This will terminate with a Master-Abort and the processor will have all 1's returned.

If the Bus Number field of CONFIG_ADDRESS does not match the bridge's bus number, then the bridge passes the write to CONFIG_DATA on through to PCI as a Type 1 configuration transaction just like any other time the bus numbers do not match.

3.2.2.3.4. Selection of a Device's Configuration Space

Accesses in the Configuration Address Space require device selection decoding to be done externally and to be signaled to the device via initialization device select, or IDSEL, which functions as a classical "chip select" signal. Each device has its own IDSEL input (except for host bus bridges, which are permitted to implement their initialization device selection internally).

Devices that respond to Type 0 configuration cycles are separated into two types and are differentiated by an encoding in the Configuration Space header. The first type (single-function device) is defined for backward compatibility and only uses its IDSEL pin and AD[1::0] to determine whether or not to respond.

A single function device asserts DEVSEL# to claim a configuration transaction when the following conditions are met:

☐ a configuration command is decoded

☐ the device's IDSEL is asserted

☐ AD[1::0] is "00" (Type 0 Configuration Command) during the Address Phase

Otherwise, the device ignores the current transaction. A single-function device may optionally respond to all function numbers as the same function or may decode the Function Number field, AD[10::08], and respond only to function 0 and not respond (Master-Abort termination) to the other function numbers.

The second type of device (multi-function device) decodes the Function Number field AD[10::08] to select one of eight possible functions on the device when determining whether or not to respond. Multi-function devices are required to do a full decode on AD[10::08] and only respond to the configuration cycle if they have implemented the Configuration Space registers for the selected function. They must not respond (Master-Abort termination) to unimplemented function numbers. They are also required to always implement function 0 in the device. Implementing other functions is optional and may be assigned in any order (i.e., a two-function device must respond to function 0 but can choose any of the other possible function numbers (1-7) for the second function).

If a device implements multiple independent functions, it asserts DEVSEL# to claim a configuration transaction when the following conditions are met:

- ☐ a configuration command is decoded
- ☐ the target's IDSEL is asserted
- ☐ AD[1::0] is "00"
- ☐ AD[10::08] match a function that is implemented

Otherwise, the transaction is ignored. For example, if functions 0 and 4 are implemented (functions 1 through 3 and 5 through 7 are not), the device would assert DEVSEL# for a configuration transaction in which IDSEL is asserted and AD[1::0] are 00 and AD[10::08] matches 000 or 100. AD[31::11] are ignored by a multi-function device during an access of its configuration registers.

The order in which configuration software probes devices residing on a bus segment is not specified. Typically, configuration software either starts with Device Number 0 and works up or starts at Device Number 31 and works down. If a single function device is detected (i.e., bit 7 in the Header Type register of function 0 is 0), no more functions for that Device Number will be checked. If a multi-function device is detected (i.e., bit 7 in the Header Type register of function 0 is 1), then all remaining Function Numbers will be checked.

Once a function has been selected, it uses AD[7::2] to address a DWORD and the byte enables to determine which bytes within the addressed DWORD are being accessed. A function must not restrict the size of the access it supports in Configuration Space. The configuration commands, like other commands, allow data to be accessed using any combination of bytes (including a byte, word, DWORD, or non-contiguous bytes) and multiple data phases in a burst. The target is required to handle any combination of byte enables. However, it is not required to handle a configuration transaction that consists of multiple data phases. If a configuration transaction consists of more than a single data phase, the target is permitted to terminate the request with Disconnect. This is not sufficient cause for the target to terminate the transaction with Target-Abort, since this is not an error condition.

If a configuration transaction has multiple data phases (burst), linear burst ordering is the only addressing mode allowed, since AD[1::0] convey configuration transaction type and not a burst addressing mode like Memory accesses. The implied address of each subsequent data phase is one DWORD larger than the previous data phase. For example, a transaction starts with AD[7::2] equal to 0000 00xxb, the sequence of a burst would be: 0000 01xxb, 0000 10xxb, 0000 11xxb, 0001 00xxb (where xx indicate whether the transaction is a Type 00 or Type 01 configuration transaction). The rest of the transaction is the same as other commands including all termination semantics. Note: The *PCI-to-PCI Bridge Architecture Specification* restricts Type 1 configuration transactions that are converted into a transaction that uses a Special Cycle command to a single data phase (no Special Cycle bursts).

If no agent responds to a configuration transaction, the request is terminated via Master-Abort (refer to Section 3.3.3.1).

3.2.2.3.5. System Generation of IDSEL

Exactly how the IDSEL pin is driven is left to the discretion of the host/memory bridge or system designer. This signal has been designed to allow its connection to one of the upper 21 address lines, which are not otherwise used in a configuration access. However, there is no specified way of determining IDSEL from the upper 21 address bits. Therefore, the IDSEL pin must be supported by all targets. Devices must not make an internal connection between an AD line and an internal IDSEL signal in order to save a pin. The only exception is the host bridge, since it defines how IDSELs are mapped. IDSEL generation behind a PCI-to-PCI bridge is specified in the *PCI-to-PCI Bridge Architecture Specification*.

The binding between a device number in the CONFIG_ADDRESS register of PC-AT compatible system and the generation of an IDSEL is not specified. Therefore, firmware must scan all 32 device numbers to ensure all components are located. Note: The hardware that converts the device number to an IDSEL is required to ensure that only a single unique IDSEL line is asserted for each device number. Configuration transactions that are not claimed by a device are terminated with Master-Abort. The master that initiated this transaction sets the received Master-Abort bit in the Status register.

IMPLEMENTATION NOTE

System Generation of IDSEL

How a system generates IDSEL is system specific; however, if no other mapping is required, the following example may be used. The IDSEL signal associated with Device Number 0 is connected to AD[16], IDSEL of Device Number 1 is connected to AD[17], and so forth until IDSEL of Device Number 15 is connected to AD[31]. For Device Numbers 17-31, the host bridge should execute the transaction but not assert any of the AD[31::16] lines but allow the access to be terminated with Master-Abort.

Twenty-one different devices can be uniquely selected for configuration accesses by connecting a different address line to each device and asserting one of the AD[31::11] lines at a time. The issue with connecting one of the upper 21 AD lines to IDSEL is an additional load on the AD line. This can be mitigated by resistively coupling IDSEL to the appropriate AD line. This does, however, create a very slow slew rate on IDSEL, causing it to be in an invalid logic state most of the time, as shown in Figure 3-4 with the "XXXX" marks. However, since it is only used on the address phase of a Type 0 configuration transaction, the address bus can be pre-driven a few clocks before FRAME#[16], thus guaranteeing IDSEL to be stable when it needs to be sampled. Pre-driving the address bus is equivalent to IDSEL stepping as discussed in Section 3.6.3. Note that if resistive coupling is used, the bridge that generates the configuration transaction is required to use IDSEL stepping or ensure that the clock period is sufficiently long to allow IDSEL to become stable before initiating the configuration transaction. For all other cycles, IDSEL is undefined and may be at a non-deterministic level during the address phase.

---

[16] The number of clocks the address bus should be pre-driven is determined from the RC time constant on IDSEL.

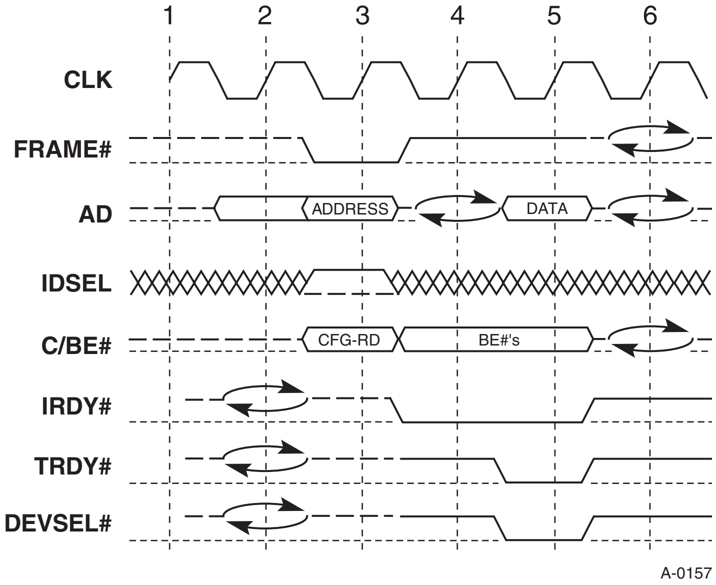


Figure 3-4: Configuration Read

3.2.3. Byte Lane and Byte Enable Usage

The bus protocol does not support automatic bus sizing on requests of a DWORD or less. (Automatic bus sizing allows a device to request the master of the current transaction to break the access into smaller pieces for the target to complete. For example, an 8-bit device that is accessed with a 16-bit request could transfer the lower 8 bits and require the master to move the upper 8 bits (of the 16-bit access) down to the lower byte lane to complete the request.) Since all PCI devices connect to the lower 32 bits for address decode, the device itself is required to provide this byte steering when required, or the driver is required to place the data on the correct byte. In general, software is aware of the characteristics of the target device and only issues appropriate length accesses.

The bus protocol requires automatic bus sizing if a master requests a 64-bit data transfer to a 32-bit target. In this case, the target does not indicate that it can do a 64-bit data transfer, and the master is required to complete the current transaction using 32-bit data transfers. For more details about 64-bit data transactions, refer to Section 3.8.

The byte enables alone are used to determine which byte lanes carry meaningful data. The byte enables are free to change between data phases but must be valid on the clock that starts each data phase and must stay valid for the entire data phase. In Figure 3-5, data phases begin on clocks 3, 5, and 7. (Changing byte enables during a read burst transaction is generally not useful, but is permitted.) The master is free to change the byte enables on each new data phase (although the read diagram does not show this). If the master changes byte enables on a read transaction, it does so with the same timing as would be used in a write transaction. If byte enables are important for the target on a read transaction, the target must wait for the byte enables to be valid on each data phase before completing the transfer; otherwise, it must return all bytes. Note: Byte enables are valid during the entire data phase independent of the state of IRDY#.

If a target supports prefetching (bit 3 is set in the Memory Base Address register -- refer to Section 6.2.5.1), it must also return all data[17] regardless of which byte enables are asserted. A target can only operate in this mode when there are no side effects (data destroyed or status changes because of the access).

PCI allows any contiguous or non-contiguous combination of byte enables. If no byte enables are asserted, the target of the access must complete the data phase by asserting TRDY# and providing parity if the transaction is a read request. The target of an access where no byte enables are asserted must complete the current data phase without any state change. On a read transaction, this means that data and status are not changed. If completing the access has no affect on the data or status, the target may complete the access by either providing data or not. The generation and checking of parity is the same regardless of the state of the byte enables for both 32-bit and 64-bit data transfers. For a discussion on parity generation and checking, refer to Section 3.7.1 (32-bit transactions) and Section 3.8 (64-bit transactions).

However, some targets may not be able to properly interpret non-contiguous patterns (e.g., expansion bus bridges that interface to 8- and 16-bit devices). Expansion bus bridges may optionally report patterns that are illegal on the expansion bus as an asynchronous error (SERR#) or break the transaction into smaller transactions that are legal for the intended agent. The target of an I/O transaction is required to signal Target-Abort if it is unable to complete the entire access defined by the byte enables.

3.2.4. Bus Driving and Turnaround

A turnaround cycle is required on all signals that are driven by more than one agent. The turnaround cycle is required to avoid contention when one agent stops driving a signal and another agent begins driving the signal. This is indicated on the timing diagrams as two arrows pointing at each others' tail. This turnaround cycle occurs at different times for different signals. For instance, IRDY#, TRDY#, DEVSEL#, STOP#, and ACK64# use the address phase as their turnaround cycle. FRAME#, REQ64#, C/BE[3::0]#, C/BE[7::4]#, AD[31::00], and AD[63::32] use the Idle state between transactions as their turnaround cycle. The turnaround cycle for LOCK# occurs one clock after the current owner releases it. PERR# has a turnaround cycle on the fourth clock after the last data phase, which is three clocks after the turnaround-cycle for the AD lines. An Idle state is when both FRAME# and IRDY# are deasserted (e.g., clock 9 in Figure 3-5).

All AD lines (including AD[63::32] when the master supports a 64-bit data path) must be driven to stable values during 64-bit transfers every address and data phase. Even byte lanes not involved in the current data transfer must physically drive stable (albeit meaningless) data onto the bus. The motivation is for parity calculations and to keep input buffers on byte lanes not involved in the transfer from switching at the threshold level and, more generally, to facilitate fast metastability-free latching. In power-sensitive applications, it is recommended that in the interest of minimizing bus switching power consumption, byte lanes not being used in the current bus phase should be driven with the same data as

---

[17] For a 32-bit data transfer, this means 4 bytes per data phase; for a 64-bit data transfer, this means 8 bytes per data phase.

contained in the previous bus phase. In applications that are not power sensitive, the agent driving the AD lines may drive whatever it desires on unused byte lanes. Parity must be calculated on all bytes regardless of the byte enables.

3.2.5. Transaction Ordering and Posting

Transaction ordering rules on PCI accomplish three things. First, they satisfy the write-results ordering requirements of the Producer-Consumer Model. This means that the results of writes from one master (the Producer) anywhere in the system are observable by another master (the Consumer) anywhere in the system only in their original order. (Different masters (Producers) in different places in the system have no fundamental need for their writes to happen in a particular order with respect to each other, since each will have a different Consumer. In this case, the rules allow for some writes to be rearranged.) Refer to Appendix E for a complete discussion of the Producer-Consumer Model. Second, they allow for some transactions to be posted to improve performance. And third, they prevent bus deadlock conditions, when posting buffers have to be flushed to meet the first requirement.

The order relationship of a given transaction with respect to other transactions is determined when it completes; i.e., when data is transferred. Transactions which terminate with Retry have not completed since no data was transferred and, therefore, have no ordering requirements relative to each other. Transactions that terminate with Master-Abort or Target-Abort are considered completed with or without data being transferred and will not be repeated by the master. The system may accept requests in any order, completing one while continuing to Retry another. If a master requires one transaction to be completed before another, the master must not attempt the second transaction until the first one is complete. If a master has only one outstanding request at a time, then that master's transactions will complete throughout the system in the same order the master executed them. Refer to Section 3.3.3.3.5 for further discussion of request ordering.

Transactions can be divided into two general groups based on how they are handled by an intermediate agent, such as a bridge. The two groups are posted and non-posted transactions. Posted transactions complete at the originating device before they reach their ultimate destination. The master will often proceed with other work, sometimes including other bus transactions, before the posted transaction reaches it ultimate destination. In essence, the intermediate agent of the access (e.g., a bridge) accepts the data on behalf of the actual target and assumes responsibility for ensuring that the access completes at the final destination. Memory writes (Memory Write and Memory Write and Invalidate commands) are allowed to be posted on the PCI bus.

Non-posted transactions reach their ultimate destination before completing at the originating device. The master cannot proceed with any other work until the transaction has completed at the ultimate destination (if a dependency exists). Memory read transactions (Memory Read, Memory Read Line, and Memory Read Multiple), I/O transactions (I/O Read and I/O Write), and configuration transactions (Configuration Read and Configuration Write) are non-posted (except as noted below for host bridges).

There are two categories of devices with different requirements for transaction ordering and posting. Each category will be presented separately.

3.2.5.1. Transaction Ordering and Posting for Simple Devices

A simple device is any device that while acting as a bus master does not require its write data to be posted at the bus interface logic. Generally devices that do not connect to local CPUs are implemented as simple devices.

The target and master state machines in the PCI interface of a simple device are completely independent. A device cannot make the completion of any transaction (either posted or non-posted) as a target contingent upon the prior completion of any other transaction as a master. Simple devices are allowed to terminate a transaction with Retry only to execute the transaction as a Delayed Transaction or for temporary conditions which are guaranteed to be resolved with time; e.g., during a video screen refresh or while a transaction buffer is filled with transactions moving in the same direction. (Refer to Section 3.5.3 for a limit on the length of time a memory write transaction can be terminated with Retry.)

IMPLEMENTATION NOTE

Deadlock When Target and Master Not Independent

The following is an example of a deadlock that could occur if devices do not make their target and master interfaces independent.

Suppose two devices, Device A and Device B, are talking directly to each other. Both devices attempt I/O writes to each other simultaneously. Suppose Device A is granted the bus first and executes its I/O write addressing Device B. Device B decodes its address and asserts DEVSEL#. Further, suppose that Device B violates the requirement for the target state machine to be independent of the master state machine and always terminates transactions as a target with Retry until its master state machine completes its outstanding requests. Since Device B also has an I/O transaction it must execute as a master, it terminates Device A's transaction with Retry.

Device B is then granted the bus, and Device B executes its I/O write addressing Device A. If Device A responds the same way Device B did, the system will deadlock.

IMPLEMENTATION NOTE

Deadlock When Posted Write Data is Not Accepted

Deadlocks can also occur when a device does not accept a memory write transaction from a bridge. As described below, a bridge is required in certain cases to flush its posting buffer as a master before it completes a transaction as a target. Suppose a PCI-to-PCI bridge contains posted memory write data addressed to a downstream device. But before the bridge can acquire the downstream bus to do the write transaction, a downstream device initiates a read from host memory. Since requirement 3 in the bridge rules presented below states that posting buffers must be flushed before a read transaction can be completed, the bridge must Retry the agent's read and attempt a write transaction. If the downstream device were to make the acceptance of the write data contingent upon the prior completion of the retried read transaction (that is, if it could not accept the posted write until it first completed the read transaction), the bus would be deadlocked.

Since certain PCI-to-PCI bridge devices designed to previous versions of this specification require their posting buffer to be flushed before starting any non-posted transaction, the same deadlock could occur if the downstream device makes the acceptance of a posted write contingent on the prior completion of any non-posted transaction.

The required independence of target and master state machines in a simple device implies that a simple device cannot internally post any outbound transactions. For example, if during the course of performing its intended function a device must execute a memory write as a master on the PCI bus, the device cannot post that memory write in the master interface of the device. More specifically, the device cannot proceed to other internal operations such as updating status registers that would be observable by another master in the system. The simple device must wait until the memory write transaction completes on the PCI bus (TRDY# asserted; Master-Abort or Target-Abort) before proceeding internally.

Simple devices are strongly encouraged to post inbound memory write transactions to speed the transaction on the PCI bus. How such a device deals with ordering of inbound posted write data is strictly implementation dependent and beyond the scope of this specification.

Simple devices do not support exclusive accesses and do not use the LOCK# signal. Refer to Appendix F for a discussion of the use of LOCK# in bridge devices.

3.2.5.2. Transaction Ordering and Posting for Bridges

A bridge device is any device that implements internal posting of outbound memory write transactions; i.e., write transactions that must be executed by the device as a master on the PCI bus. Bridges normally join two buses such as two PCI buses, a host bus and a PCI bus, or a PCI bus and a bus for a local CPU; i.e., a peripheral CPU.

Bridges are permitted to post memory write transactions moving in either direction through the bridge. The following ordering rules guarantee that the results of one master's write transactions are observable by other masters in the proper order, even though the write transaction may be posted in a bridge. They also guarantee that the bus does not deadlock when a bridge tries to empty its posting buffers.

- ☐ Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus. Even if a single burst on the originating bus is terminated with Disconnect on the destination bus so that it is broken into multiple transactions, those transactions must not allow the data phases to complete on the destination bus in any order other than their order on the originating bus.
- ☐ Write transactions flowing in one direction through a bridge have no ordering requirements with respect to writes flowing in the other direction through the bridge.
- ☐ Posted memory write buffers in both directions must be flushed before completing a read transaction in either direction. Posted memory writes originating on the *same* side of the bridge as a read transaction, and completing before the read command completes on the originating bus, must complete on the destination bus in the same order. Posted memory writes originating on the opposite side of the bridge from a read transaction and completing on the read-destination bus before the read command completes on the read-destination bus must complete on the read-origin bus in the same order. In other words, a read transaction must push ahead of it through the bridge any posted writes originating on the same side of the bridge and posted before the read. In addition, before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the opposite side and were posted before the read command completes on the read-destination bus.
- ☐ A bridge can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. A bridge can make the acceptance of a memory write transaction as a target contingent on the prior completion of a locked transaction as a master only if the bridge has already established a locked operation with its intended target; otherwise, a deadlock may occur. (Refer to Appendix F for a discussion of the use of LOCK# in bridge devices.) In all other cases, bridges are allowed to refuse to accept a memory write only for temporary conditions which are guaranteed to be resolved with time, e.g., during a video screen refresh or while the memory buffer is filled by previous memory write transactions moving in the same direction.

Host bus bridges are permitted to post I/O write transactions that originate on the host bus and complete on a PCI bus segment when they follow the ordering rules described in this specification and do not cause a deadlock. This means that when a host bus bridge posts an

I/O write transaction that originated on the host bus, it must provide a deadlock free environment when the transaction completes on PCI. The transaction will complete on the destination PCI bus before completing on the originating PCI bus.

Since memory write transactions may be posted in bridges anywhere in the system, and I/O writes may be posted in the host bus bridge, a master cannot automatically tell when its write transaction completes at the final destination. For a device driver to guarantee that a write has completed at the actual target (and not at an intermediate bridge), it must complete a read to the same device that the write targeted. The read (memory or I/O) forces all bridges between the originating master and the actual target to flush all posted data before allowing the read to complete. For additional details on device drivers, refer to Section 6.5. Refer to Section 3.10, item 6, for other cases where a read is necessary.

Interrupt requests (that use INTx#) do not appear as transactions on the PCI bus (they are sideband signals) and, therefore, have no ordering relationship to any bus transactions. Furthermore, the system is not required to use the Interrupt Acknowledge bus transaction to service interrupts. So interrupts are not synchronizing events and device drivers cannot depend on them to flush posting buffers. However, when MSI are used, they have the same ordering rules as a memory write transaction (refer to Section 6.8 for more information).

3.2.6. Combining, Merging, and Collapsing

Under certain conditions, bridges that receive (write) data may attempt to convert a transaction (with a single or multiple data phases) into a larger transaction to optimize the data transfer on PCI. The terms used when describing the action are: combining, merging, and collapsing. Each term will be defined and the usage for bridges (host, PCI-to-PCI, or standard expansion bus) will be discussed.

Combining – occurs when sequential memory write transactions (single data phase or burst and independent of active byte enables) are combined into a single PCI bus transaction (using linear burst ordering).

The combining of data is not required but is recommended whenever posting of write data is being done. Combining is permitted only when the implied ordering is not changed. Implied ordering means that the target sees the data in the same order as the original master generated it. For example, a write sequence of DWORD 1, 2, and 4 can be converted into a burst sequence. However, a write of DWORD 4, 3, and 1 cannot be combined into a burst but must appear on PCI as three separate transactions in the same order as they occurred originally. Bursts may include data phases that have no byte enables asserted. For example, the sequence DWORD 1, 2, and 4 could be combined into a burst in which data phase 1 contains the data and byte enables provided with DWORD 1. The second data phase of the burst uses data and byte enables provided with DWORD 2, while data phase 3 asserts no byte enables and provides no meaningful data. The burst completes with data phase 4 using data and byte enables provided with DWORD 4.

If the target is unable to handle multiple data phases for a single transaction, it terminates the burst transaction with Disconnect with or after each data phase. The target sees the data in the same order the originating master generated it, whether the transaction was originally generated as a burst or as a series of single data phase accesses which were combined into a burst.

Byte Merging – occurs when a sequence of individual memory writes (bytes or words) are merged into a single DWORD.

The merging of bytes within the same DWORD for 32-bit transfers or QUADWORD (eight bytes) for 64-bit transfers is not required but is recommended when posting of write data is done. Byte merging is permitted only when the bytes within a data phase are in a prefetchable address range. While similar to combining in concept, merging can be done in any order (within the same data phase) as long as each byte is only written once. For example, in a sequence where bytes 3, 1, 0, and 2 are written to the same DWORD address, the bridge could merge them into a single data phase memory write on PCI with Byte Enable 0, 1, 2, and 3 all asserted instead of four individual write transactions. However, if the sequence written to the same DWORD address were byte 1, and byte 1 again (with the same or different data), byte 2, and byte 3, the bridge cannot merge the first two writes into a single data phase because the same byte location must be written twice. However, the last three transactions could be merged into a single data phase with Byte Enable 0 being deasserted and Byte Enable 1, 2, and 3 being asserted. Merging can never be done to a range of I/O or Memory Mapped I/O addresses (not prefetchable).

Note: Merging and combining can be done independently of each other. Bytes within a DWORD may be merged and merged DWORDs can be combined with other DWORDs when conditions allow. A device can implement only byte merging, only combining, both byte merging and combining, or neither byte merging or combining.

Collapsing – is when a sequence of memory writes to the same location (byte, word, or DWORD address) are collapsed into a single bus transaction.

Collapsing is not permitted by PCI bridges (host, PCI-to-PCI, or standard expansion) except as noted below. For example, a memory write transaction with Byte Enable 3 asserted to DWORD address X, followed by a memory write access to the same address (X) as a byte, word, or DWORD, or any other combination of bytes allowed by PCI where Byte Enable 3 is asserted, cannot be merged into a single PCI transaction. These two accesses must appear on PCI as two separate and distinct transactions.

Note: The combining and merging of I/O and Configuration transactions are not allowed. The collapsing of data of any type of transaction (Configuration, Memory, or I/O) is never allowed (except where noted below).

Note: If a device cannot tolerate memory write combining, it has been designed incorrectly. If a device cannot tolerate memory write byte merging, it must mark itself as not prefetchable. (Refer to Section 6.2.5.1 for a description of prefetchable.) A device that marks itself prefetchable must tolerate combining (without reordering) and byte merging (without collapsing) of writes as described previously. A device is explicitly not required to tolerate reordering of DWORDs or collapsing of data. A prefetchable address range may have write side effects, but it may not have read side effects. A bridge (host bus, PCI-to-PCI, or standard expansion bus) cannot reorder DWORDs in any space, even in a prefetchable space.

Bridges may optionally allow data to be collapsed in a specific address range when a device driver indicates that there are no adverse side effects due to collapsing. How a device driver indicates this to the system is beyond the scope of this specification.

IMPLEMENTATION NOTE

Combining, Merging, and Collapsing

Bridges that post memory write data should consider implementing Combining and Byte Merging. The collapsing of multiple memory write transactions into a single PCI bus transaction is never allowed (except as noted above). The combining of sequential DWORD memory writes into a PCI burst has significant performance benefits. For example, a processor is doing a large number of DWORD writes to a frame buffer. When the host bus bridge combines these accesses into a single PCI transaction, the PCI bus can keep up with a host bus that is running faster and/or wider than PCI.

The merging of bytes within a single DWORD provides a performance improvement but not as significant as combining. However, for unaligned multi-byte data transfers merging allows the host bridge to merge misaligned data into single DWORD memory write transactions. This reduces (at a minimum) the number of PCI transactions by a factor of two. When the bridge merges bytes into a DWORD and then combines DWORDs into a burst, the number of transactions on PCI can be reduced even further than just by merging. With the addition of combining sequential DWORDs, the number of transactions on PCI can be reduced even further. Merging data (DWORDs) within a single cacheline appears to have minimal performance gains.

3.3. Bus Transactions

The timing diagrams in this section show the relationship of significant signals involved in 32-bit transactions. When a signal is drawn as a solid line, it is actively being driven by the current master or target. When a signal is drawn as a dashed line, no agent is actively driving it. However, it may still be assumed to contain a stable value if the dashed line is at the high rail. Tri-stated signals are indicated to have indeterminate values when the dashed line is between the two rails (e.g., AD or C/BE# lines). When a solid line becomes a dotted line, it indicates the signal was actively driven and now is tri-stated. When a solid line makes a low to high transition and then becomes a dotted line, it indicates the signal was actively driven high to precharge the bus and then tri-stated. The cycles before and after each transaction will be discussed in Section 3.4.

3.3.1. Read Transaction

Figure 3-5 illustrates a read transaction and starts with an address phase which occurs when FRAME# is asserted for the first time and occurs on clock 2. During the address phase, AD[31::00] contain a valid address and C/BE[3::0]# contain a valid bus command.

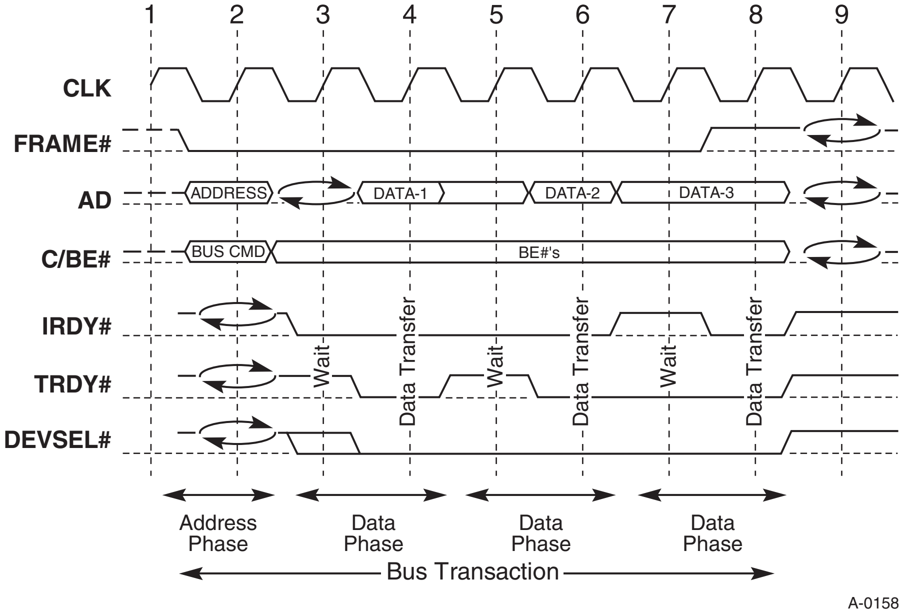


Figure 3-5: Basic Read Operation

The first clock of the first data phase is clock 3. During the data phase, C/BE# indicate which byte lanes are involved in the current data phase. A data phase may consist of wait cycles and a data transfer. The C/BE# output buffers must remain enabled (for both read and writes) from the first clock of the data phase through the end of the transaction. This ensures C/BE# are not left floating for long intervals. The C/BE# lines contain valid byte enable information during the entire data phase independent of the state of IRDY#. The C/BE# lines contain the byte enable information for data phase N+1 on the clock following the completion of the data phase N. This is not shown in Figure 3-5 because a burst read transaction typically has all byte enables asserted; however, it is shown in Figure 3-6. Notice on clock 5 in Figure 3-6, the master inserted a wait state by deasserting IRDY#. However, the byte enables for data phase 3 are valid on clock 5 and remain valid until the data phase completes on clock 8.

The first data phase on a read transaction requires a turnaround-cycle (enforced by the target via TRDY#). In this case, the address is valid on clock 2 and then the master stops driving AD. The earliest the target can provide valid data is clock 4. The target must drive the AD lines following the turnaround cycle when DEVSEL# is asserted. Once enabled, the output buffers must stay enabled through the end of the transaction. (This ensures that the AD lines are not left floating for long intervals.)

One way for a data phase to complete is when data is transferred, which occurs when both IRDY# and TRDY# are asserted on the same rising clock edge. There are other conditions that complete a data phase and these are discussed in Section 3.3.3.2. (TRDY# cannot be driven until DEVSEL# is asserted.) When either IRDY# or TRDY# is deasserted, a wait cycle is inserted and no data is transferred. As noted in Figure 3-5, data is successfully transferred on clocks 4, 6, and 8 and wait cycles are inserted on clocks 3, 5, and 7. The first data phase completes in the minimum time for a read transaction. The second data phase is extended on clock 5 because TRDY# is deasserted. The last data phase is extended because IRDY# was deasserted on clock 7.

The master knows at clock 7 that the next data phase is the last. However, because the master is not ready to complete the last transfer (IRDY# is deasserted on clock 7), FRAME# stays asserted. Only when IRDY# is asserted can FRAME# be deasserted as occurs on clock 8, indicating to the target that this is the last data phase of the transaction.

3.3.2. Write Transaction

Figure 3-6 illustrates a write transaction. The transaction starts when FRAME# is asserted for the first time which occurs on clock 2. A write transaction is similar to a read transaction except no turnaround cycle is required following the address phase because the master provides both address and data. Data phases work the same for both read and write transactions.

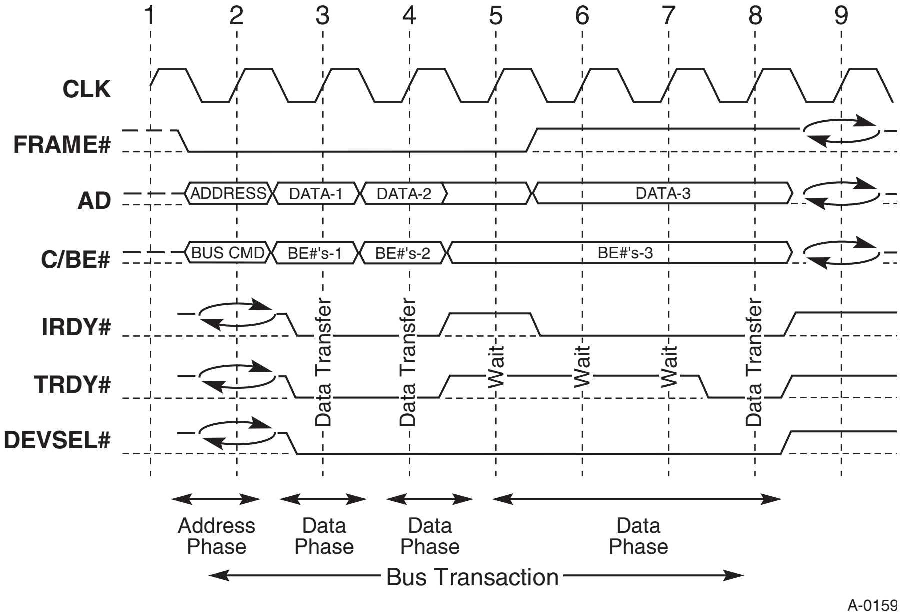


Figure 3-6: Basic Write Operation

In Figure 3-6, the first and second data phases complete with zero wait cycles. However, the third data phase has three wait cycles inserted by the target. Notice both agents insert a wait cycle on clock 5. IRDY# must be asserted when FRAME# is deasserted indicating the last data phase.

The data transfer was delayed by the master on clock 5 because IRDY# was deasserted. The last data phase is signaled by the master on clock 6, but it does not complete until clock 8.

Note: Although this allowed the master to delay data, it did not allow the byte enables to be delayed.

3.3.3. Transaction Termination

Termination of a PCI transaction may be initiated by either the master or the target. While neither can actually stop the transaction unilaterally, the master remains in ultimate control, bringing all transactions to an orderly and systematic conclusion regardless of what caused the termination. All transactions are concluded when FRAME# and IRDY# are both deasserted, indicating an Idle state (e.g., clock 9 in Figure 3-6).

3.3.3.1. Master Initiated Termination

The mechanism used in master initiated termination is when FRAME# is deasserted and IRDY# is asserted. This condition signals the target that the final data phase is in progress. The final data transfer occurs when both IRDY# and TRDY# are asserted. The transaction reaches completion when both FRAME# and IRDY# are deasserted (Idle state).

The master may initiate termination using this mechanism for one of two reasons:

Completion	refers to termination when the master has concluded its intended transaction. This is the most common reason for termination.
Timeout	refers to termination when the master's GNT# line is deasserted and its internal Latency Timer has expired. The intended transaction is not necessarily concluded. The timer may have expired because of target-induced access latency or because the intended operation was very long. Refer to Section 3.5.4 for a description of the Latency Timer operation. A Memory Write and Invalidate transaction is not governed by the Latency Timer except at cacheline boundaries. A master that initiates a transaction with the Memory Write and Invalidate command ignores the Latency Timer until a cacheline boundary. When the transaction reaches a cacheline boundary and the Latency Timer has expired (and GNT# is deasserted), the master must terminate the transaction.

A modified version of this termination mechanism allows the master to terminate the transaction when no target responds. This abnormal termination is referred to as *Master-Abort*. Although it may cause a fatal error for the application originally requesting the transaction, the transaction completes gracefully, thus preserving normal PCI operation for other agents.

Two examples of normal completion are shown in Figure 3-7. The final data phase is indicated by the deassertion of FRAME# and the assertion of IRDY#. The final data phase completes when FRAME# is deasserted and IRDY# and TRDY# are both asserted. The bus reaches an Idle state when IRDY# is deasserted, which occurs on clock 4. Because the transaction has completed, TRDY# is deasserted on clock 4 also. Note: TRDY# is not required to be asserted on clock 3, but could have delayed the final data transfer (and transaction termination) until it is ready by delaying the final assertion of TRDY#. If the target does that, the master is required to keep IRDY# asserted until the final data transfer occurs.

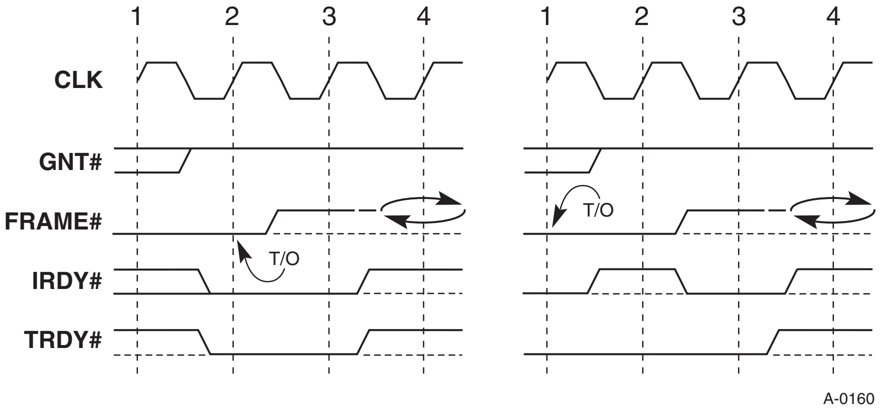


Figure 3-7: Master Initiated Termination

Both sides of Figure 3-7 could have been caused by a timeout termination. On the left side, FRAME# is deasserted on clock 3 because the timer expires, GNT# is deasserted, and the master is ready (IRDY# asserted) for the final transfer. Because GNT# was deasserted when the timer expired, continued use of the bus is not allowed except when using the Memory Write and Invalidate command (refer to Section 3.5.4), which must be stopped at the cacheline boundary. Termination then proceeds as normal. If TRDY# is deasserted on clock 2, that data phase continues until TRDY# is asserted. FRAME# must remain deasserted and IRDY# must remain asserted until the data phase completes.

The right-hand example shows a timer expiring on clock 1. Because the master is not ready to transfer data (IRDY# is deasserted on clock 2), FRAME# is required to stay asserted. FRAME# is deasserted on clock 3 because the master is ready (IRDY# is asserted) to complete the transaction on clock 3. The master must be driving valid data (write) or be capable of receiving data (read) whenever IRDY# is asserted. This delay in termination should not be extended more than two or three clocks. Also note that the transaction need not be terminated after timer expiration unless GNT# is deasserted.

Master-Abort termination, as shown in Figure 3-8, is an abnormal case (except for configuration or Special Cycle commands) of master initiated termination. A master determines that there will be no response to a transaction if DEVSEL# remains deasserted on clock 6. (For a complete description of DEVSEL# operation, refer to Section 3.6.1.) The master must assume that the target of the access is incapable of dealing with the requested transaction or that the address was bad and must not repeat the transaction. Once the master has detected the missing DEVSEL# (clock 6 in this example), FRAME# is deasserted on clock 7 and IRDY# is deasserted on clock 8. The earliest a master can terminate a transaction with Master-Abort is five clocks after FRAME# was first sampled asserted, which occurs when the master attempts a single data transfer. If a burst is attempted, the transaction is longer than five clocks. However, the master may take longer to deassert FRAME# and terminate the access. The master must support the

FRAME# -- IRDY# relationship on all transactions including Master-Abort. FRAME# cannot be deasserted before IRDY# is asserted, and IRDY# must remain asserted for at least one clock after FRAME# is deasserted even when the transaction is terminated with Master-Abort.

Alternatively, IRDY# could be deasserted on clock 7, if FRAME# was deasserted as in the case of a transaction with a single data phase. The master will normally not repeat a transaction terminated with Master-Abort. (Refer to Section 3.7.4.)

Note: If DEVSEL# had been asserted on clocks 3, 4, 5, or 6 of this example, it would indicate the request had been acknowledged by an agent and Master-Abort termination would not be permissible.

The host bus bridge, in PC compatible systems, must return all 1's on a read transaction and discard data on a write transaction when terminated with Master-Abort. The bridge is required to set the Master-Abort detected bit in the status register. Other master devices may report this condition as an error by signaling SERR# when the master cannot report the error through its device driver. A PCI-to-PCI bridge must support PC compatibility as described for the host bus bridge. When the PCI-to-PCI bridge is used in other systems, the bridge behaves like other masters and reports an error. Prefetching of read data beyond the actual request by a bridge must be totally transparent to the system. This means that when a prefetched transaction is terminated with Master-Abort, the bridge must simply stop the transaction and continue normal operation without reporting an error. This occurs when a transaction is not claimed by a target.

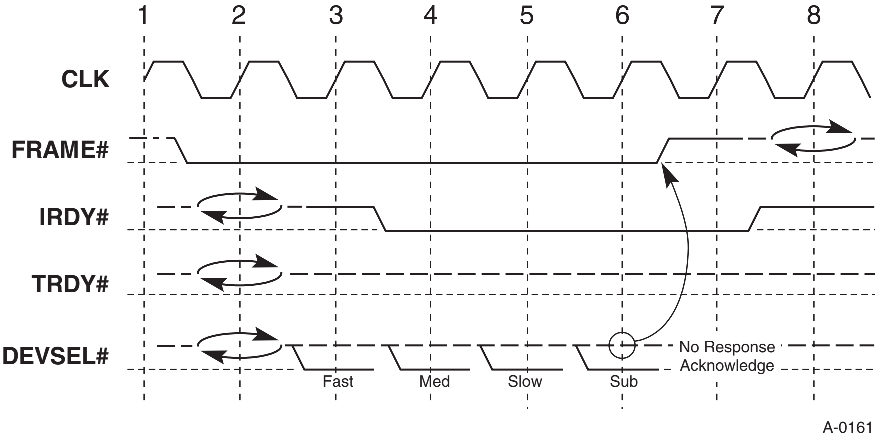


Figure 3-8: Master-Abort Termination

In summary, the following general rules govern FRAME# and IRDY# in all PCI transactions:

1. FRAME# and its corresponding IRDY# define the Busy/Idle state of the bus; when either is asserted, the bus is Busy; when both are deasserted, the bus is Idle.
2. Once FRAME# has been deasserted, it cannot be reasserted during the same transaction.
3. FRAME# cannot be deasserted unless IRDY# is asserted. (IRDY# must always be asserted on the first clock edge that FRAME# is deasserted.)

4. Once a master has asserted IRDY#, it cannot change IRDY# or FRAME# until the current data phase completes.
5. The master must deassert IRDY# the clock after the completion of the last data phase.

3.3.3.2. Target Initiated Termination

Under most conditions, the target is able to source or sink the data requested by the master until the master terminates the transaction. But when the target is unable to complete the request, it may use the STOP# signal to initiate termination of the transaction. How the target combines STOP# with other signals will indicate to the master something about the condition which lead to the termination.

The three types of target initiated termination are:

Retry refers to termination requested before any data is transferred because the target is busy and temporarily unable to process the transaction. This condition may occur, for example, because the device cannot meet the initial latency requirement, is currently locked by another master, or there is a conflict for a internal resource.

Retry is a special case of Disconnect without data being transferred on the initial data phase.

The target signals Retry by asserting STOP# and not asserting TRDY# on the initial data phase of the transaction (STOP# cannot be asserted during the turn-around cycle between the address phase and first data phase of a read transaction). When the target uses Retry, no data is transferred.

Disconnect refers to termination requested with or after data was transferred on the initial data phase because the target is unable to respond within the target subsequent latency requirement and, therefore, is temporarily unable to continue bursting. This might be because the burst crosses a resource boundary or a resource conflict occurs. Data may or may not transfer on the data phase where Disconnect is signaled. Notice that Disconnect differs from Retry in that Retry is always on the initial data phase, and no data transfers. If data is transferred with or before the target terminates the transaction, it is a Disconnect. This may also occur on the initial data phase because the target is not capable of doing a burst.

Disconnect with data may be signaled on any data phase by asserting TRDY# and STOP# together. This termination is used when the target is only willing to complete the current data phase and no more.

Disconnect without data may be signaled on any subsequent data phase (meaning data was transferred on the previous data phase) by deasserting TRDY# and asserting STOP#.

Target-Abort refers to an abnormal termination requested because the target detected a fatal error or the target will never be able to complete the request. Although it may cause a fatal error for the application originally requesting the transaction, the transaction completes gracefully, thus preserving normal operation for other agents. For example, a master requests all bytes in an I/O Address Space DWORD to be read, but the target design restricts access to a single byte in this range. Since the target cannot complete the request, the target terminates the request with Target-Abort.

Once the target has claimed an access by asserting DEVSEL#, it can signal Target-Abort on any subsequent clock. The target signals Target-Abort by deasserting DEVSEL# and asserting STOP# at the same time.

Most targets will be required to implement at least Retry capability, but any other versions of target initiated termination are optional for targets. Masters must be capable of properly dealing with them all. Retry is optional to very simple targets that:

- ☐ do not support exclusive (locked) accesses
- ☐ do not have a posted memory write buffer which needs to be flushed to meet the PCI ordering rules
- ☐ cannot get into a state where they may need to reject an access
- ☐ can always meet target initial latency

A target is permitted to signal Disconnect with data (assert STOP# and TRDY#) on the initial data phase even if the master is not bursting; i.e., FRAME# is deasserted.

3.3.3.2.1. Target Termination Signaling Rules

The following general rules govern FRAME#, IRDY#, TRDY#, STOP#, and DEVSEL# while terminating transactions.

1. A data phase completes on any rising clock edge on which IRDY# is asserted and either STOP# or TRDY# is asserted.
2. Independent of the state of STOP#, a data transfer takes place on every rising edge of clock where both IRDY# and TRDY# are asserted.
3. Once the target asserts STOP#, it must keep STOP# asserted until FRAME# is deasserted, whereupon it must deassert STOP#.
4. Once a target has asserted TRDY# or STOP#, it cannot change DEVSEL#, TRDY#, or STOP# until the current data phase completes.
5. Whenever STOP# is asserted, the master must deassert FRAME# as soon as IRDY# can be asserted.
6. If not already deasserted, TRDY#, STOP#, and DEVSEL# must be deasserted the clock following the completion of the last data phase and must be tri-stated the next clock.

Rule 1 means that a data phase can complete with or without TRDY# being asserted. When a target is unable to complete a data transfer, it can assert STOP# without asserting TRDY#.

When both FRAME# and IRDY# are asserted, the master has committed to complete two data phases. The master is unable to deassert FRAME# until the current data phase completes because IRDY# is asserted. Because a data phase is allowed to complete when STOP# and IRDY# are asserted, the master is allowed to start the final data phase by deasserting FRAME# and keeping IRDY# asserted. The master must deassert IRDY# the clock after the completion of the last data phase.

Rule 2 indicates that data transfers regardless of the state of STOP# when both TRDY# and IRDY# are asserted.

Rule 3 means that once STOP# is asserted, it must remain asserted until the transaction is complete. The last data phase of a transaction completes when FRAME# is deasserted, IRDY# is asserted, and STOP# (or TRDY#) is asserted. The target must not assume any timing relationship between the assertion of STOP# and the deassertion of FRAME#, but must keep STOP# asserted until FRAME# is deasserted and IRDY# is asserted (the last data phase completes). STOP# must be deasserted on the clock following the completion of the last data phase.

When both STOP# and TRDY# are asserted in the same data phase, the target will transfer data in that data phase. In this case, TRDY# must be deasserted when the data phase completes. As before, STOP# must remain asserted until the transaction ends whereupon it is deasserted.

If the target requires wait states in the data phase where it asserts STOP#, it must delay the assertion of STOP# until it is ready to complete the data phase.

Rule 4 means the target is not allowed to change its mind once it has committed to complete the current data phase. Committing to complete a data phase occurs when the target asserts either TRDY# or STOP#. The target commits to:

- ☐ Transfer data in the current data phase and continue the transaction (if a burst) by asserting TRDY# and not asserting STOP#
- ☐ Transfer data in the current data phase and terminate the transaction by asserting both TRDY# and STOP#
- ☐ Not transfer data in the current data phase and terminate the transaction by asserting STOP# and deasserting TRDY#
- ☐ Not transfer data in the current data phase and terminate the transaction with an error condition (Target-Abort) by asserting STOP# and deasserting TRDY# and DEVSEL#

The target has not committed to complete the current data phase while TRDY# and STOP# are both deasserted. The target is simply inserting wait states.

Rule 5 means that when the master samples STOP# asserted, it must deassert FRAME# on the first cycle thereafter in which IRDY# is asserted. The assertion of IRDY# and deassertion of FRAME# should occur as soon as possible after STOP# is asserted, preferably within one to three cycles. This assertion of IRDY# (and therefore FRAME# deassertion) may occur as a consequence of the normal IRDY# behavior of the master had the current transaction not been target terminated. Alternatively, if TRDY# is deasserted (indicating there will be no further data transfer), the master may assert IRDY# immediately (even without being prepared to complete a data transfer). If a Memory Write and Invalidate transaction is terminated by the target, the master completes the transaction (the rest of the cacheline) as soon as possible (adhering to the STOP# protocol) using the Memory Write command (since the conditions to issue Memory Write and Invalidate are no longer true).

Rule 6 requires the target to release control of the target signals in the same manner it would if the transaction had completed using master termination. Retry and Disconnect are normal termination conditions on the bus. Only Target-Abort is an abnormal termination that may have caused an error. Because the reporting of errors is optional, the bus must continue operating as though the error never occurred.

Examples of Target Termination

Retry

Figure 3-9 shows a transaction being terminated with Retry. The transaction starts with FRAME# asserted on clock 2 and IRDY# asserted on clock 3. The master requests multiple data phases because both FRAME# and IRDY# are asserted on clock 3. The target claims the transaction by asserting DEVSEL# on clock 4.

The target determines it cannot complete the master's request and also asserts STOP# on clock 4 while keeping TRDY# deasserted. The first data phase completes on clock 4 because both IRDY# and STOP# are asserted. Since TRDY# was deasserted, no data was transferred during the initial data phase. Because STOP# was asserted and TRDY# was deasserted on clock 4, the master knows the target is unwilling to transfer any data for this transaction at the present time. The master is required to deassert FRAME# as soon as IRDY# can be asserted. In this case, FRAME# is deasserted on clock 5 because IRDY# is asserted on clock 5. The last data phase completes on clock 5 because FRAME# is deasserted and STOP# is asserted. The target deasserts STOP# and DEVSEL# on clock 6 because the transaction is complete. This transaction consisted of two data phases in which no data was transferred and the master is required to repeat the request again.

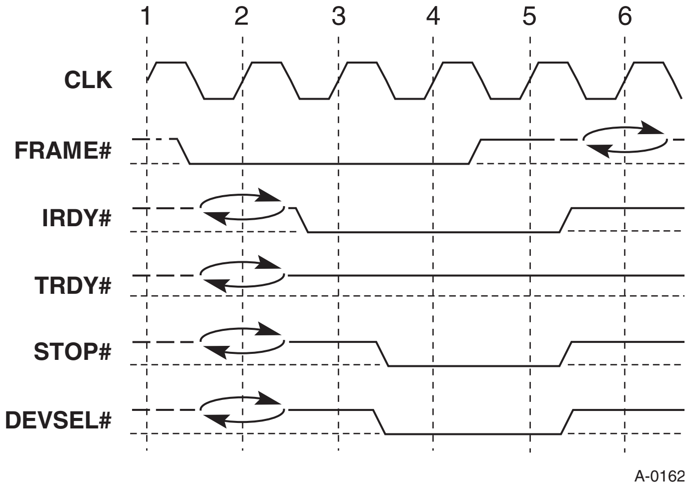


Figure 3-9: Retry

Disconnect With Data

Disconnect - A, in Figure 3-10, is where the master is inserting a wait state when the target signals Disconnect with data. This transaction starts prior to clock 1. The current data phase, which could be the initial or a subsequent data phase, completes on clock 3. The master inserts a wait state on clocks 1 and 2, while the target inserts a wait state only on clock 1. Since the target wants to complete only the current data phase, and no more, it asserts TRDY# and STOP# at the same time. In this example, the data is transferred during the last data phase. Because the master sampled STOP# asserted on clock 2, FRAME# is deasserted on clock 3 and the master is ready to complete the data phase (IRDY# is asserted). Since FRAME# is deasserted on clock 3, the last data phase completes because STOP# is asserted and data transfers because both IRDY# and TRDY# are asserted. Notice that STOP# remains asserted for both clocks 2 and 3. The target is required to keep STOP# asserted until FRAME# is deasserted.

Disconnect - B, in Figure 3-10, is almost the same as Disconnect - A, but TRDY# is not asserted in the last data phase. In this example, data was transferred on clocks 1 and 2 but not during the last data phase. The target indicates that it cannot continue the burst by asserting both STOP# and TRDY# together. When the data phase completes on clock 2, the target is required to deassert TRDY# and keep STOP# asserted. The last data phase completes, without transferring data, on clock 3 because TRDY# is deasserted and STOP# is asserted. In this example, there are three data phases, two that transfer data and one that does not.

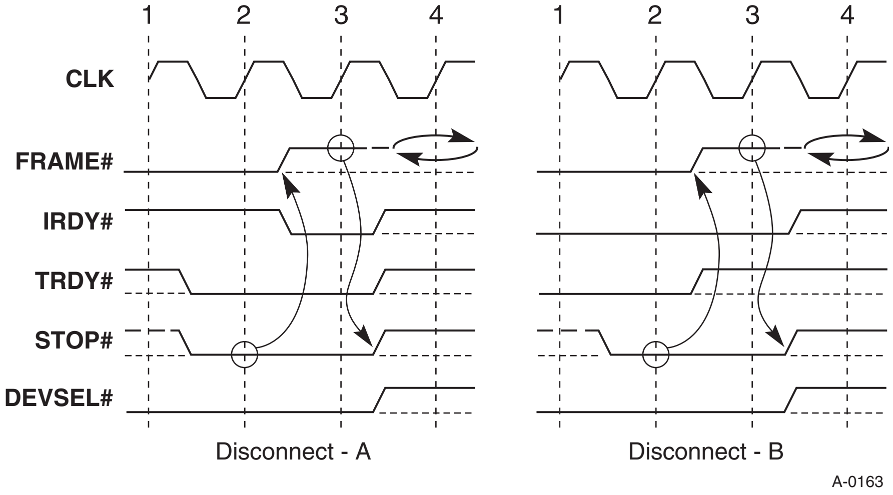


Figure 3-10: Disconnect With Data

Figure 3-11 is an example of Master Completion termination where the target blindly asserts STOP#. This is a legal termination where the master is requesting a transaction with a single data phase and the target blindly asserts STOP# and TRDY# indicating it can complete only a single data phase. The transaction starts like all transactions with the assertion of FRAME#. The master indicates that the initial data phase is the final data phase because FRAME# is deasserted and IRDY# is asserted on clock 3. The target claims the transaction, indicates it is ready to transfer data, and requests the transaction to stop by asserting DEVSEL#, TRDY#, and STOP# all at the same time.

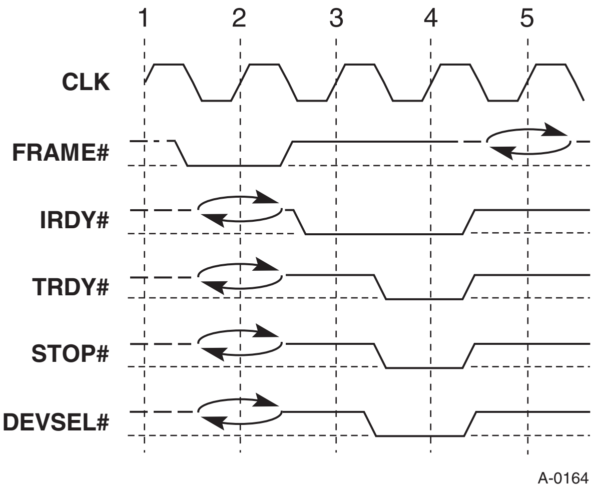


Figure 3-11: Master Completion Termination

Disconnect Without Data

Figure 3-12 shows a transaction being terminated with Disconnect without data. The transaction starts with FRAME# being asserted on clock 2 and IRDY# being asserted on clock 3. The master is requesting multiple data phases because both FRAME# and IRDY# are asserted on clock 3. The target claims the transaction by asserting DEVSEL# on clock 4.

The first data phase completes on clock 4 and the second completes on clock 5. On clock 6, the master wants to continue bursting because FRAME# and IRDY# are still asserted. However, the target cannot complete any more data phases and asserts STOP# and deasserts TRDY# on clock 6. Since IRDY# and STOP# are asserted on clock 6, the third data phase completes. The target continues to keep STOP# asserted on clock 7 because FRAME# is still asserted on clock 6. The fourth and final data phase completes on clock 7 since FRAME# is deasserted (IRDY# is asserted) and STOP# is asserted on clock 7. The bus returns to the Idle state on clock 8.

In this example, the first two data phases complete transferring data while the last two do not. This might happen if a device accepted two DWORDs of data and then determined that its buffers were full or if the burst crossed a resource boundary. The target is able to complete the first two data phases but cannot complete the third. When and if the master continues the burst, the device that owns the address of the next untransferred data will claim the access and continue the burst.

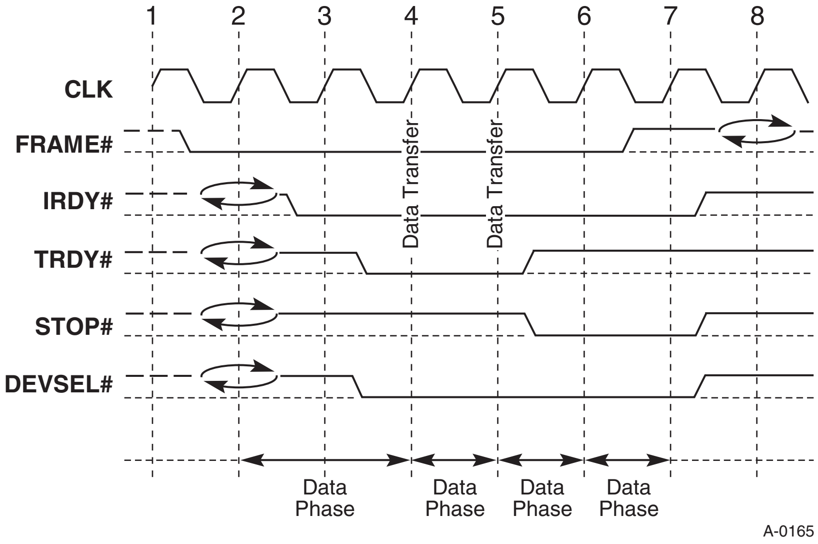


Figure 3-12: Disconnect-1 Without Data Termination

Figure 3-13 shows the same transaction as described in Figure 3-12 except that the master inserts a wait state on clock 6. Since FRAME# was not deasserted on clock 5, the master committed to at least one more data phase and must complete it. The master is not allowed simply to transition the bus to the Idle state by deasserting FRAME# and keeping IRDY# deasserted. This would be a violation of bus protocol. When the master is ready to assert IRDY#, it deasserts FRAME# indicating the last data phase, which completes on clock 7 since STOP# is asserted. This example only consists of three data phases while the previous had four. The fact that the master inserted a wait state allowed the master to complete the transaction with the third data phase. However, from a clock count, the two transactions are the same.

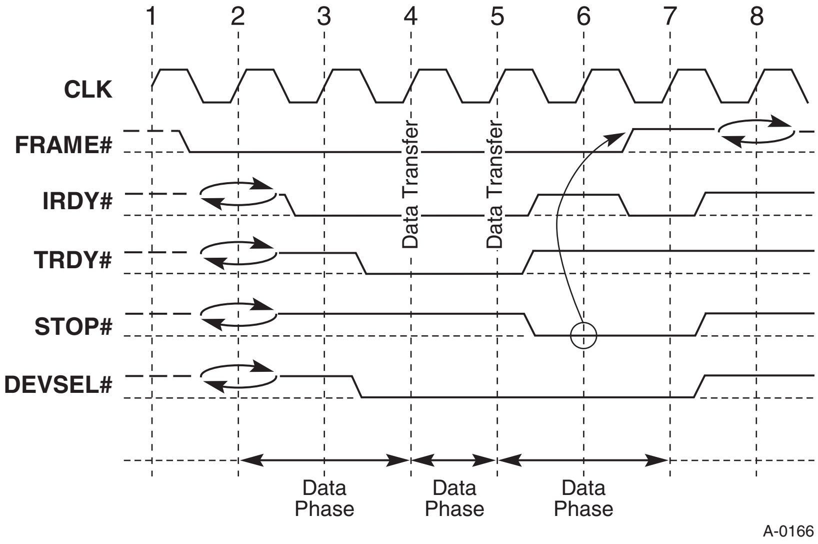


Figure 3-13: Disconnect-2 Without Data Termination

Target-Abort

Figure 3-14 shows a transaction being terminated with Target-Abort. Target-Abort indicates the target requires the transaction to be stopped and does not want the master to repeat the request again. Sometime prior to clock 1, the master asserted **FRAME#** to initiate the request and the target claimed the access by asserting **DEVSEL#**. Data phases may or may not have completed prior to clock 1. The target determines that the master has requested a transaction that the target is incapable of completing or has determined that a fatal error has occurred. Before the target can signal Target-Abort, **DEVSEL#** must be asserted for one or more clocks. To signal Target-Abort, **TRDY#** must be deasserted when **DEVSEL#** is deasserted and **STOP#** is asserted, which occurs on clock 2. If any data was transferred during the previous data phases of the current transaction, it may have been corrupted. Because **STOP#** is asserted on clock 2 and the master can assert **IRDY#** on clock 3, the master deasserts **FRAME#** on clock 3. The transaction completes on clock 3 because **IRDY#** and **STOP#** are asserted. The master deasserts **IRDY#** and the target deasserts **STOP#** on clock 4.

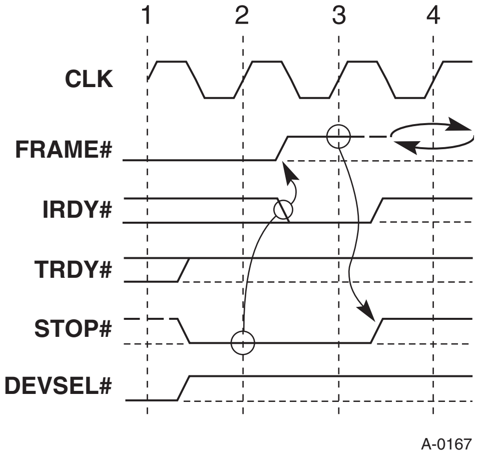


Figure 3-14: Target-Abort

3.3.3.2.2. Requirements on a Master Because of Target Termination

Although not all targets will implement all forms of target termination, masters must be capable of properly dealing with them all.

Deassertion of REQ# When Target Terminated

When the current transaction is terminated by the target either by Retry or Disconnect (with or without data), the master must deassert its REQ# signal before repeating the transaction. A device containing a single source of master activity must deassert REQ# for a minimum of two clocks, one being when the bus goes to the Idle state (at the end of the transaction where STOP# was asserted) and either the clock before or the clock after the Idle state.

Some devices contain multiple sources of master activity that share the same REQ# pin. Examples of such devices include the following:

- ☐ A single function device that contains two independent sub-functions. One that produces data and one that consumes data.
- ☐ A multi-function device.
- ☐ A PCI-to-PCI bridge that is capable of forwarding multiple Delayed Transactions from the other bus.

A device containing multiple sources of master activity that share a single REQ# pin is permitted to allow each source to use the bus (assuming that GNT# is still asserted) without deasserting REQ# even if one or more sources are target terminated (STOP# asserted). However, the device must deassert REQ# for two consecutive clocks, one of which while the bus is Idle, before any transaction that was target terminated can be repeated.

The master is not required to deassert its REQ# when the target requests the transaction to end by asserting STOP# in the last data phase. An example is Figure 3-11 which is really Master Completion termination and not target termination.

Repeat Request Terminated With Retry

A master that is target terminated with Retry must unconditionally repeat the same request until it completes; however, it is not required to repeat the transaction when terminated with Disconnect. "Same request" means that the same address (including AD[1::0]), same command, same byte enables, same write data for write transactions (even if the byte enable for that byte lane is not asserted), and, if supported, LOCK# and REQ64# that were used on the original request must be used when the access is repeated. "Unconditionally" in the above rule means the master must repeat the same transaction that was terminated with Retry independent of any subsequent events (except as noted below) until the original transaction is satisfied.

This does not mean the master must immediately repeat the same transaction. In the simplest form, the master would request use of the bus after the two clocks REQ# was deasserted and repeat the same transaction. The master is permitted to perform other bus transactions, but cannot require them to complete before repeating the original transaction.

If the device also implements target functionality, it must be able to accept accesses during this time as well.

A multi-function device is a good example of how this works. Functions 1, 2, and 3 of a single device are all requesting use of the interface. Function 1 requests a read transaction and is terminated with Retry. Once Function 1 has returned the bus to an Idle state, Function 2 may attempt a transaction (assuming GNT# is still active for the device). After Function 2 releases the bus, Function 3 may proceed if GNT# is still active. Once Function 3 completes, the device must deassert its REQ# for the two clocks before reasserting it. As illustrated above, Function 1 is not required to complete its transaction before another function can request a transaction. But Function 1 must repeat its access regardless of how the transactions initiated by Function 2 or 3 are terminated. The master of a transaction must repeat its transaction unconditionally, which means the repeat of the transaction cannot be gated by any other event or condition.

This rule applies to all transactions that are terminated by Retry regardless of how many previous transactions may have been terminated by Retry. In the example above, if Function 2 attempted to do a transaction and was terminated by Retry, it must repeat that transaction unconditionally just as Function 1 is required to repeat its transaction unconditionally. Neither Function 1 nor Function 2 can depend on the completion of the other function's transaction or the success of any transaction attempted by Function 3 to be able to repeat its original request.

A subsequent transaction (not the original request) could result in the assertion of SERR# or PERR#, or being terminated with Retry, Disconnect, Target-Abort, or Master-Abort. Any of these events would have no effect on the requirement that the master must repeat an access that was terminated with Retry.

A master should repeat a transaction terminated by Retry as soon as possible, preferably within 33 clocks. However, there are a few conditions when a master is unable to repeat the request. These conditions typically are caused when an error occurs; for example, the system asserts RST#, the device driver resets, and then re-initializes the component, or software disables the master by resetting the Bus Master bit (bit 2 in the Command register). Refer to Section 3.3.3.3.3 for a description of how a target using Delayed Transaction termination handles this error condition.

However, when the master repeats the transaction and finally is successful in transferring data, it is not required to continue the transaction past the first data phase.

IMPLEMENTATION NOTE

Potential Temporary Deadlock and Resulting Performance Impacts

The previous paragraph states that a master may perform other bus transactions, but cannot require them to complete before repeating the original transaction (one previously target terminated with Retry). If a master does not meet this requirement, it may cause temporary deadlocks resulting in significant device and system performance impacts. Devices designed prior to Revision 2.1 of this specification may exhibit this behavior. Such temporary deadlocks should eventually clear when the discard timer (refer to Section 3.3.3.3.3) expires.

3.3.3.3. Delayed Transactions

Delayed Transaction termination is used by targets that cannot complete the initial data phase within the requirements of this specification. There are two types of devices that will use Delayed Transactions: I/O controllers and bridges (in particular, PCI-to-PCI bridges). In general, I/O controllers will handle only a single Delayed Transaction at a time, while bridges may choose to handle multiple transactions to improve system performance.

One advantage of a Delayed Transaction is that the bus is not held in wait states while completing an access to a slow device. While the originating master rearbitrates for the bus, other bus masters are allowed to use the bus bandwidth that would normally be wasted holding the master in wait states. Another advantage is that all posted (memory write) data is not required to be flushed before the request is accepted. The actual flushing of the posted memory write data occurs before the Delayed Transaction completes on the originating bus. This allows posting to remain enabled while a nonpostable transaction completes and still maintains the system ordering rules.

The following discussion focuses on the basic operation and requirements of a device that supports a single Delayed Transaction at a time. Section 3.3.3.3.5 extends the basic concepts from support of a single Delayed Transaction to the support of multiple Delayed Transactions at a time.

3.3.3.3.1. Basic Operation of a Delayed Transaction

All bus commands that must complete on the destination bus before completing on the originating bus may be completed as a Delayed Transaction. These include Interrupt Acknowledge, I/O Read, I/O Write, Configuration Read, Configuration Write, Memory Read, Memory Read Line, and Memory Read Multiple commands. Memory Write and Memory Write and Invalidate commands can complete on the originating bus before completing on the destination bus (i.e., can be posted). Each command is not completed using Delayed Transaction termination and are either posted or terminated with Retry. For I/O controllers, the term *destination bus* refers to the internal bus where the resource addressed by the transaction resides. For a bridge, the destination bus means the interface that was not acting as the target of the original request. For example, the secondary bus of a bridge is the destination bus when a transaction originates on the primary bus of the bridge and targets (addresses) a device attached to the secondary bus of the bridge. However, a transaction that is moving in the opposite direction would have the primary bus as the destination bus.

A Delayed Transaction progresses to completion in three steps:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first step, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. The master of a request that is terminated with Retry cannot distinguish between a target which is completing the transaction using Delayed Transaction termination and a target which simply cannot complete the transaction at the current time. Since the master cannot tell the difference, it must reissue any request that has been terminated with Retry until the request completes (refer to Section 3.3.3.3.2.2).

During the second step, the target independently completes the request on the destination bus using the latched information from the Delayed Request. If the Delayed Request is a read, the target obtains the requested data and completion status. If the Delayed Request is a write, the target delivers the write data and obtains the completion status. The result of completing the Delayed Request on the destination bus produces a Delayed Completion, which consists of the latched information of the Delay Request and the completion status (and data if a read request). The target stores the Delayed Completion until the master repeats the initial request.

During the third step, the master successfully rearbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed. The status returned to the master is exactly the same as the target obtained when it executed (completed) the Delayed Request (i.e., Master-Abort, Target-Abort, parity error, normal, Disconnect, etc.).

3.3.3.3.2. Information Required to Complete a Delayed Transaction

To complete a transaction using Delayed Transaction termination, a target must latch the following information:

☐ address

☐ command

☐ byte enables

☐ address and data parity, if the Parity Error Response bit (bit 6 of the command register) is set

☐ REQ64# (if a 64-bit transfer)

For write transactions completed using Delayed Transaction termination, a target must also latch data from byte lanes for which the byte enable is asserted and may optionally latch data from byte lanes for which the byte enable is deasserted. Refer to Appendix F for requirements for a bridge to latch LOCK# when completing a Delayed Transaction.

On a read transaction, the address and command are available during the address phase and the byte enables during the following clock. Byte enables for both read and write transactions are valid the entire data phase and are independent of IRDY#. On a write transaction, all information is valid at the same time as a read transaction, except for the actual data, which is valid only when IRDY# is asserted.

Note: Write data is only valid when IRDY# is asserted. Byte enables are always valid for the entire data phase regardless of the state of IRDY#.

The target differentiates between transactions (by the same or different masters) by comparing the current transaction with information latched previously (for both Delayed Request(s) and Delayed Completion(s)). During a read transaction, the target is not required to use byte enables as part of the comparison, if all bytes are returned independent of the asserted byte enables and the accessed location has no read side effects (prefetchable). If the compare matches a Delayed Request (already enqueued), the target does not enqueue the request again but simply terminates the transaction with Retry indicating that the target is not yet ready to complete the request. If the compare matches a Delayed Completion, the target responds by signaling the status and providing the data if a read transaction.

The master must repeat the transaction exactly as the original request, including write data in all byte lanes (whether the corresponding byte enables are asserted or not). Otherwise, the target will assume it is a new transaction. If the original transaction is never repeated, it will eventually be discarded when the Discard Timer expires (refer to Section 3.3.3.3.3). Two masters could request the exact same transaction and the target cannot and need not distinguish between them and will simply complete the access.

Special requirements apply if a data parity error occurs while initiating or completing a Delayed Transaction. Refer to Section 3.7.5 for details about a parity error and Delayed Transactions.

3.3.3.3.3. Discarding a Delayed Transaction

A device is allowed to discard a Delayed Request from the time it is enqueued until it has been attempted on the destination bus, since the master is required to repeat the request until it completes. Once a Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus and cannot be discarded. The master is allowed to present other requests. But if it attempts more than one request, the master must continue to repeat all requests that have been attempted unconditionally until they complete. The repeating of the requests is not required to be equal, but is required to be fair.

When a Delayed Request completes on the destination bus, it becomes a Delayed Completion. The target device is allowed to discard Delayed Completions in only two cases. The first case is when the Delayed Completion is a read to a prefetchable region (or the command was Memory Read Line or Memory Read Multiple). The second case is for all Delayed Completions (read or write, prefetchable or not) when the master has not repeated the request within 2^{15} clocks. When this timer (referred to as the Discard Timer) expires, the device is required to discard the data; otherwise, a deadlock may occur.

Note: When the transaction is discarded, data may be destroyed. This occurs when the discarded Delayed Completion is a read to a non-prefetchable region.

If the Discard Timer expires, the device may choose to report an error or not. If the data is prefetchable (case 1), it is recommended that the device not report an error, since system integrity is not effected. However, if the data on a read access is not prefetchable (case 2), it is recommended that the device report the error to its device driver since system integrity is affected.

3.3.3.3.4. Memory Writes and Delayed Transactions

While completing a Delayed Request, the target is also required to complete all memory write transactions addressed to it. The target may, from time to time, retry a memory write while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full, or before the Delayed Request has completed on the destination bus (but is guaranteed to complete). However, the target cannot require the Delayed Transaction to complete on the originating bus before accepting the memory write data; otherwise, a deadlock may occur. Refer to Section 3.10, item 6, for additional information. The following implementation note describes the deadlock.

IMPLEMENTATION NOTE

Deadlock When Memory Write Data is Not Accepted

The deadlock occurs when the master and the target of a transaction reside on different buses (or segments). The PCI-to-PCI bridge[18] that connects the two buses together does not implement Delayed Transactions. The master initiates a request that is forwarded to the target by the bridge. The target responds to the request by using Delayed Transaction termination (terminated with Retry). The bridge terminates the master's request with Retry (without latching the request). Another master (on the same bus segment as the original master) posts write data into the bridge targeted at the same device as the read request. Because it is designed to the previous version of this specification, before Delayed Transactions, the bridge is required to flush the memory write data before the read can be repeated. If the target that uses Delayed Transaction termination will not accept the memory write data until the master repeats the initial read, a deadlock occurs because the bridge cannot repeat the request until the target accepts the write data. To prevent this from occurring, the target that uses the Delayed Transaction termination to meet the initial latency requirements is required to accept memory write data even though the Delayed Transaction has not completed.

3.3.3.3.5. Supporting Multiple Delayed Transactions

This section takes the basic concepts of a single Delayed Transaction as described in the previous section and extends them to support multiple Delayed Transactions at the same time. Bridges (in particular, PCI-to-PCI bridges) are the most likely candidates to handle multiple Delayed Transactions as a way to improve system performance and meet the initial latency requirements. To assist in understanding the requirements of supporting multiple Delayed Transactions, the following section focuses on a PCI-to-PCI bridge. This focus allows the same terminology to be used when describing transactions initiated on either interface of the bridge. Most other bridges (host bus bridge and standard expansion bus bridge) will typically handle only a single Delayed Transaction. Supporting multiple transactions is possible but the details may vary. The fundamental requirements in all cases are that transaction ordering be maintained as described in Section 3.2.5. and Section 3.3.3.3.4. and deadlocks will be avoided.

---

[18] This is a bridge that is built to an earlier version of this specification.

Transaction Definitions

PMW - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

DRR - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an Interrupt Acknowledge, I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple command. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Until that time, the DRR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

DWR - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write command. Note: Memory Write and Memory Write and Invalidate commands must be posted (PMW) and not be completed as DWR. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes. Until that time, the DWR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

DRC - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The DRC contains the data requested by the master and the status of the target (normal, Master-Abort, Target-Abort, parity error, etc.).

DWC - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The DWC does not contain the data of the access but only status of how it completed (normal, Master-Abort, Target-Abort, parity error, etc.). The write data has been written to the specified target.

Ordering Rules for Multiple Delayed Transactions

Table 3-3 represents the ordering rules when a bridge in the system is capable of allowing multiple transactions to proceed in each direction at the same time. The number of simultaneous transactions is limited by the implementation and not by the architecture. Because there are five types of transactions that can be handled in each direction, the following table has 25 entries. Of the 25 boxes in the table, only four are required No's, eight are required Yes's, and the remaining 13 are don't cares. The column of the table represents an access that was accepted previously by the bridge, while the row represents a transaction that was accepted subsequent to the access represented by the column. The following table specifies the ordering relationships between transactions as they cross a bridge. For an explanation as to why these rules are required or for a general discussion on system ordering rules, refer to Appendix E for details.

Table 3-3: Ordering Rules for Multiple Delayed Transactions

Row pass Col.?	PMW (Col 2)	DRR (Col 3)	DWR (Col 4)	DRC (Col 5)	DWC (Col 6)
PMW (Row 1)	No	Yes	Yes	Yes	Yes
DRR (Row 2)	No	Yes/No	Yes/No	Yes/No	Yes/No
DWR (Row 3)	No	Yes/No	Yes/No	Yes/No	Yes/No
DRC (Row 4)	No	Yes	Yes	Yes/No	Yes/No
DWC (Row 5)	Yes/No	Yes	Yes	Yes/No	Yes/No

No - indicates the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes are found in column 2 and maintain a consistent view of data in the system as described by the Producer - Consumer Model found in Appendix E. These boxes prevent PMW data from being passed by other accesses.

Yes - The four Yes boxes in Row 1 indicate the PMW must be allowed to complete before Delayed Requests or Delayed Completions moving in the same direction or a deadlock can occur. This prevents deadlocks from occurring when Delayed Transactions are used with devices designed to an earlier version of this specification. A PMW cannot be delayed from completing because a Delayed Request or a Delayed Completion was accepted prior to the PMW. The only thing that can prevent the PMW from completing is gaining access to the bus or the target terminating the attempt with Retry. Both conditions are temporary and will resolve independently of other events. If the master continues attempting to complete Delayed Requests, it must be fair in attempting to complete the PMW. There is no ordering violation when a subsequent transaction completes before a prior transaction.

The four Yes boxes in rows 4 and 5, columns 3 and 4, indicate that Delayed Completions must be allowed to pass Delayed Requests moving in the same direction. This prevents deadlocks from occurring when two bridges that support Delayed Transactions are requesting accesses to each other. If neither bridge allows Delayed Completions to pass the Delayed Requests, neither can make progress.

Yes/No - indicates the bridge may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid. How a bridge designer chooses to implement these boxes may have a cost impact on the bridge implementation or performance impact on the system.

Ordering of Delayed Transactions

The ordering of Delayed Transactions is established when the transaction completes on the originating bus (i.e., the requesting master receives a response other than Retry). Delayed Requests and Delayed Completions are intermediate steps in the process of completing a Delayed Transaction, which occur prior to the completion of the transaction on the originating bus. As a result, reordering is allowed for Delayed Requests with respect to other Delayed Requests, Delayed Requests with respect to Delayed Completions, or for Delayed Completions with respect to other Delayed Completions. However, reordering is not allowed with respect to memory write transactions, which is described in Table 3-3 (the No boxes).

In general, a master does not need to wait for one request to be completed before it issues another request. As described in Section 3.3.3.2.2., a master may have any number of requests terminated with Retry at one time, some of which may be serviced as Delayed Transactions and some not. However, if the master does issue a second request before the first is completed, the master must continue to repeat each of the requests fairly, so that each has a fair opportunity to be completed. If a master has a specific need for two transactions to be completed in a particular order, it must wait for the first one to complete before requesting the second.

3.4. Arbitration

In order to minimize access latency, the PCI arbitration approach is access-based rather than time-slot-based. That is, a bus master must arbitrate for each access it performs on the bus. PCI uses a central arbitration scheme, where each master agent has a unique request (REQ#) and grant (GNT#) signal. A simple request-grant handshake is used to gain access to the bus. Arbitration is "hidden," which means it occurs during the previous access so that no PCI bus cycles are consumed due to arbitration, except when the bus is in an Idle state.

An arbitration algorithm must be defined to establish a basis for a worst case latency guarantee. However, since the arbitration algorithm is fundamentally not part of the bus specification, system designers may elect to modify it, but must provide for the latency requirements of their selected I/O controllers and for add-in cards. Refer to Section 3.5.4 for information on latency guidelines. The bus allows back-to-back transactions by the same agent and allows flexibility for the arbiter to prioritize and weight requests. An arbiter can implement any scheme as long as it is fair and only a single GNT# is asserted on any rising clock.

The arbiter is required to implement a fairness algorithm to avoid deadlocks. In general, the arbiter must advance to a new agent when the current master deasserts its REQ#. Fairness means that each potential master must be granted access to the bus independent of other requests. However, this does not mean that all agents are required to have equal access to the bus. By requiring a fairness algorithm, there are no special conditions to handle when LOCK# is active (assuming a resource lock). A system that uses a fairness algorithm is still considered fair if it implements a complete bus lock instead of resource lock. However, the arbiter must advance to a new agent if the initial transaction attempting to establish the lock is terminated with Retry.

IMPLEMENTATION NOTE

System Arbitration Algorithm

One example of building an arbiter to implement a fairness algorithm is when there are two levels to which bus masters are assigned. In this example, the agents that are assigned to the first level have a greater need to use the bus than agents assigned to the second level (i.e., lower latency or greater throughput). Second level agents have equal access to the bus with respect to other second level agents. However, the second level agents as a group have equal access to the bus as each agent of the first level. An example of how a system may assign agents to a given level is where devices such as video, ATM, or FDDI bus masters would be assigned to Level 1 while devices such as SCSI, LAN, or standard expansion bus masters would be assigned to the second level.

The figure below is an example of a fairness arbitration algorithm that uses two levels of arbitration. The first level consists of Agent A, Agent B, and Level 2, where Level 2 is the next agent at that level requesting access to the bus. Level 2 consists of Agent X, Agent Y, and Agent Z. If all agents on level 1 and 2 have their REQ# lines asserted and continue to assert them, and if Agent A is the next to receive the bus for Level 1 and Agent X is the next for Level 2, then the order of the agents accessing the bus would be:

A, B, Level 2 (this time it is X)

A, B, Level 2 (this time it is Y)

A, B, Level 2 (this time it is Z)

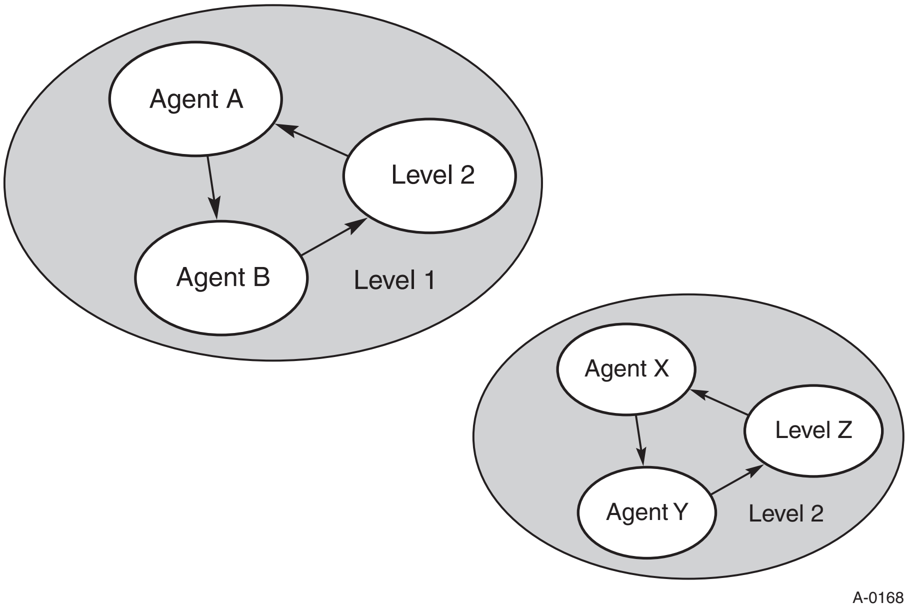
and so forth.

If only Agent B and Agent Y had their REQ#s asserted and continued to assert them, the order would be:

B, Level 2 (Y),

B, Level 2 (Y).

By requiring a fairness arbitration algorithm, the system designer can balance the needs of high performance agents such as video, ATM, or FDDI with lower performance bus devices like LAN and SCSI. Another system designer may put only multimedia devices on arbitration Level 1 and put the FDDI (or ATM), LAN, and SCSI devices on Level 2. These examples achieve the highest level of system performance possible for throughput or lowest latency without possible starvation conditions. The performance of the system can be balanced by allocating a specific amount of bus bandwidth to each agent by careful assignment of each master to an arbitration level and programming each agent's Latency Timer appropriately.



3.4.1. Arbitration Signaling Protocol

An agent requests the bus by asserting its REQ#. Agents must only use REQ# to signal a true need to use the bus. An agent must never use REQ# to "park" itself on the bus. If bus parking is implemented, it is the arbiter that designates the default owner. When the arbiter determines an agent may use the bus, it asserts the agent's GNT#.

The arbiter may deassert an agent's GNT# on any clock. An agent must ensure its GNT# is asserted on the rising clock edge it wants to start a transaction. Note: A master is allowed to start a transaction when its GNT# is asserted and the bus is in an Idle state independent of the state of its REQ#. If GNT# is deasserted, the transaction must not proceed. Once asserted, GNT# may be deasserted according to the following rules:

1. If GNT# is deasserted and FRAME# is asserted on the same clock, the bus transaction is valid and will continue.
2. One GNT# can be deasserted coincident with another GNT# being asserted if the bus is not in the Idle state. Otherwise, a one clock delay is required between the deassertion of a GNT# and the assertion of the next GNT#, or else there may be contention on the AD lines and PAR due to the current master doing IDSEL stepping.
3. While FRAME# is deasserted, GNT# may be deasserted at any time in order to service a higher priority[19] master or in response to the associated REQ# being deasserted.

---

[19] Higher priority here does not imply a fixed priority arbitration, but refers to the agent that would win arbitration at a given instant in time.

Figure 3-15 illustrates basic arbitration. Two agents are used to illustrate how an arbiter may alternate bus accesses.

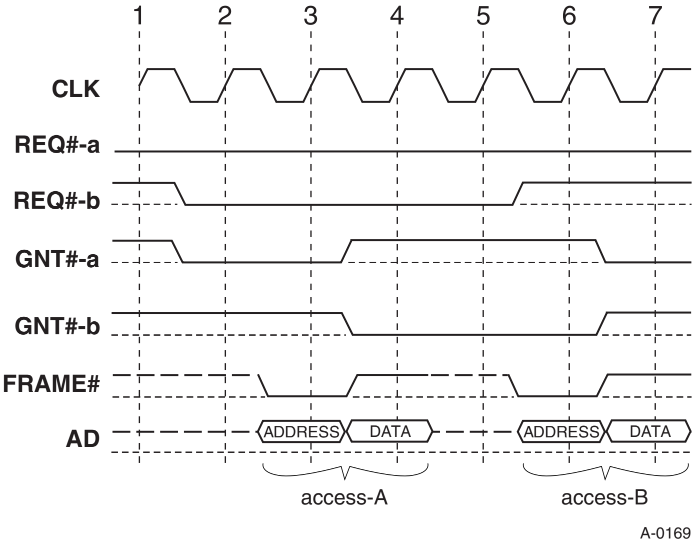


Figure 3-15: Basic Arbitration

REQ#-a is asserted prior to or at clock 1 to request use of the interface. Agent A is granted access to the bus because **GNT#**-a is asserted at clock 2. Agent A may start a transaction at clock 2 because **FRAME#** and **IRDY#** are deasserted and **GNT#**-a is asserted. Agent A's transaction starts when **FRAME#** is asserted on clock 3. Since Agent A desires to perform another transaction, it leaves **REQ#**-a asserted. When **FRAME#** is asserted on clock 3, the arbiter determines Agent B should go next and asserts **GNT#**-b and deasserts **GNT#**-a on clock 4.

When agent A completes its transaction on clock 4, it relinquishes the bus. All PCI agents can determine the end of the current transaction when both **FRAME#** and **IRDY#** are deasserted. Agent B becomes the owner on clock 5 (because **FRAME#** and **IRDY#** are deasserted) and completes its transaction on clock 7.

Notice that **REQ#**-b is deasserted and **FRAME#** is asserted on clock 6 indicating agent B requires only a single transaction. The arbiter grants the next transaction to Agent A because its **REQ#** is still asserted.

The current owner of the bus keeps **REQ#** asserted when it requires additional transactions. If no other requests are asserted or the current master has highest priority, the arbiter continues to grant the bus to the current master.

IMPLEMENTATION NOTE

Bus Parking

When no REQ#s are asserted, it is recommended not to remove the current master's GNT# to park the bus at a different master until the bus enters its Idle state. If the current bus master's GNT# is deasserted, the duration of the current transaction is limited to the value of the Latency Timer. If the master is limited by the Latency Timer, it must rearbitrate for the bus which would waste bus bandwidth. It is recommended to leave GNT# asserted at the current master (when no other REQ#s are asserted) until the bus enters its Idle state. When the bus is in the Idle state and no REQ#s are asserted, the arbiter may park the bus at any agent it desires.

GNT# gives an agent access to the bus for a single transaction. If an agent desires another access, it should continue to assert REQ#. An agent may deassert REQ# anytime, but the arbiter may interpret this to mean the agent no longer requires use of the bus and may deassert its GNT#. An agent should deassert REQ# in the same clock FRAME# is asserted if it only wants to do a single transaction. When a transaction is terminated by a target (STOP# asserted), the master must deassert its REQ# for a minimum of two clocks, one being when the bus goes to the Idle state (at the end of the transaction where STOP# was asserted), and the other being either the clock before or the clock after the Idle state. For an exception, refer to Section 3.3.3.2.2. This allows another agent to use the interface while the previous target prepares for the next access.

The arbiter can assume the current master is "broken" if it has not started an access after its GNT# has been asserted (its REQ# is also asserted) and the bus is in the Idle state for 16 clocks. The arbiter is allowed to ignore any "broken" master's REQ# and may optionally report this condition to the system. However, the arbiter may remove GNT# at any time to service a higher priority agent. A master that has requested use of the bus that does not assert FRAME# when the bus is in the Idle state and its GNT# is asserted faces the possibility of losing its turn on the bus. Note: In a busy system, a master that delays the assertion of FRAME# runs the risk of starvation because the arbiter may grant the bus to another agent. For a master to ensure that it gains access to the bus, it must assert FRAME# the first clock possible when FRAME# and IRDY# are deasserted and its GNT# is asserted.

3.4.2. Fast Back-to-Back Transactions

There are two types of fast back-to-back transactions that can be initiated by the same master: those that access the same agent and those that do not. Fast back-to-back transactions are allowed on PCI when contention on TRDY#, DEVSEL#, STOP#, or PERR# is avoided.

The first type of fast back-to-back support places the burden of avoiding contention on the master, while the second places the burden on all potential targets. The master may remove the Idle state between transactions when it can guarantee that no contention occurs. This can be accomplished when the master's current transaction is to the same target as the previous write transaction. This type of fast back-to-back transaction requires the master to understand the address boundaries of the potential target; otherwise, contention may occur. This type of fast back-to-back is optional for a master but must be decoded by a target. The target must be able to detect a new assertion of FRAME# (from the same master) without the bus going to the Idle state.

The second type of fast back-to-back support places the burden of no contention on all potential targets. The Fast Back-to-Back Capable bit in the Status register may be hardwired to a logical one (high) if, and, only if, the device, while acting as a bus target, meets the following two requirements:

1. The target must not miss the beginning of a bus transaction, nor lose the address, when that transaction is started without a bus Idle state preceding the transaction. In other words, the target is capable of following a bus state transition from a final data transfer (FRAME# high, IRDY# low) directly to an address phase (FRAME# low, IRDY# high) on consecutive clock cycles. Note: The target may or may not be selected on either or both of these transactions, but must track bus states nonetheless.[20]

2. The target must avoid signal conflicts on DEVSEL#, TRDY#, STOP#, and PERR#. If the target does not implement the fastest possible DEVSEL# assertion time, this guarantee is already provided. For those targets that do perform zero wait state decodes, the target must delay assertion of these four signals for a single clock, except in either one of the following two conditions:

   a. The current bus transaction was immediately preceded by a bus Idle state; that is, this is not a back-to-back transaction, or,

   b. The current target had driven DEVSEL# on the previous bus transaction; that is, this is a back-to-back transaction involving the same target as the previous transaction.

   Note: Delaying the assertion of DEVSEL# to avoid contention on fast back-to-back transactions does not affect the decode speed indicated in the status register. A device that normally asserts fast DEVSEL# still indicates "fast" in the status register even though DEVSEL# is delayed by one clock in this case. The status bits associated with decode time are used by the system to allow the subtractive decoding agent to move in the time when it claims unclaimed accesses. However, if the subtractive decode agent claims the access during medium or slow decode time instead of waiting for the subtractive decode time, it must delay the assertion of DEVSEL# when a fast back-to-back transaction is in progress; otherwise, contention on DEVSEL#, STOP#, TRDY#, and PERR# may occur.

For masters that want to perform fast back-to-back transactions that are supported by the target mechanism, the Fast Back-to-Back Enable bit in the Command register is required. (This bit is only meaningful in devices that act as bus masters and is fully optional.) It is a read/write bit when implemented. When set to a one (high), the bus master may start a PCI transaction using fast back-to-back timing without regard to which target is being addressed providing the previous transaction was a write transaction issued by the current bus master.

---

[20] It is recommended that this be done by returning the target state machine (refer to Appendix B) from the B_BUSY state to the IDLE state as soon as FRAME# is deasserted and the device's decode time has been met (a miss occurs) or when DEVSEL# is asserted by another target and not waiting for a bus Idle state (IRDY# deasserted).

If this bit is set to a zero (low) or not implemented, the master may perform fast back-to-back only if it can guarantee that the new transaction goes to the same target as the previous one (master based mechanism).

This bit would be set by the system configuration routine after ensuring that all targets on the same bus had the Fast Back-to-Back Capable Bit set.

Note: The master based fast back-to-back mechanism does not allow these fast cycles to occur with separate targets while the target based mechanism does.

If the target is unable to provide both of the guarantees specified above, it must not implement this bit at all, and it will automatically be returned as a zero when the Status register is read.

Fast back-to-back transactions allow agents to utilize bus bandwidth more effectively. It is recommended that targets and those masters that can improve bus utilization should implement this feature, particularly since the implementation cost is negligible.

Under all other conditions, the master must insert a minimum of one Idle bus state. (Also there is always at least one Idle bus state between transactions by different masters.)
Note: The master is required to cause an Idle state to appear on the bus when the requirements for a fast back-to-back transaction are not met or when bus ownership changes.

During a fast back-to-back transaction, the master starts the next transaction immediately without an Idle bus state (assuming its GNT# is still asserted). If GNT# is deasserted in the last data phase of a transaction, the master has lost access to the bus and must relinquish the bus to the next master. The last data phase completes when FRAME# is deasserted, and IRDY# and TRDY# (or STOP#) are asserted. The current master starts another transaction on the clock following the completion of the last data phase of the previous transaction.

It is important to note that agents not involved in a fast back-to-back transaction sequence cannot (and generally need not) distinguish intermediate transaction boundaries using only FRAME# and IRDY# (there is no bus Idle state). During fast back-to-backs only, the master and target involved need to distinguish these boundaries. When the last transaction is over, all agents will see an Idle state. However, those that do support the target based mechanism must be able to distinguish the completion of all PCI transactions and be able to detect all address phases.

In Figure 3-16, the master completes a write on clock 3 and starts the next transaction on clock 4. The target must begin sampling FRAME# on clock 4 since the previous transaction completed on clock 3; otherwise, it will miss the address of the next transaction. A device must be able to decode back-to-back operations, to determine if it is the current target, while a master may optionally support this function. A target is free to claim ownership by asserting DEVSEL#, then Retry the request.

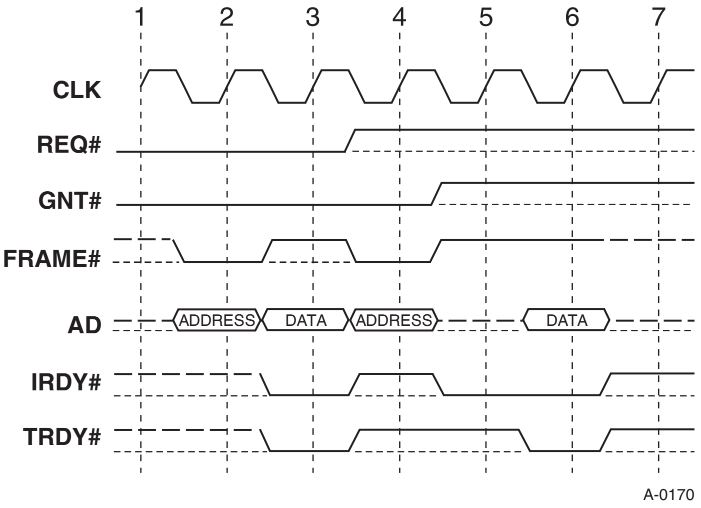


Figure 3-16: Arbitration for Back-to-Back Access

3.4.3. Arbitration Parking

The term *park* implies permission for the arbiter to assert GNT# to a selected agent when no agent is currently using or requesting the bus. The arbiter can select the default owner any way it wants (fixed, last used, etc.) or can choose not to park at all (effectively designating itself the default owner). When the arbiter asserts an agent's GNT# and the bus is in the Idle state, that agent must enable its AD[31::00], C/BE[3::0]#, and (one clock later) PAR output buffers within eight clocks (required), while two-three clocks is recommended. Devices that support the 64-bit extension are permitted but not required to park AD[63::32], C/BE[7::4]#, and PAR64. (Refer to Section 3.7.1 for a description of the timing relationship of PAR to AD). The agent is not compelled to turn on all buffers in a single clock. This requirement ensures that the arbiter can safely park the bus at some agent and know that the bus will not float. (If the arbiter does not park the bus, the central resource device in which the arbiter is embedded typically drives the bus.)

If the bus is in the Idle state and the arbiter removes an agent's GNT#, the agent has lost access to the bus except for one case. The one case is if the arbiter deasserted GNT# coincident with the agent asserting FRAME#. In this case, the master will continue the transaction. Otherwise, the agent must tri-state AD[31::00], C/BE#[3::0], and (one clock later) PAR. Unlike above, the agent must disable all buffers in a single clock to avoid possible contention with the next bus owner.

Given the above, the minimum arbitration latency achievable on PCI from the bus Idle state is as follows:

- ☐ Parked: zero clocks for parked agent, two clocks for others
- ☐ Not Parked: one clock for every agent

When the bus is parked at an agent, the agent is allowed to start a transaction without REQ# being asserted. (A master can start a transaction when the bus is in the Idle state and GNT# is asserted.) When the agent needs to do multiple transactions, it should assert REQ# to inform the arbiter that it intends to do multiple transactions. When a master requires only a single transaction, it should not assert REQ#; otherwise, the arbiter may continue to assert its GNT# when it does not require use of the bus.

3.5. Latency

PCI is a low latency, high throughput I/O bus. Both targets and masters are limited as to the number of wait states they can add to a transaction. Furthermore, each master has a programmable timer limiting its maximum tenure on the bus during times of heavy bus traffic. Given these two limits and the bus arbitration order, worst-case bus acquisition latencies can be predicted with relatively high precision for any PCI bus master. Even bridges to standard expansion buses with long access times (ISA, EISA, or MC) can be designed to have minimal impact on the PCI bus and still keep PCI bus acquisition latency predictable.

3.5.1. Target Latency

Target latency is the number of clocks the target waits before asserting TRDY#. Requirements on the initial data phase are different from those of subsequent data phases.

3.5.1.1. Target Initial Latency

Target initial latency is the number of clocks from the assertion of FRAME# to the assertion of TRDY# which completes the initial data phase, or to the assertion of STOP# in the Retry and Target-Abort cases. This number of clocks varies depending on whether the command is a read or write, and, if a write, whether it can be posted or not. A memory write command should simply be posted by the target in a buffer and written to the final destination later. In this case, the target initial latency is small because the transaction was simply a register to register transfer. Meeting target initial latency on read transactions is more difficult since this latency is a combination of the access time of the storage media (e.g., disk, DRAM, etc.) and the delay of the interface logic. Meeting initial latency on I/O and configuration write transactions are similar to read latency.

Target initial latency requirements depend on the state of system operation. The system can either be operating in initialization-time or run-time. Initialization-time begins when RST# is deasserted and completes 2^{25} PCI clocks later. Run-time follows initialization-time.

If a target is accessed during initialization-time, it is allowed to do any of the following:

- ☐ Ignore the request (except if it is a boot device)
- ☐ Claim the access and hold in wait states until it can complete the request, not to exceed the end of initialization-time
- ☐ Claim the access and terminate with Retry

If a target is accessed during run-time (RST# has been deasserted greater than 2^{25} clocks), it must complete the initial data phase of a transaction (read or write) within 16 clocks from the assertion of FRAME#. The target completes the initial data phase by asserting TRDY#

(to accept or provide the requested data) or by terminating the request by asserting STOP# within the target initial latency requirement.

Host bus bridges are granted an additional 16 clocks, to a maximum of 32 clocks, to complete the initial data phase when the access hits a modified line in a cache. However, the host bus bridge can never exceed 32 clocks on any initial data phase.

In most designs, the initial data phase latency is known when the device is designed. If the time required to complete the initial data phase will normally exceed the maximum target initial latency specification, the device must terminate the transaction with Retry as soon as possible and execute the transaction as a Delayed Transaction.

In the unusual case in which the initial data phase latency cannot be determined in advance, the target is allowed to implement a counter that causes the target to assert STOP# and to begin execution of the transaction as a Delayed Transaction on or before the sixteenth clock, if TRDY# is not asserted sooner. A target device that waits for an initial data phase latency counter to expire prior to beginning a Delayed Transaction reduces PCI bandwidth available to other agents and limits transaction efficiency. Therefore, this behavior is strongly discouraged.

IMPLEMENTATION NOTE

Working with Older Targets that Violate the Target Initial Latency Specification

All new target devices must adhere to the 16 clock initial latency requirement except as noted above. However, a new master should not depend on targets meeting the 16 clock maximum initial access latency for functional operation (in the near term), but must function normally (albeit with reduced performance) since systems and devices were designed and built against an earlier version of this specification and may not meet the new requirements. New devices should work with existing devices.

Three options are given to targets to meet the initial latency requirements. Most targets will use either Option 1 or Option 2. Those devices unable to use Option 1 or Option 2 are required to use Option 3.

Option 1 is for a device that always transfers data (asserts TRDY#) within 16 clocks from the assertion of FRAME#.

Note: The majority of I/O controllers built prior to revision 2.1 of this specification will meet the initial latency requirements using Option 1. In this case, the target always asserts TRDY# to complete the initial data phase of the transaction within 16 clocks of the assertion of FRAME#.

Option 2 is for devices that normally transfer data within 16 clocks, but under some specific conditions will exceed the initial latency requirement. Under these conditions, the device terminates the access with Retry within 16 clocks from the assertion of FRAME#.

For devices that cannot use Option 1, a small modification may be required to meet the initial latency requirements as described by Option 2. This option is used by a target that can normally complete the initial data phase within 16 clocks (same as Option 1), but occasionally will take longer and uses the assertion of STOP# to meet the initial latency requirement. It then becomes the responsibility of the master to attempt the transaction again at a later time. A target is permitted to do this only when there is a high probability the target will be able to complete the transaction when the master repeats the request; otherwise, the target must use Option 3.

IMPLEMENTATION NOTE

An Example of Option 2

Consider a simple graphic device that normally responds to a request within 16 clocks but under special conditions, such as refreshing the screen, the internal bus is “busy” and prevents data from transferring. In this case, the target terminates the access with Retry knowing the master will repeat the transaction and the target will most likely be able to complete the transfer then.

The device could have an internal signal that indicates to the bus interface unit that the internal bus is busy and data cannot be transferred at this time. This allows the device to claim the access (asserts DEVSEL#) and immediately terminate the access with Retry. By doing this instead of terminating the transaction 16 clocks after the assertion of FRAME#, other agents can use the bus.

Option 3 is for a device that frequently cannot transfer data within 16 clocks. This option requires the device to use Delayed Transactions which are discussed in detail in Section 3.3.3.3.

Those devices that cannot meet the requirements of Option 1 or 2 are required to use Option 3. This option is used by devices that under normal conditions cannot complete the transaction within the initial latency requirements. An example could be an I/O controller that has several internal functions contending with the PCI interface to access an internal resource. Another example could be a device that acts like a bridge to another device or bus where the initial latency to complete the access may be greater than 16 clocks. The most common types of bridges are host bus bridges, standard expansion bus bridges, and PCI-to-PCI bridges.

IMPLEMENTATION NOTE

Using More Than One Option to Meet Initial Latency

A combination of the different options may be used based on the access latency of a particular device. For example, a graphics controller may meet the initial latency requirements using Option 1 when accessing configuration or internal (I/O or memory mapped) registers. However, it may be required to use Option 2 or in some cases Option 3 when accessing the frame buffer.

3.5.1.2. Target Subsequent Latency

Target subsequent latency is the number of clocks from the assertion of IRDY# and TRDY# for one data phase to the assertion of TRDY# or STOP# for the next data phase in a burst transfer. The target is required to complete a subsequent data phase within eight clocks from the completion of the previous data phase. This requires the target to complete the data phase either by transferring data (TRDY# asserted), by doing target Disconnect without data (STOP# asserted, TRDY# deasserted), or by doing Target-Abort (STOP# asserted, DEVSEL# deasserted) within the target subsequent latency requirement.

In most designs, the latency to complete a subsequent data phase is known when the device is being designed. In this case, the target must manipulate TRDY# and STOP# so as to end the transaction (subsequent data phase) upon completion of data phase "N" (where N=1, 2, 3, ...), if incremental latency to data phase "N+1" is greater than eight clocks. For example, assume a PCI master read from an expansion bus takes a minimum of 15 clocks to complete each data phase. Applying the rule for N = 1, the incremental latency to data phase 2 is 15 clocks; thus, the target must terminate upon completion of data phase 1 (i.e., a target this slow must break attempted bursts on data phase boundaries).

For designs where the latency to complete a subsequent data phase cannot be determined in advance, the target is allowed to implement a counter that causes the target to assert STOP# before or during the eighth clock if TRDY# is not asserted. If TRDY# is asserted before the count expires, the counter is reset and the target continues the transaction.

3.5.2. Master Data Latency

Master data latency is the number of clocks the master takes to assert IRDY# indicating it is ready to transfer data. All masters are required to assert IRDY# within eight clocks of the assertion of FRAME# on the initial data phase and within eight clocks on all subsequent data phases. Generally in the first data phase of a transaction, there is no reason for a master to delay the assertion of IRDY# more than one or two clocks for a write transaction. The master should never delay the assertion of IRDY# on a read transaction. If the master has no buffer available to store the read data, it should delay requesting use of the bus until a buffer is available. On a write transaction, the master should have the data available before requesting the bus to transfer the data. Data transfers on PCI should be done as register to register transfers to maximize performance.

3.5.3. Memory Write Maximum Completion Time Limit

A target may, from time to time, terminate a memory write transaction with Retry while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full or during a video screen refresh. However, a target is not permitted to terminate memory write transactions with Retry indefinitely.

After a target terminates a memory write transaction with Retry, it is required to be ready to complete at least one data phase of a memory write within a specified number of PCI clock cycles from the first Retry termination. This specified number of clock cycles is 334 clocks for systems running at 33 MHz or slower and 668 clocks for systems running at 66 MHz. This time limit, which translates to 10 microseconds at maximum frequencies (33 MHz and 66 MHz), is called the Maximum Completion Time. If a target is presented with multiple memory write requests, the Maximum Completion Time is measured from the time the first memory write transaction is terminated with Retry until the time the first data phase of any memory write to the target is completed with something other than Retry. Once a non-Retry termination has occurred, the Maximum Completion Time limit starts over again with the next Retry termination.

The actual time that the data phase completes will also depend upon when the master repeats the transaction. Targets must be designed to meet the Maximum Completion Time requirements assuming the master will repeat the memory write transaction precisely at the limit of the Maximum Completion Time.

IMPLEMENTATION NOTE

Meeting Maximum Completion Time Limit by Restricting Use of the Device

Some target hardware designs may not be able to process every memory write transaction within the Maximum Completion Time. An example is writing to a command queue where commands can take longer than the Maximum Completion Time to complete. Subsequent writes to such a target when it is currently processing a previous write could experience completion times that are longer than the Maximum Completion Time. Devices that take longer than the Maximum Completion Time to process some memory write transaction must restrict the usage of the device to prevent write transactions when the device cannot complete them within the Maximum Completion Time. This is typically done by the device driver and is accomplished by limiting the rate at which memory writes are issued to the device, or by reading the device to determine that a buffer is available before the write transaction is issued.

Bridge devices (Base Class = 06h) are exempt from the Maximum Completion Time requirement for any requests that move data across the bridge. Bridge devices must follow the Maximum Completion Time requirement for transactions that address locations within (or associated with) the bridge.

The Maximum Completion Time requirement is not in effect during device initialization time, which is defined as the 2^{25} PCI clocks immediately following the deassertion of RST#.

Even though targets are required to complete memory write transactions within the Maximum Completion Time, masters cannot rely on memory write transactions completing within this time. A transaction may flow through a PCI-to-PCI bridge or be one of multiple transactions to a target. In both of these cases, the actual completion time may exceed the normal limit.

3.5.4. Arbitration Latency

Arbitration latency is the number of clocks from when a master asserts its REQ# until the bus reaches an Idle state *and* the master's GNT# is asserted. In a lightly loaded system, arbitration latency will generally just be the time for the bus arbiter to assert the master's GNT#. If a transaction is in progress when the master's GNT# is asserted, the master must wait the additional time for the current transaction to complete.

The total arbitration latency for a master is a function of how many other masters are granted the bus before it, and how long each one keeps the bus. The number of other masters granted the bus is determined by the bus arbiter as discussed in Section 3.4. Each master's tenure on the bus is limited by its master Latency Timer when its GNT# has been deasserted.

The master Latency Timer is a programmable timer in each master's Configuration Space (refer to Section 6.2.4). It is required for each master that is capable of bursting more than two data phases. Each master's Latency Timer is cleared and suspended whenever it is not asserting FRAME#. When a master asserts FRAME#, it enables its Latency Timer to count. The master's behavior upon expiration of the Latency Timer depends on what command is being used and the state of FRAME# and GNT# when the Latency Timer expires.

- ☐ If the master deasserts FRAME# prior to or on the same clock that the counter expires, the Latency Timer is meaningless. The cycle terminates as it normally would when the current data phase completes.
- ☐ If FRAME# is asserted when the Latency Timer expires, and the command *is not* Memory Write and Invalidate, the master must initiate transaction termination when GNT# is deasserted, following the rules described in Section 3.3.3.1. In this case, the master has committed to the target that it will complete the current data phase and one more (the final data phase is indicated when FRAME# is deasserted).
- ☐ If FRAME# is asserted when the Latency Timer expires, the command *is* Memory Write and Invalidate, and the current data phase *is not* transferring the last DWORD of the current cacheline when GNT# is deasserted, the master must terminate the transaction at the end of the current cacheline (or when STOP# is asserted).
- ☐ If FRAME# is asserted when the Latency Timer expires, the command *is* Memory Write and Invalidate, and the current data phase *is* transferring the last DWORD of the current cacheline when GNT# is deasserted, the master must terminate the transaction at the end of the *next* cacheline. (This is required since the master committed to the target at least one more data phase, which would be the beginning of the next cacheline which it must complete, unless STOP# is asserted.)

In essence, the value programmed into the Latency Timer represents a minimum guaranteed number of clocks allotted to the master, after which it must surrender tenure as soon as possible after its GNT# is deasserted. The actual duration of a transaction (assuming its GNT# is deasserted) can be from a minimum of the Latency Timer value plus one clock to a maximum of the Latency Timer value plus the number of clocks required to complete an entire cacheline transfer (unless the target asserts STOP#).

3.5.4.1. Bandwidth and Latency Considerations

In PCI systems, there is a tradeoff between the desire to achieve low latency and the desire to achieve high bandwidth (throughput). High throughput is achieved by allowing devices to use long burst transfers. Low latency is achieved by reducing the maximum burst transfer length. The following discussion is provided (for a 32-bit bus) to illustrate this tradeoff.

A given PCI bus master introduces latency on PCI each time it uses the PCI bus to do a transaction. This latency is a function of the behavior of both the master and the target device during the transaction as well as the state of the master's GNT# signal. The bus command used, transaction burst length, master data latency for each data phase, and the Latency Timer are the primary parameters which control the master's behavior. The bus command used, target latency, and target subsequent latency are the primary parameters which control the target's behavior.

A master is required to assert its IRDY# within eight clocks for any given data phase (initial and subsequent). For the first data phase, a target is required to assert its TRDY# or STOP# within 16 clocks from the assertion of FRAME# (unless the access hits a modified cacheline in which case 32 clocks are allowed for host bus bridges). For all subsequent data phases in a burst transfer, the target must assert its TRDY# or STOP# within eight clocks. If the effects of the Latency Timer are ignored, it is a straightforward exercise to develop equations for the worst case latencies that a PCI bus master can introduce from these specification requirements.

latency_max (clocks) = 32 + 8 * (n-1) if a modified cacheline is hit (for a host bus bridge only)

or = 16 + 8 * (n-1) if not a modified cacheline

where n is the total number of data transfers in the transaction

However, it is more useful to consider transactions that exhibit typical behavior. PCI is designed so that data transfers between a bus master and a target occur as register to register transfers. Therefore, bus masters typically do not insert wait states since they only request transactions when they are prepared to transfer data. Targets typically have an initial access latency less than the 16 (32 for modified cacheline hit for host bus bridge) clock maximum allowed. Once targets begin transferring data (complete their first data phase), they are typically able to sustain burst transfers at full rate (one clock per data phase) until the transaction is either completed by the master or the target's buffers are filled or are temporarily empty. The target can use the target Disconnect protocol to terminate the burst transaction early when its buffers fill or temporarily empty during the transaction. Using these more realistic considerations, the worst case latency equations can be modified to give a typical latency (assuming that the target's initial data phase latency is eight clocks) again ignoring the effects of the Latency Timer.

latency_typical (clocks) = 8 + (n-1)

If a master were allowed to burst indefinitely with a target which could absorb or source the data indefinitely, then there would be no upper bound on the latency which a master could introduce into a PCI system. However, the master Latency Timer provides a mechanism to constrain a master's tenure on the bus (when other bus masters need to use the bus).

In effect, the Latency Timer controls the tradeoff between high throughput (higher Latency Timer values) and low latency (lower Latency Timer values). Table 3-4 shows the latency for different burst length transfers using the following assumptions:

- ☐ The initial latency introduced by the master or target is eight clocks.
- ☐ There is no latency on subsequent data phases (IRDY# and TRDY# are always asserted).
- ☐ The number of data phases are powers of two because these are easy to correlate to cacheline sizes.
- ☐ The Latency Timer values were chosen to expire during the next to last data phase, which allows the master to complete the correct number of data phases.

For example, with a Latency Timer of 14 and a target initial latency of 8, the Latency Timer expires during the seventh data phase. The transaction completes with the eighth data phase.

Table 3-4: Latency for Different Burst Length Transfers

Data Phases	Bytes Transferred	Total Clocks	Latency Timer (clocks)	Bandwidth (MB/s)	Latency (μs)
8	32	16	14	60	.48
16	64	24	22	80	.72
32	128	40	38	96	1.20
64	256	72	70	107	2.16

In Table 3-4, the columns have the following meaning:

Data Phases	Number of data phases completed during transaction
Bytes Transferred	Total number of bytes transferred during transaction (assuming 32-bit transfers)
Total Clocks	Total number of clocks used to complete the transfer **total_clocks = 8 + (n-1) + 1** (Idle time on bus)
Latency Timer	Latency Timer value in clocks such that the Latency Timer expires in next to last data phase **latency_timer = total_clocks - 2**
Bandwidth	Calculated bandwidth in MB/s **bandwidth = bytes_transferred/(total clocks * 30 ns)**

Latency	Latency in microseconds introduced by transaction

latency = total clocks * 30 ns

Table 3-4 clearly shows that as the burst length increases, the amount of data transferred increases. Note: The amount of data doubles between each row in the table, while the latency increases by less than double. The amount of data transferred between the first row and the last row increases by a factor of 8, while the latency increases by a factor of 4.5. The longer the transaction (more data phases), the more efficiently the bus is being used. However, this increase in efficiency comes at the expense of larger buffers.

3.5.4.2. Determining Arbitration Latency

Arbitration latency is the number of clocks a master must wait after asserting its REQ# before it can begin a transaction. This number is a function of the arbitration algorithm of the system; i.e., the sequence in which masters are given access to the bus and the value of the Latency Timer of each master. Since these factors will vary from system to system, the best an individual master can do is to pick a configuration that is considered the typical case and apply the latency discussion to it to determine the latency a device will experience.

Arbitration latency is also affected by the loading of the system and how efficient the bus is being used. The following two examples illustrate a lightly and heavily loaded system where the bus (PCI) is 32-bit. The lightly loaded example is the more typical case of systems today, while the second is more of a theoretical maximum.

Lightly Loaded System

For this example, assume that no other REQ#s are asserted and the bus is either in the Idle state or that a master is currently using the bus. Since no other REQ#s are asserted, as soon as Agent A's REQ# is asserted, the arbiter will assert its GNT# on the next evaluation of the REQ# lines. In this case, Agent A's GNT# will be asserted within a few clocks. Agent A gains access to the bus when the bus is in the Idle state (assuming its GNT# is still active).

Heavily Loaded System

This example will use the arbiter described in the implementation note in Section 3.4. Assume that all agents have their REQ# lines asserted and all want to transfer more data than their Latency Timers allow. To start the sequence, assume that the next bus master is Agent A on level 1 and Agent X on level 2. In this example, Agent A has a very small number of clocks before it gains access to the bus, while Agent Z has the largest number. In this example, Agents A and B each get a turn before an Agent at Level 2. Therefore, Agents A and B each get three turns on the bus, and Agents X and Y each get one turn before Agent Z gets a turn. Arbitration latency (in this example) can be as short as a few clocks for Agent A or (assuming a Latency Timer of 22 clocks) as long as 176 clocks (8 masters * 22 clocks/master) for Agent Z. Just to keep this in perspective, the heavily loaded system is constantly moving about 90 MB/s of data (assuming target initial latency of eight clocks and target subsequent latency of one clock).

As seen in the example, a master experiences its maximum arbitration latency when all the other masters use the bus up to the limits of their Latency Timers. The probability of this happening increases as the loading of the bus increases. In a lightly loaded system, fewer masters will need to use the bus or will use it less than their Latency Timer would allow, thus allowing quicker access by the other masters.

How efficiently each agent uses the bus will also affect average arbitration latencies. The more wait states a master or target inserts on each transaction, the longer each transaction will take, thus increasing the probability that each master will use the bus up to the limit of its Latency Timer.

The following examples illustrate the impact on arbitration latency as the efficiency of the bus goes down due to wait states being inserted. In both examples, the system has a single arbitration level, the Latency Timer is set to 22 and there are five masters that have data to move. A Latency Timer of 22 allows each master to move a 64-byte cacheline if initial latency is only eight clocks and subsequent latency is one clock. The high bus efficiency example illustrates that the impact on arbitration latency is small when the bus is being used efficiently.

System with High Bus Efficiency

In this example, each master is able to move an entire 64-byte cacheline before its respective Latency Timer expires. This example assumes that each master is ready to transfer another cacheline just after it completes its current transaction. In this example, the Latency Timer has no affect. It takes the master

[(1 idle clock) + (8 initial TRDY# clocks)+ (15 subsequent TRDY# clocks)]
* 30 ns/clock = 720 ns

to complete each cacheline transfer.

If all five masters use the same number of clocks, then each master will have to wait for the other four, or

720 ns/master * 4 other masters = 2.9 μs

between accesses. Each master moves data at about 90 MB/s.

The Low Bus Efficiency example illustrates the impact on arbitration latency as a result of the bus being used inefficiently. The first effect is that the Latency Timer expires. The second effect is that is takes two transactions to complete a single cacheline transfer which causes the loading to increase.

System with Low Bus Efficiency

This example keeps the target initial latency the same but increases the subsequent latency (master or target induced) from 1 to 2. In this example, the Latency Timer will expire before the master has transferred the full 64-byte cacheline. When the Latency Timer expires, GNT# is deasserted, and FRAME# is asserted, the master must stop the transaction prematurely and completes the final two data phases it has committed to complete (unless a MWI command in which case it completes the current cacheline). Each master's tenure on the bus would be

[(1 idle clock) + (22 Latency Timer clocks)+
(2 * 2 subsequent TRDY# clocks)]
* 30 ns/clock = 810 ns

and each master has to wait

810 ns/master * 4 other masters = 3.2 µs

between accesses. However, the master only took slightly more time than the High Bus Efficiency example, but only completed nine data phases (36 bytes, just over half a cacheline) instead of 16 data phases. Each master moves data at only about 44 MB/s.

The arbitration latency in the Low Bus Efficiency example is 3 µs instead of 2.9 µs as in the High Bus Efficiency example; but it took the master two transactions to complete the transfer of a single cacheline. This doubled the loading of the system without increasing the data throughput. This resulted from simply adding a single wait state to each data phase.

Also, note that the above description assumes that all five masters are in the same arbitration level. When a master is in a lower arbitration level or resides behind a PCI-to-PCI bridge, it will experience longer latencies between accesses when the primary PCI bus is in use.

The maximum limits of a target and master data latency in this specification are provided for instantaneous conditions while the recommendations are used for normal behavior. An example of an instantaneous condition is when the device is unable to continue completing a data phase on each clock. Rather than stopping the transfer (introducing the overhead of re-arbitration and target initial latency), the target would insert a couple of wait states and continue the burst by completing a data phase on each clock. The maximum limits are not intended to be used on every data phase, but rather on those rare occasions when data is temporarily unable to transfer.

The following discussion assumes that devices are compliant with the specification and have been designed to minimize their impact on the bus. For example, a master is required to assert IRDY# within eight clocks for all data phases; however, it is recommended that it assert IRDY# within one or two clocks.

Example of a System

The following system configuration and the bandwidth each device requires are generous and exceed the needs of current implementations. The system that will be used for a discussion about latency is a workstation comprised of:

Host bus bridge (with integrated memory controller)
Graphics device (VGA and enhanced graphics)
Frame grabber (for video conferencing)
LAN connection
Disk (a single spindle, IDE or SCSI)
Standard expansion bus bridge (PCI to ISA)
A PCI-to-PCI bridge for providing more than three add-in slots

The graphics controller is capable of sinking 50 MB/s. This assumes that the host bus bridge generates 30 MB/s and the frame grabber generates 20 MB/s.

The LAN controller requires only about 4 MB/s (100 Mb) on average (workstation requirements) and is typically much less.

The disk controller can move about 5 MB/s.

The standard expansion bus provides a cost effective way of connecting standard I/O controllers (i.e., keyboard, mouse, serial, parallel ports, etc.) and masters on this bus place a maximum of about 4 MB/s (aggregate plus overhead) on PCI and will decrease in future systems.

The PCI-to-PCI bridge, in and of itself, does not use PCI bandwidth, but a place holder of 9 MB/s is allocated for devices that reside behind it.

The total bandwidth needs of the system is about 72 MB/s (50 + 4 + 5 + 4 + 9) if all devices want to use the bus at the same time.

To show that the bus can handle all the devices, these bandwidth numbers will be used in the following discussion. The probability of all devices requiring use of the bus at the same time is extremely low, and the typical latency will be much lower than the worst cases number discussed. For this discussion, the typical numbers used are at a steady state condition where the system has been operating for a while and not all devices require access to the bus at the same time.

Table 3-5 lists the requirements of each device in the target system and how many transactions each device must complete to sustain its bandwidth requirements within 10 μs time slices.

The first column identifies the device generating the data transfer.

The second column is the total bandwidth the device needs.

The third column is the approximate number of bytes that need to be transferred during this 10 μs time slice.

The fourth column is the amount of time required to move the data.

The last column indicates how many different transactions that are required to move the data. This assumes that the entire transfer cannot be completed as a single transaction.

Table 3-5: Example System

Device	Bandwidth (MB/s)	Bytes/10 μs	Time Used (μs)	Number of Transactions per Slice	Notes
Graphics	50	500	6.15	10	1
LAN	4	40	0.54	1	2
Disk	5	50	0.63	1	3
ISA bridge	4	40	0.78	2	4
PCI-to PCI bridge	9	90	1.17	2	5
Total	72	720	9.27	16	

Notes:

1. Graphics is a combination of host bus bridge and frame grabber writing data to the frame buffer. The host moves 300 bytes using five transactions with 15 data phases each, assuming eight clocks of target initial latency. The frame grabber moves 200 bytes using five transactions with 10 data phases each, assuming eight clocks of target initial latency.
2. The LAN uses a single transaction with 10 data phases with eight clocks of target initial latency.
3. The disk uses a single transaction with 13 data phases with eight clocks of target initial latency.
4. The ISA bridge uses two transactions with five data phases each with eight clocks of target initial latency.
5. The PCI-to-PCI bridge uses two transactions. One transaction is similar to the LAN and the second is similar to the disk requirements.

If the targeted system only needs full motion video or a frame grabber but not both, then replace the Graphics row in Table 3-5 with the appropriate row in Table 3-6. In either case, the total bandwidth required on PCI is reduced.

Table 3-6: Frame Grabber or Full Motion Video Example

Device	Bandwidth (MB/s)	Bytes/10 μs	Time Used (μs)	Number of Transactions per Slice	Notes
Host writing to the frame buffer	40	400	4.2	5	1
Frame grabber	20	200	3.7	5	2

Notes:

1. The host uses five transactions with 20 data phases each, assuming eight clocks of target initial latency.
2. The frame grabber uses five transactions with 10 data phases each, assuming eight clocks of target initial latency.

The totals for Table 3-5 indicate that within a 10 μs window, all the devices listed in the table move the data they required for that time slice. In a real system, not all devices need to move data all the time, but they may be able to move more data in a single transaction. When devices move data more efficiently, the latency each device experiences is reduced.

If the above system supported the arbiter illustrated in the System Arbitration Algorithm Implementation Note (refer to Section 3.4), the frame grabber (or graphics device when it is a master) and the PCI-to-PCI bridge would be put in the highest level. All other devices would be put in the lower level (i.e., level two). Table 3-5 shows that if all devices provide 10 μs of buffering, they would not experience underruns or overruns. However, for devices that move large blocks of data and are generally given higher priority in a system, then a latency of 3 μs is reasonable. (When only two agents are at the highest level, each experiences about 2 μs of delay between transactions. The table assumes that the target is able to consume all data as a single transaction.)

3.5.4.3. Determining Buffer Requirements

Each device that interfaces to the bus needs buffering to match the rate the device produces or consumes data with the rate that it can move data across the bus. The size of buffering can be determined by several factors based on the functionality of the device and the rate at which it handles data. As discussed in the previous section, the arbitration latency a master experiences and how efficiently data is transferred on the bus will affect the amount of buffering a device requires.

In some cases, a small amount of buffering is required to handle errors, while more buffering may give better bus utilization. For devices which do not use the bus very much (devices which rarely require more than 5 MB/s), it is recommended that a minimum of four DWORDs of buffering be supported to ensure that transactions on the bus are done with reasonable efficiency. Moving data as entire cachelines is the preferred transfer size. Transactions less than four DWORDs in length are inefficient and waste bus bandwidth. For devices which use the bus a lot (devices which frequently require more than 5 MB/s), it is recommended that a minimum of 32 DWORDs of buffering be supported to ensure that transactions on the bus are done efficiently. Devices that do not use the bus efficiently will have a negative impact on system performance and a larger impact on future systems.

While these recommendations are minimums, the real amount of buffering a device needs is directly proportional to the difficulty required to recover from an underrun or overrun. For example, a disk controller would provide sufficient buffering to move data efficiently across PCI, but would provide no additional buffering for underruns and overruns (since they will not occur). When data is not available to write to the disk, the controller would just wait until data is available. For reads, when a buffer is not available, it simply does not accept any new data.

A frame grabber must empty its buffers before new data arrives or data is destroyed. For systems that require good video performance, the system designer needs to provide a way for that agent to be given sufficient bus bandwidth to prevent data corruption. This can be accomplished by providing an arbiter that has different levels and/or adjusting the Latency Timer of other masters to limit their tenure on the bus.

The key for future systems is to have all devices use the bus as efficiently as possible. This means to move as much data as possible (preferably several cachelines) in the smallest number of clocks (preferably one clock subsequent latency). As devices do this, the entire system experiences greater throughput and lower latencies. Lower latencies allow smaller buffers to be provided in individual devices. Future benchmarks will allow system designers to distinguish between devices that use the bus efficiently and those that do not. Those that do will enable systems to be built that meet the demands of multimedia systems.

3.6. Other Bus Operations

3.6.1. Device Selection

DEVSEL# is driven by the target of the current transaction as shown in Figure 3-17 to indicate that it is responding to the transaction. DEVSEL# may be driven one, two, or three clocks following the address phase. Each target indicates the DEVSEL# timing it uses in its Configuration Space Status register described in Section 6.2.3. DEVSEL# must be asserted with or prior to the edge at which the target enables its TRDY#, STOP#, and data if a read transaction. In other words, a target must assert DEVSEL# (claim the transaction) before or coincident with signaling any other target response. Once DEVSEL# has been asserted, it cannot be deasserted until the last data phase has completed, except to signal Target-Abort. Refer to Section 3.3.3.2 for more information.

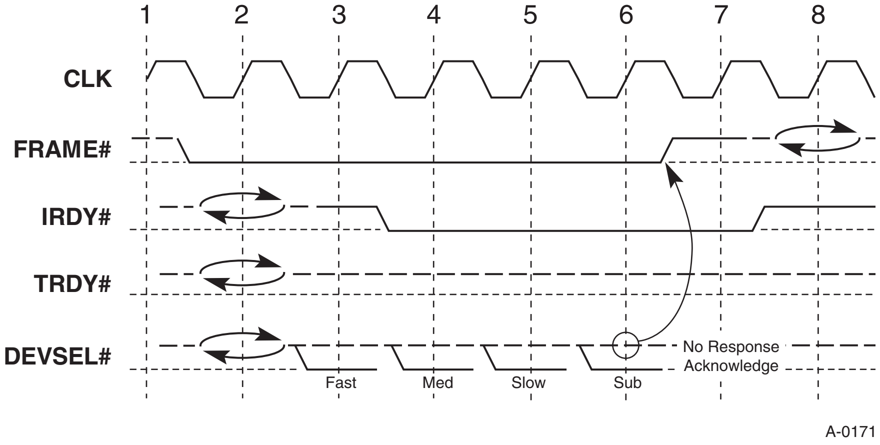


Figure 3-17: **DEVSEL#** Assertion

If no agent asserts DEVSEL# within three clocks of FRAME#, the agent doing subtractive decode may claim and assert DEVSEL#. If the system does not have a subtractive decode agent, the master never sees DEVSEL# asserted and terminates the transaction per the Master-Abort mechanism (refer to Section 3.3.3.1).

A target must do a full decode before driving/asserting DEVSEL#, or any other target response signal. It is illegal to drive DEVSEL# prior to a complete decode and then let the decode combinationally resolve on the bus. (This could cause contention.) A target must qualify the AD lines with FRAME# before DEVSEL# can be asserted on commands other than configuration. A target must qualify IDSEL with FRAME# and AD[1::0] before DEVSEL# can be asserted on a configuration command.

It is expected that most (perhaps all) target devices will be able to complete a decode and assert DEVSEL# within one or two clocks of FRAME# being asserted (fast and medium in the figure).

Accordingly, the subtractive decode agent may provide an optional device dependent configuration register that can be programmed to pull in by one or two clocks the edge at which it asserts DEVSEL#, allowing faster access to the expansion bus. Use of such an option is limited by the slowest positive decode agent on the bus.

If the first byte addressed by the transaction maps into the target's address range, it asserts DEVSEL# to claim the access. But if the master attempts to continue the burst transaction across the resource boundary, the target is required to signal Disconnect.

When a target claims an I/O access and the byte enables indicate one or more bytes of the access are outside the target's address range, it must signal Target-Abort. (Refer to Section 3.3.3.2 for more information.) To deal with this type of I/O access problem, a subtractive decode device (expansion bus bridge) may do one of the following:

- ☐ Do positive decode (by including a byte map) on addresses for which different devices share common DWORDs, additionally using byte enables to detect this problem and signal Target-Abort.
- ☐ Pass the full access to the expansion bus, where the portion of the access that cannot be serviced will quietly drop on the floor. (This occurs only when the first addressed target resides on the expansion bus and the other is on PCI.)

3.6.2. Special Cycle

The Special Cycle command provides a simple message broadcast mechanism on PCI. In addition to communicating processor status, it may also be used for logical sideband signaling between PCI agents, when such signaling does not require the precise timing or synchronization of physical signals.

A good paradigm for the Special Cycle command is that of a “logical wire” which only signals single clock pulses; i.e., it can be used to set and reset flip flops in real time implying that delivery is guaranteed. This allows the designer to define necessary sideband communication without requiring additional pins. As with sideband signaling in general, implementation of Special Cycle command support is optional.

The Special Cycle command contains no explicit destination address, but is broadcast to all agents on the same bus segment. Each receiving agent must determine whether the message is applicable to it. PCI agents will never assert DEVSEL# in response to a Special Cycle command.

Note: Special Cycle commands do not cross PCI-to-PCI bridges. If a master desires to generate a Special Cycle command on a specific bus in the hierarchy, it must use a Type 1 configuration write command to do so. Type 1 configuration write commands can traverse PCI-to-PCI bridges in both directions for the purpose of generating Special Cycle commands on any bus in the hierarchy and are restricted to a single data phase in length. However, the master must know the specific bus on which it desires to generate the Special Cycle command and cannot simply do a broadcast to one bus and expect it to propagate to all buses. Refer to Section 3.2.2.3.1 for more information.

A Special Cycle command may contain optional, message dependent data, which is not interpreted by the PCI sequencer itself, but is passed, as necessary, to the hardware application connected to the PCI sequencer. In most cases, explicitly addressed messages should be handled in one of the three physical address spaces on PCI and not with the Special Cycle command.

Using a message dependent data field can break the logical wire paradigm mentioned above and create delivery guarantee problems. However, since targets only accept messages they recognize and understand, the burden is placed on them to fully process the message in the minimum delivery time (six bus clocks) or to provide any necessary buffering for messages they accept. Normally this buffering is limited to a single flip-flop. This allows delivery to be guaranteed. In some cases, it may not be possible to buffer or process all messages that could be received. In this case, there is no guarantee of delivery.

A Special Cycle command is like any other bus command where there is an address phase and a data phase. The address phase starts like all other commands with the assertion of FRAME# and completes like all other commands when FRAME# and IRDY# are deasserted. The uniqueness of this command compared to the others is that no agent responds with the assertion of DEVSEL# and the transaction concludes with a Master-Abort termination. Master-Abort is the normal termination for Special Cycle transactions and no errors are reported for this case of Master-Abort termination. This command is basically a broadcast to all agents, and interested agents accept the command and process the request.

The address phase contains no valid information other than the command field. There is no explicit address; however, AD[31::00] are driven to a stable level and parity is generated. During the data phase, AD[31::00] contain the message type and an optional data field. The message is encoded on the least significant 16 lines, namely AD[15::00]. The optional data field is encoded on the most significant 16 lines, namely AD[31::16], and is not required on all messages. The master of a Special Cycle command can insert wait states like any other command while the target cannot (since no target claimed the access by asserting DEVSEL#). The message and associated data are only valid on the first clock IRDY# is asserted. The information contained in, and the timing of, subsequent data phases are message dependent. When the master inserts a wait state or performs multiple data phases, it must extend the transaction to give potential targets sufficient time to process the message. This means the master must guarantee the access will not complete for at least four clocks (may be longer) after the last valid data completes. For example, a master keeps IRDY# deasserted for two clocks for a single data phase Special Cycle command. Because the master inserted wait states, the transaction cannot be terminated with Master-Abort on the fifth clock after FRAME# (the clock after subtractive decode time) like usual, but must be extended at least an additional two clocks. When the transaction has multiple data phases, the master cannot terminate the Special Cycle command until at least four clocks after the last valid data phase. Note: The message type or optional data field will indicate to potential targets the amount of data to be transferred. The target must latch data on the first clock IRDY# is asserted for each piece of data transferred.

During the address phase, C/BE[3::0]# = 0001 (Special Cycle command) and AD[31::00] are driven to random values and must be ignored. During the data phase, C/BE[3::0]# are asserted and AD[31::00] are as follows:

AD[15::00]	Encoded message
AD[31::16]	Message dependent (optional) data field

The PCI bus sequencer starts this command like all others and terminates it with a Master-Abort. The hardware application provides all the information like any other command and starts the bus sequencer. When the sequencer reports that the access terminated with a Master-Abort, the hardware application knows the access completed. In this case, the Received Master Abort bit in the configuration Status register (Section 6.2.3) must not be set. The quickest a Special Cycle command can complete is five clocks. One additional clock is required for the turnaround cycle before the next access. Therefore, a total of six clocks is required from the beginning of a Special Cycle command to the beginning of another access.

There are a total of 64 K messages. The message encodings are defined and described in Appendix A.

3.6.3. IDSEL Stepping

The ability of an agent to spread assertion of qualified signals over several clocks is referred to as *stepping*. All agents must be able to handle **IDSEL** stepping while generating it is optional. Refer to Section 4.2.4 for conditions associated with indeterminate signal levels on the rising edge of **CLK**.

Stepping is only permitted on **IDSEL** pins as a result of being driven by an **AD** signal through a series resistor. **IDSEL** is qualified by the combination of **FRAME#** and a decoded Type 0 configuration command. Output buffer technologies with a slow tri-state enable to **AD** and **C/BE#** active time are permitted to enable the **AD** and **C/BE#** buffers a clock cycle prior to the assertion of **FRAME#** for non-configuration transactions (these devices must still meet the T_{off} time of Table 4-6).

Figure 3-18 illustrates a master delaying the assertion of **FRAME#** until it has driven the **AD** lines and the associated **IDSEL**. The master is both permitted and required to drive **AD** and **C/BE#** once ownership has been granted and the bus is in the Idle state. However, by delaying assertion of **FRAME#**, the master runs the risk of losing its turn on the bus. As with any master, **GNT#** must be asserted on the rising clock edge before **FRAME#** is asserted. If **GNT#** were deasserted, on the clock edges marked "A", the master is required to immediately tri-state its signals because the arbiter has granted the bus to another agent. (The new master would be at a higher priority level.) If **GNT#** were deasserted on the clock edges marked "B" or "C", **FRAME#** will have already been asserted and the transaction continues.

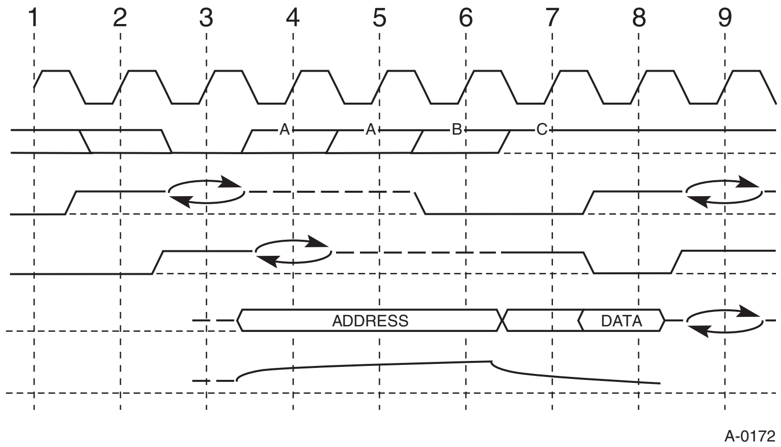


Figure 3-18: IDSEL Stepping

3.6.4. Interrupt Acknowledge

The PCI bus supports an Interrupt Acknowledge cycle as shown in Figure 3-19. This figure illustrates an x86 Interrupt Acknowledge cycle on PCI where a single byte enable is asserted and is presented only as an example. In general, the byte enables determine which bytes are involved in the transaction. During the address phase, AD[31::00] do not contain a valid address but must be driven with stable data, PAR is valid, and parity may be checked. An Interrupt Acknowledge transaction has no addressing mechanism and is implicitly targeted to the interrupt controller in the system. As defined in the *PCI-to-PCI Bridge Architecture Specification*, the Interrupt Acknowledge command is not forwarded to another PCI segment. The Interrupt Acknowledge cycle is like any other transaction in that DEVSEL# must be asserted one, two, or three clocks after the assertion of FRAME# for positive decode and may also be subtractively decoded by a standard expansion bus bridge. Wait states can be inserted and the request can be terminated, as discussed in Section 3.3.3.2. The vector must be returned when TRDY# is asserted.

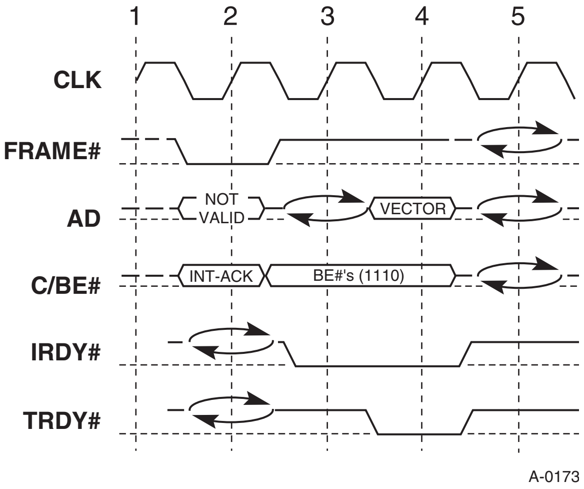


Figure 3-19: Interrupt Acknowledge Cycle

Unlike the traditional 8259 dual cycle acknowledge, PCI runs a single cycle acknowledge. Conversion from the processor's two cycle format to the PCI one cycle format is easily done in the bridge by discarding the first Interrupt Acknowledge request from the processor.

3.7. Error Functions

PCI provides for parity and other system errors to be detected and reported. A single system may include devices that have no interest in errors (particularly parity errors) and agents that detect, signal, and recover from errors. PCI error reporting allows agents that recover from parity errors to avoid affecting the operation of agents that do not. To allow this range of flexibility, the generation of parity is required on all transactions by all agents. The detection and reporting of errors is generally required, with limited exclusions for certain classes of PCI agents as listed in Section 3.7.2.

3.7.1. Parity Generation

Parity on PCI provides a mechanism to determine for each transaction if the master is successful in addressing the desired target and if data transfers correctly between them. To ensure that the correct bus operation is performed, the four command lines are included in the parity calculation. To ensure that correct data is transferred, the four byte enables are also included in the parity calculation. The agent that is responsible for driving AD[31::00] on any given bus phase is also responsible for driving even parity on PAR. The following requirements also apply when the 64-bit extensions are used (refer to Section 3.8 for more information).

During address and data phases, parity covers AD[31::00] and C/BE[3::0]# lines regardless of whether or not all lines carry meaningful information. Byte lanes not actually transferring data are still required to be driven with stable (albeit meaningless) data and are included in the parity calculation. During configuration, Special Cycle or Interrupt Acknowledge transactions some (or all) address lines are not defined but are required to be driven to stable values and are included in the parity calculation.

Parity is generated according to the following rules:

☐ Parity is calculated the same on all PCI transactions regardless of the type or form.

☐ The number of "1"s on AD[31::00], C/BE[3::0]#, and PAR equals an even number.

☐ Parity generation is not optional; it must be done by all PCI-compliant devices.

On any given bus phase, PAR is driven by the agent that drives AD[31::00] and lags the corresponding address or data by one clock. Figure 3-20 illustrates both read and write transactions with parity. The master drives PAR for the address phases on clocks 3 and 7. The target drives PAR for the data phase on the read transaction (clock 5) and the master drives PAR for the data phase on the write transaction (clock 8). Note: Other than the one clock lag, PAR behaves exactly like AD[31::00] including wait states and turnaround cycles.

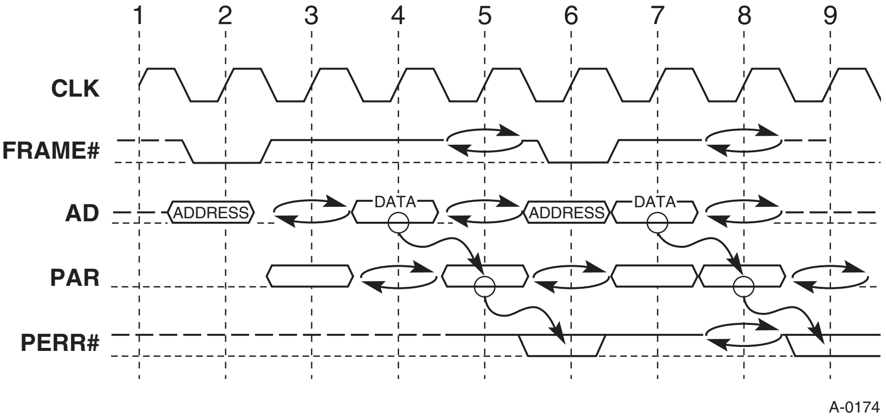


Figure 3-20: Parity Operation

3.7.2. Parity Checking

Parity must be checked to determine if the master successfully addressed the desired target and if data transferred correctly. All devices are required to check parity, except devices in the following two classes for which parity checking is optional:

- ☐ Devices that are designed exclusively for use on the system board; e.g., chip sets. System vendors have control over the use of these devices since they will never appear on add-in cards.
- ☐ Devices that never deal with or contain or access any data that represents permanent or residual system or application state; e.g., human interface and video/audio devices. These devices only touch data that is a temporary representation (e.g., pixels) of permanent or residual system or application state. Therefore, they are not prone to create system integrity problems in the event of undetected failure.

3.7.3. Address Parity Errors

A device is said to have detected an address parity error if the device's parity checking logic detects an error in a single address cycle or either address phase of a dual address cycle.

If a device detects an address parity error, in some cases, it will assert SERR# (refer to Section 3.7.4.2), and, in all cases, it will set the Detected Parity Error bit (Status register, bit 15) (refer to Section 3.7.4.4).

If a device detects an address parity error, and the device's Parity Error Response bit (Command register, bit 6) is set, and the device's address decoder indicates that the device is selected, the device must do one of the following:

- ☐ claim the transaction and terminate it as if there was no address/command error
- ☐ claim the transaction and terminate with Target-Abort
- ☐ not claim the transaction and let it terminate with Master-Abort

An error in the address phase of a transaction may affect any or all of the address bits, the command bits, and the parity bit. Since devices monitoring the transaction cannot determine which bits are actually in error, use of a transaction that contained an address parity error may cause unpredictable results.

The target is not allowed to claim a transaction and terminate it with Retry solely because of an address parity error or a write[21] data parity error. However, the occurrence of a parity error does not prevent the target from terminating the transaction with Retry for other reasons.

3.7.4. Error Reporting

PCI provides for the detection and signaling of two kinds of errors: data parity errors and other system errors. It is intended that data parity errors be reported up through the access and device driver chain whenever possible. This error reporting chain from target to bus master to device driver to device manager to operating system is intended to allow error recovery options to be implemented at any level. Since it is generally not possible to associate system errors with a specific access chain, they are reported via a separate system error signal (refer to Section 3.7.4.2).

PCI devices are enabled to report data parity errors by the Parity Error Response bit (bit 6 of the Command register). This bit is required in all devices except those not required to check parity (refer to Section 3.7.2). If the Parity Error Response bit is set, devices must respond to and report data parity errors for all bus operations (except those that occur during a Special Cycle transaction). If the Parity Error Response bit is cleared, an agent that detects a data parity error must ignore the error and complete the transaction as though parity was correct. In this case, no special handling of the data parity error can occur.

Two signals (pins) and two status bits are used in the PCI error reporting scheme. Each will be discussed separately.

3.7.4.1. Data Parity Error Signaling on PERR#

PERR# is used for signaling data parity errors on all transactions except Special Cycle transactions. Data parity errors that occur during a Special Cycle transaction are reported on SERR# as described in Section 3.7.4.2. PERR# is required for all devices except those not required to check parity (refer to Section 3.7.2).

If parity error response is enabled (bit 6 of the Command register is set) and a data parity error is detected by a master during a read transaction, the master must assert PERR#. If parity error response is enabled (bit 6 of the Command register is set) and a data parity error is detected by a target during a write transaction, the target must assert PERR#. Masters use this information to record the occurrence of the error for the device driver. PERR# is both an input and output signal for a master and only an output signal for a target.

A device asserting PERR# must do so two clocks after the completion of a data phase in which an error occurs, as shown in Figure 3-20. If the receiving agent inserts wait states,

---

[21] Targets check data parity only on write transactions.

that agent is permitted to assert **PERR#** as soon as a data parity error is detected. In other words, if the target is inserting wait states during a write transaction, the target is permitted to assert **PERR#** two clocks after data is valid (**IRDY#** asserted) but before the data transfers (**TRDY#** is also asserted). If the master is inserting wait states during a read transaction, the master is permitted to assert **PERR#** two clocks after data is valid (**TRDY#** is asserted) but before the data transfers (**IRDY#** is also asserted). Once **PERR#** is asserted, it must remain asserted until two clocks following the completion of the data phase (**IRDY#** and **TRDY#** both asserted). Note that the master is required to provide valid byte enables during every clock cycle of every data phase for both read and write transactions independent of **IRDY#**.

If a master asserts **PERR#** prior to completion of a read data phase, it must eventually assert **IRDY#** to complete the data phase. If a target asserts **PERR#** prior to completion of a write data phase, it must eventually assert **TRDY#** to complete the data phase. The target cannot terminate the data phase by signaling Retry, Disconnect without data, or Target-Abort after signaling **PERR#**. A master knows a data parity error occurred on a write data phase anytime **PERR#** is asserted, which may be prior to the completion of the data phase. But the master only knows the data phase was error free two clocks following the completion of the data phase.

Both masters and targets are permitted either to continue a burst transaction or stop it after detecting a data parity error. During a burst transaction in which multiple data phases are completed without intervening wait states, **PERR#** will be qualified on multiple consecutive clocks accordingly and may be asserted in any or all of them.

PERR# is a sustained tri-state signal that is bused to all PCI agents. It must be actively driven to the correct value on each qualified clock edge by the agent receiving the data. At the end of each bus operation, **PERR#** must actively be driven high for one clock period by the agent receiving data, starting two clocks after the **AD** bus turnaround cycle (e.g., clock 7 in Figure 3-20). The **PERR#** turnaround cycle occurs one clock later (clock 8 in Figure 3-20). **PERR#** cannot be driven (enabled) for the current transaction until at least three clocks after the address phase (which is one clock long for single address cycles and two clocks long for dual address cycles). Note that the target of a write transaction must not drive any signal until after asserting **DEVSEL#**; for example, for decode speed "slow" the target must not drive **PERR#** until four clocks after the address phase.

3.7.4.2. Other Error Signaling on SERR#

If a device is enabled to assert **SERR#** (i.e., SERR# Enable, bit 8 of the Command register, is set), and the device's Parity Error Response bit (Command register, bit 6) is set, the device must assert **SERR#** if any of the following conditions occurs:

- ☐ The device's parity checking logic detects an error in a single address cycle or either address phase of a dual address cycle (regardless of the intended target).
- ☐ The device monitors Special Cycle transactions, and the Special Cycles bit (Command register, bit 3) is set, and the device's parity checking logic detects a data parity error.

Refer to Section 6.8.2.1 for the cases in which a device must assert **SERR#** if that device is the master of a Message Signaled Interrupt transaction resulting in **PERR#** assertion, a Master-Abort, or a Target-Abort.

SERR# may optionally be used to report other internal errors that might jeopardize system or data integrity. It must be assumed, however, that signaling on SERR# will generate a critical system interrupt (e.g., NMI or Machine Check) and is, therefore, fatal. Consequently, care should be taken in using SERR# to report non-parity or system errors.

SERR# is required for all devices except those not required to check parity (refer to Section 3.7.2). SERR# is an open drain signal that is wire-ORed with all other PCI agents and, therefore, may be simultaneously driven by multiple agents. An agent reporting an error on SERR# drives it active for a single clock and then tri-states it. (Refer to Section 2.2.5 for more details.) Since open drain signaling cannot guarantee stable signals on every rising clock edge, once SERR# is asserted, its logical value must be assumed to be indeterminate until the signal is sampled in the deasserted state on at least two successive rising clock edges.

3.7.4.3. Master Data Parity Error Status Bit

The Master Data Parity Error bit (Status register, bit 8) must be set by the master if its Parity Error Response bit (Command register, bit 6) is set and either of the following two conditions occurs:

- ☐ The master detects a data parity error on a read transaction.
- ☐ The master samples PERR# asserted on a write transaction.

If the Parity Error Response bit is cleared, the master must not set the Master Data Parity Error bit, even if the master detects a parity error or the target asserts PERR#.

Targets never set the Master Data Parity Error bit.

3.7.4.4. Detected Parity Error Status Bit

The Detected Parity Error bit (Status register, bit 15) must be set by a device whenever its parity checking logic detects a parity error, regardless of the state the Parity Error Response bit (bit 6 of the command register). The Detected Parity Error bit is required to be set by the device when any of the following conditions occurs:

- ☐ The device's parity checking logic detects an error in a single address cycle or either address phase of a dual address cycle.
- ☐ The device's parity checking logic detects a data parity error and the device is the target of a write transaction.
- ☐ The device's parity checking logic detects a data parity error and the device is the master of a read transaction.

3.7.5. Delayed Transactions and Data Parity Errors

This section presents additional requirements for error handling that are unique to a target completing a transaction as a Delayed Transaction. Data parity error requirements presented in previous sections apply to Delayed Transactions as well.

A data parity error can occur during any of the three steps of a Delayed Transaction, the master request step, the target completion step, or the master completion step (refer to Section 3.3.3.3.1). The requirements for handling the error vary depending upon the step in which the error occurred. Errors that occur during the target completion phase are specific to the target device and are handled in a device-specific manner (not specified here).[22] Device behavior for errors that occur during the master request step or master completion step depend upon whether the Delayed Transaction is a read[23] or a write[24] transaction.

During a read transaction, the target device sources the data, and parity is not valid until TRDY# is asserted. Therefore, a data parity error cannot occur during the master request phase or any subsequent reattempt by the master that is terminated with Retry. During the master completion step of read transaction, the target sources data and data parity and the master checks parity and conditionally asserts PERR# as for any other (not delayed) transaction (refer to Section 3.7.4).

During a write transaction, the master sources the write data and must assert IRDY# when the data is valid independent of the response by the target device (refer to Section 3.2.1). Therefore, a data parity error may occur both in the master request and the master completion steps. In addition, it is possible for a data parity error to be either constant (i.e., the same error occurs each time the master repeats the transaction) or transient (i.e., the error occurs on some but not other repetitions of the transaction by the master). The data parity error reporting methods for write Delayed Transactions described in the following sections are designed to detect and report both constant and transient data parity errors, and to prevent transient data parity errors from causing a deadlock condition.

If a target detects a data parity error on a write transaction that would otherwise have been handled as a Delayed Transaction, the target is required to do the following:

1. Complete the data phase in which the error occurred by asserting TRDY#. If the master is attempting a burst, the target must also assert STOP#.
2. Report the error as described in Section 3.7.4.1.
3. Discard the transaction. No Delayed Write Request is enqueued, and no Delayed Write Completion is retired.

---

[22] If the actual target resides on a PCI bus segment generated by a PCI-to-PCI bridge, the target completion phase occurs across a PCI bus segment. In this case, the *PCI-to-PCI Bridge Architecture Specification* details additional requirements for error handling during the target completion phase of a read Delayed Transaction.

[23] Memory Read, Memory Read Line, Memory Read Multiple, Configuration Read, I/O Read, or Interrupt Acknowledge.

[24] Configuration Write or I/O Write, but never Memory Write and Invalidate or Memory Write.

If the target detects a data parity error during the initial request phase of a Delayed Write Transaction, no Delayed Request is ever enqueued.

If the target enqueues a good Delayed Write Request and later detects a data parity error during a subsequent repetition of the transaction, the target does not retire any Delayed Write Completions, even if the transaction appears to match one previously enqueued. (It is impossible to determine whether the transaction really matches a previously enqueued one, since an error is present.) This causes the target to have an orphan Delayed Write Completion, because the master believes the transaction has completed, but the target is waiting for the original (error free) request to be repeated. The orphan completion is discarded when the target's Discard Timer expires (refer to Section 3.3.3.3.3). While waiting for the discard timer to expire, some target implementations will not be able to accept a new Delayed Transaction, since the target is not required to handle multiple Delayed Transactions at the same time. However, since this condition is temporary, a deadlock cannot occur. While in this condition, the device is required to complete transactions that use memory write[25] commands (refer to Section 3.3.3.3.4).

3.7.6. Error Recovery

The action that a system takes as a result of the assertion of SERR# is not controlled by this specification. The assertion of SERR# by a device indicates that the device has encountered an error from which it cannot recover. The system may optionally stop execution at that point, if it does not have enough information to contain and recover from the error condition.

The PCI parity error signals and status bits are designed to provide a method for data parity errors to be detected and reported (if enabled). On a write transaction, the target always signals data parity errors back to the master on PERR#. On a read transaction, the master asserts PERR# to indicate to the system that an error was detected. In both cases, the master has the ability to promote the error to its device driver or the operating system or to attempt recovery using hardware and/or software methods.

The system designer may elect to report all data parity errors to the operating system by asserting SERR# when the central resource samples PERR# asserted. Note that when this option is used, recovery is not possible.

---

[25] This includes two commands: Memory Write and Invalidate and Memory Write.

IMPLEMENTATION NOTE

Recovery from Data Parity Errors

It is optional for PCI masters and systems to attempt recovery from data parity errors. The following are examples of how data parity error recovery may be attempted:

- ☐ **Recovery by the master.** If the master of the transaction in which the parity error was detected has sufficient knowledge that the transaction can be repeated without side effects, then the master may simply repeat the transaction. If no error occurs on the repeated transaction, reporting of the parity error (to the operating system or device driver) is unnecessary. If the error persists, or if the master is not capable of recovering from the data parity error, the master must inform its device driver. This can be accomplished by generating an interrupt, modifying a status register, setting a flag, or other suitable means. When the master does not have a device driver, it may report the error by asserting SERR#.

  Note: Most devices have side effects when accessed, and, therefore, it is unlikely that recovery is possible by simply repeating a transaction. However, in applications where the master understands the behavior of the target, it may be possible to recover from the error by repetition of the transaction.

- ☐ **Recovery by the device driver.** The device driver may support an error recovery mechanism such that the data parity error can be corrected. In this case, the reporting of the error to the operating system is not required. For example, the driver may be able to repeat an entire block transfer by reloading the master with the transfer size, source, and destination addresses of the data. If no error occurs on the repeated block transfer, then the error is not reported. When the device driver does not have sufficient knowledge that the access can be repeated without side effects, it must report the error to the operating system.

- ☐ **Recovery (or error handling) by the operating system.** Once the data parity error has been reported to the operating system, no other agent or mechanism can recover from the error. How the operating system handles the data parity error is operating system dependent.

3.8. 64-Bit Bus Extension

PCI supports a high 32-bit bus, referred to as the 64-bit extension to the standard low 32-bit bus. The 64-bit bus provides additional data bandwidth for agents that require it. The high 32-bit extension for 64-bit devices needs an additional 39 signal pins: REQ64#, ACK64#, AD[63::32], C/BE[7::4]#, and PAR64. These signals are defined in Section 2.2.8. 32-bit agents work unmodified with 64-bit agents. 64-bit agents must default to 32-bit mode unless a 64-bit transaction is negotiated. Hence, 64-bit transactions are totally transparent to 32-bit devices. Note: 64-bit addressing does not require a 64-bit data path (refer to Section 3.9).

64-bit transactions on PCI are dynamically negotiated (once per transaction) between the master and target. This is accomplished by the master asserting REQ64# and the target responding to the asserted REQ64# by asserting ACK64#. Once a 64-bit transaction is negotiated, it holds until the end of the transaction. ACK64# must not be asserted unless REQ64# was sampled asserted during the same transaction. REQ64# and ACK64# are externally pulled up to ensure proper behavior when mixing 32- and 64-bit agents. Refer to Section 3.8.1 for the operation of 64-bit devices in a 32-bit system.

During a 64-bit transaction, all PCI protocol and timing remain intact. Only memory transactions make sense when doing 64-bit data transfers. Interrupt Acknowledge and Special Cycle[26] commands are basically 32-bit transactions and must not be used with a REQ64#. The bandwidth requirements for I/O and configuration transactions cannot justify the added complexity, and, therefore, only memory transactions support 64-bit data transfers.

All memory transactions and other bus transfers operate the same whether data is transferred 32 or 64 bits at a time. 64-bit agents can transfer from one to eight bytes per data phase, and all combinations of byte enables are legal. As in 32-bit mode, byte enables may change on every data phase. The master initiating a 64-bit data transaction must use a double DWORD (Quadword or 8 byte) referenced address (AD[2] must be "0" during the address phase).

When a master requests a 64-bit data transfer (REQ64# asserted), the target has three basic responses and each is discussed in the following paragraphs.

1. Complete the transaction using the 64-bit data path (ACK64# asserted).
2. Complete the transaction using the 32-bit data path (ACK64# deasserted).
3. Complete a single 32-bit data transfer (ACK64# deasserted, STOP# asserted).

The first option is where the target responds to the master that it can complete the transaction using the 64-bit data path by asserting ACK64#. The transaction then transfers data using the entire data bus and up to 8 bytes can be transferred in each data phase. It behaves like a 32-bit bus except more data transfers each data phase.

The second option occurs when the target cannot perform a 64-bit data transfer to the addressed location (it may be capable in a different space). In this case, the master is required to complete the transaction acting as a 32-bit master and not as a 64-bit master.

---

[26] Since no agent claims the access by asserting DEVSEL# and, therefore, cannot respond with ACK64#.

The master has two options when the target does not respond by asserting ACK64# when the master asserts REQ64# to start a write transaction. The first option is that the master quits driving the upper AD lines and only provides data on the lower 32 AD lines. The second option is the master continues presenting the full 64 bits of data on each even DWORD address boundary. On the odd DWORD address boundary, the master drives the same data on both the upper and lower portions of the bus.

The third and last option is where the target is only 32 bits and cannot sustain a burst for this transaction. In this case, the target does not respond by asserting ACK64#, but terminates the transaction by asserting STOP#. If this is a Retry termination (STOP# asserted and TRDY# deasserted), the master repeats the same request (as a 64-bit request) at a later time. If this is a Disconnect termination (STOP# and TRDY# asserted), the master must repeat the request as a 32-bit master since the starting address is now on a odd DWORD boundary. If the target completed the data transfer such that the next starting address would be an even DWORD boundary, the master would be free to request a 64-bit data transfer. Caution should be used when a 64-bit request is presented and the target transfers a single DWORD as a 32-bit agent. If the master were to continue the burst with the same address, but with the lower byte enables deasserted, no forward progress would be made because the target would not transfer any new data, since the lower byte enables are deasserted. Therefore, the transaction would continue to be repeated forever without making progress.

64-bit parity (PAR64) works the same for the high 32-bits of the 64-bit bus as the 32-bit parity (PAR) works for the low 32-bit bus. PAR64 covers AD[63::32] and C/BE[7::4]# and has the same timing and function as PAR. (The number of "1"s on AD[63::32], C/BE[7::4]#, and PAR64 equal an even number.) PAR64 must be valid one clock after each address phase on any transaction in which REQ64# is asserted. (All 64-bit targets qualify address parity checking of PAR64 with REQ64#.) 32-bit devices are not aware of activity on 64-bit bus extension signals.

For 64-bit devices checking parity on data phases, PAR64 must be additionally qualified with the successful negotiation of a 64-bit transaction. PAR64 is required for 64-bit data phases; it is not optional for a 64-bit agent.

In the following two figures, a 64-bit master requests a 64-bit transaction utilizing a single address phase. This is the same type of addressing performed by a 32-bit master (in the low 4 GB address space). The first, Figure 3-21, is a read where the target responds with ACK64# asserted and the data is transferred in 64-bit data phases. The second, Figure 3-22, is a write where the target does not respond with ACK64# asserted and the data is transferred in 32-bit data phases (the transaction defaulted to 32-bit mode). These two figures are identical to Figure 3-5 and Figure 3-6 except that 64-bit signals have been added and in Figure 3-21 data is transferred 64-bits per data phase. The same transactions are used to illustrate that the same protocol works for both 32- and 64-bit transactions.

AD[63::32] and C/BE[7::4]# are reserved during the address phase of a single address phase transaction. AD[63::32] contain data and C/BE[7::4]# contain byte enables for the upper four bytes during 64-bit data phases of these transactions. AD[63::32] and C/BE[7::4]# are defined during the two address phases of a dual address cycle (DAC) and during the 64-bit data phases (refer to Section 3.9 for details).

Figure 3-21 illustrates a master requesting a 64-bit read transaction by asserting REQ64# (which exactly mirrors FRAME#). The target acknowledges the request by asserting ACK64# (which mirrors DEVSEL#). Data phases are stretched by both agents deasserting their ready lines. 64-bit signals require the same turnaround cycles as their 32-bit counterparts.

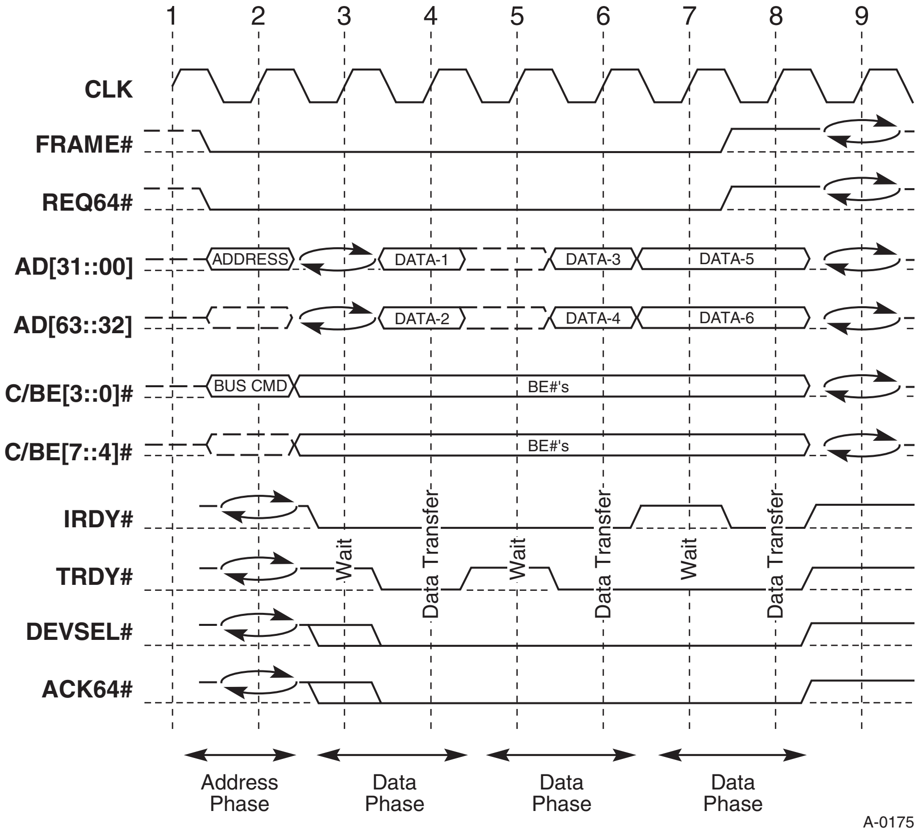


Figure 3-21: 64-bit Read Request With 64-bit Transfer

Figure 3-22 illustrates a master requesting a 64-bit transfer. The 32-bit target is not connected to REQ64# or ACK64#, and ACK64# is kept in the deasserted state with a pull-up. As far as the target is concerned, this is a 32-bit transfer. The master converts the transaction from 64- to 32-bits. Since the master is converting 64-bit data transfers into 32-bit data transfers, there may or may not be any byte enables asserted during any data phase of the transaction. Therefore, all 32-bit targets must be able to handle data phases with no byte enables asserted. The target should not use Disconnect or Retry because a data phase is encountered that has no asserted byte enables, but should assert TRDY# and complete the data phase. However, the target is allowed to use Retry or Disconnect because it is internally busy and unable to complete the data transfer independent of which byte enables are asserted. The master resends the data that originally appeared on AD[63::32] during the first data phase on AD[31::00] during the second data phase. The subsequent data phases appear exactly like the 32-bit transfer. (If the 64-bit signals are removed, Figure 3-22 and Figure 3-6 are identical.)

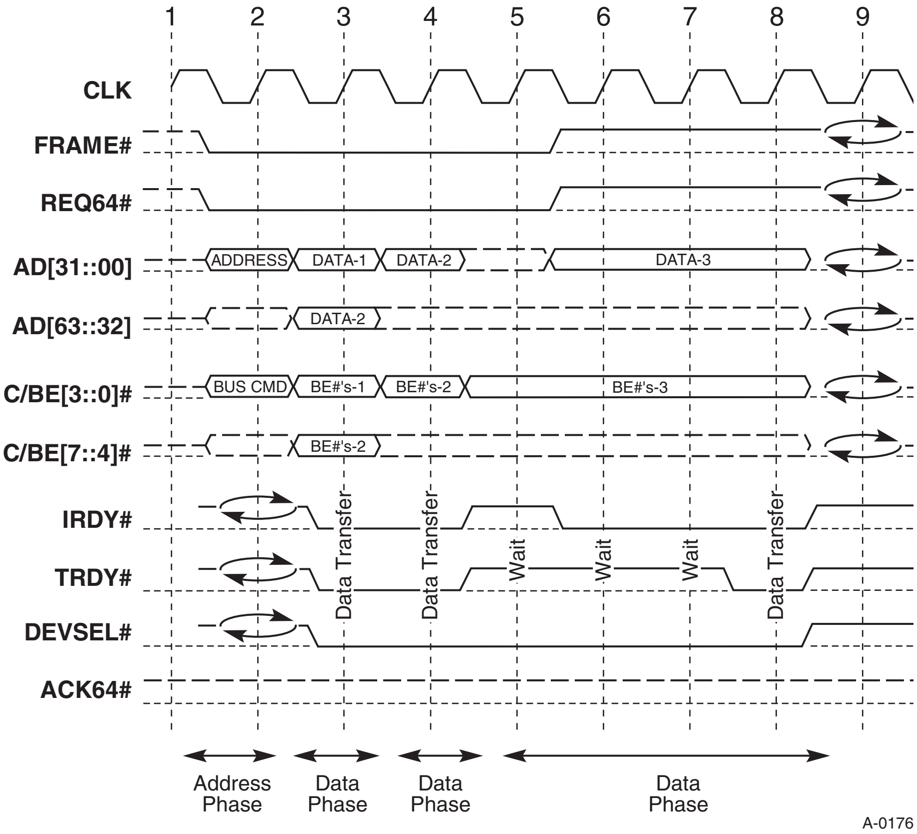


Figure 3-22: 64-bit Write Request With 32-bit Transfer

Using a single data phase with 64-bit transfers may not be very effective. Since the master does not know how the transaction will be resolved with ACK64# until DEVSEL# is returned, it does not know the clock on which to deassert FRAME# for a 64-bit single data phase transaction. IRDY# must remain deasserted until FRAME# signaling is resolved. The single 64-bit data phase may have to be split into two 32-bit data phases when the target is only 32-bits, which means a two phase 32-bit transfer is at least as fast as a one phase 64-bit transfer.

3.8.1. Determining Bus Width During System Initialization

REQ64# is used during reset to distinguish between parts that are connected to a 64-bit data path, and those that are not. PCI add-in card slots that support only a 32-bit data path must not connect REQ64# to any other slots or devices. (The REQ64# and ACK64# pins are located in the 32-bit portion of the connector.) Each 32-bit-only connector must have an individual pull-up resistor for REQ64# on the system board. ACK64# is bused to all 64-bit devices and slots on the system board and pulled up with a single resistor located on the system board. ACK64# for each 32-bit slots must be deasserted either by connecting it to the ACK64# signal connecting the 64-bit devices and slots or by individual pull-up resistors on the system board.

REQ64# is bused to all devices on the system board (including PCI connector slots) that support a 64-bit data path. This signal has a single pull-up resistor on the system board. The central resource must drive REQ64# low (asserted) during the time that RST# is asserted, according to the timing specification in Section 4.3.2. Devices that see REQ64# asserted on the rising edge of RST# are connected to the 64-bit data path, and those that do not see REQ64# asserted are not connected. This information may be used by the component to stabilize floating inputs during runtime, as described below.

REQ64# has setup and hold time requirements relative to the deasserting (high-going) edge of RST#. While RST# is asserted, REQ64# is asynchronous with respect to CLK.

When a 64-bit data path is provided, AD[63::32], C/BE[7::4]#, and PAR64 require either pull-up resistors or input "keepers," because they are not used in transactions with 32-bit devices and may, therefore, float to the threshold level causing oscillation or high power drain through the input buffer. This pull-up or keeper function must be part of the system board central resource, not the add-in card, (refer to Section 4.3.3) to ensure a consistent solution and avoid pull-up current overload.

When the 64-bit data path is present on a device but not connected (as in a 64-bit add-in card plugged into a 32-bit PCI slot), that PCI device must insure that its inputs do not oscillate, and that there is not a significant power drain through the input buffer both before and after the rising edge of RST#. This can be done in a variety of ways; e.g., biasing the input buffer or actively driving the outputs continuously (since they are not connected to anything). External resistors on an add-in card or any solution that violates the input leakage specification are prohibited.

While RST# is asserted, the PCI device floats its output buffers for the extended data path, AD[63::32], C/BE[7::4]#, and PAR64, unless the device input buffers cannot tolerate their inputs floating for an indefinitely long RST# period. If the device input buffers cannot tolerate this, the component must control its inputs while RST# is asserted. In this case, the device is permitted to enable its outputs continuously while RST# is asserted and REQ64# is deasserted (indicating a 32-bit bus), but must drive them to a logic low level (in case the bus connection is actually 64-bits wide and REQ64# has not yet settled to its final value). After the device detects that REQ64# is deasserted at the rising edge of RST#, the device must continue to control the extended bus to protect the device input buffers.

3.9. 64-bit Addressing

PCI supports memory addressing beyond the low 4 GB by defining a mechanism to transfer a 64-bit address from the master of the transaction to the target. No additional pins are required for a 32- or 64-bit device to support 64-bit addressing. Devices that support only 32-bit addresses are mapped into the low 4 GB of the address space and work transparently with devices that generate 64-bit addresses. Only memory transactions support 64-bit addressing.

The width of the address is independent of the width of the bus on either the master or the target. If both the master and target support a 64-bit bus, the entire 64-bit address could theoretically be provided in a single clock. However, the master is required in all cases to use two clocks to communicate a 64-bit address, since the width of the target's bus is not known during the address phase.

The standard PCI bus transaction supports a 32-bit address, Single Address Cycle (SAC), where the address is valid for a single clock when FRAME# is first sampled asserted. To support the transfer of a 64-bit address, a Dual Address Cycle (DAC) bus command is used, accompanied with one of the defined bus commands to indicate the desired data phase activity for the transaction. The DAC uses two clocks to transfer the entire 64-bit address on the AD[31::00] signals. When a 64-bit master uses DAC (64-bit addressing), it must provide the upper 32 bits of the address on AD[63::32] and the associated command for the transaction on C/BE[7::4]# during both address phases of the transaction to allow 64-bit targets additional time to decode the transaction.

Figure 3-23 illustrates a DAC for a read transaction. In a basic SAC read transaction, a turnaround cycle follows the address phase. In the DAC read transaction, an additional address phase is inserted between the standard address phase and the turnaround cycle. In the figure, the first and second address phases occur on clock 2 and 3 respectively. The turnaround cycle between the address and data phases is delayed until clock 4.
Note: FRAME# must be asserted during both address phases even for nonbursting single data phase transactions. To adhere to the FRAME# - IRDY# relationship, FRAME# cannot be deasserted until IRDY# is asserted. IRDY# cannot be asserted until the master provides data on a write transaction or is ready to accept data on a read transaction.

A DAC is decoded by a potential target when a "1101" is present on C/BE[3::0]# during the first address phase. If a 32-bit target supports 64-bit addressing, it stores the address that was transferred on AD[31::00] and prepares to latch the rest of the address on the next clock. The actual command used for the transaction is transferred during the second address phase on C/BE[3::0]#. A 64-bit target is permitted to latch the entire address on the first address phase. Once the entire address is transferred and the command is latched, the target determines if DEVSEL# is to be asserted. The target can do fast, medium, or slow decode one clock delayed from SAC decoding. A subtractive decode agent adjusts to the delayed device selection timing either by ignoring the entire transaction or by delaying its own assertion of DEVSEL#. If the bridge does support 64-bit addressing, it will delay asserting its DEVSEL# (if it does support 64-bit addressing). The master (of a DAC) will also delay terminating the transaction with Master-Abort for one additional clock.

The execution of an exclusive access is the same for either DAC or SAC. In either case, LOCK# is deasserted during the address phase (first clock) and asserted during the second clock (which is the first data phase for SAC and the second address phase for a DAC). Agents monitoring the transaction understand the lock resource is busy, and the target knows the master is requesting a locked operation. For a target that supports both SAC and DAC, the logic that handles LOCK# is the same.

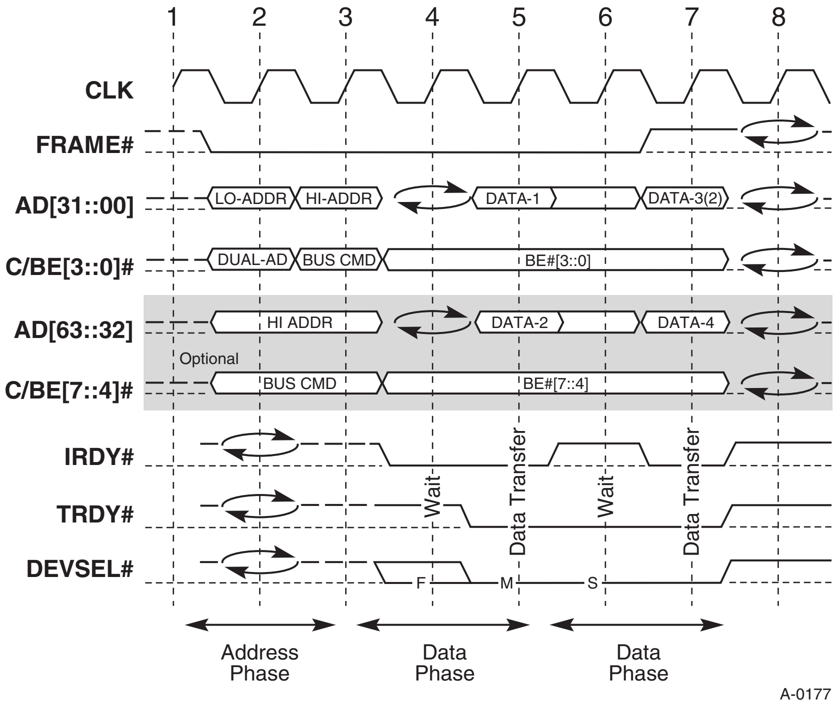


Figure 3-23: 64-Bit Dual Address Read Cycle

The master communicates a 64-bit address as shown in Figure 3-23, regardless of whether the target supports a 32-bit or 64-bit bus. The shaded area in Figure 3-23 is used only when the master of the access supports a 64-bit bus. The master drives the entire address (lower address on AD[31::00] and upper address on AD[63::32]) and both commands (DAC "1101" on C/BE[3::0]# and the actual bus command on C/BE[7::4]#), all during the initial address phase. On the second address phase, the master drives the upper address on AD[31::00] (and AD[63::32]) while the bus command is driven on C/BE[3::0]# (and C/BE[7::4]#). The master cannot determine if the target supports a 64-bit data path until the entire address has been transferred and, therefore, must assume a 32-bit target while providing the address.

If both the master and target support a 64-bit bus, then 64-bit addressing causes no additional latency when determining DEVSEL#, since all required information for command decoding is supplied in the first address phase. For example, a 64-bit target that normally performs a medium DEVSEL# decode for a SAC can decode the full 64-bit address from a 64-bit master during the first address phase of the DAC and perform a fast DEVSEL# decode. If either the master or the target does not support a 64-bit data path, one additional clock of delay will be encountered.

A master that supports 64-bit addressing must generate a SAC, instead of a DAC, when the upper 32 bits of the address are zero. This allows masters that generate 64-bit addresses to communicate with 32-bit addressable targets via SAC. The type of addressing (SAC or DAC) depends on whether the address is in the low 4-GB address range or not, and not by the target's bus width capabilities.

A 64-bit addressable target must act like a 32-bit addressable target (respond to SAC transactions) when mapped in the lower 4 GB address space. If a 32-bit master must access targets mapped above the lower 4 GB address space, that master must support 64-bit addressing using DAC.

3.10. Special Design Considerations

This section describes topics that merit additional comments or are related to PCI but are not part of the basic operation of the bus.

1. **Third party DMA**
   Third party DMA is not supported on PCI since sideband signals are not supported on the connector. The intent of PCI is to group together the DMA function in devices that need master capability and, therefore, third party DMA is not supported.
2. **Snooping PCI transactions**
   Any transaction generated by an agent on PCI may be snooped by any other agent on the same bus segment. Snooping does not work when the agents are on different PCI bus segments. In general, the snooping agent cannot drive any PCI signal, but must be able to operate independently of the behavior of the current master or target.
3. **Illegal protocol behavior**
   A device is not encouraged actively to check for protocol errors. However, if a device does detect illegal protocol events (as a consequence of the way it is designed), the design may return its state machines (target or master) to an Idle state as quickly as possible in accordance with the protocol rules for deassertion and tri-state of signals driven by the device.
4. **VGA palette snoop**
   The active VGA device always responds to a read of the color palette, while either the VGA or graphics agent will be programmed to respond to write transactions to the color palette and the other will snoop it. When a device (VGA or graphics) has been programmed to snoop a write to the VGA palette register, it must only latch the data when IRDY# and TRDY# are both asserted on the same rising clock edge or when a Master-Abort occurs. The first option is the normal case when a VGA and graphics device are present in the same system. The second option occurs when no device on the current bus has been programmed to positively respond to this range of addresses. This occurs when the PCI segment is given the first right of refusal and a subtractive decode device is not present. In some systems, this access is still forwarded to another bus which will complete the access. In this type of system, a device that has been programmed to snoop writes to the palette should latch the data when the transaction is terminated with Master-Abort.

The palette snoop bit will be set by the system firmware when it detects both a VGA device and a graphics accelerator device that are on separate add-in cards on the same bus or on the same path but on different buses.

- ☐ When both agents are PCI devices that reside on the same bus, either device can be set to snoop and the other will be set to positively respond.
- ☐ When both are PCI devices that reside on different buses but on the same path, the first device found in the path will be set to snoop and the other device may be set to positively respond or snoop the access. (Either option works in a PC-AT compatible system since a write transaction on a PCI segment, other than the primary PCI bus, that is terminated with Master-Abort is simply terminated and the data is dropped and Master-Aborts are not reported.)
- ☐ When one device is on PCI and the other is behind the subtractive decode device, the PCI device will be set to snoop and the subtractive decode device will automatically claim the access and forward it.

The only case where palette snooping would be turned off is when only a VGA device (no graphics device) is present in the system, or both the VGA and graphics devices are integrated together into single device or add-in card.

Note: Palette snooping does not work when the VGA and graphics devices reside on different buses that are not on the same path. This occurs because only a single agent per bus segment may claim the access. Therefore, one agent will never see the access because its bridge cannot forward the access. When a device has been programmed to snoop the access, it cannot insert wait states or delay the access in any way and, therefore, must be able to latch and process the data without delay.

For more information on PCI support of VGA devices, refer to Appendix A of the *PCI-to-PCI Bridge Architecture Specification*.

5. **Potential deadlock scenario when using PCI-to-PCI bridges**
Warning: A potential deadlock will occur when all the following conditions exist in a system:

1. When PCI-to-PCI bridges are supported in the system. (Note: If an add-in card connector is supported, PCI-to-PCI bridges may be present in the system.)
2. A read access originated by the host bridge targets a PCI device that requires more than a single data phase to complete. (Eight-byte transfer or an access that crosses a DWORD boundary when targeting an agent that responds to this request as 32-bit agent or resides on a 32-bit PCI segment.)

The deadlock occurs when the following steps are met:

1. A burst read is initiated on PCI by the host bridge and only the first data phase completes. (This occurs because either the target or the PCI-to-PCI bridge in the path terminates the request with Disconnect.)
2. The request passes through a PCI-to-PCI bridge and the PCI-to-PCI bridge allows posted write data (moving toward main memory) after the initial read completes.
3. The host bridge that originated the read request blocks the path to main memory.

The deadlock occurs because the PCI-to-PCI bridge cannot allow a read to transverse it while holding posted write data. The host bridge that initiated the PCI access cannot allow the PCI-to-PCI bridge to flush data until it completes the second read, because there is no way to "back-off" the originating agent without losing data. It must be assumed the read data was obtained from a device that has destructive read side effects. Therefore, discarding the data and repeating the access is not an option.

If all these conditions are met, the deadlock will occur. If the system allows all the conditions to exist, then the host bridge initiating the read request must use LOCK# to guarantee that the read access will complete without the deadlock conditions being met. The fact that LOCK# is active for the transaction causes the PCI-to-PCI bridge to turn-off posting until the lock operation completes. (A locked operation completes when LOCK# is deasserted while FRAME# is deasserted.)

Note: The use of LOCK# is only supported by PCI-to-PCI bridges moving downstream (away from the processor). Therefore, this solution is only applicable to host bus bridges.

Another deadlock that is similar to the above deadlock occurs doing an I/O Write access that straddles an odd DWORD boundary. The same condition occurs as the read deadlock when the host bridge cannot allow access to memory until the I/O write completes. However, LOCK# cannot be used to prevent this deadlock since locked accesses must be initiated with a read access.

6. **Potential data inconsistency when an agent uses delayed transaction termination**

Delayed Completion transactions on PCI are matched by the target with the requester by comparing addresses, bus commands, and byte enables, and if a write, write data. As a result, when two masters access the same address with the same bus command and byte enables, it is possible that one master will obtain the data assuming that it is a read which was actually requested by the other master. In a prefetchable region, this condition can occur even if the byte enables, and in some cases, the commands of the two transactions do not match. A prefetchable region can be defined by the target using range registers or by the master using the Memory Read Line or Memory Read Multiple commands. Targets completing read accesses in a prefetchable memory range ignore the byte enables and can also alias the memory read commands when completing the delayed read request.

If no intervening write occurs between the read issued by the two masters, there is no data consistency issue. However, if a master completes a memory write and then requests a read of the same location, there is a possibility that the read will return a snapshot of that location which actually occurred prior to the write (due to a Delayed Read Request by another master queued prior to the write).

This is only a problem when multiple masters on one side of a bridge are polling the same location on the other side of the bridge and one of the masters also writes the location. Although it is difficult to envision a real application with these characteristics, consider the sequence below:

1. Master A attempts a read to location X and a bridge responds to the request using Delayed Transaction semantics (queues a Delayed Read Request).
2. The bridge obtains the requested read data and the Delayed Request is now stored as a Delayed Completion in the bridge.
3. Before Master A is able to complete the read request (obtain the results stored in the Delayed Completion in the bridge), Master B does a memory write to Location X and the bridge posts the memory write transaction.
4. Master B then reads location X using the same address, byte enables, and bus command as Master A's original request. Note that if the transaction reads from a prefetchable location, the two commands can be confused by the bridge even if the byte enable patterns and read commands are different.
5. The bridge completes Master B's read access and delivers read data which is a snapshot of Location X prior to the memory write of Location X by Master B.

Since both transactions are identical, the bridge provides the data to the wrong master. If Master B takes action on the read data, then an error may occur, since Master B will see the value before the write. However, if the purpose of the read by Master B was to ensure that the write had completed at the destination, no error occurs and the system is coherent since the read data is not used (dummy read). If the purpose of the read is only to flush the write posted data, it is recommended that the read be to a different DWORD location of the same device. Then the reading of stale data does not exist. If the read is to be compared to decide what to do, it is recommended that the first read be discarded and the decision be based on the second read.

The above example applies equally to an I/O controller that uses Delayed Transaction termination. In the above example, replace the word "bridge" with "I/O controller" and the same potential problem exists.

A similar problem can occur if the two masters are not sharing the same location, but locations close to each other, and one master begins reading at a *smaller* address than the one actually needed. If the smaller address coincides exactly with the address of the other master's read from the near location, then the two masters' reads can be swapped by a device using Delayed Transaction termination. If there is an intervening write cycle, then the second master may receive stale data; i.e., the results from the read which occurred before the write cycle. The result of this example is the same as the first example since the start addresses are the same. To avoid this problem, the master must address the data actually required and not start at a smaller address.

In summary, this problem can only occur if two masters on one side of a bridge are sharing locations on the other side of the bridge. Although typical applications are not configured this way, the problem can be avoided if a master doing a read fetches only the actual data it needs and does *not* prefetch data *before* the desired data, or if the master does a dummy read after the write to guarantee that the write completes.

Another data inconsistency situation can occur when a single master changes its behavior based on a new transaction it receives after having a request terminated with Retry. The following sequence illustrates the data inconsistency:

1. A master is informed that pointer 1 at DWORD Location X is valid. (Pointer 2 at Location Y, the next sequential DWORD location, is not valid.)
2. The master initiates a memory read to Location X and is terminated with Retry. (The master intends to read only pointer 1, since pointer 2 is invalid.)
3. The host bridge begins to fetch the contents of Location X as a Delayed Transaction.
4. The host bridge completes the read request, prefetching beyond Location X to include Location Y and places the Delayed Read Completion in the outbound queue.
5. The CPU updates pointer 2 in Location Y in memory.
6. The CPU uses a memory write to inform the master that pointer 2 is valid. The host bridge posts the memory write. Ordering rule number 7 in Appendix E requires the host bridge to allow the posted memory write transaction to pass the Delayed Read Completion of Location X (including the stale value from Location Y).
7. The host bridge executes the posted memory write on the PCI bus informing the master that pointer 2 in now valid.
8. The master repeats the original memory read to Location X, but because pointer 2 in now valid, it extends the transaction and obtains two DWORDS including Location Y.

   The data the master received from Location Y is stale. To prevent this data inconsistency from occurring, the master is not allowed to extend a memory read transaction beyond its original intended limits after it has been terminated with Retry.

7. Peer-to-peer transactions crossing multiple host bridges

PCI host bridges may, but are not required to, support PCI peer-to-peer transactions that traverse multiple PCI host bridges.

8. The effect of PCI-to-PCI bridges on the PCI clock specification

The timing parameters for CLK for PCI add-in card connectors are specified at the input of the device in the slot. Refer to Section 4.2.3.1 and Section 7.6.4.1 for more information. Like all signals on the connector, only a single load is permitted on CLK in each slot. An add-in card that uses several devices behind a PCI-to-PCI bridge must accommodate the clock buffering requirements of that bridge. For example, if the bridge's clock buffer affects the duty cycle of CLK, the rest of the devices on the add-in card must accept the different duty cycle. It is the responsibility of the add-in card designer to choose components with compatible CLK specifications.

The system must always guarantee the timing parameters for CLK specified in Section 4.2.3.1 and Section 7.6.4.1 at the input of the device in a PCI add-in card slot, even if the system board places PCI slots on the secondary side of a PCI-to-PCI bridge.

It is the responsibility of the system board designer to choose clock sources and PCI-to-PCI bridges that will guarantee this specification for all slots.

9. **Devices cannot drive and receive signals at the same time**

   Bus timing requires that no device both drive and receive a signal on the bus at the same time. System timing analysis considers the worst signal propagation case to be when one device drives a signal and the signal settles at the input of all other devices on the bus. In most cases, the signal will not settle at the driving device until some time after it has settled at all other devices. Refer to Section 4.3.5 and Section 7.7.5 for a description of T_{prop}.

   Logic internal to a device must never use the signal received from the bus while that device is driving the bus. If internal logic requires the state of a bus signal while the device is driving the bus, that logic must use the internal signal (the one going to the output buffer of the device) rather than the signal received from the device input buffer. For example, if logic internal to a device continuously monitors the state of FRAME# on the bus, that logic must use the signal from the device input buffer when the device is not the current bus master, and it must use the internally generated FRAME# when the device is the current bus master.

4

4. Electrical Specification

4.1. Overview

This chapter defines all the electrical characteristics and constraints of PCI components, systems, and add-in cards, including pin assignment on the add-in card connector, when the operating frequency is at or below 33 MHz. PCI-X is the preferred replacement of PCI when higher operating frequencies are required. Refer to the PCI-X *Addendum to the PCI Local Bus Specification* for PCI-X requirements and to Chapter 7 for information on existing 66-MHz PCI requirements. PCI-X is capable of supporting four add-in card slots at 66 MHz making it an easier migration path than the two slot capability of 66 MHz PCI. This chapter is divided into major sections covering integrated circuit components (Section 4.2), systems or system boards (Section 4.3), and add-in cards (Section 4.4). Each section contains the requirements that must be met by the respective product, as well as the assumptions it may make about the environment provided. While every attempt was made to make these sections self-contained, there are invariably dependencies between sections so that it is necessary that all vendors be familiar with all three areas. The PCI electrical definition provides for both 3.3V and 5V signaling environments. These should not be confused with 3.3V and 5V component technologies. A "3.3V component" can be designed to work in a 5V signaling environment and vice versa; component technologies can be mixed in either signaling environment. The signaling environments cannot be mixed; all components on a given PCI bus must use the same signaling convention of 3.3V or 5V.

4.1.1. Transition Road Map

With this revision of the PCI specification the migration from the 5 volt to the 3.3 volt signaling environment is complete. The 33 MHz systems plus PCI-X, Low Profile, Mini PCI, and PCI 66 systems support only 3.3V signaling by requiring the system board add-in card connectors to be keyed for the 3.3V signaling environment. The 5V keyed system board connector and the 5V keyed add-in card are no longer supported.

The 3.3V keyed system board connector defines the 3.3V signaling environment for the system. The 3.3V add-in card is designed to work only in the 3.3V signaling environment. The Universal add-in card (retained for backward compatibility with 5V systems) is capable of detecting the signaling environment in use and adapting itself to that environment. It can, therefore, be plugged into either connector type (see Figure 4-1). Both add-in card types define connections to both 3.3V and 5V power supplies and may contain either 3.3V and/or 5V components. The distinction between add-in card types is the signaling protocol they use, not the power rails they connect to nor the component technology they contain.

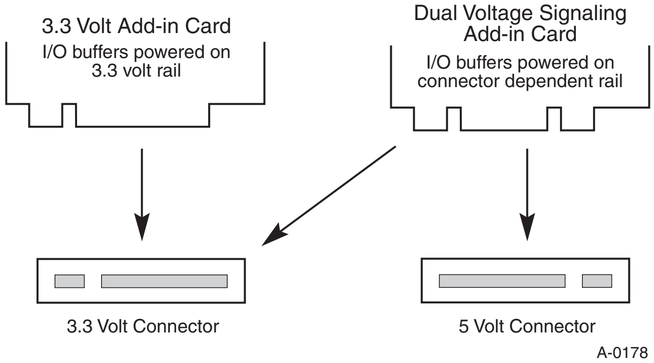

3.3 Volt Add-in Card
I/O buffers powered on 3.3 volt rail
Dual Voltage Signaling Add-in Card
I/O buffers powered on connector dependent rail
3.3 Volt Connector
5 Volt Connector
A-0178

Figure 4-1: Add-in Card Connectors

PCI components on the Universal add-in card must use I/O buffers that can be compliant with either the 3.3V or 5V signaling environment. While there are multiple buffer implementations that can achieve this dual environment compliance, dual voltage buffers capable of operating from either power rail can be used. They should be powered from "I/O" designated power pins[27] on PCI connectors that will always be connected to the power rail associated with the signaling environment in use. This means that in the 3.3V signaling environment (system board connectors keyed for 3.3V), these buffers are powered on the 3.3V rail. When the same add-in card is plugged into a 5V connector (used in systems compliant to previous revisions of this specification), these buffers are powered on the 5V rail. This enables the Universal add-in card to be compliant with either signaling environment.

4.1.2. Dynamic vs. Static Drive Specification

The PCI bus has two electrical characteristics that motivate a different approach to specifying I/O buffer characteristics. First, PCI is a CMOS bus, which means that steady state currents (after switching transients have died out) are minimal. In fact, the majority of the DC drive current is spent on pull-up resistors. Second, PCI is based on reflected wave rather than incident wave signaling. This means that bus drivers are sized to only switch the bus half way to the required high or low voltage. The electrical wave propagates down the bus, reflects off the unterminated end and back to the point of origin, thereby doubling the initial voltage excursion to achieve the required voltage level. The bus driver is actually in the middle of its switching range during this propagation time, which lasts up to 10 ns, one third of the bus cycle time at 33 MHz.

PCI bus drivers spend this relatively large proportion of time in transient switching, and the DC current is minimal, so the typical approach of specifying buffers based on their DC

---

[27] When “5V tolerant” 3.3V parts are used on the Universal add-in card, its I/O buffers may optionally be connected to the 3.3V rail rather than the "I/O" designated power pins; but high clamp diodes may still be connected to the "I/O" designated power pins. (Refer to the last paragraph of Section 4.2.1.2 - "Clamping directly to the 3.3V rail with a simple diode must never be used in the 5V signaling environment.") Since the effective operation of these high clamp diodes may be critical to both signal quality and device reliability, the designer must provide enough "I/O" designated power pins on a component to handle the current spikes associated with the 5V maximum AC waveforms (Section 4.2.1.3).

current sourcing capability is not useful. PCI bus drivers are specified in terms of their AC switching characteristics rather than DC drive. Specifically, the voltage to current relationship (V/I curve) of the driver through its active switching range is the primary means of specification. These V/I curves are targeted at achieving acceptable switching behavior in typical configurations of six loads on the system board and two add-in card connectors or two loads on the system board and four add-in card connectors. However, it is possible to achieve different or larger configurations depending on the actual equipment practice, layout arrangement, loaded impedance of the system board, etc.

4.2. Component Specification

This section specifies the electrical and timing parameters for PCI components; i.e., integrated circuit devices. Both 3.3V and 5V rail-to-rail signaling environments are defined. The 3.3V environment is based on V_{CC}-relative switching voltages and is an optimized CMOS approach. The 5V environment, on the other hand, is based on absolute switching voltages in order to be compatible with TTL switching levels. The intent of the electrical specification is that components connect directly together, whether on the system board or an add-in card, without any external buffers.

These specifications are intended to provide a design definition of PCI component electrical compliance and are not, in general, intended as actual test specifications. Some of the elements of this design definition cannot be tested in any practical way but must be guaranteed by design characterization. It is the responsibility of component designers and ASIC vendors to devise an appropriate combination of device characterization and production tests, correlated to the parameters herein, in order to guarantee the PCI component complies with this design definition. All component specifications have reference to a packaged component and, therefore, include package parasitics. Unless specifically stated otherwise, component parameters apply at the package pins, not at bare silicon pads[28] nor at add-in card edge connectors.

The intent of this specification is that components operate within the "commercial" range of environmental parameters. However, this does not preclude the option of other operating environments at the vendor's discretion.

---

[28] It may be desirable to perform some production tests at bare silicon pads. Such tests may have different parameters than those specified here and must be correlated back to this specification.

PCI output buffers are specified in terms of their V/I curves. Limits on acceptable V/I curves provide for a maximum output impedance that can achieve an acceptable first step voltage in typical configurations and for a minimum output impedance that keeps the reflected wave within reasonable bounds. Pull-up and pull-down sides of the buffer have separate V/I curves, which are provided with the parametric specification. The effective buffer strength is primarily specified by an AC drive point, which defines an acceptable first step voltage, both high going and low going, together with required currents to achieve that voltage in typical configurations. The DC drive point specifies steady state conditions that must be maintained, but in a CMOS environment these are minimal and do not indicate real output drive strength. The shaded areas on the V/I curves shown in Figure 4-2 and Figure 4-4 define the allowable range for output characteristics.

DC parameters must be sustainable under steady state (DC) conditions. AC parameters must be guaranteed under transient switching (AC) conditions, which may represent up to 33% of the clock cycle. The sign on all current parameters (direction of current flow) is referenced to a ground *inside* the component; that is, positive currents flow into the component while negative currents flow out of the component. The behavior of reset (RST#) is described in Section 4.3.2 (system specification) rather than in this (component) section.

The optional PME# signal is unique because of its use when the system is in a low-power state in which the PCI bus and all the peripherals attached to it will sometimes have power removed. This requires careful design to ensure that a voltage applied to the PME# pin will never cause damage to the part, even if the component's V_{cc} pins are not powered. Additionally, the device must ensure that it does not pull the signal to ground unless the PME# signal is being intentionally asserted. See the *PCI Power Management Interface Specification* for requirements for PME# in systems that support it.

4.2.1. 5V Signaling Environment

4.2.1.1. DC Specifications

Table 4-1 summarizes the DC specifications for 5V signaling.

Table 4-1: DC Specifications for 5V Signaling

Symbol	Parameter	Condition	Min	Max	Units	Notes
V_{cc}	Supply Voltage		4.75	5.25	V	
V_{ih}	Input High Voltage		2.0	V_{cc}+0.5	V	
V_{il}	Input Low Voltage		-0.5	0.8	V	
I_{ih}	Input High Leakage Current	V_{in} = 2.7		70	μA	1
I_{il}	Input Low Leakage Current	V_{in} = 0.5		-70	μA	1
V_{oh}	Output High Voltage	I_{out} = -2 mA	2.4		V	

Symbol	Parameter	Condition	Min	Max	Units	Notes
V_{ol}	Output Low Voltage	I_{out} = 3 mA, 6 mA		0.55	V	2
C_{in}	Input Pin Capacitance			10	pF	3
C_{clk}	CLK Pin Capacitance		5	12	pF	
C_{IDSEL}	IDSEL Pin Capacitance			8	pF	4
L_{pin}	Pin Inductance			20	nH	5
I_{Off}	PME# input leakage	$V_o \leq 5.25$ V V_{cc} off or floating	–	1	µA	6

Notes:

1. Input leakage currents include hi-Z output leakage for all bi-directional buffers with tri-state outputs.
2. Signals without pull-up resistors must have 3 mA low output current. Signals requiring pull up must have 6 mA; the latter include, FRAME#, TRDY#, IRDY#, DEVSEL#, STOP#, SERR#, PERR#, LOCK#, INTA#, INTB#, INTC#, INTD#, and, when used, AD[63::32], C/BE[7::4]#, PAR64, REQ64#, and ACK64#.
3. Absolute maximum pin capacitance for a PCI input is 10 pF (except for CLK, SMBDAT, and SMBCLK) with an exception granted to system board-only devices up to 16 pF, in order to accommodate PGA packaging. This means, in general, that components for add-in cards need to use alternatives to ceramic PGA packaging (i.e., PQFP, SGA, etc.). Pin capacitance for SMBCLK and SMBDAT is not specified; however, the maximum capacitive load is specified for the add-in card in Section 8.2.5.
4. Lower capacitance on this input-only pin allows for non-resistive coupling to AD[xx].
5. This is a recommendation, not an absolute requirement. The actual value should be provided with the component data sheet.
6. This input leakage is the maximum allowable leakage into the PME# open drain driver when power is removed from V_{cc} of the component. This assumes that no event has occurred to cause the device to attempt to assert PME#.

Refer to Section 3.8.1 for special requirements for AD[63::32], C/BE[7::4]#, and PAR64 when they are not connected (as in a 64-bit add-in card installed in a 32-bit connector).

4.2.1.2. AC Specifications

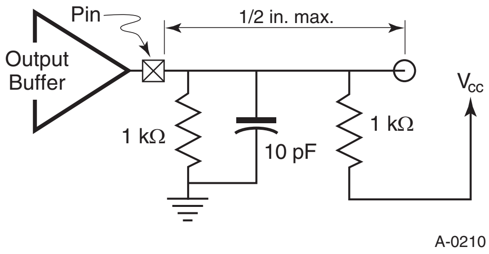
Table 4-2 summarizes the AC specifications for 5V signaling.

Table 4-2: AC Specifications for 5V Signaling

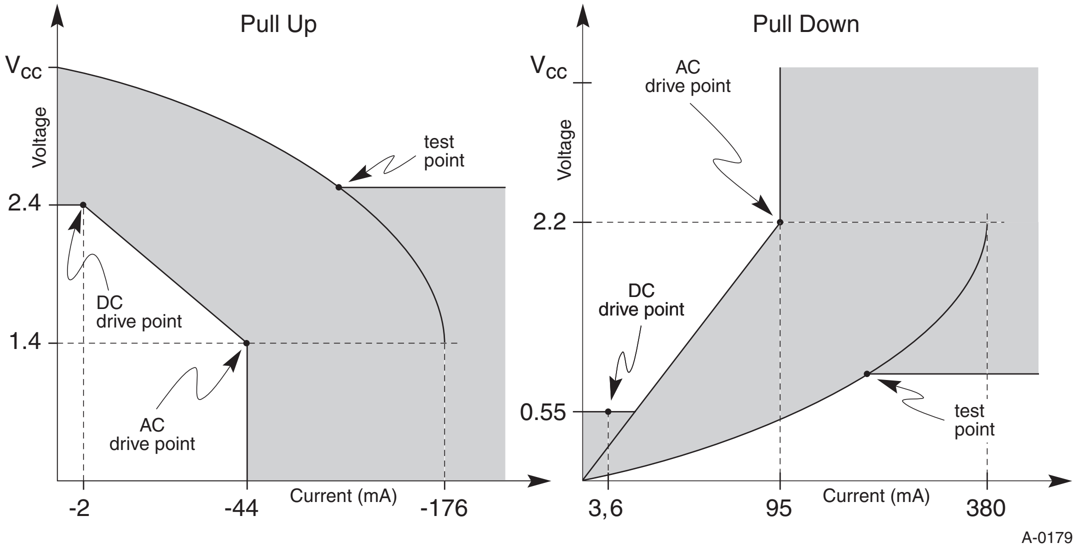
Symbol	Parameter	Condition	Min	Max	Units	Notes
$I_{oh}(AC)$	Switching Current High	$0 < V_{out} \leq 1.4$	-44		mA	1
		$1.4 < V_{out} < 2.4$	$-44+(V_{out}-1.4)/0.024$		mA	1, 2
		$3.1 < V_{out} < V_{cc}$		Eqt'n A		1, 3
	(Test Point)	$V_{out} = 3.1$		-142	mA	3
$I_{ol}(AC)$	Switching Current Low	$V_{out} \geq 2.2$	95		mA	1
		$2.2 > V_{out} > 0.55$	$V_{out}/0.023$		mA	1
		$0.71 > V_{out} > 0$		Eqt'n B		1, 3
	(Test Point)	$V_{out} = 0.71$		206	mA	3
I_{cl}	Low Clamp Current	$-5 < V_{in} \leq -1$	$-25+(V_{in}+1)/0.015$		mA	
$slew_r$	Output Rise Slew Rate	0.4 V to 2.4 V load	1	5	V/ns	4
$slew_f$	Output Fall Slew Rate	2.4 V to 0.4 V load	1	5	V/ns	4

Notes:

1. Refer to the V/I curves in Figure 4-2. Switching current characteristics for REQ# and GNT# are permitted to be one half of that specified here; i.e., half size output drivers may be used on these signals. This specification does not apply to CLK and RST# which are system outputs. "Switching Current High" specifications are not relevant to SERR#, PME#, INTA#, INTB#, INTC#, and INTD# which are open drain outputs.
2. Note that this segment of the minimum current curve is drawn from the AC drive point directly to the DC drive point rather than toward the voltage rail (as is done in the pull-down curve). This difference is intended to allow for an optional N-channel pull-up.
3. Maximum current requirements must be met as drivers pull beyond the first step voltage. Equations defining these maximums (A and B) are provided with the respective diagrams in Figure 4-2. The equation-defined maximums should be met by design. In order to facilitate component testing, a maximum current test point is defined for each side of the output driver.
4. This parameter is to be interpreted as the cumulative edge rate across the specified range, rather than the instantaneous rate at any point within the transition range. The specified load (diagram below) is optional; i.e., the designer may elect to meet this parameter with an unloaded output. However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline).



The minimum and maximum drive characteristics of PCI output buffers are defined by V/I curves. These curves should be interpreted as traditional "DC" transistor curves with the following exceptions: the "DC Drive Point" is the only position on the curves at which steady state operation is intended, while the higher current parts of the curves are only reached momentarily during bus switching transients. The "AC Drive Point" (the real definition of buffer strength) defines the minimum instantaneous current curve required to switch the bus with a single reflection. From a quiescent or steady state, the current associated with the AC drive point must be reached within the output delay time, T_{val}. Note, however, that this delay time also includes necessary logic time. The partitioning of T_{val} between clock distribution, logic, and output buffer is not specified, but the faster the buffer (as long as it does not exceed the maximum rise/fall slew rate specification), the more time is allowed for logic delay inside the part. The "Test Point" defines the maximum allowable instantaneous current curve in order to limit switching noise and is selected roughly on a 22 Ω load line.



Equation A:

$I_{oh} = 11.9*(V_{out}-5.25)*(V_{out}+2.45)$

for $V_{cc} > V_{out} > 3.1$ V

Equation B:

$I_{ol} = 78.5*V_{out}*(4.4-V_{out})$

for $0 \text{ V} < V_{out} < 0.71$ V

Figure 4-2: V/I Curves for 5V Signaling

Adherence to these curves is evaluated at worst case conditions. The minimum pull up curve is evaluated at minimum V_{cc} and high temperature. The minimum pull down curve is evaluated at maximum V_{CC} and high temperature. The maximum curve test points are evaluated at maximum V_{CC} and low temperature.

Inputs are required to be clamped to ground. Clamps to the 5V rail are optional, but may be needed to protect 3.3V input devices (refer to Section 4.2.1.3). Clamping directly to the 3.3V rail with a simple diode must never be used in the 5V signaling environment. When dual power rails are used, parasitic diode paths can exist from one supply to another. These diode paths can become significantly forward biased (conducting) if one of the power rails goes out of specification momentarily. Diode clamps to a power rail, as well as to output pull-up devices, must be able to withstand short circuit current until drivers can be tri-stated. Refer to Section 4.3.2 for more information.

4.2.1.3. Maximum AC Ratings and Device Protection

Maximum AC waveforms are included here as examples of worst case AC operating conditions. It is recommended that these waveforms be used as qualification criteria, against which the long term reliability of a device is evaluated. This is not intended to be used as a production test; it is intended that this level of robustness be guaranteed by design. This section covers AC operating conditions only; DC conditions are specified in Section 4.2.1.1.

The PCI environment contains many reactive elements and, in general, must be treated as a non-terminated, transmission line environment. The basic premise of the environment requires that a signal reflect at the end of the line and return to the driver before the signal is considered switched. As a consequence of this environment, under certain conditions of drivers, device topology, system board impedance, add-in card impedance, etc., the "open circuit" voltage at the pins of PCI devices will exceed the expected ground-to-V_{CC} voltage range by a considerable amount. The technology used to implement PCI can vary from vendor to vendor, so it cannot be assumed that the technology is naturally immune to these effects. This under-/over-voltage specification provides a synthetic worst-case AC environment, against which the long term reliability of a device can be evaluated.

All input, bi-directional, and tri-state outputs used on each PCI device must be capable of continuous exposure to the following synthetic waveform which is applied with the equivalent of a zero impedance voltage source driving a series resistor directly into each input or tri-stated output pin of the PCI device. The waveform provided by the voltage source (or open circuit voltage) and the resistor value are shown in Figure 4-3. The open circuit waveform is a composite of simulated worst cases;[29] some had narrower pulse widths, while others had lower voltage peaks. The resistor is calculated to provide a worst case current into an effective (internal) clamp diode. Note that:

- ☐ The voltage waveform is supplied at the resistor shown in the evaluation setup, not the package pin.
- ☐ With effective clamping, the waveform at the package pin will be greatly reduced.

---

[29] Waveforms based on worst case (strongest) driver, maximum and minimum system configurations, with no internal clamp diodes.

- ☐ The upper clamp is optional, but if used, it must be connected to the 5V supply or the VI/O plane of the add-in card but never[30] the 3.3V supply.
- ☐ For devices built in "3 volt technology," the upper clamp is, in practice, required for device protection.
- ☐ In order to limit signal ringing in systems that tend to generate large overshoots, system board vendors may wish to use layout techniques to lower circuit impedance.

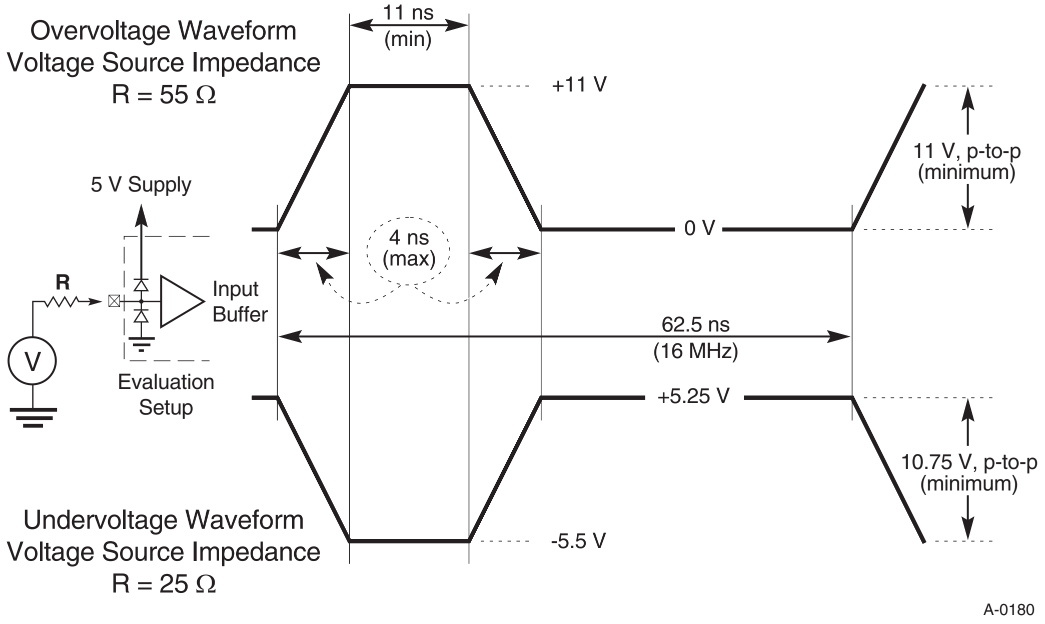


Figure 4-3: Maximum AC Waveforms for 5V Signaling

---

[30] It is possible to use alternative clamps, such as a diode stack to the 3.3V rail or a circuit to ground, if it can be insured that the I/O pin will never be clamped below the 5V level.

4.2.2. 3.3V Signaling Environment

4.2.2.1. DC Specifications

Table 4-3 summarizes the DC specifications for 3.3V signaling.

Table 4-3: DC Specifications for 3.3V Signaling

Symbol	Parameter	Condition	Min	Max	Units	Notes
V_{cc}	Supply Voltage		3.0	3.6	V	
V_{ih}	Input High Voltage		$0.5V_{cc}$	$V_{cc} + 0.5$	V	
V_{il}	Input Low Voltage		-0.5	$0.3V_{cc}$	V	
V_{ipu}	Input Pull-up Voltage		$0.7V_{cc}$		V	1
I_{il}	Input Leakage Current	$0 < V_{in} < V_{cc}$		±10	µA	2
V_{oh}	Output High Voltage	$I_{out} = -500\ \mu A$	$0.9V_{cc}$		V	
V_{ol}	Output Low Voltage	$I_{out} = 1500\ \mu A$		$0.1V_{cc}$	V	
C_{in}	Input Pin Capacitance			10	pF	3
C_{clk}	CLK Pin Capacitance		5	12	pF	
C_{IDSEL}	IDSEL Pin Capacitance			8	pF	4
L_{pin}	Pin Inductance			20	nH	5
I_{Off}	PME# input leakage	$V_o \leq 3.6$ V V_{cc} off or floating	–	1	µA	6

Notes:

1. This specification should be guaranteed by design. It is the minimum voltage to which pull-up resistors are calculated to pull a floated network. Applications sensitive to static power utilization must assure that the input buffer is conducting minimum current at this input voltage.
2. Input leakage currents include hi-Z output leakage for all bi-directional buffers with tri-state outputs.
3. Absolute maximum pin capacitance for a PCI input is 10 pF (except for CLK, SMBDAT, and SMBCLK) with an exception granted to system board-only devices up to 16 pF in order to accommodate PGA packaging. This would mean, in general, that components for add-in cards need to use alternatives to ceramic PGA packaging; i.e., PQFP, SGA, etc. Pin capacitance for SMBCLK and SMBDAT is not specified; however, the maximum capacitive load is specified for the add-in card in Section 8.2.5.
4. Lower capacitance on this input-only pin allows for non-resistive coupling to AD[xx].
5. This is a recommendation, not an absolute requirement. The actual value should be provided with the component data sheet.
6. This input leakage is the maximum allowable leakage into the PME# open drain driver when power is removed from V_{cc} of the component. This assumes that no event has occurred to cause the device to attempt to assert PME#.

Refer to Section 3.8.1 for special requirements for AD[63::32], C/BE[7::4]#, and PAR64 when they are not connected (as in a 64-bit add-in card installed in a 32-bit connector).

4.2.2.2. AC Specifications

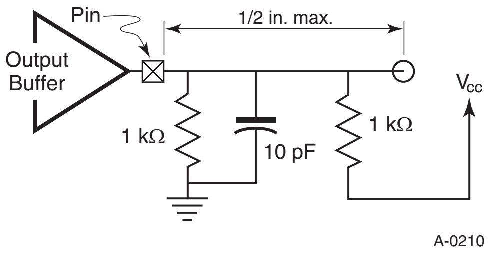
Table 4-4 summarizes the AC specifications for 3.3V signaling.

Table 4-4: AC Specifications for 3.3V Signaling

Symbol	Parameter	Condition	Min	Max	Units	Notes
I_{oh}(AC)	Switching Current High	$0 < V_{out} \leq 0.3V_{cc}$	$-12V_{cc}$		mA	1
		$0.3V_{cc}<V_{out}<0.9V_{cc}$	$-17.1(V_{cc}-V_{out})$		mA	1
		$0.7V_{cc} < V_{out} < V_{cc}$		Eqt'n C		1, 2
	(Test Point)	$V_{out} = 0.7V_{cc}$		$-32V_{cc}$	mA	2
I_{ol}(AC)	Switching Current Low	$V_{cc} > V_{out} \geq 0.6V_{cc}$	$16V_{cc}$		mA	1
		$0.6V_{cc}>V_{out}>0.1V_{cc}$	$26.7V_{out}$		mA	1
		$0.18V_{cc}>V_{out}>0$		Eqt'n D		1, 2
	(Test Point)	$V_{out} = 0.18V_{cc}$		$38V_{cc}$	mA	2
I_{cl}	Low Clamp Current	$-3 < V_{in} \leq -1$	$-25+(V_{in}+1)/0.015$		mA	
I_{ch}	High Clamp Current	$V_{cc}+4 > V_{in} \geq V_{cc}+1$	$25+(V_{in}-V_{cc}-1)/0.015$		mA	
$slew_r$	Output Rise Slew Rate	$0.2V_{cc}$ - $0.6V_{cc}$ load	1	4	V/ns	3
$slew_f$	Output Fall Slew Rate	$0.6V_{cc}$ - $0.2V_{cc}$ load	1	4	V/ns	3

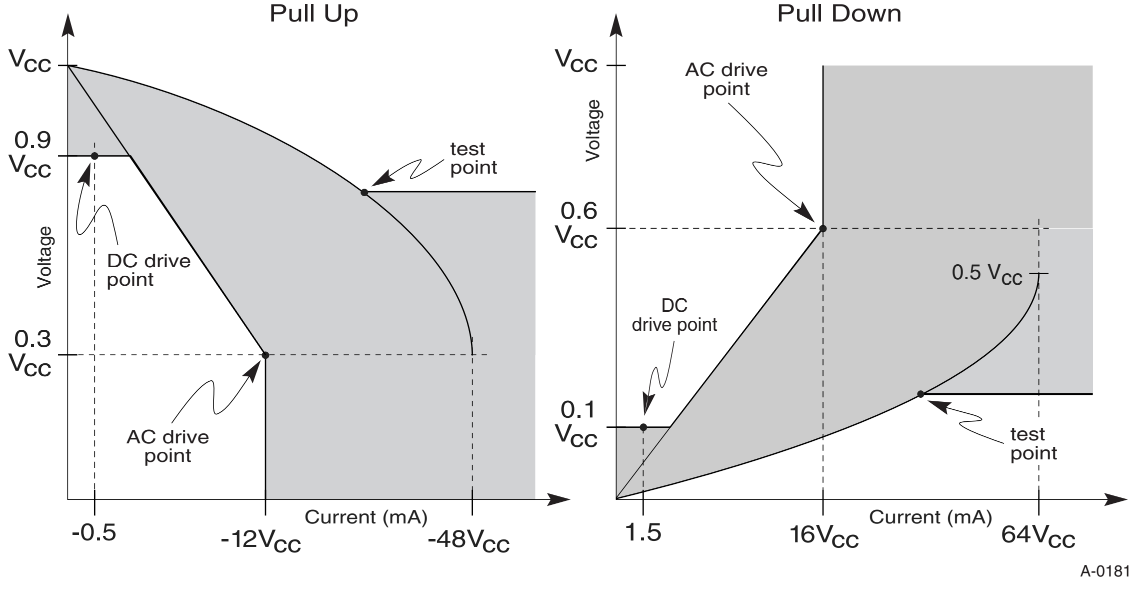
Notes:

1. Refer to the V/I curves in Figure 4-4. Switching current characteristics for REQ# and GNT# are permitted to be one half of that specified here; i.e., half size output drivers may be used on these signals. This specification does not apply to CLK and RST# which are system outputs. "Switching Current High" specifications are not relevant to SERR#, PME#, INTA#, INTB#, INTC#, and INTD# which are open drain outputs.
2. Maximum current requirements must be met as drivers pull beyond the first step voltage. Equations defining these maximums (C and D) are provided with the respective diagrams in Figure 4-4. The equation-defined maximums should be met by design. In order to facilitate component testing, a maximum current test point is defined for each side of the output driver.
3. This parameter is to be interpreted as the cumulative edge rate across the specified range, rather than the instantaneous rate at any point within the transition range. The specified load (diagram below) is optional; i.e., the designer may elect to meet this parameter with an unloaded output. However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline). Rise slew rate does not apply to open drain outputs.



The minimum and maximum drive characteristics of PCI output buffers are defined by V/I curves. These curves should be interpreted as traditional "DC" transistor curves with the following exceptions: the "DC Drive Point" is the only position on the curves at which steady state operation is intended, while the higher current parts of the curves are only reached momentarily during bus switching transients. The "AC Drive Point" (the real definition of buffer strength) defines the minimum instantaneous current curve required to switch the bus with a single reflection. From a quiescent or steady state, the current associated with the AC drive point must be reached within the output delay time, T_{val}. Note, however, that this delay time also includes necessary logic time. The partitioning of T_{val} between clock distribution, logic, and output buffer is not specified, but the faster the buffer (as long as it does not exceed the maximum rise/fall slew rate specification), the more time is allowed for logic delay inside the part. The "Test Point" defines the maximum allowable instantaneous current curve in order to limit switching noise and is selected roughly on a 22 Ω load line.

Adherence to these curves is evaluated at worst case conditions. The minimum pull up curve is evaluated at minimum V_{cc} and high temperature. The minimum pull down curve is evaluated at maximum V_{cc} and high temperature. The maximum curve test points are evaluated at maximum V_{cc} and low temperature.



Equation C:

$I_{oh} = (98.0/V_{cc})*(V_{out}-V_{cc})*(V_{out}+0.4V_{cc})$

for $V_{cc} > V_{out} > 0.7V_{cc}$

Equation D:

$I_{ol} = (256/V_{cc})*V_{out}*(V_{cc}-V_{out})$

for $0\ V < V_{out} < 0.18V_{cc}$

Figure 4-4: V/I Curves for 3.3V Signaling

Inputs are required to be clamped to both ground and V_{CC} (3.3V) rails. When dual power rails are used, parasitic diode paths could exist from one supply to another. These diode paths can become significantly forward biased (conducting) if one of the power rails goes out of specification momentarily. Diode clamps to a power rail, as well as output pull-up devices, must be able to withstand short circuit current until drivers can be tri-stated. Refer to Section 4.3.2 for more information.

4.2.2.3. Maximum AC Ratings and Device Protection

Refer to the "Maximum AC Ratings" section in the 5V signaling environment. Maximum AC waveforms are included here as examples of worst case AC operating conditions. It is recommended that these waveforms be used as qualification criteria against which the long term reliability of a device is evaluated. This is not intended to be used as a production test; it is intended that this level of robustness be guaranteed by design. This section covers AC operating conditions only; DC conditions are specified in Section 4.2.2.1.

All input, bi-directional, and tri-state outputs used on each PCI device must be capable of continuous exposure to the following synthetic waveform, which is applied with the equivalent of a zero impedance voltage source driving a series resistor directly into each input or tri-stated output pin of the PCI device. The waveform provided by the voltage source (or open circuit voltage) and the resistor value are shown in Figure 4-5. The open circuit waveform is a composite of simulated worst cases; some had narrower pulse widths, while others had lower voltage peaks. The resistor is calculated to provide a worst case current into an effective (internal) clamp diode. Note that:

- ☐ The voltage waveform is supplied at the resistor shown in the evaluation setup, not the package pin.
- ☐ With effective clamping, the waveform at the package pin will be greatly reduced.
- ☐ In order to limit signal ringing in systems that tend to generate large overshoots, system board vendors may wish to use layout techniques to lower circuit impedance.

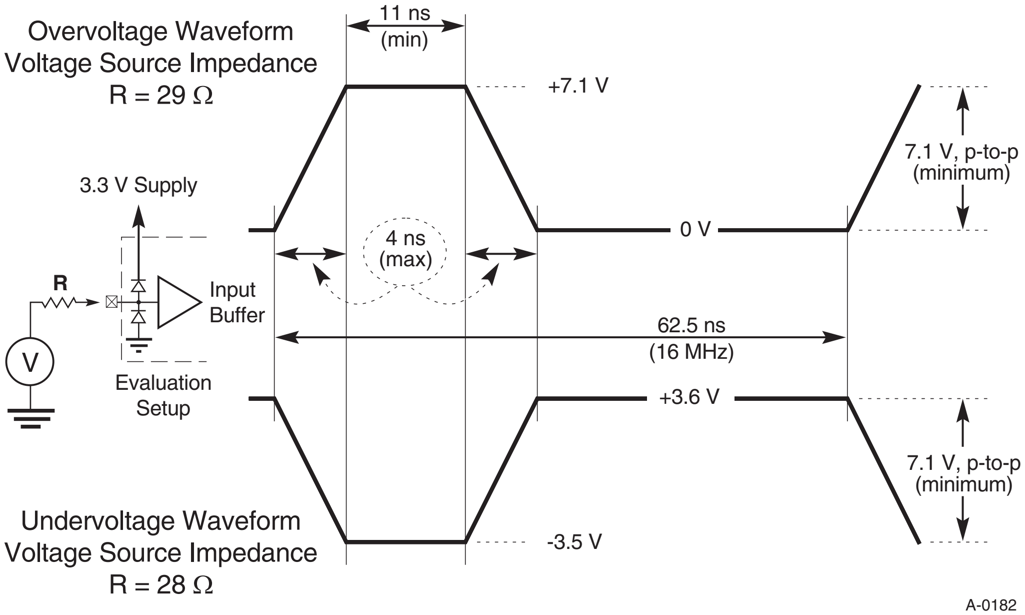


Figure 4-5: Maximum AC Waveforms for 3.3V Signaling

4.2.3. Timing Specification

4.2.3.1. Clock Specification

The clock waveform must be delivered to each PCI component in the system. In the case of add-in cards, compliance with the clock specification is measured at the add-in card component not at the connector slot. Figure 4-6 shows the clock waveform and required measurement points for both 5V and 3.3V signaling environments. Table 4-5 summarizes the clock specifications. Refer to item 8 in Section 3.10 for special considerations when using PCI-to-PCI bridges on add-in cards or when add-in card slots are located downstream of a PCI-to-PCI bridge.

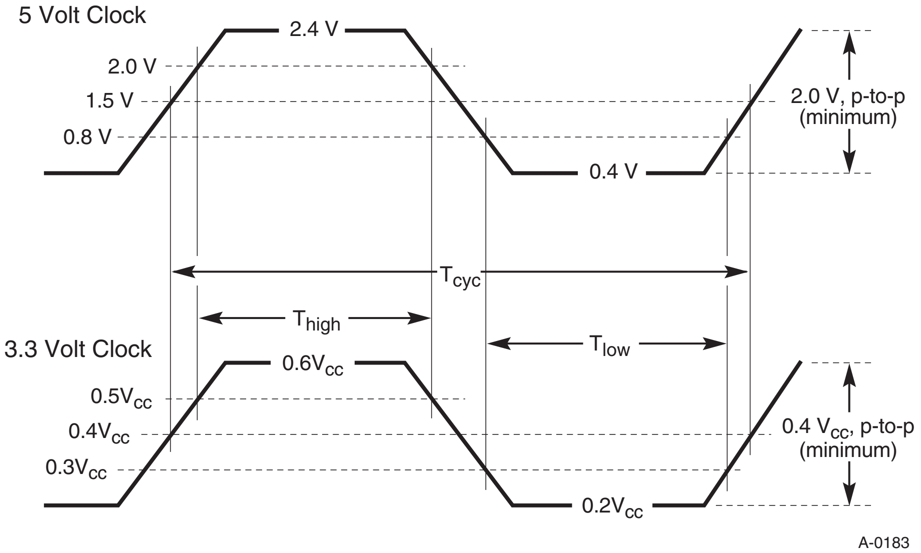


Figure 4-6: Clock Waveforms

Table 4-5: Clock and Reset Specifications

Symbol	Parameter	Min	Max	Units	Notes
T_{cyc}	CLK Cycle Time	30	∞	ns	1
T_{high}	CLK High Time	11		ns	
T_{low}	CLK Low Time	11		ns	
-	CLK Slew Rate	1	4	V/ns	2
-	RST# Slew Rate	50	-	mV/ns	3

Notes:

1. In general, all PCI components must work with any clock frequency between nominal DC and 33 MHz. Device operational parameters at frequencies under 16 MHz may be guaranteed by design rather than by testing. The clock frequency may be changed at any time during the operation of the system so long as the clock edges remain "clean" (monotonic) and the minimum cycle and high and low times are not violated. For example, the use of spread spectrum techniques to reduce EMI emissions is included in this requirement. Refer to Section 7.6.4.1 for the spread spectrum requirements for 66 MHz. The clock may only be stopped in a low state. A variance on this specification is allowed for components designed for use on the system board only. These components may operate at any single fixed frequency up to 33 MHz and may enforce a policy of no frequency changes.

2. Rise and fall times are specified in terms of the edge rate measured in V/ns. This slew rate must be met across the minimum peak-to-peak portion of the clock waveform as shown in Figure 4-7.

3. The minimum RST# slew rate applies only to the rising (deassertion) edge of the reset signal and ensures that system noise cannot render an otherwise monotonic signal to appear to bounce in the switching range. RST# waveforms and timing are discussed in Section 4.3.2.

4.2.3.2. *Timing Parameters*

Table 4-6 provides the timing parameters for 3.3V and 5V signaling environments.

Table 4-6: 3.3V and 5V Timing Parameters

Symbol	Parameter	Min	Max	Units	Notes
T_{val}	CLK to Signal Valid Delay - bused signals	2	11	ns	1, 2, 3
T_{val}(ptp)	CLK to Signal Valid Delay - point to point	2	12	ns	1, 2, 3
T_{on}	Float to Active Delay	2		ns	1, 7
T_{off}	Active to Float Delay		28	ns	1, 7
T_{su}	Input Setup Time to CLK - bused signals	7		ns	3, 4, 8
T_{su}(ptp)	Input Setup Time to CLK - point to point	10, 12		ns	3, 4
T_h	Input Hold Time from CLK	0		ns	4
T_{rst}	Reset active time after power stable	1		ms	5
$T_{rst\text{-}clk}$	Reset active time after CLK STABLE	100		μs	5
$T_{rst\text{-}off}$	Reset Active to Output Float delay		40	ns	5, 6,7
T_{rrsu}	REQ64# to RST# Setup time	10*T_{cyc}		ns	
T_{rrh}	RST# to REQ64# Hold time	0	50	ns	
T_{rhfa}	RST# High to First configuration Access	2^{25}		clocks	
T_{rhff}	RST# High to First FRAME# assertion	5		clocks	
T_{pvrh}	Power valid to RST# high	100		ms	

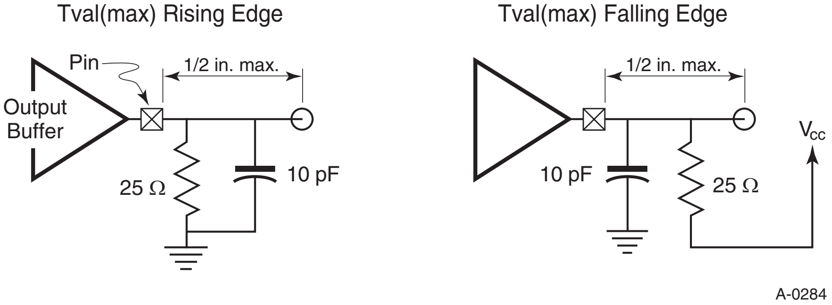
Notes:

1. See the timing measurement conditions in Figure 4-7.
2. For parts compliant to the 3.3V signaling environment:

   Minimum times are evaluated with the same load used for slew rate measurement (as shown in Table 4-4, note 3); maximum times are evaluated with the following load circuits, for high-going and low-going edges respectively.

   For parts compliant to the 5V signaling environment:

   Minimum times are evaluated with 0 pF equivalent load; maximum times are evaluated with 50 pF equivalent load. Actual test capacitance may vary, but results must be correlated to these specifications. Note that faster buffers may exhibit some ring back when attached to a 50 pF lump load which should be of no consequence as long as the output buffers are in full compliance with slew rate and V/I curve specifications.



3. REQ# and GNT# are point-to-point signals and have different output valid delay and input setup times than do bused signals. GNT# has a setup of 10; REQ# has a setup of 12. All other signals are bused.
4. See the timing measurement conditions in Figure 4-8.
5. CLK is stable when it meets the requirements in Section 4.2.3.1. RST# is asserted and deasserted asynchronously with respect to CLK. Refer to Section 4.3.2 for more information.
6. All output drivers must be asynchronously floated when RST# is active. Refer to Section 3.8.1 for special requirements for AD[63::32], C/BE[7::4]#, and PAR64 when they are not connected (as in a 64-bit add-in card installed in a 32-bit connector).
7. For purposes of Active/Float timing measurements, the Hi-Z or "off" state is defined to be when the total current delivered through the component pin is less than or equal to the leakage current specification.
8. Setup time applies only when the device is not driving the pin. Devices cannot drive and receive signals at the same time. Refer to Section 3.10, item 9, for additional details.

4.2.3.3. Measurement and Test Conditions

Figures 4-7 and 4-8 define the conditions under which timing measurements are made. The component test guarantees that all timings are met with minimum clock slew rate (slowest edge) and voltage swing. The design must guarantee that minimum timings are also met with maximum clock slew rate (fastest edge) and voltage swing. In addition, the design must guarantee proper input operation for input voltage swings and slew rates that exceed the specified test conditions.

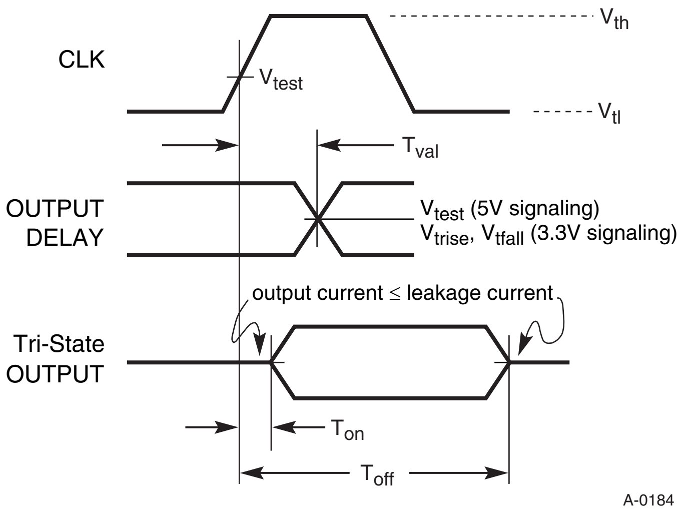

CLK
V_{th}
V_{test}
V_{tl}
T_{val}
OUTPUT DELAY
V_{test} (5V signaling)
V_{trise}, V_{tfall} (3.3V signaling)
output current ≤ leakage current
Tri-State OUTPUT
T_{on}
T_{off}
A-0184

Figure 4-7: Output Timing Measurement Conditions

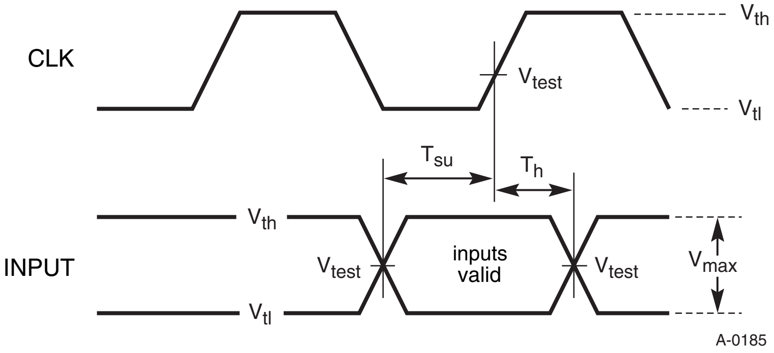


Figure 4-8: Input Timing Measurement Conditions

Table 4-7: Measure Condition Parameters

Symbol	3.3V Signaling	5V Signaling	Units
V_{th}	$0.6V_{cc}$	2.4	V (Note)
V_{tl}	$0.2V_{cc}$	0.4	V (Note)
V_{test}	$0.4V_{cc}$	1.5	V
V_{trise}	$0.285V_{cc}$	n/a	V
V_{tfall}	$0.615V_{cc}$	n/a	V
V_{max}	$0.4V_{cc}$	2.0	V (Note)
Input Signal Edge Rate	1 V/ns		

Note:
The input test for the 3.3V environment is done with $0.1V_{cc}$ of overdrive; the test for the 5V environment is done with 400 mV of overdrive (over V_{ih} and V_{il}). Timing parameters must be met with no more overdrive than this. V_{max} specifies the maximum peak-to-peak waveform allowed for measuring input timing. Production testing may use different voltage values, but must correlate results back to these parameters.

4.2.4. Indeterminate Inputs and Metastability

At times, various inputs may be indeterminate. Components must avoid logical operational errors and metastability by sampling inputs only on "qualified" clock edges. In general, synchronous signals are assumed to be valid and determinate only at the clock edge on which they are "qualified" (refer to Section 3.2).

System designs must assure that floating inputs are biased away from the switching region in order to avoid logical, electrical, thermal, or reliability problems. In general, it is not possible to avoid situations where low slew rate signals (e.g., resistively coupled IDSEL) pass through the switching region at the time of a clock edge, but they must not be allowed to remain at the threshold point for many clock periods. Frequently, a pre-charged bus may be assumed to retain its state while not driven for a few clock periods during bus turnaround.

There are specific instances when signals are known to be indeterminate. These must be carefully considered in any design.

All AD[31::00], C/BE[3::0]#, and PAR pins are indeterminate when tri-stated for bus turnaround. This will sometimes last for several cycles while waiting for a device to respond at the beginning of a transaction.

The IDSEL pin is indeterminate at all times except during configuration cycles. If a resistive connection to an AD line is used, it may tend to float around the switching region much of the time.

The PME# and SERR# pins must be considered indeterminate for a number of cycles after they have been deasserted.

Nearly all signals will be indeterminate for as long as RST# is asserted and for a period of time after it is released. Pins with pull-up resistors will eventually resolve high.

4.2.5. Vendor Provided Specification

The vendor of a PCI system is responsible for electrical simulation of the PCI bus and components to guarantee proper operation. To help facilitate this effort, component vendors are encouraged to make the following information available: (It is recommended that component vendors make this information electronically available in the IBIS model format.)

- ☐ Pin capacitance for all pins.
- ☐ Pin inductance for all pins.
- ☐ Output V/I curves under switching conditions. Two curves should be given for each output type used: one for driving high, the other for driving low. Both should show best-typical-worst curves. Also, "beyond-the-rail" response is critical, so the voltage range should span -3 V to 7 V for 3.3V signaling and -5 V to 10 V for 5V signaling.
- ☐ Input V/I curves under switching conditions. A V/I curve of the input structure when the output is tri-stated is also important. This plot should also show best-typical-worst curves over the range of 0 to V_{CC}.
- ☐ Rise/fall slew rates for each output type.
- ☐ Complete absolute maximum data, including operating and non-operating temperature, DC maximums, etc.

In addition to this component information, connector vendors are encouraged to make available accurate simulation models of PCI connectors.

4.2.6. Pinout Recommendation

This section provides a recommended pinout for PCI components. Since add-in card stubs are so limited, layout issues are greatly minimized if the component pinout aligns exactly with the add-in card (connector) pinout. Components for use only on system boards are encouraged also to follow this same signal ordering to allow layouts with minimum stubs. Figure 4-9 shows the recommended pinout for a typical PQFP PCI component. Note that the pinout is exactly aligned with the signal order on the add-in card connector. Placement and number of power and ground pins is device-dependent.

The additional signals needed in 64-bit versions of the bus continue wrapping around the component in a counter-clockwise direction in the same order they appear on the 64-bit connector extension.

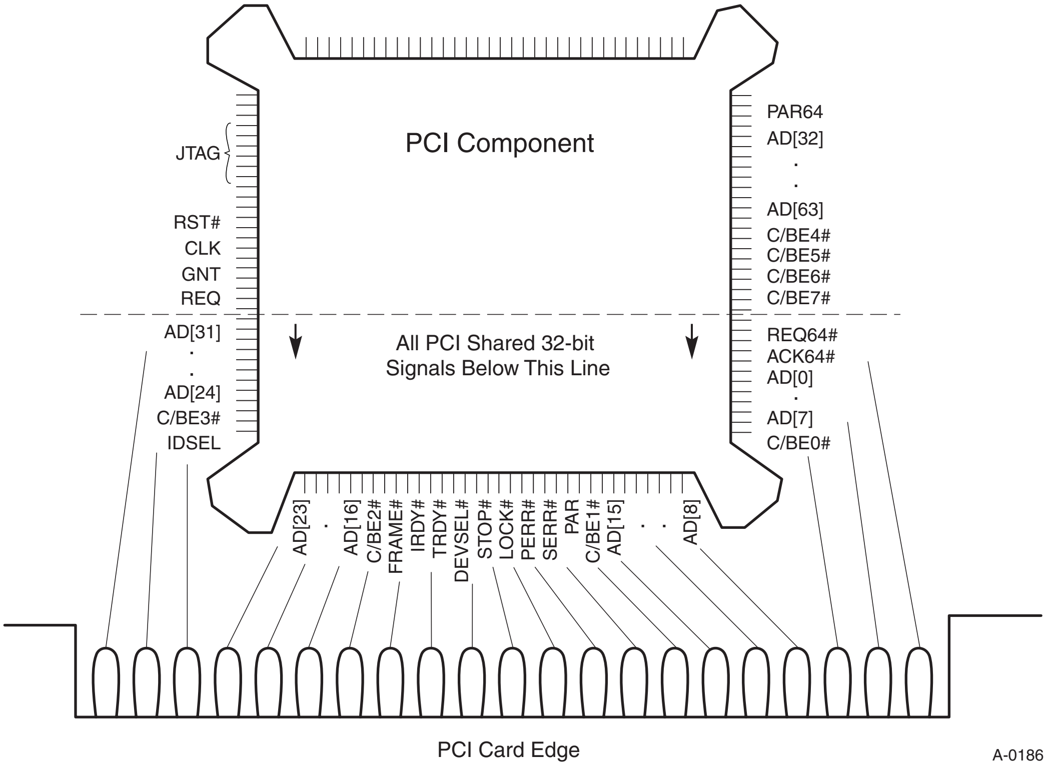


Figure 4-9: Suggested Pinout for PQFP PCI Component

Placing the IDSEL input as close as possible to AD[31::11] allows the option for a non-resistive[31] connection of IDSEL to the appropriate address line with a small additional load. Note that this pin has a lower capacitance specification that in some cases will constrain its placement in the package.

---

[31] Non-resistive connections of IDSEL to one of the AD[xx] lines create a technical violation of the single load per add-in card rule. PCI protocol provides for pre-driving of address lines in configuration cycles, and it is recommended that this be done in order to allow a resistive coupling of IDSEL. In absence of this, signal performance must be derated for the extra IDSEL load.

4.3. System Board Specification

4.3.1. Clock Skew

The maximum allowable clock skew is 2 ns. This specification applies not only at a single threshold point, but at all points on the clock edge that fall in the switching range defined in Table 4-8 and Figure 4-10. The maximum skew is measured between any two components[32] rather than between connectors. To correctly evaluate clock skew, the system designer must take into account clock distribution on the add-in card which is specified in Section 4.4.

Table 4-8: Clock Skew Parameters

Symbol	3.3V Signaling	5V Signaling	Units
V_{test}	0.4 V_{cc}	1.5	V
T_{skew}	2 (max)	2 (max)	ns

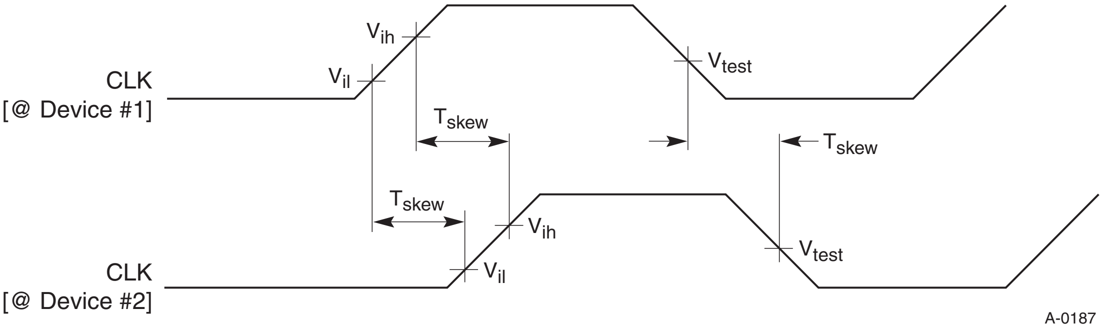


Figure 4-10: Clock Skew Diagram

4.3.2. Reset

The assertion and deassertion of the PCI reset signal (RST#) is asynchronous with respect to CLK. The rising (deassertion) edge of the RST# signal must be monotonic (bounce free) through the input switching range and must meet the minimum slew rate specified in Table 4-5. The PCI specification does not preclude the implementation of a synchronous RST#, if desired. The timing parameters for reset are listed in Table 4-6 with the exception of the T_{fail} parameter. This parameter provides for system reaction to one or both of the power rails going out of specification. If this occurs, parasitic diode paths could short circuit

---

[32] The system designer must address an additional source of skew. This clock skew occurs between two components that have clock input trip points at opposite ends of the V_{il} - V_{ih} range. In certain circumstances, this can add to the clock skew measurement as described here. In all cases, total clock skew must be limited to the specified number.

active output buffers. Therefore, RST# is asserted upon power failure in order to float the output buffers.

The value of T_{fail} is the minimum of:

- 500 ns (maximum) from either power rail going out of specification (exceeding specified tolerances by more than 500 mV)
- 100 ns (maximum) from the 5V rail falling below the 3.3V rail by more than 300 mV

The system must assert RST# during power up or in the event of a power failure. In order to minimize possible voltage contention between 5V and 3.3V parts, RST# must be asserted as soon as possible during the power up sequence. Figure 4-11 shows a worst case assertion of RST# asynchronously following the "power good" signal.[33] After RST# is asserted, PCI components must asynchronously disable (float) their outputs but are not considered reset until both T_{rst} and $T_{rst\text{-}clk}$ parameters have been met. The first rising edge of RST# after power-on for any device must be no less than T_{pvrh} after all the power supply voltages are within their specified limits for that device. If RST# is asserted while the power supply voltages remain within their specified limits, the minimum pulse width of RST# is T_{rst}. Figure 4-11 shows RST# signal timing.

The system must guarantee that the bus remains in the idle state for a minimum time delay following the deassertion of RST# to a device before the system will permit the first assertion of FRAME#. This time delay is included in Table 4-6 as Reset High to First FRAME# assertion (T_{rhff}). If a device requires longer than the specified time (T_{rhff}) after the deassertion of RST# before it is ready to participate in the bus signaling protocol, then the device's state machines must remain in the reset state until all of the following are simultaneously true:

- RST# is deasserted.
- The device is ready to participate in the bus signaling protocol.
- The bus is in the idle state.

IMPLEMENTATION NOTE

Reset

An example of a device that could require more than T_{rhff} to be ready to participate in the bus signaling protocol is a 66-MHz device that includes a Phase-Locked Loop (PLL) for distribution of the PCI clock internally to the device. The device may inhibit clocks to the PCI interface until the PLL has locked. Since the PLL could easily require more than T_{rhff} to lock, this could result in the clocks being enabled to the PCI interface of this device in the middle of a burst transfer between two other devices. When this occurs, the 66-MHz device would have to detect an idle bus condition before enabling the target selection function.

---

[33] Component vendors should note that a fixed timing relationship between RST# and power sequencing cannot be guaranteed in all cases.

Some PCI devices must be prepared to respond as a target T_{rhff} time after RST# deasserts. For example, devices in the path between the CPU and the boot ROM (not expansion ROM) must be prepared to respond as a target T_{rhff} time after RST# deasserts.

All other devices must be prepared to respond as a target not more than T_{rhfa} after the deassertion of RST#. It is recommended that the system wait at least T_{rhfa} following the deassertion of RST# to a device before the first access to that device, unless the device is in the path between the CPU and the boot ROM or the system knows that the device is ready sooner.

Software that accesses devices prior to the expiration of T_{rhfa} must be prepared for the devices either not to respond at all (resulting in Master-Abort) or for the devices to respond with Retry until the expiration of T_{rhfa}. At no time can a device return invalid data. Devices are exempt from the Maximum Retry Time specification and the target initial latency requirement until the expiration of T_{rhfa}.

IMPLEMENTATION NOTE

Trhfa

Devices are encouraged to complete their initialization and be ready to accept their first cycle (generally a Configuration Read cycle) as soon as possible after the deassertion of RST#. Some system implementations will access devices prior to waiting the full value of T_{rhfa}, if they know in advance that all devices are ready sooner. For example, a system with no PCI slots would only need to wait for the initialization requirements of the embedded devices. Similarly, an intelligent add-in card that initialized its own embedded PCI devices would only need to wait for the initialization requirements of those devices.

In some cases, such as intelligent add-in cards, the add-in card designer must select devices that initialize in less than the full value of T_{rhfa}. For example, suppose an intelligent add-in card is not ready to respond as a target to the first Configuration transaction from the host CPU until after the local CPU has configured the local devices. In this case, the local devices must initialize fast enough to enable the local CPU to complete its initialization in time for the first access from the host CPU.

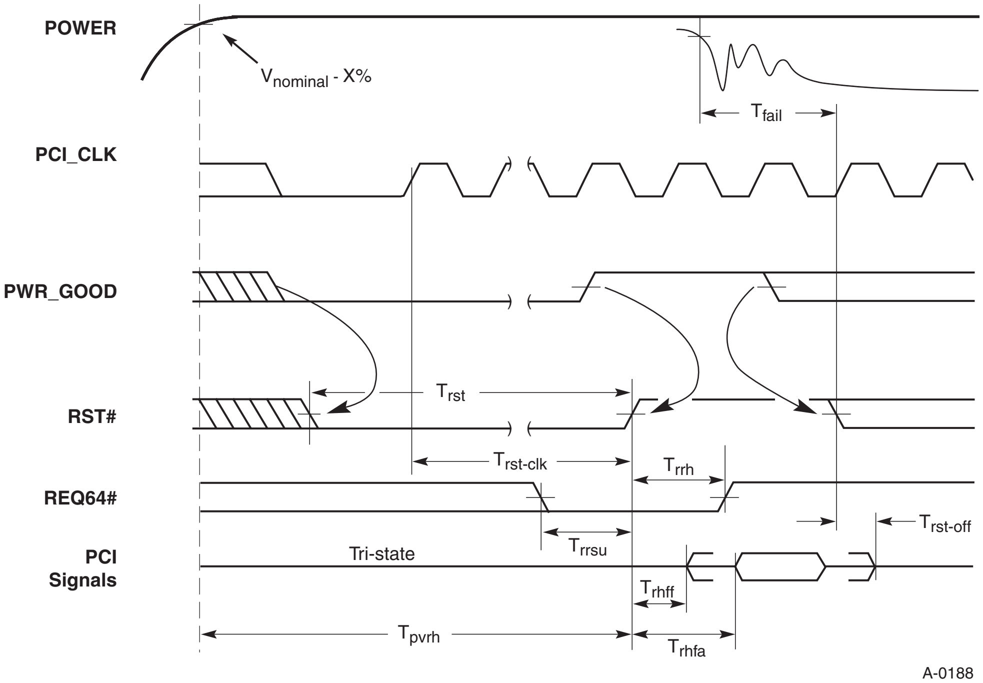


Figure 4-11: Reset Timing [34]

Refer to Section 3.8.1 for special requirements for AD[63::32], C/BE[7::4]#, and PAR64 when they are not connected (as in a 64-bit add-in card installed in a 32-bit connector).

4.3.3. Pull-ups

PCI control signals always require pull-up resistors (pulled up to V_{CC} of the signaling environment) on the system board (not the add-in card) to ensure that they contain stable values when no agent is actively driving the bus. This includes FRAME#, TRDY#, IRDY#, DEVSEL#, STOP#, SERR#, PERR#, LOCK#, INTA#, INTB#, INTC#, INTD#, REQ64#, and ACK64#. The point-to-point and shared 32-bit signals do not require pull-ups; bus parking ensures their stability. Refer to Section 3.8.1 for special requirements for terminating AD[63::32], C/BE[7::4]#, and PAR64. Refer to Section 4.3.7 for pull-up and decoupling requirements for PRSNT1# and PRSNT2#. Refer to Section 7.7.7 for pull-up and decoupling requirements for M66EN.

A system that does not support the optional SMBus interface must provide individual pull-up resistors (~5 kΩ) on the SMBCLK and SMBDAT pins for the system board connectors. A system that supports the SMBus interface must provide pull-up devices (passive or active) on SMBCLK and SMBDAT as defined in the SMBus 2.0 Specification. Refer to Section 8 of this specification. The pull-ups must be connected to the power source attached to the 3.3Vaux pin of the PCI connector for systems with the optional auxiliary power supply and

---

[34] This reset timing figure optionally shows the "PWR_GOOD" signal as a pulse which is used to time the RST# pulse. In many systems, "PWR_GOOD" may be a level, in which case the RST# pulse must be timed in another way.

to +3.3V supply for systems without the optional supply. If boundary scan is not implemented on the system board, **TMS** and **TDI** must be independently bused and pulled up, each with ~5 kΩ resistors, and **TRST#** and **TCK** must be independently bused and pulled down, each with ~5 kΩ resistors. **TDO** must be left open.

The formulas for minimum and maximum pull-up resistors are provided below. R_{min} is primarily driven by I_{ol}, the DC low output current, whereas the number of loads only has a secondary effect. On the other hand, R_{max} is primarily driven by the number of loads present. The specification provides for a minimum R value that is calculated based on 16 loads (believed to be a worst case) and a typical R value that is calculated as the maximum R value with 10 loads. The maximum R value is provided by formula only and will be the highest in a system with the smallest number of loads.

$R_{min} = [V_{cc(max)} - V_{ol}] / [I_{ol} + (16 * I_{il})]$, where 16 = max number of loads

$R_{max} = [V_{cc(min)} - V_x] / [num\_loads * I_{ih}]$, where Vx = 2.7 V for 5V signaling and $V_x = 0.7V_{cc}$ for 3.3V signaling.

Table 4-9 provides minimum and typical values for both 3.3V and 5V signaling environments. The typical values have been derated for 10% resistors at nominal values.

Table 4-9: Minimum and Typical Pull-up Resistor Values

Signaling Rail	R_{min}	$R_{typical}$	R_{max}
3.3V	2.42 kΩ	8.2 kΩ @ 10%	see formula
5V	963 Ω	2.7 kΩ @ 10%	see formula

The central resource, or any component containing an arbitration unit, may require a weak pull-up on each unconnected **REQ#** pin and each **REQ#** pin connected to a PCI slot in order to insure that these signals do not float. Values for this pull-up shall be specified by the central resource vendor.

Systems utilizing **PME#** must provide a pull-up on that signal. The resistor value used is calculated using the formulas above but substituting I_{Off} for I_{ih}.

4.3.4. Power

4.3.4.1. Power Requirements

All PCI connectors require four power rails: +5V, +3.3V, +12V, and -12V. Systems that provide PCI connectors are required to provide all four rails in every system with the current budget specified in Table 4-10. Systems may optionally supply 3.3Vaux power, as specified in the *PCI Bus Power Management Interface Specification*. Systems that do not support PCI bus power management must treat the 3.3Vaux pin as reserved.

Current requirements per connector for the two 12V rails are provided in Table 4-10. There are no specific system requirements for current per connector on the 3.3V and 5V rails; this is system dependent. Note that Section 4.4.2.2 requires that an add-in card must limit its total power consumption to 25 watts (from all power rails). The system provides a total power budget for add-in cards that can be distributed between connectors in an arbitrary way. The PRSNTn# pins on the connector allow the system to optionally assess the power demand of each add-in card and determine if the installed configuration will run within the total power budget. Refer to Section 4.4.1 for further details.

Table 4-10 specifies the tolerances of supply rails. Note that these tolerances are to be guaranteed at the components not the supply.

Table 4-10: Power Supply Rail Tolerances

Power Rail	Add-in Cards (Short and Long)
3.3 V ±0.3 V	7.6 A max. (system dependent)
5 V ± 5 %	5 A max. (system dependent)
12 V ±5%	500 mA max.
-12 V ±10%	100 mA max.

4.3.4.2. Sequencing

There is no specified sequence in which the four power rails are activated or deactivated. They may come up and go down in any order. The system must assert RST# both at power up and whenever either the 3.3V or 5V rails go out of specification (per Section 4.3.2). During reset, all PCI signals are driven to a "safe" state, as described in Section 4.3.2.

4.3.4.3. Decoupling

All power planes must be decoupled to ground to provide:

- ☐ Reasonable management of the switching currents (dI/dt) to which the plane and its supply path are subjected
- ☐ An AC return path in a manner consistent with high-speed signaling techniques

This is platform dependent and not detailed in the specification.

4.3.5. System Timing Budget

When computing a total PCI load model, careful attention must be paid to maximum trace length and loading of add-in cards, as specified in Section 4.4.3. Also, the maximum pin capacitance of 10 pF must be assumed for add-in cards, whereas the actual pin capacitance may be used for system board devices.

The total clock period can be divided into four segments. Valid output delay (T_{val}) and input setup time (T_{su}) are specified by the component specification. Total clock skew (T_{skew}) and maximum bus propagation time (T_{prop}) are system parameters. T_{prop} is specified as 10 ns but may be increased to 11 ns by lowering clock skew; that is, T_{prop} plus T_{skew} together may not exceed 12 ns; however, under no circumstance may T_{skew} exceed 2 ns. Furthermore, by using clock rates slower than 33 MHz, some systems may build larger PCI topologies having T_{prop} values larger than those specified here. Since component times (T_{val} and T_{su}) and clock skew are fixed, any increase in clock cycle time allows an equivalent increase in T_{prop}. For example, at 25 MHz (40 ns clock period), T_{prop} may be increased to 20 ns. Note that this tradeoff affects system boards only; all add-in card designs must assume 33 MHz operation.

In 3.3V signaling environments, T_{prop} is measured as shown in Figure 4-12. It begins at the time the signal at the output buffer would have crossed the threshold point of V_{trise} or V_{tfall} had the output been driving the specified $T_{val\ load}$. The end of T_{prop} for any particular input is determined by one of the following two measurement methods. The method that produces the longer value for T_{prop} must be used.

Method 1: The end of T_{prop} is the time when the signal at the input crosses V_{test} for the last time in Figure 4-12 a and d.

Method 2: Construct a line with a slope equal to the Input Signal Edge Rate shown in Table 4-7 and crossing through the point where the signal at the input crosses V_{ih} (high going) or V_{il} (low going) for the last time. The end of T_{prop} is the time when the constructed line crosses V_{test} in Figure 4-12 b, c, e, and f.

IMPLEMENTATION NOTE

Determining the End of T_{prop}

The end of T_{prop} is always determined by the calculation method that produces the longer value of T_{prop}. The shape of the waveform at the input buffer will determine which measurement method will produce the longer value.

When the signal rises or falls from V_{test} to the last crossing of V_{ih} (high going) or V_{il} (low going) *faster* than the Input Signal Edge Rate shown in Table 4-7 (Table 7-5 for 66 MHz operation), Method 1 will produce the longer value as shown in Figure 4-12 a and d.

When the signal rises or falls from V_{test} to the last crossing of V_{ih} (high going) or V_{il} (low going) *slower* than the Input Signal Edge Rate, Method 2 will produce the longer value. In other words, if the signal plateaus or rises (high going) or falls (low going) slowly after crossing V_{test}, or rings back across V_{ih} (high going) or V_{il} (low going) significantly after crossing V_{test}, Method 2 will produce the longer value as shown in Figure 4-12 b, c, e, and f.

Refer to Table 4-3 and Table 4-7 for the values of parameters in Figure 4-12 for 33-MHz mode and for 66-MHz operation, defined in Chapter 7.

For a given driver location, the worst case that must be considered when determining the value of T_{prop} is when the signal settles at all other devices on the bus. The value of T_{prop} is not affected by the time the signal settles at the driver output, since devices are not permitted to drive and receive a signal at the same time. Refer to Section 3.10, item 9 for additional details.

In many system layouts, correct PCI signal propagation relies on diodes embedded in PCI components to limit reflections and successfully meet T_{prop}. In configurations where unterminated trace ends propagate a significant distance from a PCI component (e.g., a section of unpopulated add-in card connectors), it may be necessary to add active (e.g., diode) termination at the unloaded end of the bus in order to insure adequate signal quality. Note that since the signaling protocol depends on the initial reflection, passive termination does not work.

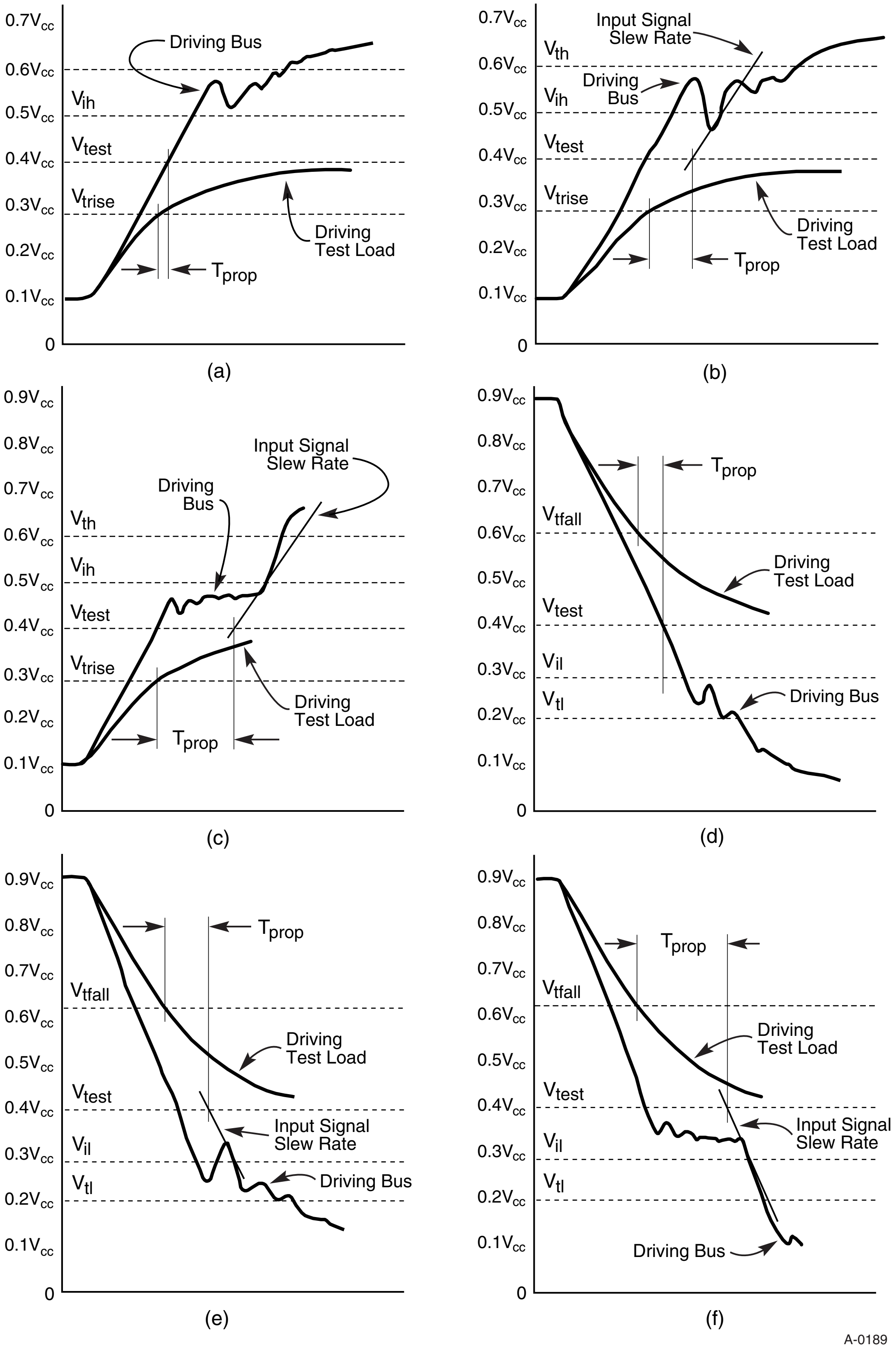


Figure 4-12: Measurement of T_{prop}, 3.3 Volt Signaling

4.3.6. Physical Requirements

4.3.6.1. Routing and Layout Recommendations for Four-Layer System Boards

The power pins have been arranged on the connector to facilitate layouts on four layer system boards. A "split power plane" is permitted - creating a 3.3V island in the 5V plane, which connects all the 3.3V PCI connector pins and may optionally have a power distribution "finger" reaching to the power supply connector. Although this is a standard technique, routing high speed signals directly over this plane split can cause signal integrity problems. The split in the plane disrupts the AC return path for the signal creating an impedance discontinuity.

A recommended solution is to arrange the signal level layouts so that no high speed signal (e.g., 33 MHz) is referenced to both planes. Signal traces should either remain entirely over the 3.3V plane or entirely over the 5V plane. Signals that must cross from one domain to the other should be routed on the opposite side of the system board so that they are referenced to the ground plane which is not split. If this is not possible, and signals must be routed over the plane split, the two planes should be capacitively tied together (5V plane decoupled directly to 3.3V plane) with 0.01 μF high-speed capacitors for each four signals crossing the split and the capacitor should be placed not more than 0.25 inches from the point the signals cross the split.

4.3.6.2. System Board Impedance

There is no bare board impedance specification for system boards. The system designer has two primary constraints in which to work:

- ☐ The length and signal velocity must allow a full round trip time on the bus within the specified propagation delay of 10 ns. (Refer to Section 4.3.5.)
- ☐ The loaded impedance seen at any drive point on the network must be such that a PCI output device (as specified by its V/I curve) can meet input device specifications with a single reflection of the signal. This includes loads presented by add-in cards.

Operating frequency may be traded off for additional round trips on the bus to build configurations that might not comply with the two constraints mentioned above. This option is neither recommended nor specifically precluded.

4.3.7. Connector Pin Assignments

The PCI connector contains all the signals defined for PCI components, plus two pins that are related to the connector only. These pins, PRSNT1# and PRSNT2#, are described in Section 4.4.1. System boards must decouple both of these pins individually to ground with 0.01 μF high-speed capacitors, because one or both of the pins also provide an AC return path. These pins may not be bused or otherwise connected to each other on the system board. Further use of these pins on the system board is optional. If the system board design accesses these pins to obtain add-in card information, each pin must have an appropriate pull-up resistor (of approximately 5 kΩ) on the system board. The connector pin assignments are shown in Table 4-11. Pins labeled "Reserved" must be left unconnected on all connectors.

Pin 38B is a special pin that has logical significance in PCI-X capable slots. In PCI-X slots, pin 38B must be handled as indicated in the PCIXCAP Connection section of the *PCI-X Addendum to the PCI Local Bus Specification*. For all other PCI connectors, this pin must be treated in all respects as a standard ground pin; i.e., the connector pin must be connected to the ground plane.

Pin 49B is a special purpose pin that has logical significance in 66-MHz-capable slots, and, in such slots, it must be separately bused, pulled up, and decoupled as described in Section 7.7.7. For all other PCI connectors, this pin must be treated in all respects as a standard ground pin; i.e., the connector pin must be connected to the ground plane.

Table 4-11: PCI Connector Pinout

	3.3V System Environment		
Pin	**Side B**	**Side A**	**Comments**
1	-12V	TRST#	32-bit connector start
2	TCK	+12V	
3	Ground	TMS	
4	TDO	TDI	
5	+5V	+5V	
6	+5V	INTA#	
7	INTB#	INTC#	
8	INTD#	+5V	
9	PRSNT1#	Reserved	
10	Reserved	+3.3V (I/O)	
11	PRSNT2#	Reserved	
12	CONNECTOR KEY		3.3 volt key
13	CONNECTOR KEY		3.3 volt key
14	Reserved	3.3Vaux	
15	Ground	RST#	
16	CLK	+3.3V (I/O)	
17	Ground	GNT#	
18	REQ#	Ground	
19	+3.3V (I/O)	PME#	
20	AD[31]	AD[30]	
21	AD[29]	+3.3V	
22	Ground	AD[28]	
23	AD[27]	AD[26]	
24	AD[25]	Ground	
25	+3.3V	AD[24]	
26	C/BE[3]#	IDSEL	
27	AD[23]	+3.3V	
28	Ground	AD[22]	
29	AD[21]	AD[20]	
30	AD[19]	Ground	
31	+3.3V	AD[18]	
32	AD[17]	AD[16]	
33	C/BE[2]#	+3.3V	
34	Ground	FRAME#	
35	IRDY#	Ground	
36	+3.3V	TRDY#	
37	DEVSEL#	Ground	
38	PCIXCAP	STOP#	
39	LOCK#	+3.3V	
40	PERR#	SMBCLK	
41	+3.3V	SMBDAT	
42	SERR#	Ground	

	3.3V System Environment		
Pin	**Side B**	**Side A**	**Comments**
43	+3.3V	PAR	
44	C/BE[1]#	AD[15]	
45	AD[14]	+3.3V	
46	Ground	AD[13]	
47	AD[12]	AD[11]	
48	AD[10]	Ground	
49	M66EN	AD[09]	66 MHz/gnd
50	Ground	Ground	
51	Ground	Ground	
52	AD[08]	C/BE[0]#	
53	AD[07]	+3.3V	
54	+3.3V	AD[06]	
55	AD[05]	AD[04]	
56	AD[03]	Ground	
57	Ground	AD[02]	
58	AD[01]	AD[00]	
59	+3.3V (I/O)	+3.3V (I/O)	
60	ACK64#	REQ64#	
61	+5V	+5V	
62	+5V	+5V	32-bit connector end
	CONNECTOR KEY		64-bit spacer
	CONNECTOR KEY		64-bit spacer
63	Reserved	Ground	64-bit connector start
64	Ground	C/BE[7]#	
65	C/BE[6]#	C/BE[5]#	
66	C/BE[4]#	+3.3V (I/O)	
67	Ground	PAR64	
68	AD[63]	AD[62]	
69	AD[61]	Ground	
70	+3.3V (I/O)	AD[60]	
71	AD[59]	AD[58]	
72	AD[57]	Ground	
73	Ground	AD[56]	
74	AD[55]	AD[54]	
75	AD[53]	+3.3V (I/O)	
76	Ground	AD[52]	
77	AD[51]	AD[50]	
78	AD[49]	Ground	
79	+3.3V (I/O)	AD[48]	
80	AD[47]	AD[46]	
81	AD[45]	Ground	
82	Ground	AD[44]	

	3.3V System Environment		
Pin	**Side B**	**Side A**	**Comments**
83	AD[43]	AD[42]	
84	AD[41]	+3.3V (I/O)	
85	Ground	AD[40]	
86	AD[39]	AD[38]	
87	AD[37]	Ground	
88	+3.3V (I/O)	AD[36]	
89	AD[35]	AD[34]	
90	AD[33]	Ground	
91	Ground	AD[32]	
92	Reserved	Reserved	
93	Reserved	Ground	
94	Ground	Reserved	64-bit connector end

Pins labeled "+3.3V (I/O)" are special power pins for defining and driving the PCI signaling rail on the Universal add-in card. On the system board, these pins are connected to the main +3.3V plane.

Refer to Section 3.8.1 for special requirements for the connection of REQ64# on 32-bit-only slot connectors.

4.4. Add-in Card Specification

4.4.1. Add-in Card Pin Assignment

The PCI connector contains all the signals defined for PCI components, plus two pins that are related to the connector only. These are PRSNT1# and PRSNT2#. They are used for two purposes: indicating that an add-in card is physically present in the slot and providing information about the total power requirements of the add-in card. Table 4-12 defines the required setting of the PRSNT# pins for add-in cards.

Table 4-12: Present Signal Definitions

PRSNT1#	PRSNT2#	Add-in Card Configuration
Open	Open	No add-in card present
Ground	Open	Add-in card present, 25 W maximum
Open	Ground	Add-in card present, 15 W maximum
Ground	Ground	Add-in card present, 7.5 W maximum

In providing a power level indication, the add-in card must indicate total maximum power consumption for the add-in card, including all supply voltages. The add-in cards may optionally draw all this power from either the 3.3V or 5V power rail. Furthermore, if the add-in card is configurable (e.g., sockets for memory expansion, etc.), the pin strapping must indicate the total power consumed by a fully configured add-in card, which may be more than that consumed in its shipping configuration.

Add-in cards that do not implement JTAG Boundary Scan are required to connect TDI and TDO (pins 4A and 4B) so the scan chain is not broken.

Pin 38B is a special pin that has logical significance in PCI-X capable add-in cards. In PCI-X slots, pin 38B must be handled as indicated in the PCIXCAP Connection section of the *PCI-X Addendum to the PCI Local Bus Specification*. For all other PCI add-in cards, this pin must be treated in all respects as a standard ground pin; i.e., the edge finger must be plated and connected to the ground plane of the add-in card.

Pin 49B is a special purpose pin that has logical significance in 66-MHz-capable add-in cards, and, in such add-in cards, it must be connected and decoupled as described in Section 7.8. For all other add-in cards, this pin must be treated in all respects as a standard ground pin; i.e., the edge finger must be plated and connected to the ground plane of the add-in card.

Table 4-13: PCI Add-in Card Pinout

	Universal Add-in Card		**3.3V Add-in Card**		
Pin	**Side B**	**Side A**	**Side B**	**Side A**	**Comments**
1	-12V	TRST#	-12V	TRST#	32-bit start
2	TCK	+12V	TCK	+12V	
3	Ground	TMS	Ground	TMS	
4	TDO	TDI	TDO	TDI	
5	+5V	+5V	+5V	+5V	
6	+5V	INTA#	+5V	INTA#	
7	INTB#	INTC#	INTB#	INTC#	
8	INTD#	+5V	INTD#	+5V	
9	PRSNT1#	Reserved	PRSNT1#	Reserved	
10	Reserved	+$V_{I/O}$	Reserved	+3.3V	
11	PRSNT2#	Reserved	PRSNT2#	Reserved	
12	KEYWAY		KEYWAY		3.3V key
13	KEYWAY		KEYWAY		3.3V key
14	Reserved	3.3Vaux	Reserved	3.3Vaux	
15	Ground	RST#	Ground	RST#	
16	CLK	+$V_{I/O}$	CLK	+3.3V	
17	Ground	GNT#	Ground	GNT#	
18	REQ#	Ground	REQ#	Ground	
19	+VI/O	PME#	+3.3V	PME#	
20	AD[31]	AD[30]	AD[31]	AD[30]	
21	AD[29]	+3.3V	AD[29]	+3.3V	
22	Ground	AD[28]	Ground	AD[28]	
23	AD[27]	AD[26]	AD[27]	AD[26]	
24	AD[25]	Ground	AD[25]	Ground	
25	+3.3V	AD[24]	+3.3V	AD[24]	
26	C/BE[3]#	IDSEL	C/BE[3]#	IDSEL	
27	AD[23]	+3.3V	AD[23]	+3.3V	
28	Ground	AD[22]	Ground	AD[22]	
29	AD[21]	AD[20]	AD[21]	AD[20]	
30	AD[19]	Ground	AD[19]	Ground	
31	+3.3V	AD[18]	+3.3V	AD[18]	
32	AD[17]	AD[16]	AD[17]	AD[16]	
33	C/BE[2]#	+3.3V	C/BE[2]#	+3.3V	
34	Ground	FRAME#	Ground	FRAME#	
35	IRDY#	Ground	IRDY#	Ground	
36	+3.3V	TRDY#	+3.3V	TRDY#	
37	DEVSEL#	Ground	DEVSEL#	Ground	
38	PCIXCAP	STOP#	PCIXCAP	STOP#	
39	LOCK#	+3.3V	LOCK#	+3.3V	
40	PERR#	SMBCLK	PERR#	SMBCLK	
41	+3.3V	SMBDAT	+3.3V	SMBDAT	

	Universal Add-in Card		**3.3V Add-in Card**		
Pin	**Side B**	**Side A**	**Side B**	**Side A**	**Comments**
42	SERR#	Ground	SERR#	Ground	
43	+3.3V	PAR	+3.3V	PAR	
44	C/BE[1]#	AD[15]	C/BE[1]#	AD[15]	
45	AD[14]	+3.3V	AD[14]	+3.3V	
46	Ground	AD[13]	Ground	AD[13]	
47	AD[12]	AD[11]	AD[12]	AD[11]	
48	AD[10]	Ground	AD[10]	Ground	
49	M66EN	AD[09]	M66EN	AD[09]	66 MHz/gnd
50	KEYWAY		Ground	Ground	5V key
51	KEYWAY		Ground	Ground	5V key
52	AD[08]	C/BE[0]#	AD[08]	C/BE[0]#	
53	AD[07]	+3.3V	AD[07]	+3.3V	
54	+3.3V	AD[06]	+3.3V	AD[06]	
55	AD[05]	AD[04]	AD[05]	AD[04]	
56	AD[03]	Ground	AD[03]	Ground	
57	Ground	AD[02]	Ground	AD[02]	
58	AD[01]	AD[00]	AD[01]	AD[00]	
59	$+V_{I/O}$	$+V_{I/O}$	+3.3V	+3.3V	
60	ACK64#	REQ64#	ACK64#	REQ64#	
61	+5V	+5V	+5V	+5V	
62	+5V	+5V	+5V	+5V	32-bit end
	KEYWAY		KEYWAY		64-bit spacer
	KEYWAY		KEYWAY		64-bit spacer
63	Reserved	Ground	Reserved	Ground	64-bit start
64	Ground	C/BE[7]#	Ground	C/BE[7]#	
65	C/BE[6]#	C/BE[5]#	C/BE[6]#	C/BE[5]#	
66	C/BE[4]#	+VI/O	C/BE[4]#	+3.3V	
67	Ground	PAR64	Ground	PAR64	
68	AD[63]	AD[62]	AD[63]	AD[62]	
69	AD[61]	Ground	AD[61]	Ground	
70	$+V_{I/O}$	AD[60]	+3.3V	AD[60]	
71	AD[59]	AD[58]	AD[59]	AD[58]	
72	AD[57]	Ground	AD[57]	Ground	
73	Ground	AD[56]	Ground	AD[56]	
74	AD[55]	AD[54]	AD[55]	AD[54]	
75	AD[53]	+VI/O	AD[53]	+3.3V	
76	Ground	AD[52]	Ground	AD[52]	
77	AD[51]	AD[50]	AD[51]	AD[50]	
78	AD[49]	Ground	AD[49]	Ground	
79	$+V_{I/O}$	AD[48]	+3.3V	AD[48]	
80	AD[47]	AD[46]	AD[47]	AD[46]	
81	AD[45]	Ground	AD[45]	Ground	

	Universal Add-in Card		3.3V Add-in Card		
Pin	**Side B**	**Side A**	**Side B**	**Side A**	**Comments**
82	Ground	AD[44]	Ground	AD[44]	
83	AD[43]	AD[42]	AD[43]	AD[42]	
84	AD[41]	$+V_{I/O}$	AD[41]	+3.3V	
85	Ground	AD[40]	Ground	AD[40]	
86	AD[39]	AD[38]	AD[39]	AD[38]	
87	AD[37]	Ground	AD[37]	Ground	
88	$+V_{I/O}$	AD[36]	+3.3V	AD[36]	
89	AD[35]	AD[34]	AD[35]	AD[34]	
90	AD[33]	Ground	AD[33]	Ground	
91	Ground	AD[32]	Ground	AD[32]	
92	Reserved	Reserved	Reserved	Reserved	
93	Reserved	Ground	Reserved	Ground	
94	Ground	Reserved	Ground	Reserved	64-bit end

Table 4-14: Pin Summary–32-bit Add-in Card

Pin Type	Universal Add-in Card	3.3V Add-in Card
Ground	18 (Note)	22 (Note)
+5 V	8	8
+3.3 V	12	17
I/O pwr	5	0
Reserv'd	4	4

Note: If the PCIXCAP and M66EN pins are implemented, the number of ground pins for a Universal add-in card is 16 and the number of ground pins for a 3.3V add-in card is 20.

Table 4-15: Pin Summary–64-bit Add-in Card (incremental pins)

Pin Type	Universal Add-in Card	3.3V Add-in Card
Ground	16	16
+5 V	0	0
+3.3 V	0	6
I/O pwr	6	0
Reserv'd	5	5

Pins labeled "+$V_{I/O}$" are special power pins for defining and driving the PCI signaling rail on the Universal add-in card. On this add-in card, the PCI component's I/O buffers must be powered from these special power pins only[35]—not from the other +3.3V or +5V power pins.

4.4.2. Power Requirements

4.4.2.1. Decoupling

All +5V, +3.3V, and +$V_{I/O}$ pins must be decoupled to ground as specified below, regardless of whether those voltages are used by the add-in card or not. (High-frequency return currents from signal pins close to these power supply pins depend upon these pins to be AC grounds to avoid signal integrity problems.) All +5V, +3.3V, +$V_{I/O}$, and ground pins that connect to planes on the add-in card must connect to the plane via with a maximum trace length of 0.25 inches and a minimum trace width of 0.02 inches.

Power supply voltages that connect to full add-in card power planes must have adequate high frequency decoupled to ground (equivalent to at least 0.01 μF at each power pin) or require the addition of discrete capacitors to fill this requirement. Smaller power planes like the +$V_{I/O}$ plane will not have adequate high frequency decoupling and require a minimum of one high-frequency 0.047 μF decoupling capacitor at each power supply pin. Power supply voltages that are not used on the add-in card must be decoupled to ground as follows:

- ☐ Each pin must be connected to a minimum of one high-frequency 0.01 μF decoupling capacitor.
- ☐ The connection between each pin and the decoupling capacitor must have a maximum trace length of 0.25 inches and a minimum trace width of 0.02 inches.
- ☐ Connector pins are permitted to share the same decoupling capacitor provided that requirements 1 and 2 are met.

All edge fingers must be copper plated and all ground edge fingers must be connected to the ground plane on the add-in card.

Maximum decoupling capacitance values are specified in Table 4-16. An add-in card must not exceed these limits if the add-in card is designed to be hot-inserted into a system, as described in *PCI Hot-Plug Specification, Revision 1.1*. Add-in cards that do not support hot-insertion are not required to support these limits.

---

[35] When “5V tolerant parts” are used on the Universal add-in card, its I/O buffers may optionally be connected to the 3.3V rail rather than the "I/O" designated power pins; but high clamp diodes may still be connected to the "I/O" designated power pins. (Refer to the last paragraph of Section 4.2.1.2 - "Clamping directly to the 3.3V rail with a simple diode must never be used in the 5V signaling environment.") Since the effective operation of these high clamp diodes may be critical to both signal quality and device reliability, the designer must provide enough "I/O" designated power pins on a component to handle the current spikes associated with the 5V maximum AC waveforms (Section 4.2.1).

Table 4-16: Maximum Slot Decoupling Capacitance and Voltage Slew Rate Limits

Supply Voltage	Maximum Operating Current	Maximum Add-in Card Decoupling Capacitance	Minimum Supply Voltage Slew Rate	Maximum Supply Voltage Slew Rate
+5 V	5 A (Note 1)	3000 µF	25 V/s	3300 V/s
+3.3 V	7.6 A (Note 1)	3000 µF	16.5 V/s	3300 V/s
+12 V	500 mA	300 µF	60 V/s	33000 V/s
-12 V	100 mA	150 µF	60 V/s	66000 V/s
3.3 Vaux	375 mA (Note 2)	150 µF	16.5 V/s	3300 V/s
+$V_{I/O}$ (Note 3)	1.5 A	150 µF	16.5 V/s	13000 V/s

Notes:

1. Value shown is for the case when all power is supplied from a single supply voltage, and the connection of PRSNT1# and PRSNT2# indicate the add-in card consumes maximum power. See Section 4.4.2.2 for the total maximum power consumption allowed for the add-in card.
2. This parameter is specified by *PCI Bus Power Management Interface Specification, Rev. 1.1* and is included here for reference purposes only.
3. These limit applies only when +$V_{I/O}$ is not connected to +5V or +3.3V, as described in PCI-X 2.0. When +$V_{I/O}$ is connected to +5V or +3.3V, the limits specified for +5V and +3.3V apply to the combination.

4.4.2.2. Power Consumption

The maximum power allowed for any add-in card is 25 watts, and represents the total power drawn from all power rails provided at the connector (+3.3V, +5V, +$V_{I/O}$, +12V, -12V, +3.3V_{aux}). The add-in card may optionally draw all this power from either the +3.3V or +5V rail.

Power supply slew rates are specified in Table 4-16. Add-in cards are required to operate after the power is applied with any slew rate within these limit, unless that add-in card is designed never to be hot-inserted into a system, as described in *PCI Hot-Plug Specification, Revision 1.1*.

It is anticipated that many systems will not provide a full 25 watts per connector for each power rail, because most add-in cards will typically draw much less than this amount. For this reason, it is recommended that add-in cards that consume more than 10 watts power up in and reset to a reduced-power state that consumes 10 watts or less. While in this state, the add-in card must provide full access to its PCI Configuration Space and must perform required bootstrap functions, such as basic text mode on a video add-in card. All other add-in card functions can be suspended if necessary. This power saving state can be achieved in a variety of ways. For example:

- ☐ Clock rates on the add-in card can be reduced, which reduces performance but does not limit functionality.
- ☐ Power planes to non-critical parts could be shut off with a FET, which could limit functional capability.

After the driver for the add-in card has been initialized, it may place the add-in card into a fully powered, full function/performance state using a device dependent mechanism of choice (probably register based). In advanced power managed systems, the device driver may be required to report the target power consumption before fully enabling the add-in card in order to allow the system to determine if it has a sufficient power budget for all add-in cards in the current configuration.

Add-in cards must never source any power back to the system board, except in the case where an add-in card has been specifically designed to provide a given system's power. In some cases, add-in cards capable of 3.3V PCI signaling have multiple mechanisms that indirectly source power back to the system and will violate this requirement if not properly controlled. For example, add-in cards containing components with bus clamps to the 3.3V rail may create a "charge pump" which directs excess bus switching energy back into the system board. Alternately, I/O output buffers operating on the 3.3V rail, but used in a 5V signaling environment, may bleed the excess charge off the bus and into the 3.3V power net when they drive the bus "high" after it was previously driven to the 5V rail. Unintentional power sourcing by any such mechanism must be managed by proper decoupling and sufficient local load on the supply (bleed resistor or otherwise) to dissipate any power "generated" on the add-in card. This requirement does not apply to noise generated on the power rail as long as the net DC current accumulated over any two clock periods is zero.

4.4.3. Physical Requirements

4.4.3.1. Trace Length Limits

Trace lengths from the top of the add-in card's edge connector to the PCI device are as follows:

- ☐ The maximum trace lengths for all 32-bit interface signals are limited to 1.5 inches for 32-bit and 64-bit add-in cards. This includes all signal groups (refer to Section 2.2.) except those listed as "System Pins," "Interrupt Pins," "SMBus," and "JTAG Pins." The trace length of REQ64# and ACK64# (on the 32 bit connector) are limited to 1.5 inches also.
- ☐ The trace lengths of the additional signals used in the 64-bit extension are limited to 2 inches on all 64-bit add-in cards.
- ☐ The trace length for the PCI CLK signal is 2.5 inches ± 0.1 inches for 32-bit and 64-bit add-in cards and must be routed to only one load.

4.4.3.2. Routing Recommendations for Four-Layer Add-in Cards

The power pins have been arranged on the connector to facilitate layouts on four layer add-in cards. A "split power plane" is permitted, as described in Section 4.3.6.1. Although this is a standard technique, routing high speed signals directly over this plane split can cause signal integrity problems. The split in the plane disrupts the AC return path for the signal creating an impedance discontinuity.

A recommended solution is to arrange the signal level layouts so that no high speed signal (e.g., 33 MHz) is referenced to both planes. Signal traces should either remain entirely over the 3.3V plane or entirely over the 5V plane. Signals that must cross from one domain to the other should be routed on the opposite side of the add-in card so that they are referenced to the ground plane which is not split. If this is not possible, and signals must be routed over the plane split, the two planes should be capacitively tied together (5V plane decoupled directly to 3.3V plane) with 0.01 μF high-speed capacitors for each four signals crossing the split and the capacitor should be placed not more than 0.25 inches from the point the signals cross the split.

4.4.3.3. Impedance

The unloaded characteristic impedance (Z_0) of the shared PCI signal traces on the add-in card shall be controlled to be in the 60 Ω - 100 Ω range if the device input capacitance (C_{in}) exceeds 8 pF. If C_{in} is 8 pF or less, the range for Z_0 is 51 Ω - 100 Ω. The trace velocity must be between 150 ps/inch and 190 ps/inch.

4.4.3.4. Signal Loading

Shared PCI signals must be limited to one load on the add-in card. Violation of add-in card trace length or loading limits will compromise system signal integrity. It is specifically a violation of this specification for add-in cards to:

- ☐ Attach an expansion ROM directly (or via bus transceivers) on any PCI pins.
- ☐ Attach two or more PCI devices on an add-in card, unless they are placed behind a PCI-to-PCI bridge.
- ☐ Attach any logic (other than a single PCI device) that "snoops" PCI pins.
- ☐ Use PCI component sets that place more than one load on each PCI pin; e.g., separate address and data path components.
- ☐ Use a PCI component that has more than 10 pF capacitance per pin.
- ☐ Attach any pull-up resistors or other discrete devices to the PCI signals, unless they are placed behind a PCI-to-PCI bridge.

The SMBus signal group is exempt from this requirement. Refer to Section 8.2.5 for SMBus signal loading requirements.

5

5. Mechanical Specification

5.1. Overview

The PCI add-in card is based on a raw add-in card design (see Figures 5-1 to 5-4) that is easily implemented in existing chassis designs from multiple manufacturers. PCI add-in cards have three basic form factors: standard length, short length, and low profile. The standard length card provides 49 square inches of real estate. The fixed and variable height short length cards were chosen for panel optimization to provide the lowest cost for a function. The fixed and variable height short cards also provide the lowest cost to implement in a system, the lowest energy consumption, and allow the design of smaller systems. The interconnect for the PCI add-in card has been defined for both the 32-bit and 64-bit interfaces.

The basic 32-bit connector contains 120 pins. The logical numbering of pins shows 124 pin identification numbers, but four pins (A12, A13, B12, and B13) are not present and are replaced by the keying location (see Table 4-11). Universal add-in cards, add-in cards built to work in both 3.3V and 5V system signaling environments, have two key slots so that they can plug into the 3.3V keyed system board connectors or the 5V keyed system connectors supported by the prior revisions of this specification. A 64-bit extension, built onto the same connector molding, extends the total number of pins to 184. The 32-bit connector subset defines the system signaling environment. 32-bit add-in cards and 64-bit add-in cards are inter-operable within the system's signaling voltage classes defined by the keying in the 32-bit connector subset. A 32-bit add-in card identifies itself for 32-bit transfers on the 64-bit connector. A 64-bit add-in card in a 32-bit connector must configure for 32-bit transfers.

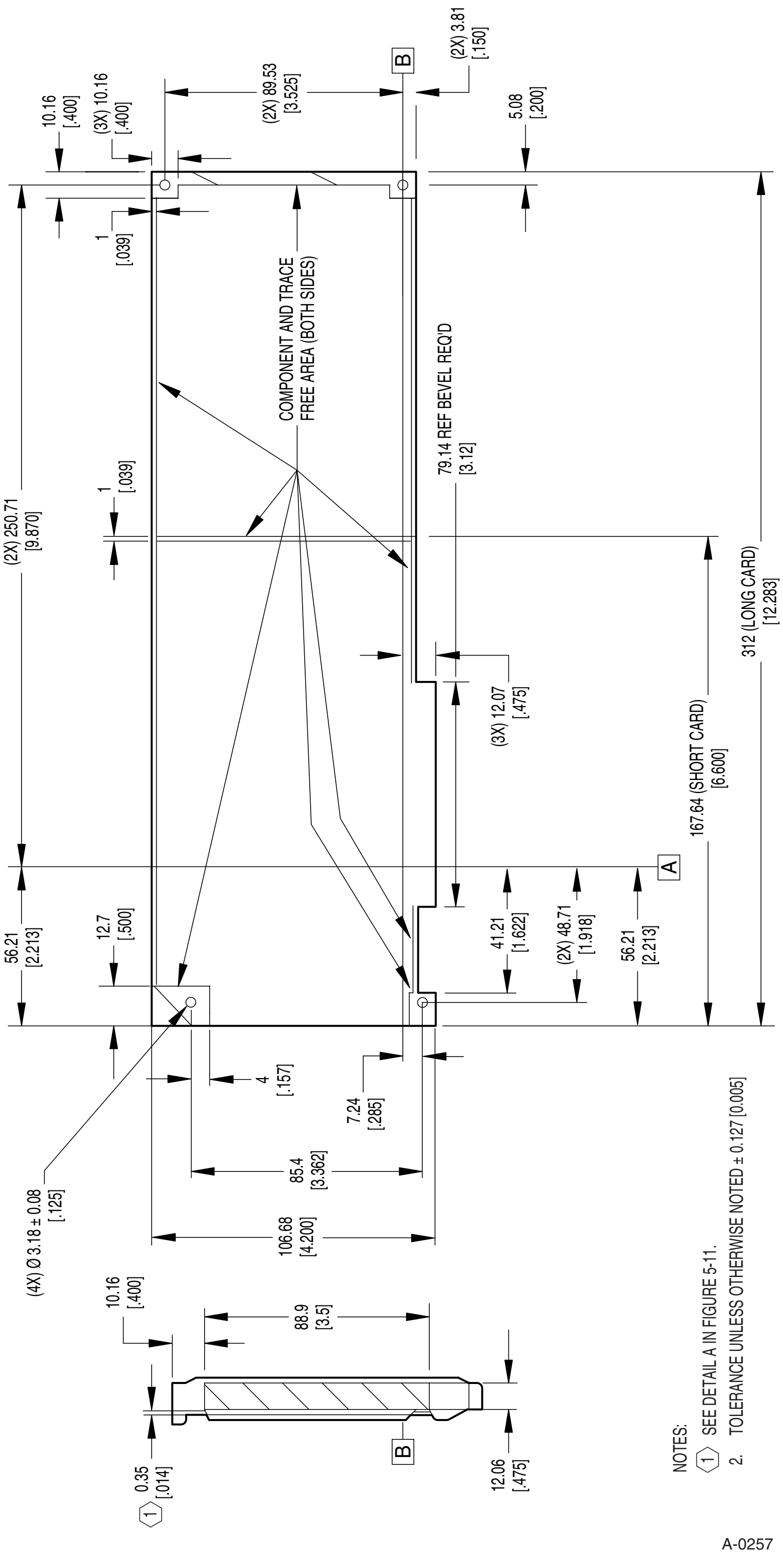
Maximum add-in card power dissipation is encoded on the PRSNT1# and PRSNT2# pins of the add-in card. This hard encoding can be read by system software upon initialization. The system's software can then make a determination whether adequate cooling and supply current is available in that system for reliable operation at start-up and initialization time. Supported power levels and their encoding are defined in Chapter 4.

The PCI add-in card includes a mounting bracket for add-in card location and retention. The backplate is the interface between the add-in card and the system that provides for cable escapement. The retainer fastens to the front edge of the PCI add-in card to provide support via a standard PCI add-in card guide.

5.2. Add-in Card Physical Dimensions and Tolerances

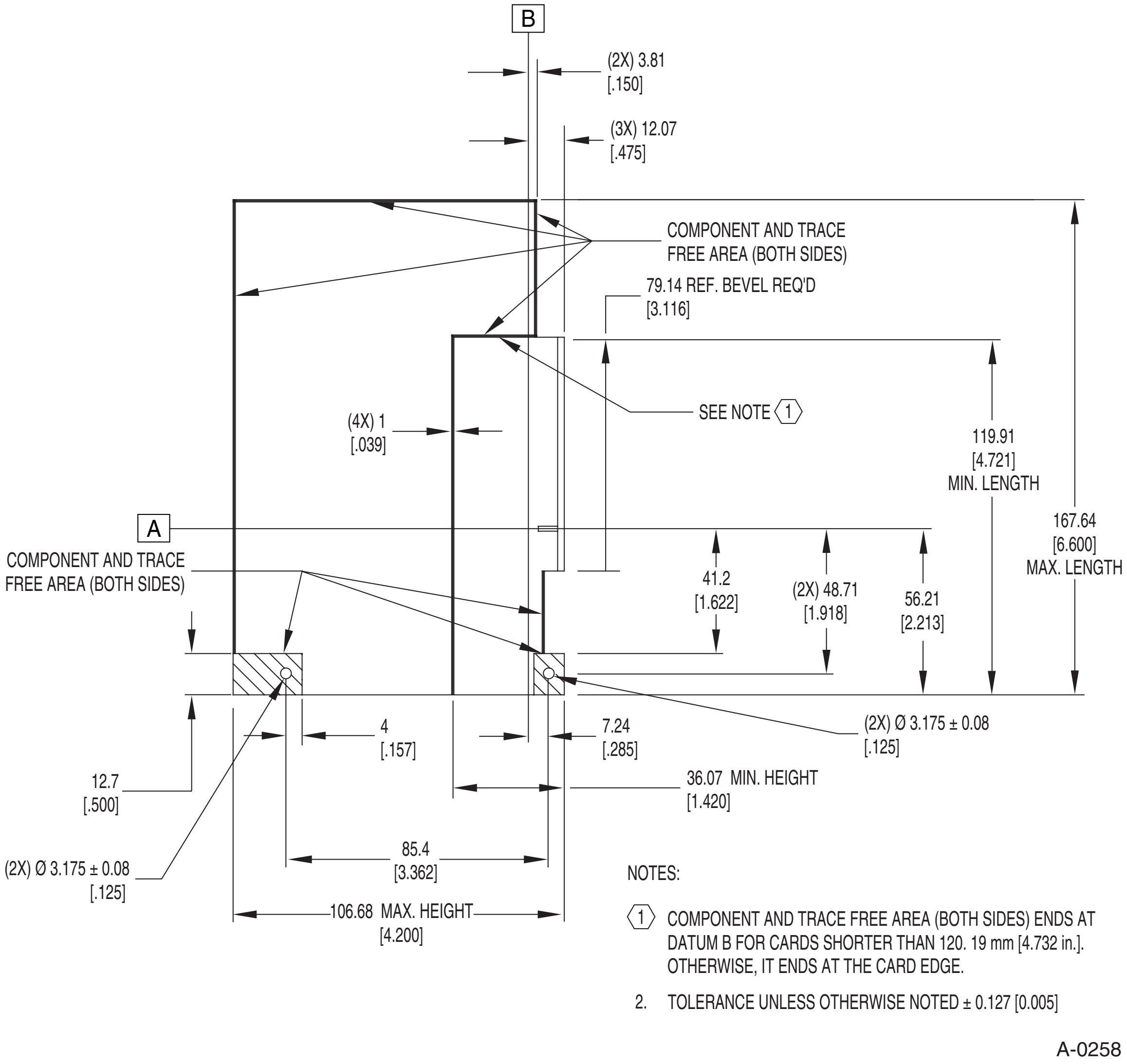
The maximum component height on the primary component side of the PCI add-in card is not to exceed 0.570 inches (14.48 mm). The maximum component height on the back side of the add-in card is not to exceed 0.105 inches (2.67 mm). Datum A on the illustrations is used to locate the add-in card to the system board and to the frame interfaces: the back of the frame and the add-in card guide. Datum A is carried through the locating key on the card edge and the locating key on the connector.

See Figures 5-1 through 5-12 for PCI add-in card physical dimensions. This revision of this specification supports only 3.3V or Universal keyed add-in cards.



NOTES:
1. SEE DETAIL A IN FIGURE 5-11.
2. TOLERANCE UNLESS OTHERWISE NOTED ± 0.127 [0.005]

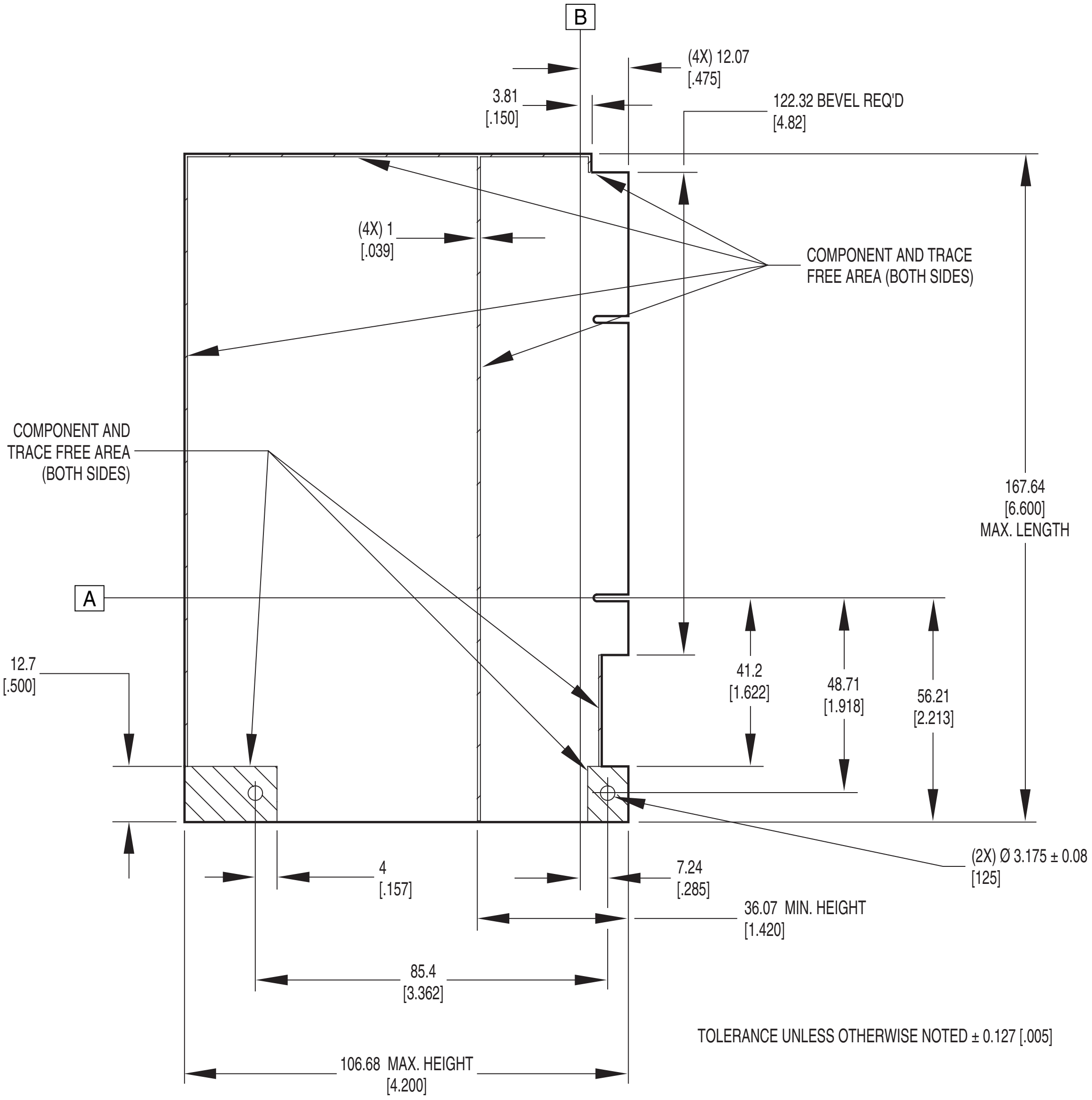
Figure 5-1: PCI Raw Add-in Card (3.3V, 32-bit)



NOTES:

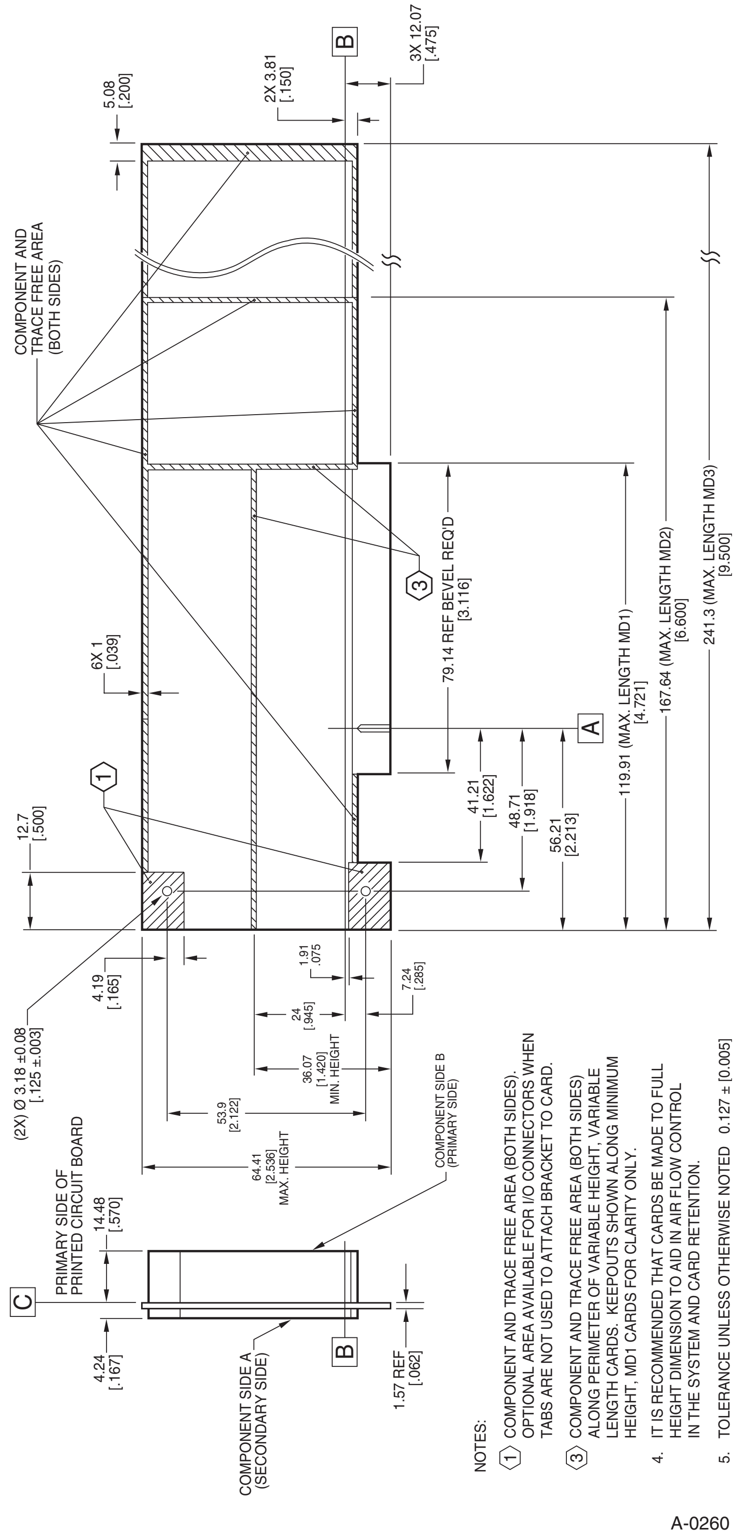
1. COMPONENT AND TRACE FREE AREA (BOTH SIDES) ENDS AT DATUM B FOR CARDS SHORTER THAN 120. 19 mm [4.732 in.]. OTHERWISE, IT ENDS AT THE CARD EDGE.
2. TOLERANCE UNLESS OTHERWISE NOTED ± 0.127 [0.005]

Figure 5-2: PCI Raw Variable Height Short Add-in Card (3.3V, 32-bit)



A-0259

Figure 5-3: PCI Raw Variable Height Short Add-in Card (3.3V, 64-bit)



NOTES:

1. COMPONENT AND TRACE FREE AREA (BOTH SIDES). OPTIONAL AREA AVAILABLE FOR I/O CONNECTORS WHEN TABS ARE NOT USED TO ATTACH BRACKET TO CARD.
3. COMPONENT AND TRACE FREE AREA (BOTH SIDES) ALONG PERIMETER OF VARIABLE HEIGHT, VARIABLE LENGTH CARDS. KEEPOUTS SHOWN ALONG MINIMUM HEIGHT, MD1 CARDS FOR CLARITY ONLY.
4. IT IS RECOMMENDED THAT CARDS BE MADE TO FULL HEIGHT DIMENSION TO AID IN AIR FLOW CONTROL IN THE SYSTEM AND CARD RETENTION.
5. TOLERANCE UNLESS OTHERWISE NOTED 0.127 ± [0.005]

Figure 5-4: PCI Raw Low Profile Add-in Card (3.3V, 32-bit)

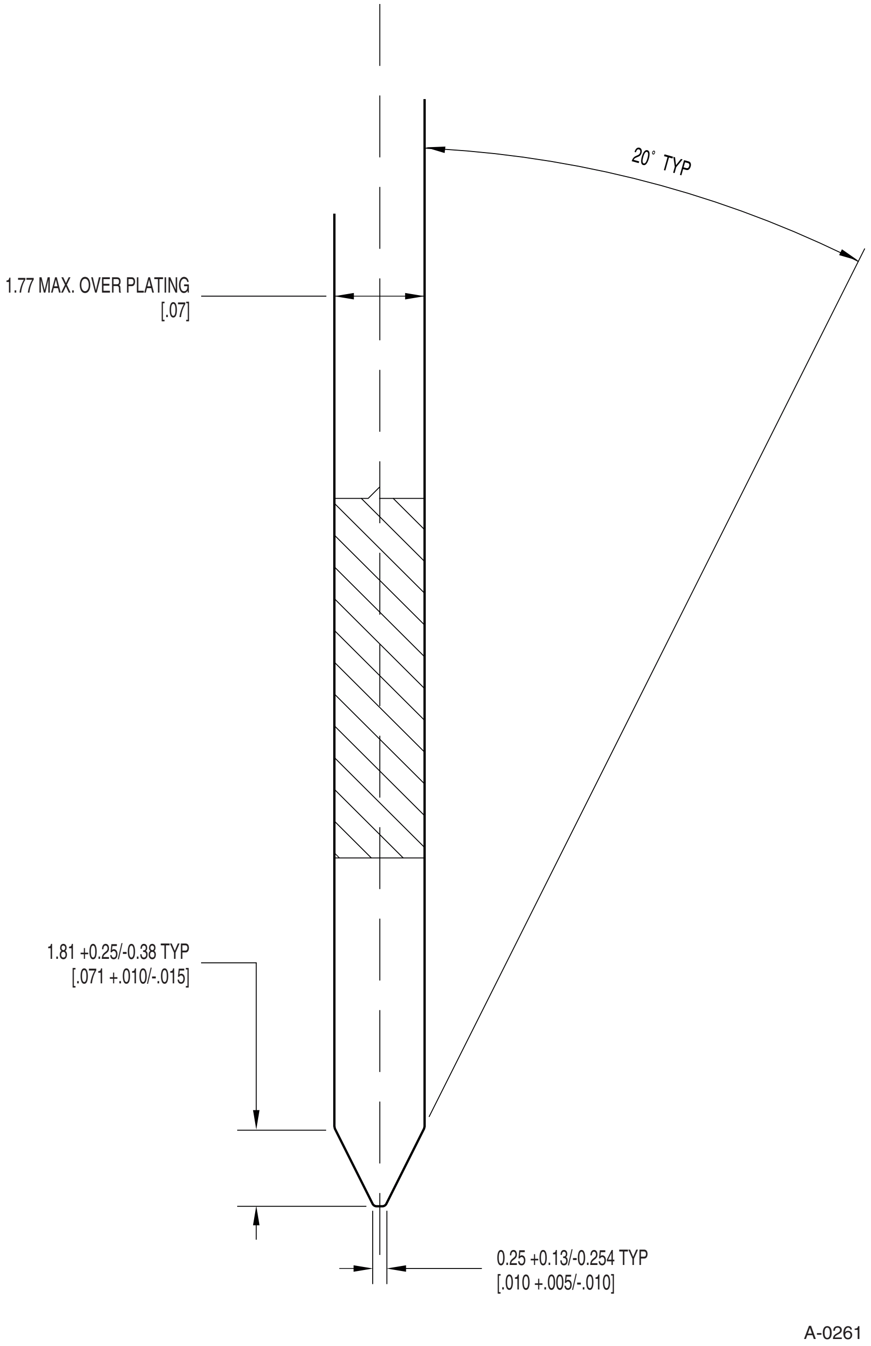
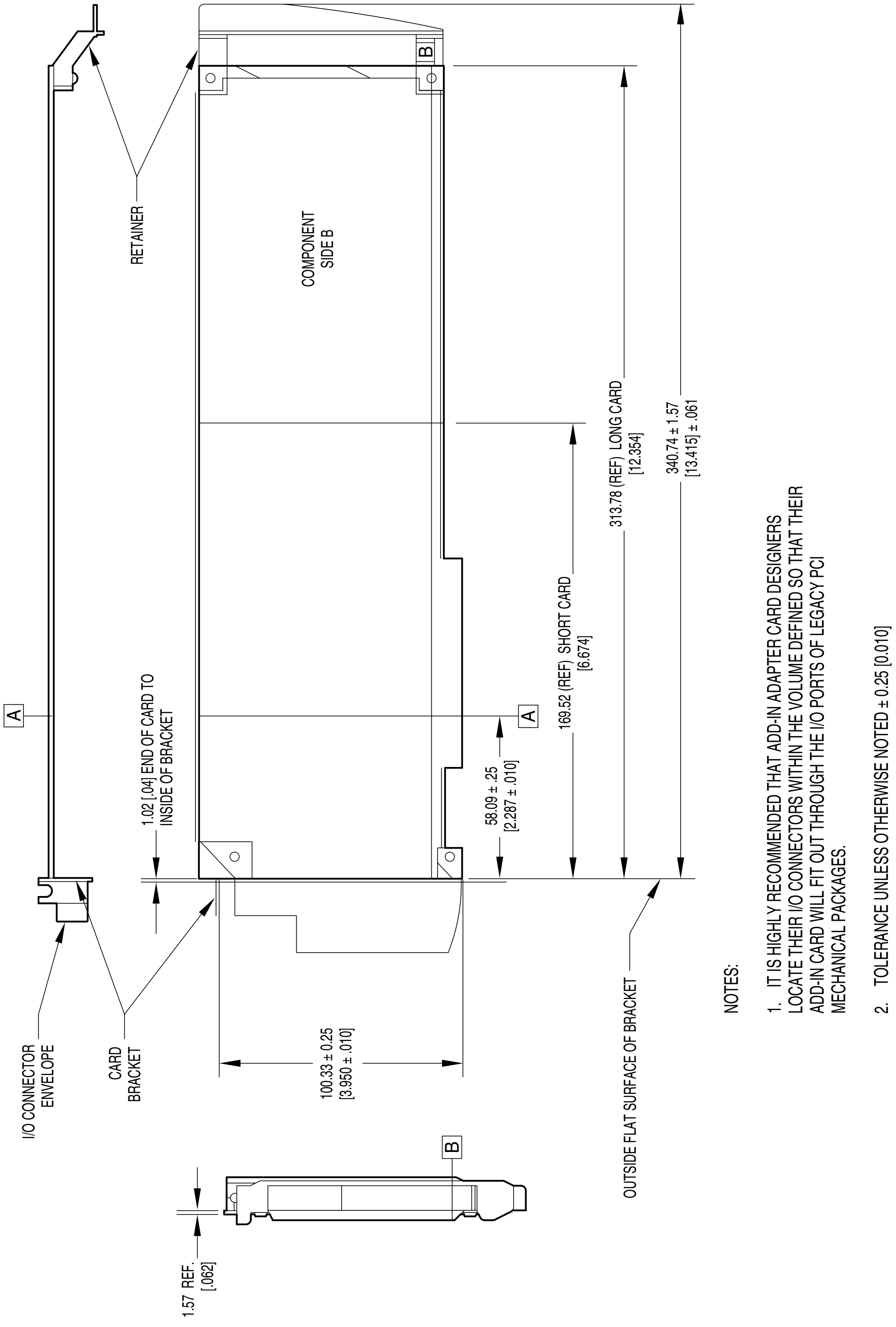


Figure 5-5: PCI Add-in Card Edge Connector Bevel



NOTES:

1. IT IS HIGHLY RECOMMENDED THAT ADD-IN ADAPTER CARD DESIGNERS LOCATE THEIR I/O CONNECTORS WITHIN THE VOLUME DEFINED SO THAT THEIR ADD-IN CARD WILL FIT OUT THROUGH THE I/O PORTS OF LEGACY PCI MECHANICAL PACKAGES.

2. TOLERANCE UNLESS OTHERWISE NOTED ± 0.25 [0.010]

Figure 5-6: PCI Add-in Card Assembly (3.3V)

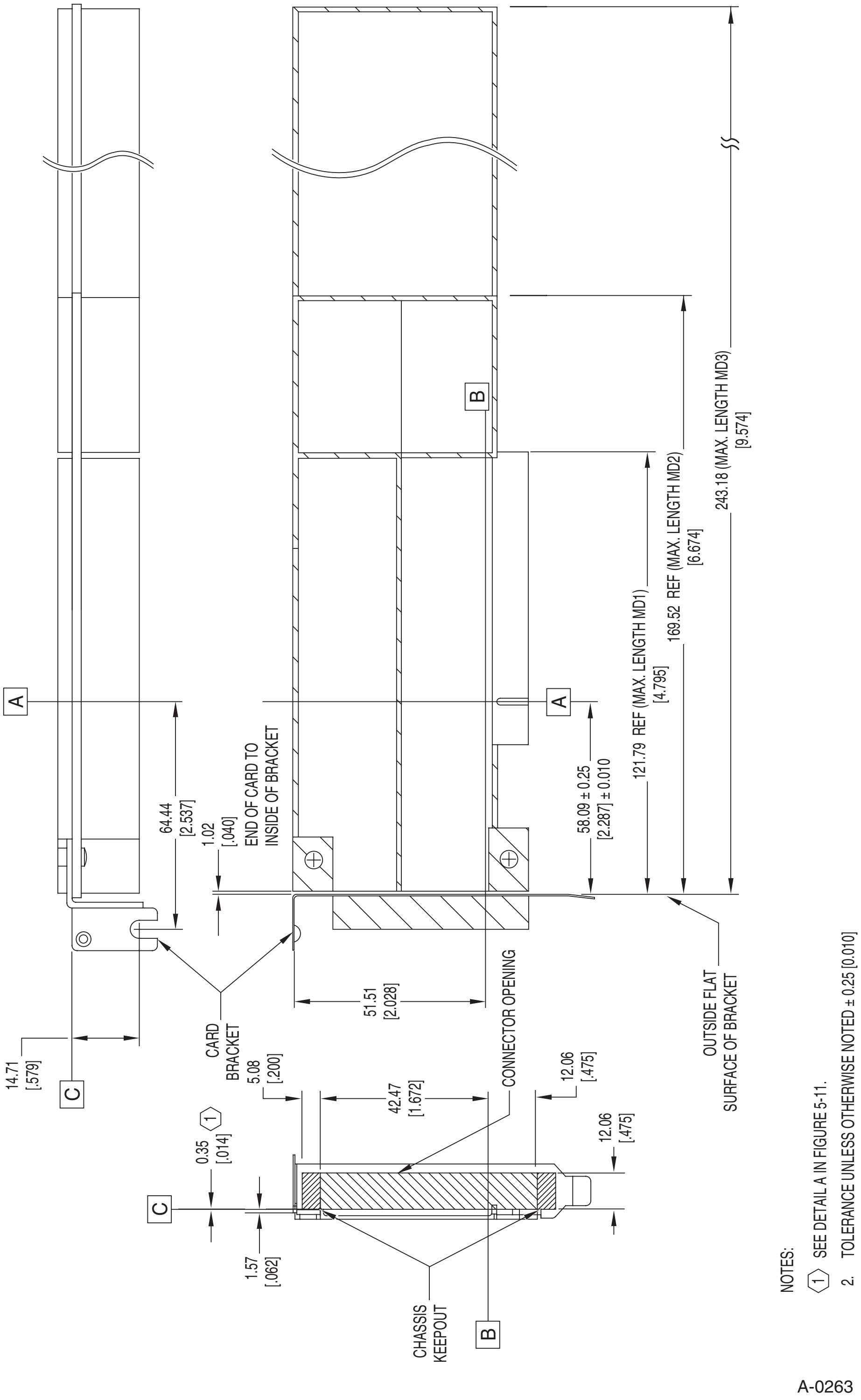


Figure 5-7: Low Profile PCI Add-in Card Assembly (3.3V)

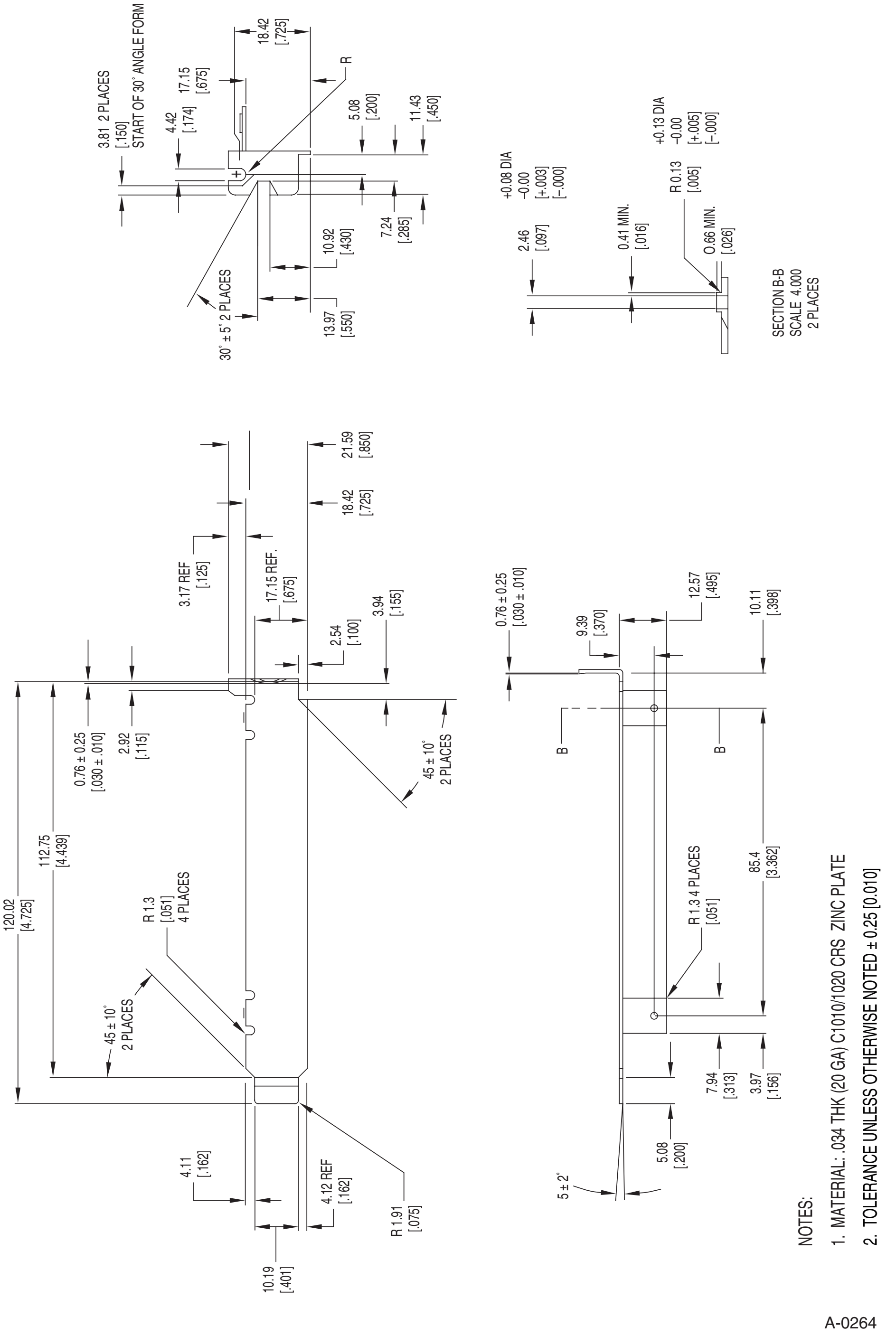
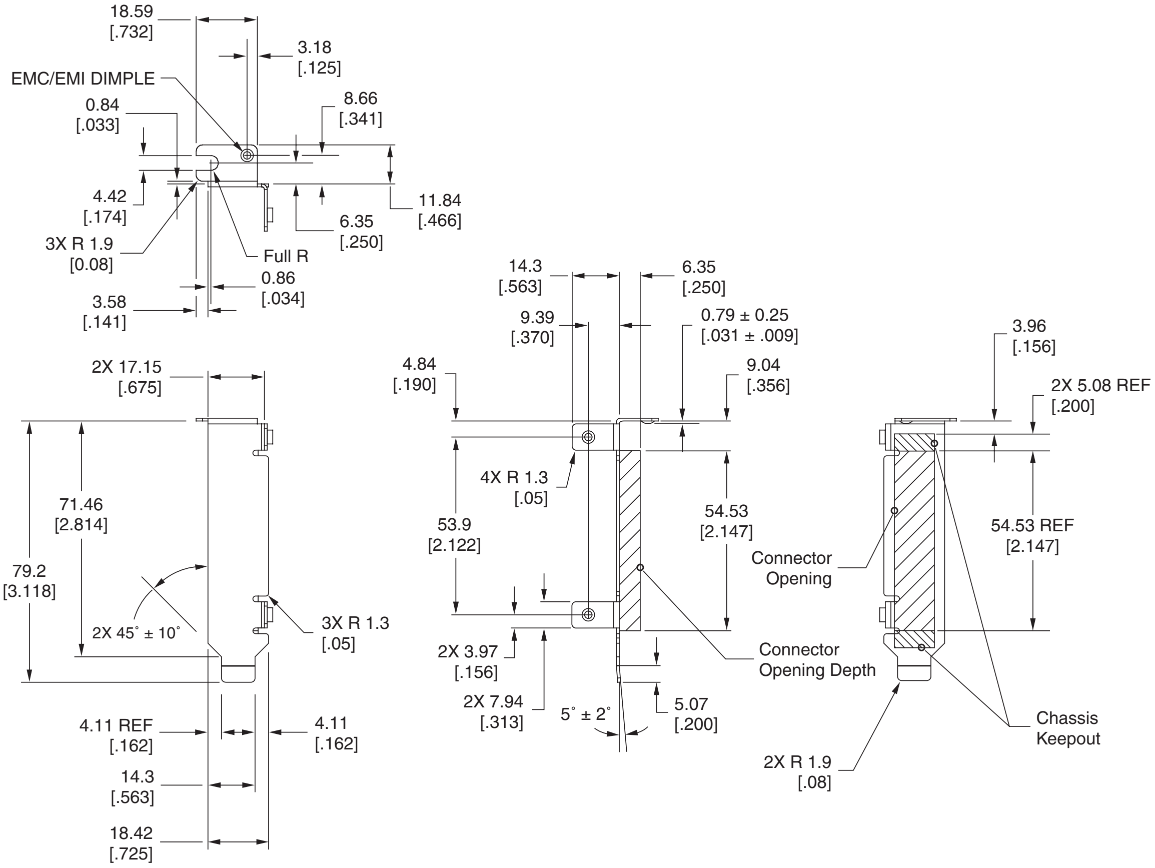


Figure 5-8: PCI Standard Bracket[36, 37]

---

[36] It is highly recommended that add-in card designers implement I/O brackets which mount on the backside of PCI add-in cards as soon as possible to reduce their exposure to causing EMC leakage in systems.

[37] It is highly recommended that system designers initiate requirements which include the use of this new design by their add-in card suppliers.

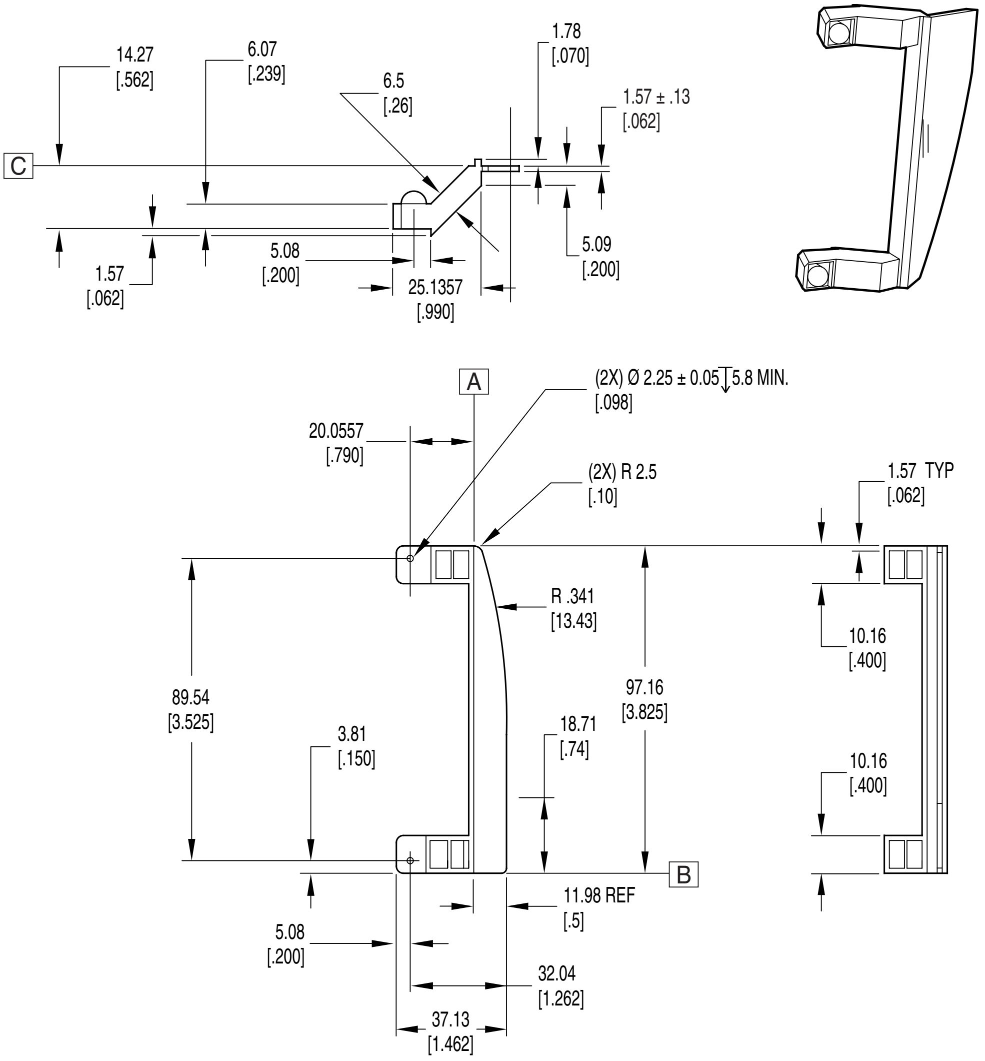


NOTES:

1. MATERIAL: .034 THK (20 GA) C1010/1020 CRS ZINC PLATE
2. TOLERANCE UNLESS OTHERWISE NOTED ± 0.25 [0.010]

A-0265

Figure 5-9: PCI Low Profile Bracket



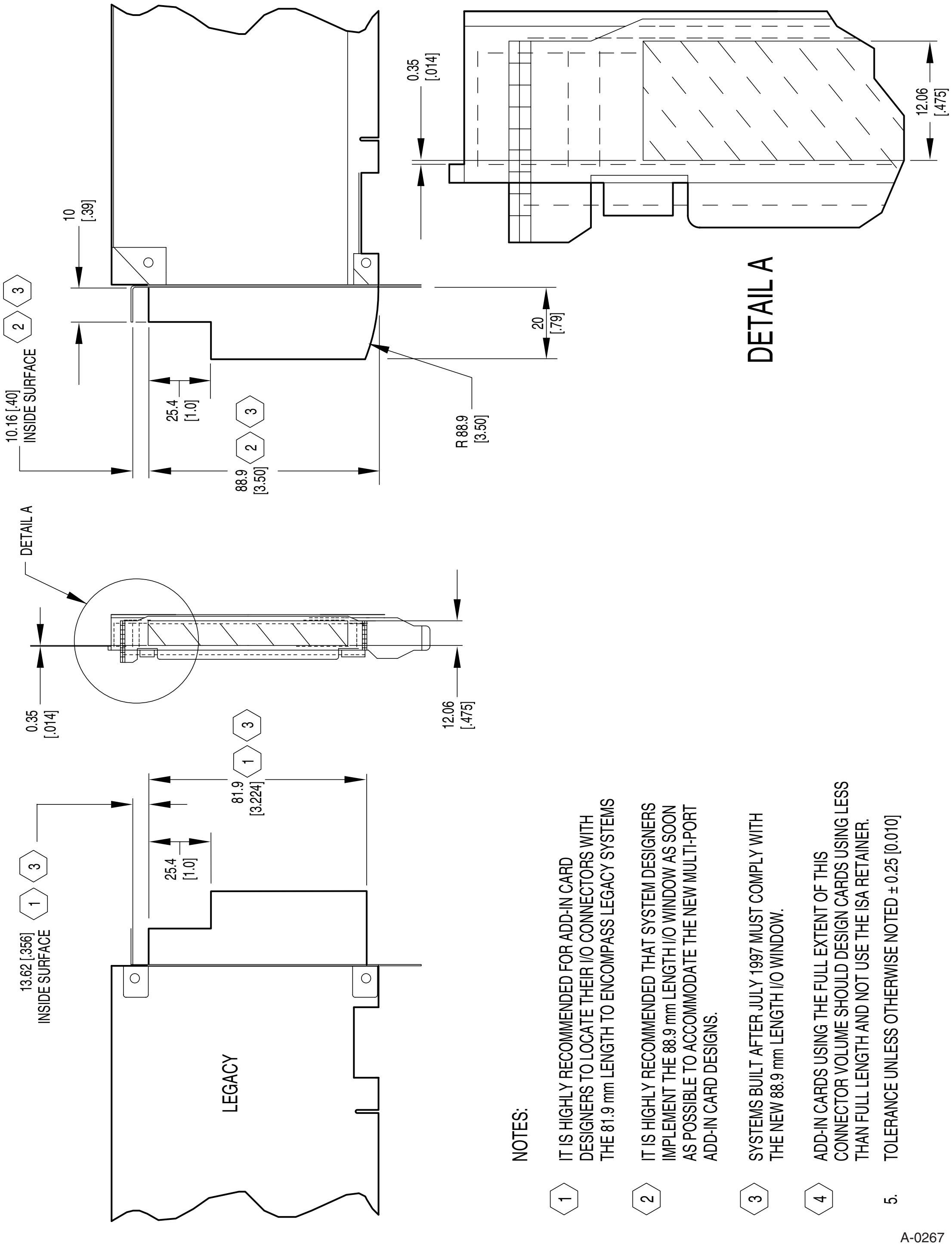
NOTES:

1. CONTINUED USE OF THE ISA RETAINER DESCRIBED IN REVISION 2.1 OF THIS SPECIFICATION IS PERMITTED, BUT IT IS HIGHLY RECOMMENDED THAT ADD-IN CARD DESIGNERS IMPLEMENT THIS DESIGN AS SOON AS POSSIBLE TO AVOID INSTALLATION PROBLEMS WHEN LARGE I/O CONNECTORS ARE PRESENT. THIS PART SHOULD BE CONSIDERED AS AN ALTERNATIVE PART FOR FULL LENGTH ADD-IN CARDS IN LEGACY SYSTEMS ONLY. IT IS HIGHLY RECOMMENDED THAT IF THE FULL CONNECTOR VOLUME DEFINED IN FIGURE 5-11 IS USED, THAT CARDS LESS THAN FULL LENGTH SHOULD BE DESIGNED TO AVOID USE OF THE ISA RETAINER.

2. TOLERANCE UNLESS OTHERWISE NOTED 0.25 [.010]

A-0266

Figure 5-10: PCI Standard Retainer



NOTES:

1. IT IS HIGHLY RECOMMENDED FOR ADD-IN CARD DESIGNERS TO LOCATE THEIR I/O CONNECTORS WITH THE 81.9 mm LENGTH TO ENCOMPASS LEGACY SYSTEMS
2. IT IS HIGHLY RECOMMENDED THAT SYSTEM DESIGNERS IMPLEMENT THE 88.9 mm LENGTH I/O WINDOW AS SOON AS POSSIBLE TO ACCOMMODATE THE NEW MULTI-PORT ADD-IN CARD DESIGNS.
3. SYSTEMS BUILT AFTER JULY 1997 MUST COMPLY WITH THE NEW 88.9 mm LENGTH I/O WINDOW.
4. ADD-IN CARDS USING THE FULL EXTENT OF THIS CONNECTOR VOLUME SHOULD DESIGN CARDS USING LESS THAN FULL LENGTH AND NOT USE THE ISA RETAINER.
5. TOLERANCE UNLESS OTHERWISE NOTED ± 0.25 [0.010]

Figure 5-11: I/O Window Height

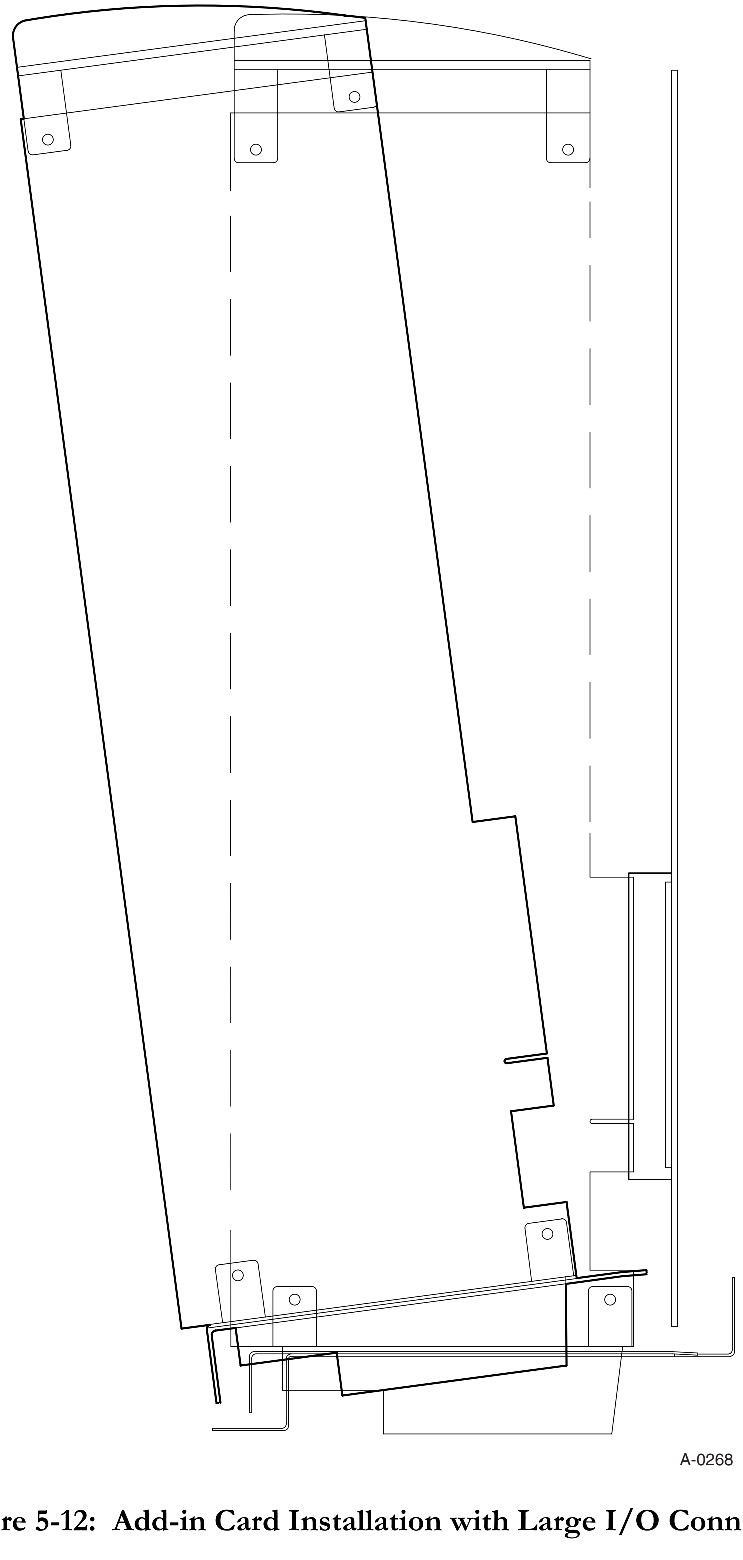


Figure 5-12: Add-in Card Installation with Large I/O Connector

5.3. Connector Physical Description

There are two 3.3V connectors that can be used (32 bit and 64 bit) depending on the PCI implementation. In the connector drawings, the recommended system board layout details are given as nominal dimensions. Layout detail tolerances should be consistent with the connector supplier's recommendations and good engineering practice.

See Figures 5-13 through 5-16 for connector dimensions and layout recommendations. See Figures 5-17 through 5-21 for card edge connector dimensions and tolerances. Tolerances for add-in cards are given so that interchangeable add-in cards can be manufactured.

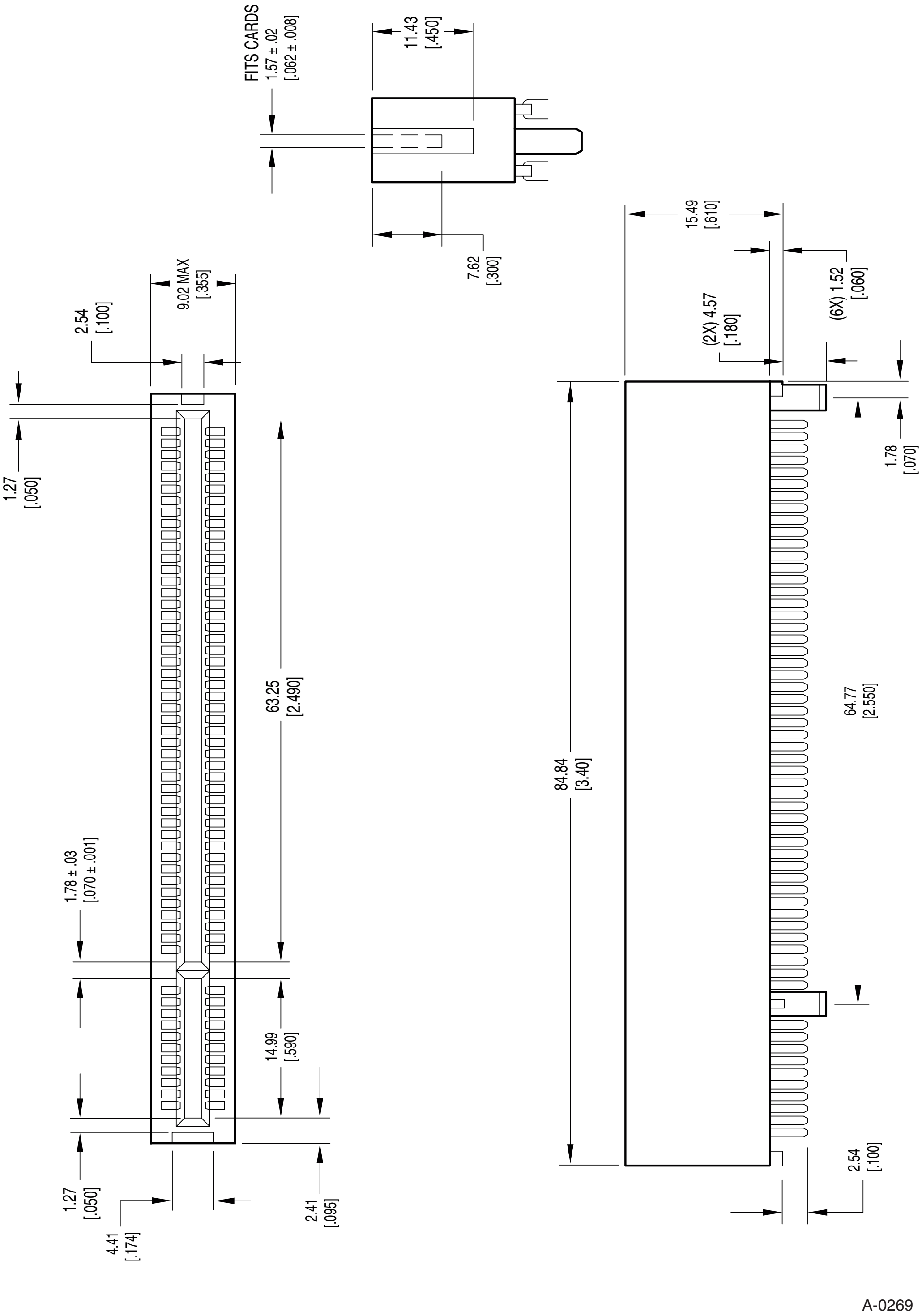


Figure 5-13: 32-bit Connector

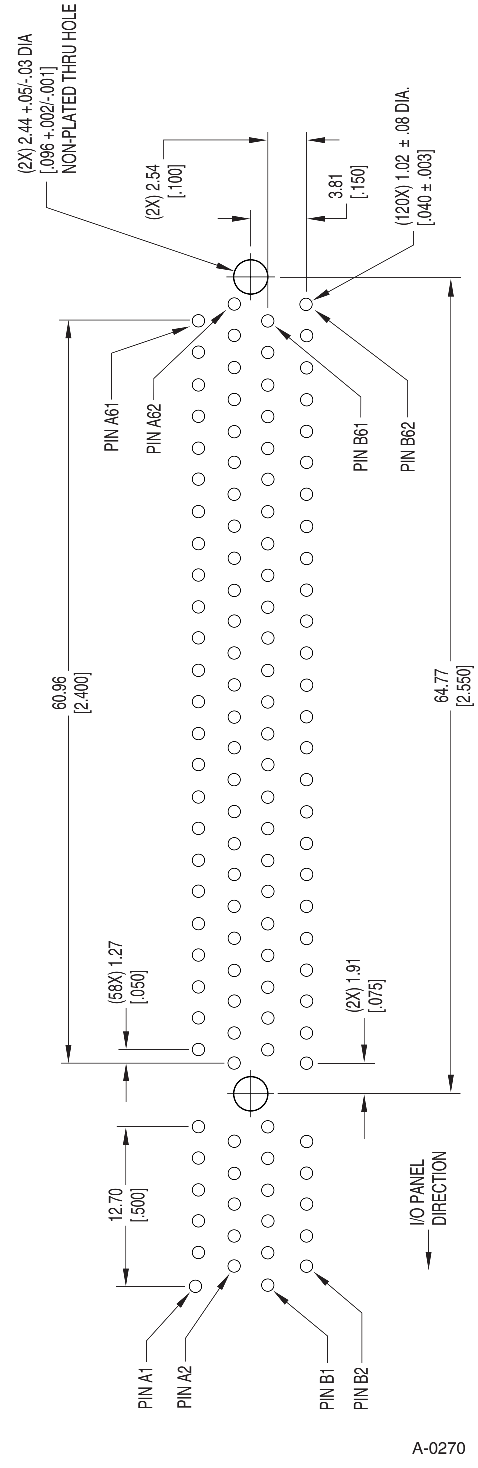


Figure 5-14: 3.3V/32-bit Connector Layout Recommendation

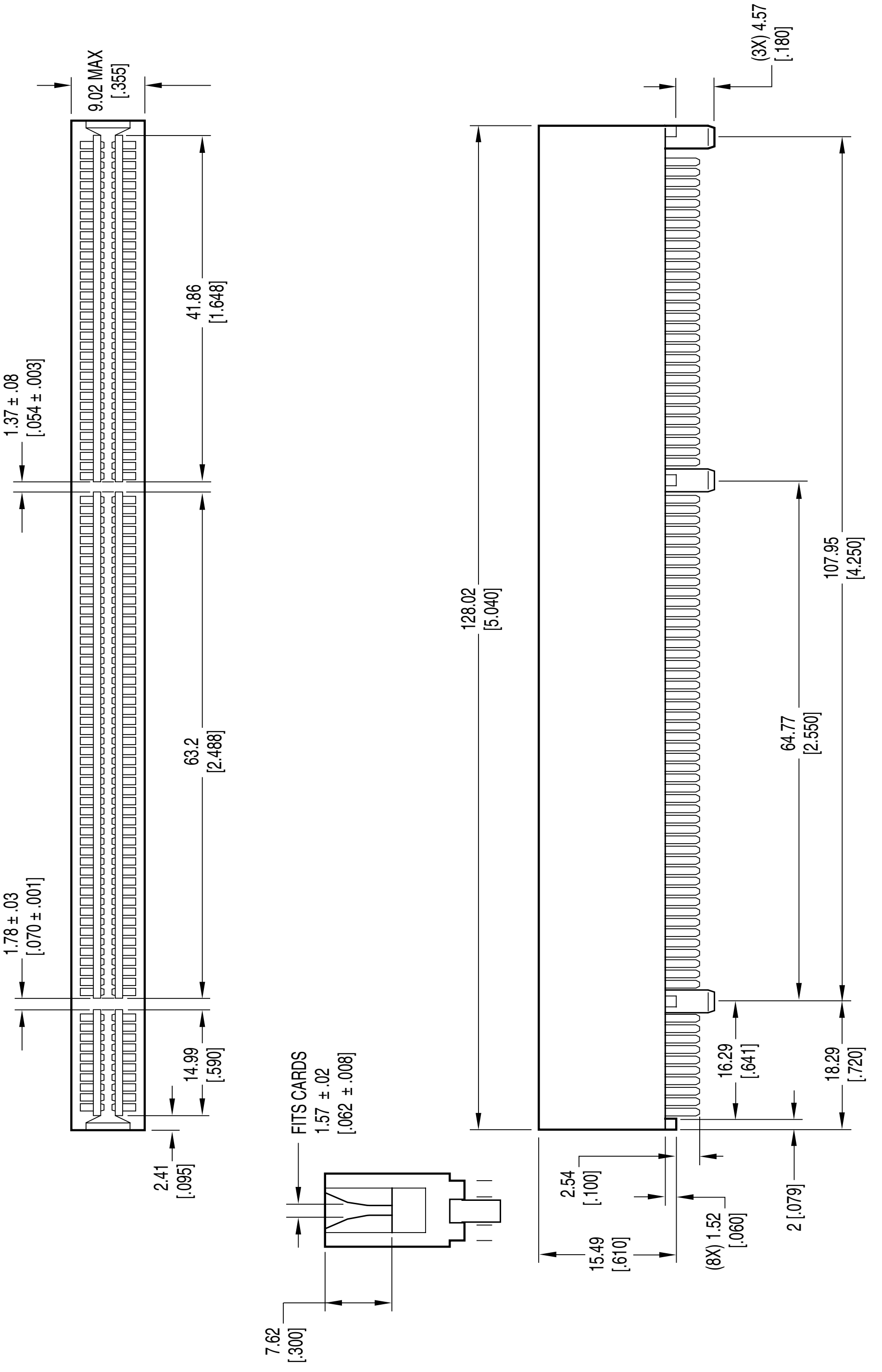


Figure 5-15: 3.3V/64-bit Connector

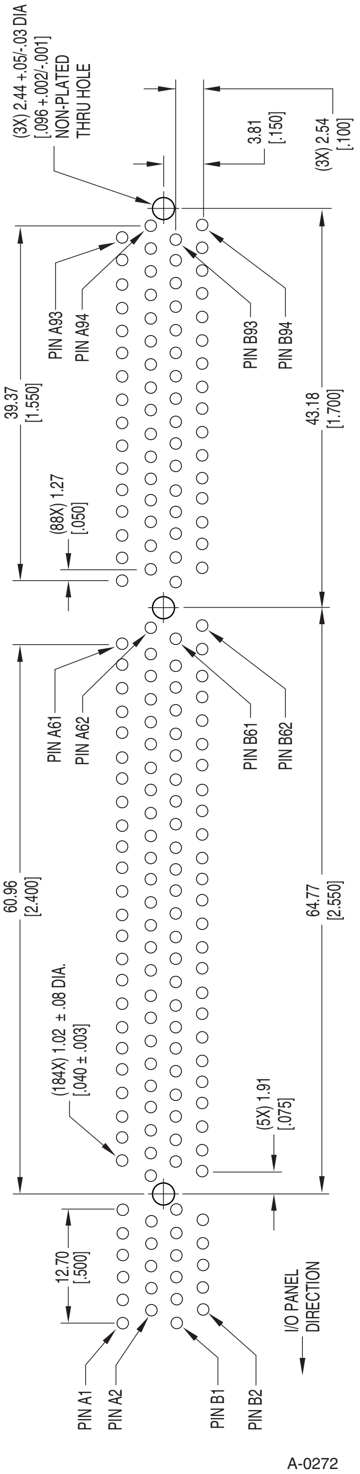


Figure 5-16: 3.3V/64-bit Connector Layout Recommendation

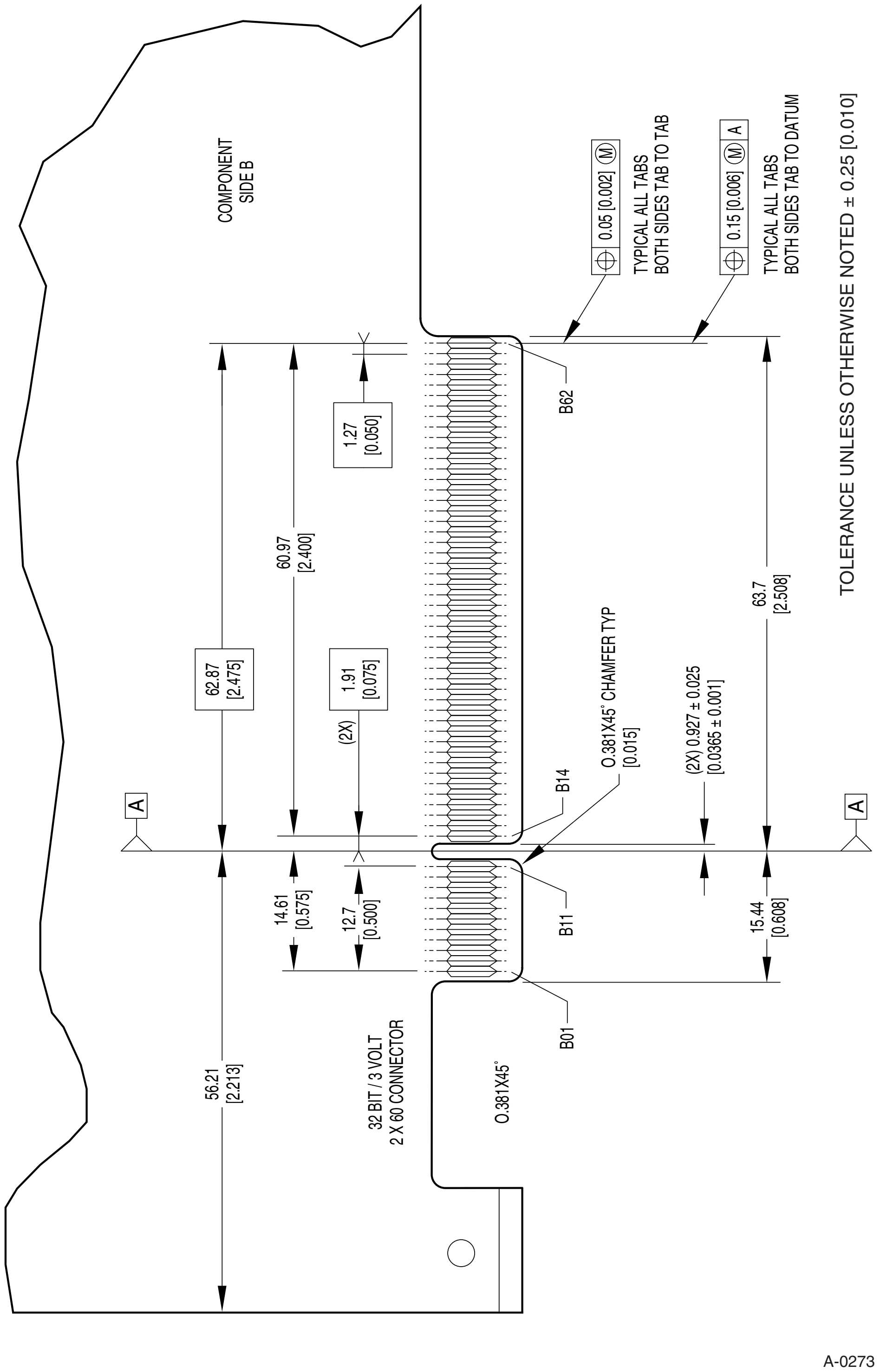

COMPONENT SIDE B
0.05 [0.002] Ⓜ
TYPICAL ALL TABS BOTH SIDES TAB TO TAB
0.15 [0.006] Ⓜ A
TYPICAL ALL TABS BOTH SIDES TAB TO DATUM
TOLERANCE UNLESS OTHERWISE NOTED ± 0.25 [0.010]
B62
1.27 [0.050]
60.97 [2.400]
62.87 [2.475]
1.91 [0.075]
(2X)
63.7 [2.508]
0.381X45° CHAMFER TYP [0.015]
(2X) 0.927 ± 0.025 [0.0365 ± 0.001]
B14
A
A
14.61 [0.575]
12.7 [0.500]
B11
15.44 [0.608]
B01
56.21 [2.213]
32 BIT / 3 VOLT 2 X 60 CONNECTOR
0.381X45°
A-0273

Figure 5-17: 3.3V/32-bit Add-in Card Edge Connector Dimensions and Tolerances

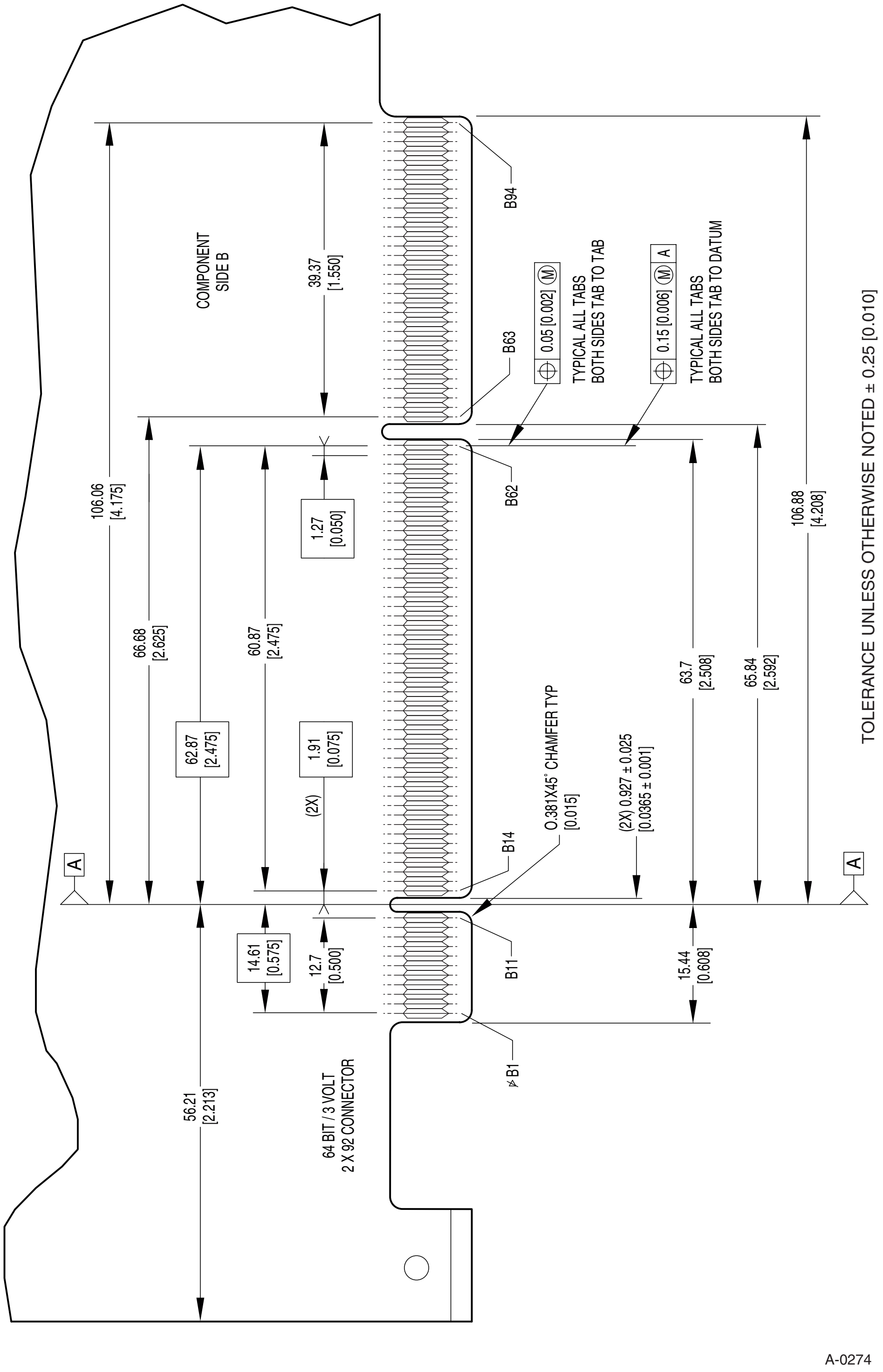


Figure 5-18: 3.3V/64-bit Add-in Card Edge Connector Dimensions and Tolerances

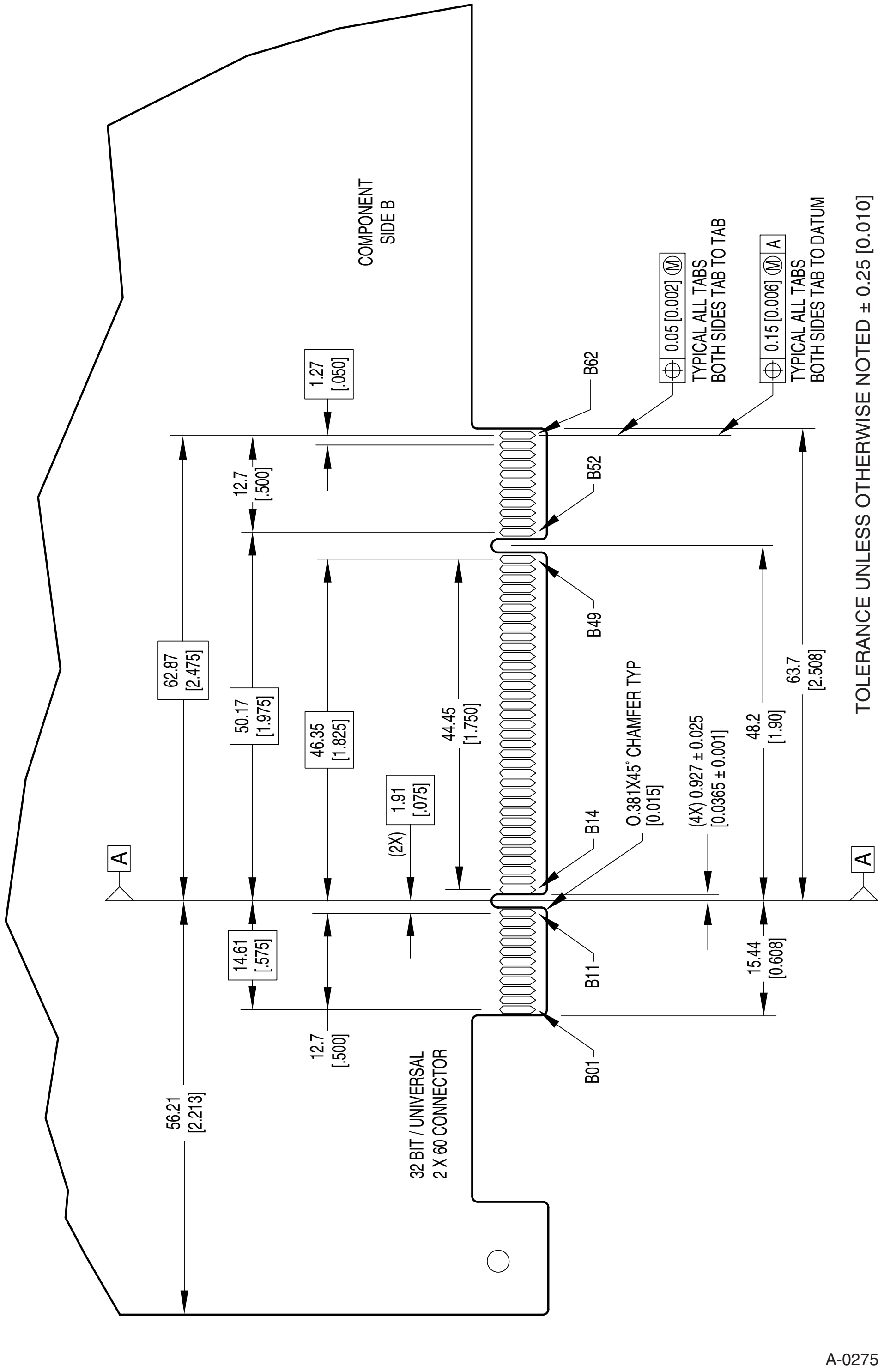


Figure 5-19: Universal 32-bit Add-in Card Edge Connector Dimensions and Tolerances

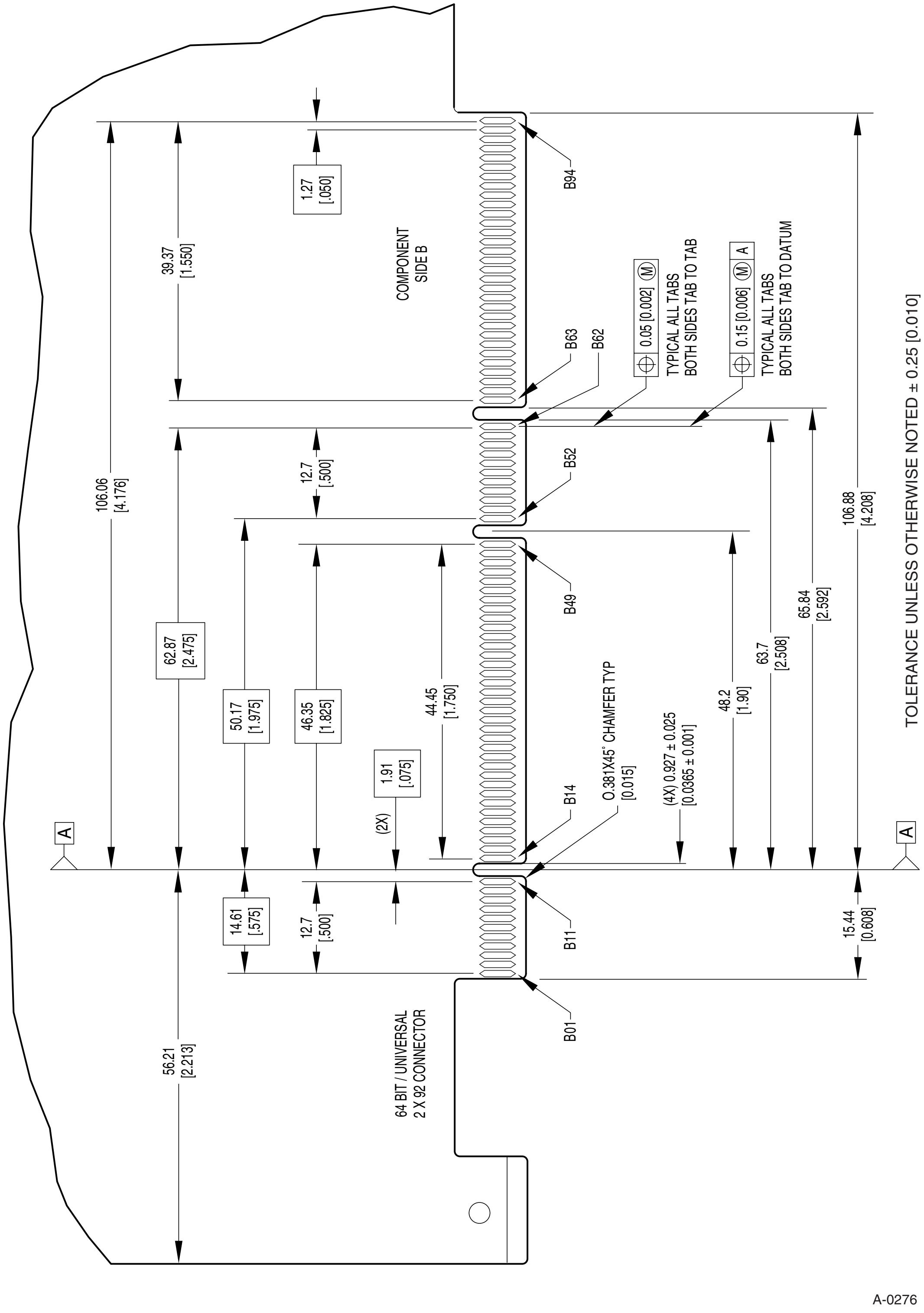


Figure 5-20: Universal 64-bit Add-in Card Edge Connector Dimensions and Tolerances

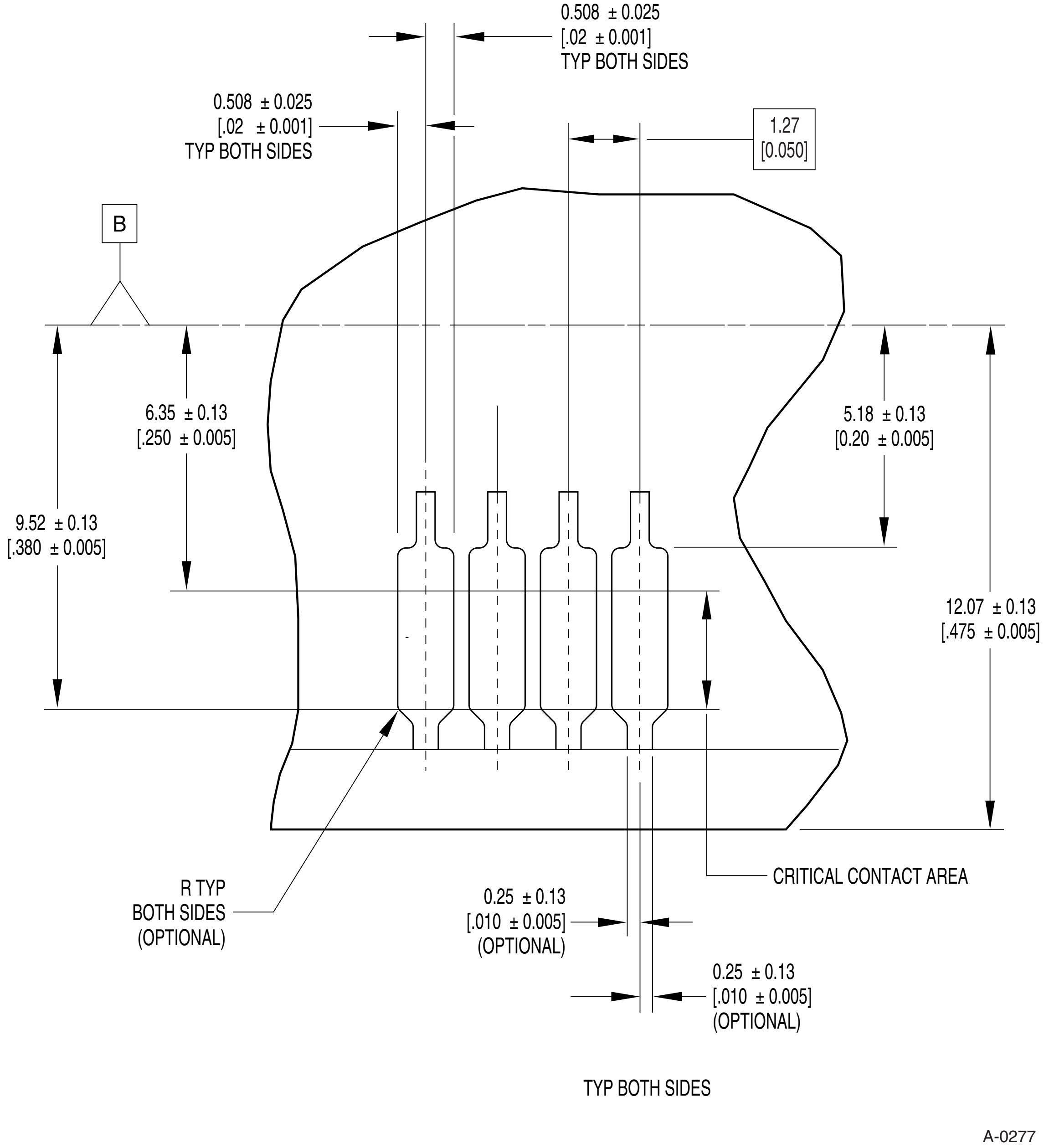


Figure 5-21: PCI Add-in Card Edge Connector Contacts

5.4. Connector Physical Requirements

Table 5-1: Connector Physical Requirements

Part	Materials
Connector Housing	High-temperature thermoplastic, UL flammability rating 94V-0, color: white.
Contacts	Phosphor bronze.
Contact Finish ⟨3⟩	0.000030 inch minimum gold over 0.000050 inch minimum nickel in the contact area. Alternate finish: gold flash over 0.000040 inch (1 micron) minimum palladium or palladium-nickel over nickel in the contact area.

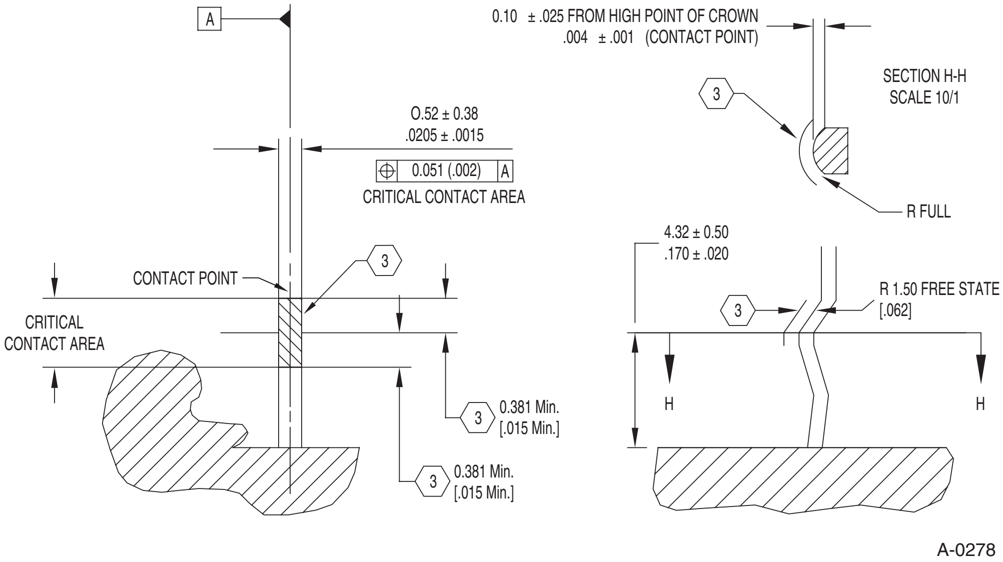


Figure 5-22: Connector Contact Detail

5.5. Connector Performance Specification

Table 5-2: Connector Mechanical Performance Requirements

Parameter	Specification
Durability	100 mating cycles without physical damage or exceeding low level contact resistance requirement when mated with the recommended add-in card edge.
Mating Force	6 oz. (1.7 N) max. avg. per opposing contact pair using MIL-STD-1344, Method 2013.1 and gauge per MIL-C-21097 with profile as shown in add-in card specification.
Contact Normal Force	75 grams minimum.

Table 5-3: Connector Electrical Performance Requirements

Parameter	Specification
Contact Resistance	(low signal level) 30 mΩ max. initial, 10 mΩ max. increase through testing. Contact resistance, test per MIL-STD-1344, Method 3002.1.
Insulation Resistance	1000 MΩ min. per MIL STD 202, Method 302, Condition B.
Dielectric Withstand Voltage	500 VAC RMS. per MIL-STD-1344, Method D3001.1 Condition 1.
Capacitance	2 pF max. @ 1 MHz.
Current Rating	1 A, 30 °C rise above ambient.
Voltage Rating	125 V.
Certification	UL Recognition and CSA Certification required.

Table 5-4: Connector Environmental Performance Requirements

Parameter	Specification
Operating Temperature	-40 °C to 105 °C
Thermal Shock	-55 °C to 85 °C, 5 cycles per MIL-STD-1344, Method 1003.1.
Flowing Mixed Gas Test	Battelle, Class II. Connector mated with an add-in card and tested per Battelle method.

5.6. System Board Implementation

Two types of system board implementations are supported by the PCI add-in card design: add-in card connectors mounted on the system board and add-in card connectors mounted on a riser card.

See Figure 5-23 for system board details. The standard card spacing for PCI connectors is 0.8 inches.

See Figures 5-24 through 5-27 for riser card details. In the connector drawings and riser connector footprints, the details are given as nominal dimensions. The tolerances should be consistent with the connector supplier's recommendations and good engineering practice.

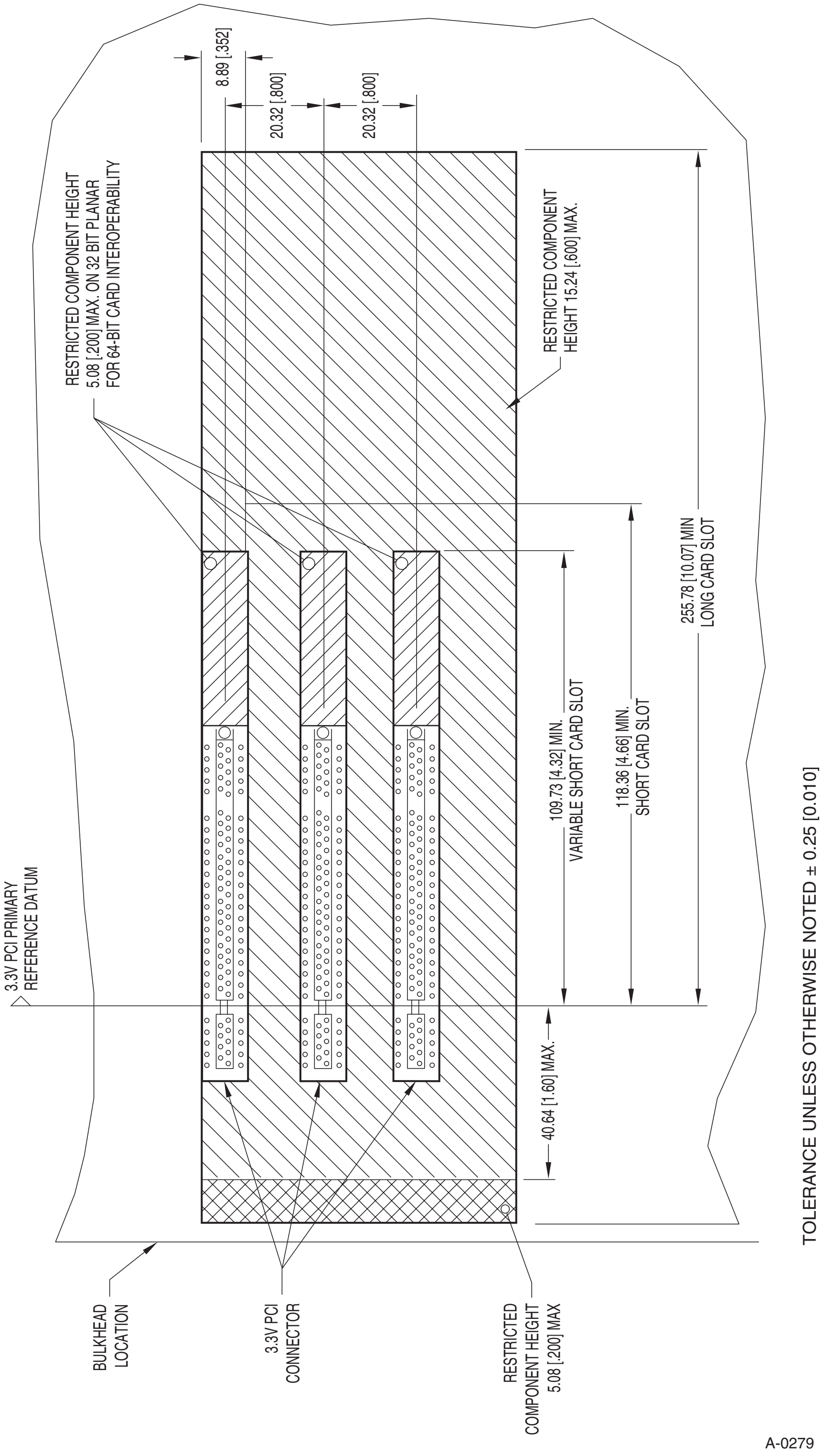

8.89 [.352]
20.32 [.800]
20.32 [.800]
RESTRICTED COMPONENT HEIGHT 5.08 [.200] MAX. ON 32 BIT PLANAR FOR 64-BIT CARD INTEROPERABILITY
RESTRICTED COMPONENT HEIGHT 15.24 [.600] MAX.
255.78 [10.07] MIN LONG CARD SLOT
118.36 [4.66] MIN. SHORT CARD SLOT
109.73 [4.32] MIN. VARIABLE SHORT CARD SLOT
3.3V PCI PRIMARY REFERENCE DATUM
40.64 [1.60] MAX.
TOLERANCE UNLESS OTHERWISE NOTED ± 0.25 [0.010]
BULKHEAD LOCATION
3.3V PCI CONNECTOR
RESTRICTED COMPONENT HEIGHT 5.08 [.200] MAX
A-0279

Figure 5-23: PCI Connector Location on System Board

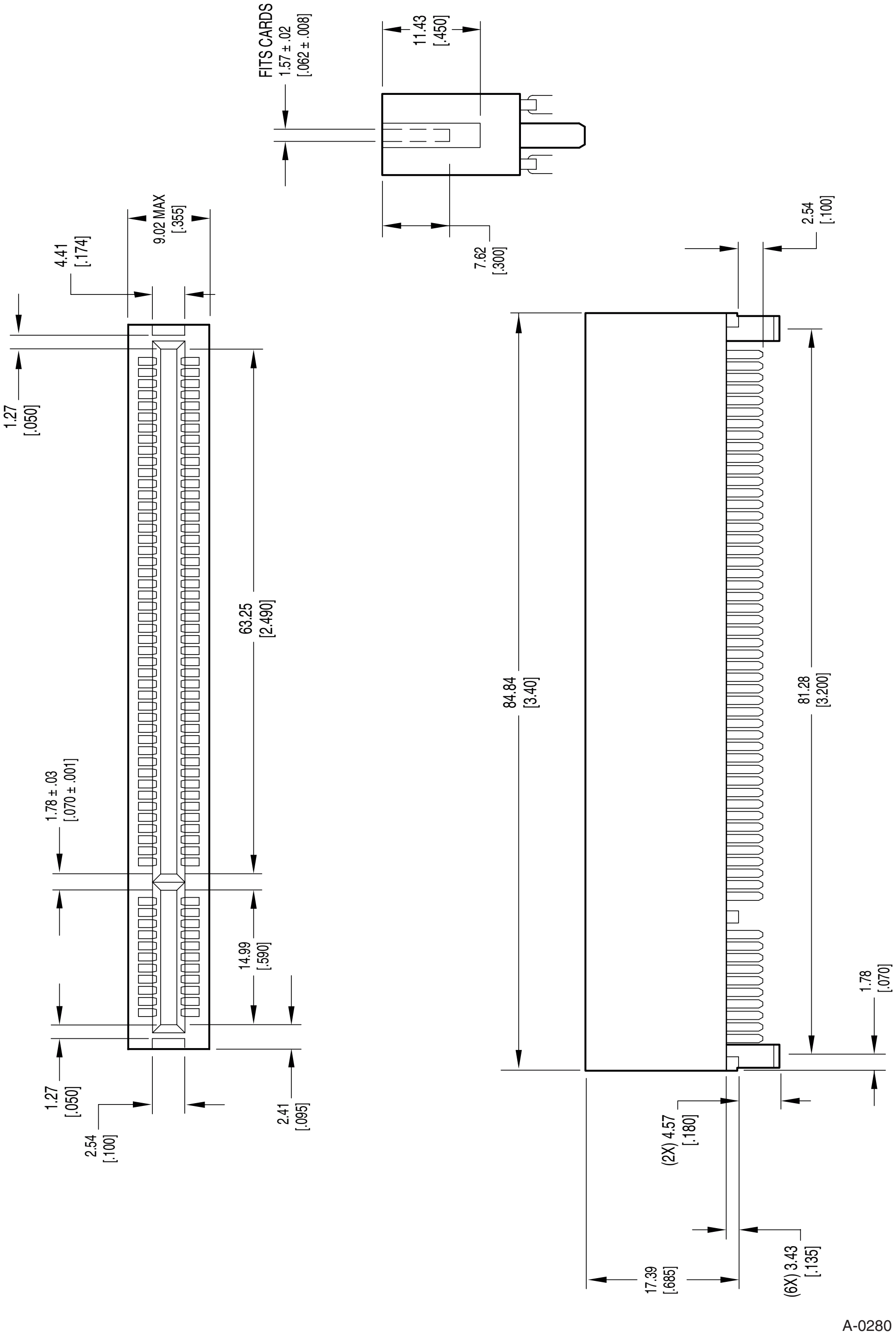


Figure 5-24: 32-bit PCI Riser Connector[38]

---

[38] It is highly recommended that LPX system chassis designers utilize the PCI riser connector when implementing riser cards on LPX systems and using connectors on 0.800-inch spacing.

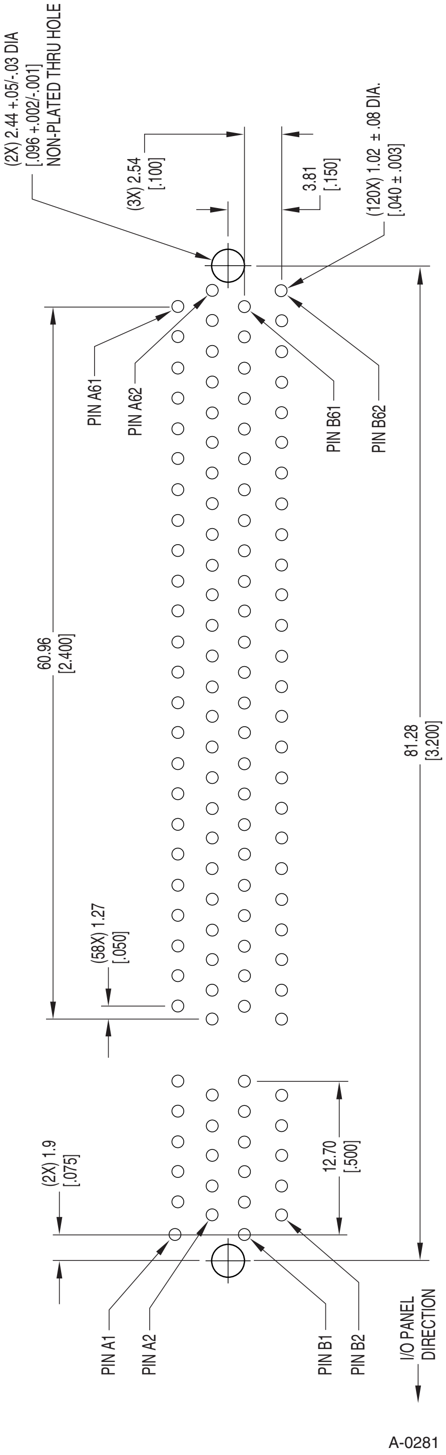


Figure 5-25: 32-bit/3.3V PCI Riser Connector Footprint

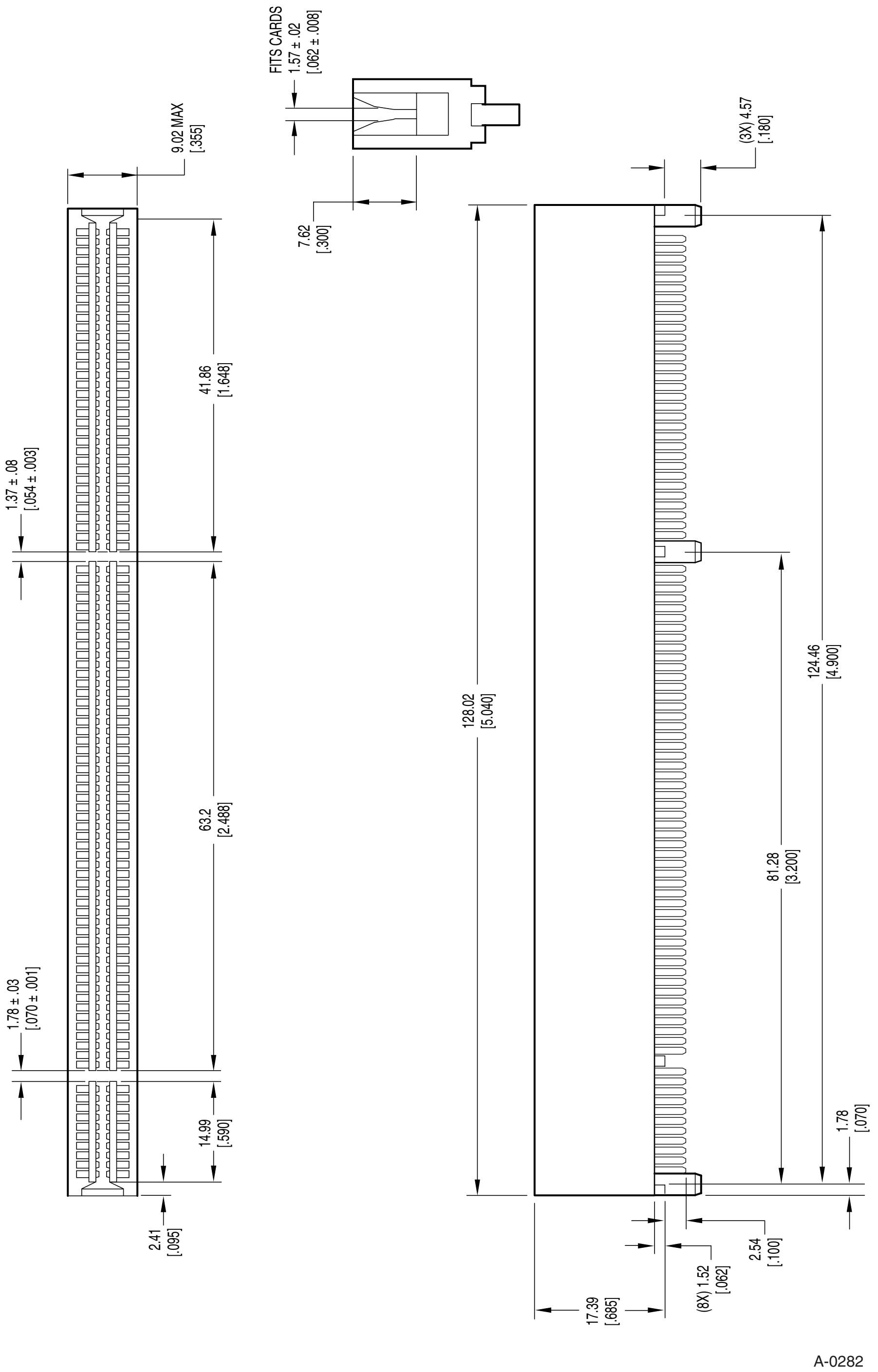


Figure 5-26: 64-bit/3.3V PCI Riser Connector[38]

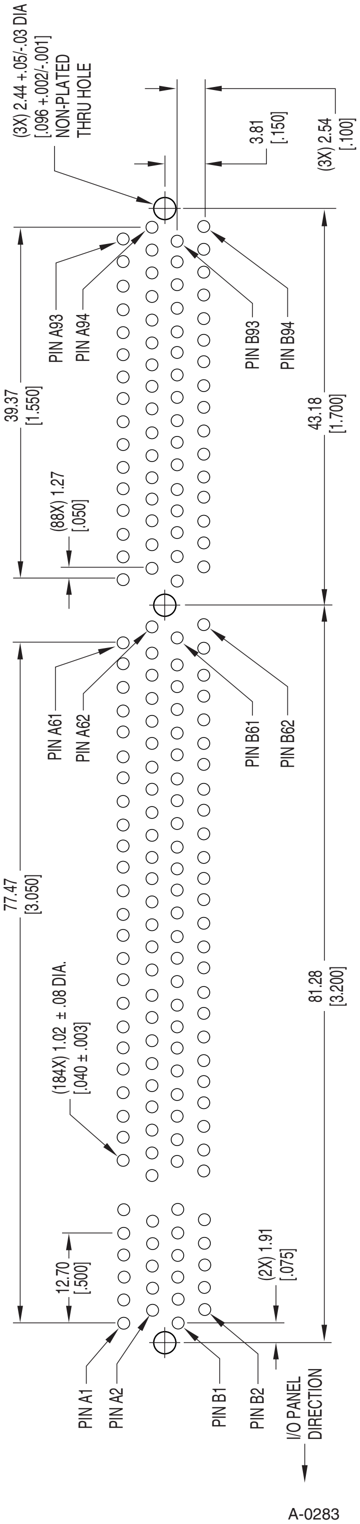


Figure 5-27: 64-bit/3.3V PCI Riser Connector Footprint

6

6. Configuration Space

This chapter defines the programming model and usage rules for the configuration register space in PCI compliant devices. This chapter is limited to the definition of PCI compliant components for a wide variety of system types. System dependent issues for specific platforms, such as mapping various PCI address spaces into host CPU address spaces, access ordering rules, requirements for host-to-PCI bus bridges, etc., are not described in this chapter.

The intent of the PCI Configuration Space definition is to provide an appropriate set of configuration "hooks" which satisfies the needs of current and anticipated system configuration mechanisms, without specifying those mechanisms or otherwise placing constraints on their use. The criteria for these configuration "hooks" are:

- Sufficient support to allow future configuration mechanisms to provide:
    - Full device relocation, including interrupt binding
    - Installation, configuration, and booting without user intervention
    - System address map construction by device independent software
- Minimize the silicon burden created by required functions
- Leverage commonality with a template approach to common functions, without precluding devices with unique requirements

All PCI devices (except host bus bridges) must implement Configuration Space. Multifunction devices must provide a Configuration Space for each function implemented (refer to Section 6.2.1).

6.1. Configuration Space Organization

This section defines the organization of Configuration Space registers and imposes a specific record structure or template on the 256-byte space. This space is divided into a predefined header region and a device dependent region.[39] Devices implement only the necessary and relevant registers in each region. A device's Configuration Space must be accessible at all times, not just during system boot.

The predefined header region consists of fields that uniquely identify the device and allow the device to be generically controlled. The predefined header portion of the Configuration Space is divided into two parts. The first 16 bytes are defined the same for all types of

---

[39] The device dependent region contains device specific information and is not described in this document.

devices. The remaining bytes can have different layouts depending on the base function that the device supports. The Header Type field (located at offset 0Eh) defines what layout is provided. Currently three Header Types are defined, 00h which has the layout shown in Figure 6-1, 01h which is defined for PCI-to-PCI bridges and is documented in the *PCI to PCI Bridge Architecture Specification*, and 02h which is defined for CardBus bridges and is documented in the *PC Card Standard*[40].

System software may need to scan the PCI bus to determine what devices are actually present. To do this, the configuration software must read the Vendor ID in each possible PCI "slot." The host bus to PCI bridge must unambiguously report attempts to read the Vendor ID of non-existent devices. Since 0 FFFFh is an invalid Vendor ID, it is adequate for the host bus to PCI bridge to return a value of all 1's on read accesses to Configuration Space registers of non-existent devices. (Note that these accesses will be terminated with a Master-Abort.)

All PCI devices must treat Configuration Space write operations to reserved registers as no-ops; that is, the access must be completed normally on the bus and the data discarded. Read accesses to reserved or unimplemented registers must be completed normally and a data value of 0 returned.

Figure 6-1 depicts the layout of a Type 00h predefined header portion of the 256-byte Configuration Space. Devices must place any necessary device specific registers after the predefined header in Configuration Space. All multi-byte numeric fields follow *little-endian* ordering; that is, lower addresses contain the least significant parts of the field. Software must take care to deal correctly with bit-encoded fields that have some bits reserved for future use. On reads, software must use appropriate masks to extract the defined bits, and may not rely on reserved bits being any particular value. On writes, software must ensure that the values of reserved bit positions are preserved; that is, the values of reserved bit positions must first be read, merged with the new values for other bit positions and the data then written back. Section 6.2 describes the registers in the Type 00h predefined header portion of the Configuration Space.

---

[40] The *PC Card Standard* is available from PCMCIA and contact information can be found at http://www.pc-card.com

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cacheline Size	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

A-0191

Figure 6-1: Type 00h Configuration Space Header

All PCI compliant devices must support the Vendor ID, Device ID, Command, Status, Revision ID, Class Code, and Header Type fields in the header. Refer to Section 6.2.4 for the requirements for Subsystem ID and Subsystem Vendor ID. Implementation of the other registers in a Type 00h predefined header is optional (i.e., they can be treated as reserved registers) depending on device functionality. If a device supports the function that the register is concerned with, the device must implement it in the defined location and with the defined functionality.

6.2. Configuration Space Functions

PCI has the potential for greatly increasing the ease with which systems may be configured. To realize this potential, all PCI devices must provide certain functions that system configuration software can utilize. This section also lists the functions that need to be supported by PCI devices via registers defined in the predefined header portion of the Configuration Space. The exact format of these registers (i.e., number of bits implemented) is device specific. However, some general rules must be followed. All registers must be capable of being read, and the data returned must indicate the value that the device is actually using.

Configuration Space is intended for configuration, initialization, and catastrophic error handling functions. Its use should be restricted to initialization software and error handling software. All operational software must continue to use I/O and/or Memory Space accesses to manipulate device registers.

6.2.1. Device Identification

Five fields in the predefined header deal with device identification. All PCI devices are required to implement these fields. Generic configuration software will be able to easily determine what devices are available on the system's PCI bus(es). All of these registers are read-only.

Vendor ID	This field identifies the manufacturer of the device. Valid vendor identifiers are allocated by the PCI SIG to ensure uniqueness. 0 FFFFh is an invalid value for Vendor ID.
Device ID	This field identifies the particular device. This identifier is allocated by the vendor.
Revision ID	This register specifies a device specific revision identifier. The value is chosen by the vendor. Zero is an acceptable value. This field should be viewed as a vendor defined extension to the *Device ID*.
Header Type	This byte identifies the layout of the second part of the predefined header (beginning at byte 10h in Configuration Space) and also whether or not the device contains multiple functions. Bit 7 in this register is used to identify a multi-function device. If the bit is 0, then the device is single function. If the bit is 1, then the device has multiple functions. Bits 6 through 0 identify the layout of the second part of the predefined header. The encoding 00h specifies the layout shown in Figure 6-1. The encoding 01h is defined for PCI-to-PCI bridges and is defined in the document *PCI to PCI Bridge Architecture Specification*. The encoding 02h is defined for a CardBus bridge and is documented in the *PC Card Standard*. All other encodings are reserved.

Class Code The Class Code register is read-only and is used to identify the generic function of the device and, in some cases, a specific register-level programming interface. The register is broken into three byte-size fields. The upper byte (at offset 0Bh) is a base class code which broadly classifies the type of function the device performs. The middle byte (at offset 0Ah) is a sub-class code which identifies more specifically the function of the device. The lower byte (at offset 09h) identifies a specific register-level programming interface (if any) so that device independent software can interact with the device. Encodings for base class, sub-class, and programming interface are provided in Appendix D. All unspecified encodings are reserved.

6.2.2. Device Control

The Command register provides coarse control over a device's ability to generate and respond to PCI cycles. When a 0 is written to this register, the device is logically disconnected from the PCI bus for all accesses except configuration accesses. All devices are required to support this base level of functionality. Individual bits in the Command register may or may not be implemented depending on a device's functionality. For instance, devices that do not implement an I/O Space will not implement a writable element at bit location 0 of the Command register. Devices typically power up with all 0's in this register, but Section 6.6 explains some exceptions. Figure 6-2 shows the layout of the register and Table 6-1 explains the meanings of the different bits in the Command register.

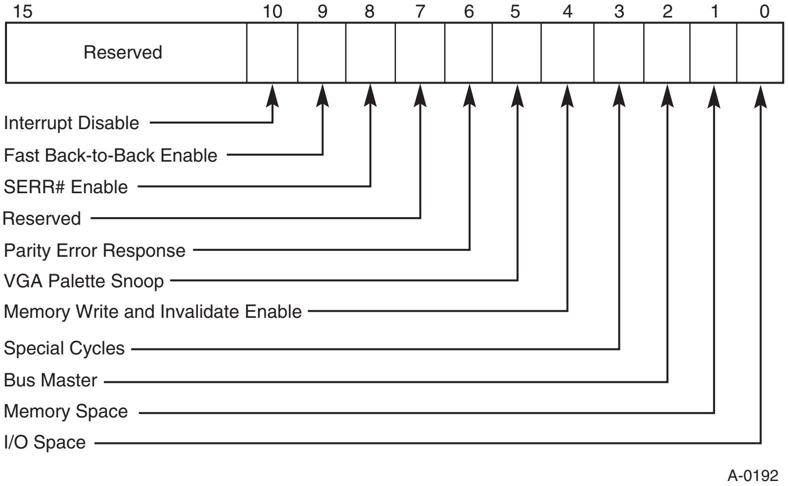


Figure 6-2: Command Register Layout

Table 6-1: Command Register Bits

Bit Location	Description
0	Controls a device's response to I/O Space accesses. A value of 0 disables the device response. A value of 1 allows the device to respond to I/O Space accesses. State after RST# is 0.
1	Controls a device's response to Memory Space accesses. A value of 0 disables the device response. A value of 1 allows the device to respond to Memory Space accesses. State after RST# is 0.
2	Controls a device's ability to act as a master on the PCI bus. A value of 0 disables the device from generating PCI accesses. A value of 1 allows the device to behave as a bus master. State after RST# is 0.
3	Controls a device's action on Special Cycle operations. A value of 0 causes the device to ignore all Special Cycle operations. A value of 1 allows the device to monitor Special Cycle operations. State after RST# is 0.
4	This is an enable bit for using the Memory Write and Invalidate command. When this bit is 1, masters may generate the command. When it is 0, Memory Write must be used instead. State after RST# is 0. This bit must be implemented by master devices that can generate the Memory Write and Invalidate command.
5	This bit controls how VGA compatible and graphics devices handle accesses to VGA palette registers. When this bit is 1, palette snooping is enabled (i.e., the device does not respond to palette register writes and snoops the data). When the bit is 0, the device should treat palette write accesses like all other accesses. VGA compatible devices should implement this bit. Refer to Section 3.10 for more details on VGA palette snooping.
6	This bit controls the device's response to parity errors. When the bit is set, the device must take its normal action when a parity error is detected. When the bit is 0, the device sets its Detected Parity Error status bit (bit 15 in the Status register) when an error is detected, but does not assert PERR# and continues normal operation. This bit's state after RST# is 0. Devices that check parity must implement this bit. Devices are still required to generate parity even if parity checking is disabled.
7	Hardwire this bit to 0.[41]
8	This bit is an enable bit for the SERR# driver. A value of 0 disables the SERR# driver. A value of 1 enables the SERR# driver. This bit's state after RST# is 0. All devices that have an SERR# pin must implement this bit. Address parity errors are reported only if this bit and bit 6 are 1.

---

[41] This bit cannot be assigned any new meaning in new designs. In an earlier version of this specification, bit 7 was used and devices may have hardwired it to 0, 1, or implemented a read/write bit.

Bit Location	Description
9	This optional read/write bit controls whether or not a master can do fast back-to-back transactions to different devices. Initialization software will set the bit if all targets are fast back-to-back capable. A value of 1 means the master is allowed to generate fast back-to-back transactions to different agents as described in Section 3.4.2. A value of 0 means fast back-to-back transactions are only allowed to the same agent. This bit's state after RST# is 0.
10	This bit disables the device/function from asserting INTx#. A value of 0 enables the assertion of its INTx# signal. A value of 1 disables the assertion of its INTx# signal. This bit's state after RST# is 0. Refer to Section 6.8.1.3 for control of MSI.
11-15	Reserved.

6.2.3. Device Status

The Status register is used to record status information for PCI bus related events. The definition of each of the bits is given in Table 6-2 and the layout of the register is shown in Figure 6-3. Devices may not need to implement all bits, depending on device functionality. For instance, a device that acts as a target, but will never signal Target-Abort, would not implement bit 11. Reserved bits should be read-only and return zero when read.

Reads to this register behave normally. Writes are slightly different in that bits can be reset, but not set. A one bit is reset (if it is not read-only) whenever the register is written, and the write data in the corresponding bit location is a 1. For instance, to clear bit 14 and not affect any other bits, write the value 0100_0000_0000_0000b to the register.

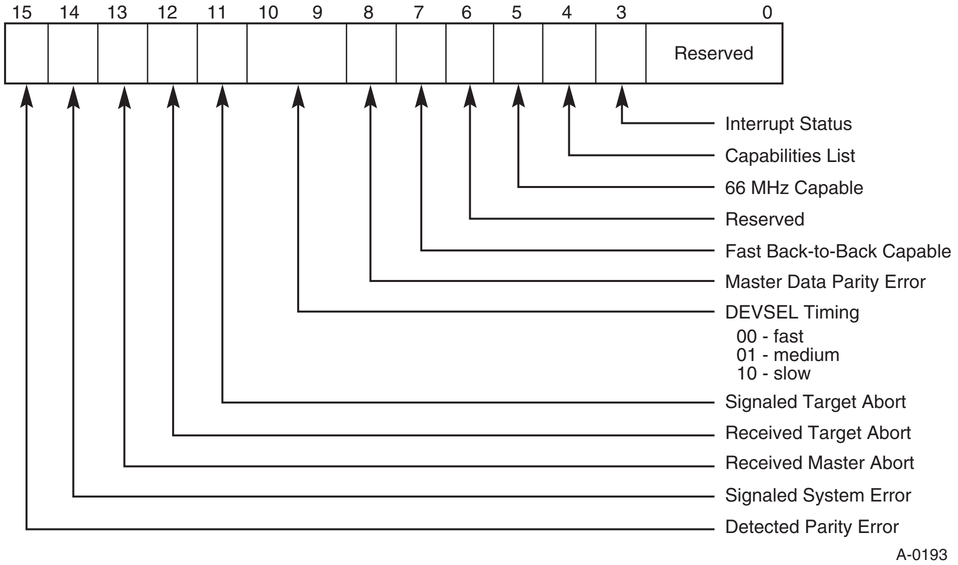


Figure 6-3: Status Register Layout

Table 6-2: Status Register Bits

Bit Location	Description
0-2	Reserved.
3	This read-only bit reflects the state of the interrupt in the device/function. Only when the Interrupt Disable bit in the command register is a 0 and this Interrupt Status bit is a 1, will the device's/function's INTx# signal be asserted. Setting the Interrupt Disable bit to a 1 has no effect on the state of this bit.
4	This optional read-only bit indicates whether or not this device implements the pointer for a New Capabilities linked list at offset 34h. A value of zero indicates that no New Capabilities linked list is available. A value of one indicates that the value read at offset 34h is a pointer in Configuration Space to a linked list of new capabilities. Refer to Section 6.7 for details on New Capabilities.
5	This optional read-only bit indicates whether or not this device is capable of running at 66 MHz as defined in Chapter 7. A value of zero indicates 33 MHz. A value of 1 indicates that the device is 66 MHz capable.
6	This bit is reserved.[42]
7	This optional read-only bit indicates whether or not the target is capable of accepting fast back-to-back transactions when the transactions are not to the same agent. This bit can be set to 1 if the device can accept these transactions and must be set to 0 otherwise. Refer to Section 3.4.2 for a complete description of requirements for setting this bit.
8	This bit is only implemented by bus masters. It is set when three conditions are met: 1) the bus agent asserted PERR# itself (on a read) or observed PERR# asserted (on a write); 2) the agent setting the bit acted as the bus master for the operation in which the error occurred; and 3) the Parity Error Response bit (Command register) is set.
9-10	These bits encode the timing of DEVSEL#. Section 3.6.1 specifies three allowable timings for assertion of DEVSEL#. These are encoded as 00b for fast, 01b for medium, and 10b for slow (11b is reserved). These bits are read-only and must indicate the slowest time that a device asserts DEVSEL# for any bus command except Configuration Read and Configuration Write.
11	This bit must be set by a target device whenever it terminates a transaction with Target-Abort. Devices that will never signal Target-Abort do not need to implement this bit.
12	This bit must be set by a master device whenever its transaction is terminated with Target-Abort. All master devices must implement this bit.
13	This bit must be set by a master device whenever its transaction (except for Special Cycle) is terminated with Master-Abort. All master devices must implement this bit.

---

[42] In Revision 2.1 of this specification, this bit was used to indicate whether or not a device supported User Definable Features.

Bit Location	Description
14	This bit must be set whenever the device asserts SERR#. Devices who will never assert SERR# do not need to implement this bit.
15	This bit must be set by the device whenever it detects a parity error, even if parity error handling is disabled (as controlled by bit 6 in the Command register).

6.2.4. Miscellaneous Registers

This section describes the registers that are device independent and only need to be implemented by devices that provide the described function.

CacheLine Size

This read/write register specifies the system cacheline size in units of DWORDs. This register must be implemented by master devices that can generate the Memory Write and Invalidate command (refer to Section 3.1.1). The value in this register is also used by master devices to determine whether to use Read, Read Line, or Read Multiple commands for accessing memory (refer to Section 3.1.2).

Slave devices that want to allow memory bursting using cacheline wrap addressing mode (refer to Section 3.2.2.2) must implement this register to know when a burst sequence wraps to the beginning of the cacheline.

This field must be initialized to 0 at RST#.

A device may limit the number of cacheline sizes that it can support. For example, it may accept only powers of 2 less than 128. If an unsupported value is written to the CacheLine Size register, the device should behave as if a value of 0 was written.

Latency Timer

This register specifies, in units of PCI bus clocks, the value of the Latency Timer for this PCI bus master (refer to Section 3.5.4). This register must be implemented as writable by any master that can burst more than two data phases. This register may be implemented as read-only for devices that burst two or fewer data phases, but the hardwired value must be limited to 16 or less. A typical implementation would be to build the five high-order bits (leaving the bottom three as read-only), resulting in a timer granularity of eight clocks. At RST#, the register must be initialized to 0 (if programmable).

Built-in Self Test (BIST)

This optional register is used for control and status of BIST. Devices that do not support BIST must always return a value of 0 (i.e., treat it as a reserved register). A device whose BIST is invoked must not prevent normal operation of the PCI bus. Figure 6-4 shows the register layout and Table 6-3 describes the bits in the register.

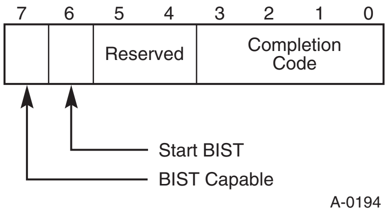


Figure 6-4: BIST Register Layout

Table 6-3: BIST Register Bits

Bit Location	Description
7	Return 1 if device supports BIST. Return 0 if the device is not BIST capable.
6	Write a 1 to invoke BIST. Device resets the bit when BIST is complete. Software should fail the device if BIST is not complete after 2 seconds.
5-4	Reserved. Device returns 0.
3-0	A value of 0 means the device has passed its test. Non-zero values mean the device failed. Device-specific failure codes can be encoded in the non-zero value.

CardBus CIS Pointer

This optional register is used by those devices that want to share silicon between CardBus and PCI. The field is used to point to the Card Information Structure (CIS) for the CardBus card.

For a detailed explanation of the CIS, refer to the PCMCIA v2.10 specification. The subject is covered under the heading Card Metaformat and describes the types of information provided and the organization of this information.

Interrupt Line

The Interrupt Line register is an eight-bit register used to communicate interrupt line routing information. The register is read/write and must be implemented by any device (or device function) that uses an interrupt pin. POST software will write the routing information into this register as it initializes and configures the system.

The value in this register tells which input of the system interrupt controller(s) the device's interrupt pin is connected to. The device itself does not use this value, rather it is used by device drivers and operating systems. Device drivers and operating systems can use this information to determine priority and vector information. Values in this register are system architecture specific.[43]

Interrupt Pin

The Interrupt Pin register tells which interrupt pin the device (or device function) uses. A value of 1 corresponds to INTA#. A value of 2 corresponds to INTB#. A value of 3 corresponds to INTC#. A value of 4 corresponds to INTD#. Devices (or device functions) that do not use an interrupt pin must put a 0 in this register. The values 05h through FFh are reserved. This register is read-only. Refer to Section 2.2.6 for further description of the usage of the INTx# pins.

MIN_GNT and *MAX_LAT*

These read-only byte registers are used to specify the device's desired settings for Latency Timer values. For both registers, the value specifies a period of time in units of ¼ microsecond. Values of 0 indicate that the device has no major requirements for the settings of Latency Timers.

MIN_GNT is used for specifying how long a burst period the device needs assuming a clock rate of 33 MHz. MAX_LAT is used for specifying how often the device needs to gain access to the PCI bus.

Devices should specify values that will allow them to most effectively use the PCI bus as well as their internal resources. Values should be chosen assuming that the target does not insert any wait-states.

IMPLEMENTATION NOTE

Choosing MIN_GNT and MAX_LAT

A fast Ethernet controller (100 Mbs) has a 64-byte buffer for each transfer direction. Optimal usage of these internal resources is achieved when the device treats each buffer as two 32-byte ping-pong buffers. Each 32-byte buffer has eight DWORDS of data to be transferred, resulting in eight data phases on the PCI bus. These eight data phases translate to ¼ microsecond at 33 MHz, so the MIN_GNT value for this device is "1". When moving data, the device will need to empty or fill a 32-byte buffer every 3.2 μs (assuming a throughput of 10 MB/s). This would correspond to a MAX_LAT value of 12.

---

[43] For x86 based PCs, the values in this register correspond to IRQ numbers (0-15) of the standard dual 8259 configuration. The value 255 is defined as meaning "unknown" or "no connection" to the interrupt controller. Values between 15 and 254 are reserved.

Subsystem Vendor ID and *Subsystem ID*

These registers are used to uniquely identify the add-in card or subsystem where the PCI device resides. They provide a mechanism for add-in card vendors to distinguish their add-in cards from one another even though the add-in cards may have the same PCI controller on them (and, therefore, the same Vendor ID and Device ID).

Implementation of these registers is required for all PCI devices except those that have a base class 6 with sub class 0-4 (0, 1, 2, 3, 4), or a base class 8 with sub class 0-3 (0, 1, 2, 3). Subsystem Vendor IDs can be obtained from the PCI SIG and are used to identify the vendor of the add-in card or subsystem.[44] Values for the Subsystem ID are vendor specific.

Values in these registers must be loaded and valid prior to the system firmware or any system software accessing the PCI Configuration Space. How these values are loaded is not specified but could be done during the manufacturing process or loaded from external logic (e.g., strapping options, serial ROMs, etc.). These values must not be loaded using expansion ROM software because expansion ROM software is not guaranteed to be run during POST in all systems. Devices are responsible for guaranteeing the data is valid before allowing reads to these registers to complete. This can be done by responding to any accesses with Retry until the data is valid.

If a device is designed to be used exclusively on the system board, the system vendor may use system specific software to initialize these registers after each power-on.

Capabilities Pointer

This optional register is used to point to a linked list of new capabilities implemented by this device. This register is only valid if the “Capabilities List” bit in the Status Register is set. If implemented, the bottom two bits are reserved and should be set to 00b. Software should mask these bits off before using this register as a pointer in Configuration Space to the first entry of a linked list of new capabilities. Refer to Section 6.7 for a description of this data structure.

6.2.5. Base Addresses

One of the most important functions for enabling superior configurability and ease of use is the ability to relocate PCI devices in the address spaces. At system power-up, device independent software must be able to determine what devices are present, build a consistent address map, and determine if a device has an expansion ROM. Each of these areas is covered in the following sections.

---

[44] A company requires only one Vendor ID. That value can be used in either the Vendor ID field of configuration space (offset 00h) or the Subsystem Vendor ID field of configuration space (offset 2Ch). It is used in the Vendor ID field (offset 00h) if the company built the silicon. It is used in the Subsystem Vendor ID field (offset 2Ch) if the company built the add-in card. If a company builds both the silicon and the add-in card, then the same value would be used in both fields.

6.2.5.1. Address Maps

Power-up software needs to build a consistent address map before booting the machine to an operating system. This means it has to determine how much memory is in the system, and how much address space the I/O controllers in the system require. After determining this information, power-up software can map the I/O controllers into reasonable locations and proceed with system boot. In order to do this mapping in a device independent manner, the base registers for this mapping are placed in the predefined header portion of Configuration Space.

Bit 0 in all Base Address registers is read-only and used to determine whether the register maps into Memory or I/O Space. Base Address registers that map to Memory Space must return a 0 in bit 0 (see Figure 6-5). Base Address registers that map to I/O Space must return a 1 in bit 0 (see Figure 6-6).

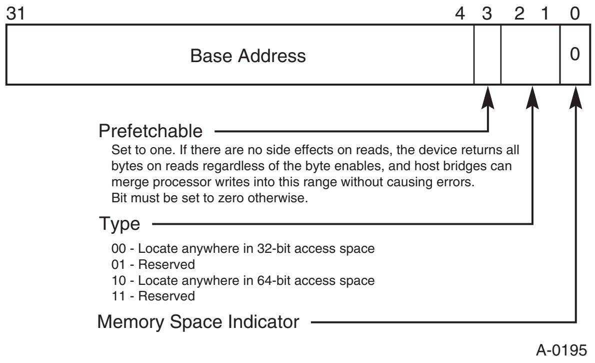

31
4
3
2
1
0
Base Address
0
Prefetchable
Set to one. If there are no side effects on reads, the device returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes into this range without causing errors. Bit must be set to zero otherwise.
Type
00 - Locate anywhere in 32-bit access space
01 - Reserved
10 - Locate anywhere in 64-bit access space
11 - Reserved
Memory Space Indicator
A-0195

Figure 6-5: Base Address Register for Memory

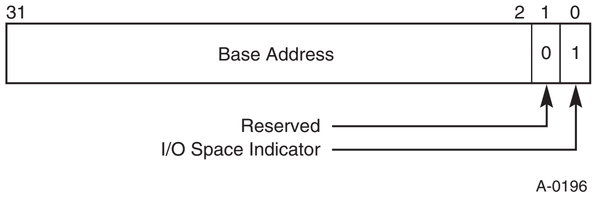


Figure 6-6: Base Address Register for I/O

Base Address registers that map into I/O Space are always 32 bits wide with bit 0 hardwired to a 1. Bit 1 is reserved and must return 0 on reads and the other bits are used to map the device into I/O Space.

Base Address registers that map into Memory Space can be 32 bits or 64 bits wide (to support mapping into a 64-bit address space) with bit 0 hardwired to a 0. For Memory Base Address registers, bits 2 and 1 have an encoded meaning as shown in Table 6-4. Bit 3 should be set to 1 if the data is prefetchable and reset to 0 otherwise. A device can mark a range as prefetchable if there are no side effects on reads, the device returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes (refer to Section 3.2.3) into this range[45] without causing errors. Bits 0-3 are read-only.

Table 6-4: Memory Base Address Register Bits 2/1 Encoding

Bits 2/1	Meaning
00	Base register is 32 bits wide and mapping can be done anywhere in the 32-bit Memory Space.
01	Reserved[46]
10	Base register is 64 bits wide and can be mapped anywhere in the 64-bit address space.
11	Reserved

The number of upper bits that a device actually implements depends on how much of the address space the device will respond to. A 32-bit register can be implemented to support a single memory size that is a power of 2 from 16 bytes to 2 GB. A device that wants a 1 MB memory address space (using a 32-bit base address register) would build the top 12 bits of the address register, hardwiring the other bits to 0.

Power-up software can determine how much address space the device requires by writing a value of all 1's to the register and then reading the value back. The device will return 0's in all don't-care address bits, effectively specifying the address space required. Unimplemented Base Address registers are hardwired to zero.

This design implies that all address spaces used are a power of two in size and are naturally aligned. Devices are free to consume more address space than required, but decoding down to a 4 KB space for memory is suggested for devices that need less than that amount. For instance, a device that has 64 bytes of registers to be mapped into Memory Space may consume up to 4 KB of address space in order to minimize the number of bits in the address decoder. Devices that do consume more address space than they use are not required to respond to the unused portion of that address space. Devices that map control functions into I/O Space must not consume more than 256 bytes per I/O Base Address register. The upper 16 bits of the I/O Base Address register may be hardwired to zero for devices intended for 16-bit I/O systems, such as PC compatibles. However, a full 32-bit decode of I/O addresses must still be done.

---

[45] Any device that has a range that behaves like normal memory should mark the range as prefetchable. A linear frame buffer in a graphics device is an example of a range that should be marked prefetchable.

[46] The encoding to support memory space below 1 MB was supported in previous versions of this specification. System software should recognize this encoding and handle appropriately.

IMPLEMENTATION NOTE

Sizing a 32-bit Base Address Register Example

Decode (I/O or memory) of a register is disabled via the command register before sizing a Base Address register. Software saves the original value of the Base Address register, writes 0 FFFF FFFFh to the register, then reads it back. Size calculation can be done from the 32-bit value read by first clearing encoding information bits (bit 0 for I/O, bits 0-3 for memory), inverting all 32 bits (logical NOT), then incrementing by 1. The resultant 32-bit value is the memory/I/O range size decoded by the register. Note that the upper 16 bits of the result is ignored if the Base Address register is for I/O and bits 16-31 returned zero upon read. The original value in the Base Address register is restored before re-enabling decode in the command register of the device.

64-bit (memory) Base Address registers can be handled the same, except that the second 32-bit register is considered an extension of the first; i.e., bits 32-63. Software writes 0FFFFFFFFh to both registers, reads them back, and combines the result into a 64-bit value. Size calculation is done on the 64-bit value.

A type 00h predefined header has six DWORD locations allocated for Base Address registers starting at offset 10h in Configuration Space. A device may use any of the locations to implement Base Address registers. An implemented 64-bit Base Address register consumes two consecutive DWORD locations. Software looking for implemented Base Address registers must start at offset 10h and continue upwards through offset 24h. A typical device will require one memory range for its control functions. Some graphics devices may use two ranges, one for control functions and another for a frame buffer. A device that wants to map control functions into both memory and I/O Spaces at the same time must implement two Base Address registers (one memory and one I/O). The driver for that device might only use one space in which case the other space will be unused. Devices are recommended always to map control functions into Memory Space.

6.2.5.2. Expansion ROM Base Address Register

Some PCI devices, especially those that are intended for use on add-in cards in PC architectures, require local EPROMs for expansion ROM (refer to the *PCI Firmware Specification, Revision 3.0* for a definition of ROM contents). The four-byte register at offset 30h in a type 00h predefined header is defined to handle the base address and size information for this expansion ROM. Figure 6-7 shows how this word is organized. The register functions exactly like a 32-bit Base Address register except that the encoding (and usage) of the bottom bits is different. The upper 21 bits correspond to the upper 21 bits of the Expansion ROM base address. The number of bits (out of these 21) that a device actually implements depends on how much address space the device requires. For instance, a device that requires a 64 KB area to map its expansion ROM would implement the top 16 bits in the register, leaving the bottom 5 (out of these 21) hardwired to 0. Devices that support an expansion ROM must implement this register.

Device independent configuration software can determine how much address space the device requires by writing a value of all 1's to the address portion of the register and then reading the value back. The device will return 0's in all don't-care bits, effectively specifying the size and alignment requirements. The amount of address space a device requests must not be greater than 16 MB.

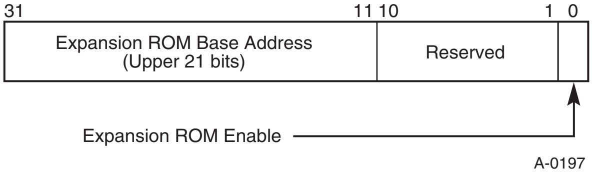


Figure 6-7: Expansion ROM Base Address Register Layout

Bit 0 in the register is used to control whether or not the device accepts accesses to its expansion ROM. When this bit is 0, the device's expansion ROM address space is disabled. When the bit is 1, address decoding is enabled using the parameters in the other part of the base register. This allows a device to be used with or without an expansion ROM depending on system configuration. The Memory Space bit in the Command register has precedence over the Expansion ROM enable bit. A device must respond to accesses to its expansion ROM only if both the Memory Space bit and the Expansion ROM Base Address Enable bit are set to 1. This bit's state after RST# is 0.

In order to minimize the number of address decoders needed, a device may share a decoder between the Expansion ROM Base Address register and other Base Address registers.[47] When expansion ROM decode is enabled, the decoder is used for accesses to the expansion ROM and device independent software must not access the device through any other Base Address registers.

6.3. PCI Expansion ROMs

As part of the update from PCI 2.3 to PCI 3.0, Sections 6.3 through 6.3.3.1.3 of PCI 2.3 were deleted from this specification and moved to the new PCI Firmware Specification.

---

[47] Note that it is the address decoder that is shared, not the registers themselves. The Expansion ROM Base Address register and other Base Address registers must be able to hold unique values at the same time.

6.4. Vital Product Data

Vital Product Data (VPD) is the information that uniquely defines items such as the hardware, software, and microcode elements of a system. The VPD provides the system with information on various FRUs (Field Replaceable Unit) including Part Number, Serial Number, and other detailed information. VPD also provides a mechanism for storing information such as performance and failure data on the device being monitored. The objective, from a system point of view, is to collect this information by reading it from the hardware, software, and microcode components.

Support of VPD within add-in cards is optional depending on the manufacturer. The definition of PCI VPD presents no impact to existing PCI devices and minimal impact to future PCI devices which optionally include VPD. Though support of VPD is optional, add-in card manufacturers are encouraged to provide VPD due to its inherent benefits for the add-in card, system manufacturers, and for Plug and Play.

The mechanism for accessing VPD and the description of VPD data structures is documented in Appendix I.

6.5. Device Drivers

There are two characteristics of PCI devices that may make PCI device drivers different from "standard" or existing device drivers. The first characteristic is that PCI devices are relocatable (i.e., not hardwired) in the address spaces. PCI device drivers (and other configuration software) should use the mapping information stored in the device's Configuration Space registers to determine where the device was mapped. This also applies to determining interrupt line usage.

The second characteristic is that PCI interrupts are shareable. PCI device drivers are required to support shared interrupts, since it is very likely that system implementations will connect more than one device to a single interrupt line. The exact method for interrupt sharing is operating system specific and is not elaborated here.

Some systems may not guarantee that data is delivered to main memory before interrupts are delivered to the CPU. If not handled properly, this can lead to data consistency problems (loss of data). This situation is most often associated with the implementation of posting buffers in bridges between the PCI bus and other buses.

There are three ways that data and interrupt consistency can be guaranteed:

1. The system hardware can guarantee that posting buffers are flushed before interrupts are delivered to the processor.
2. The device signaling the interrupt can perform a read of the just-written data before signaling the interrupt. This causes posting buffers to be flushed.
3. The device driver can perform a read to any register in the device before accessing the data written by the device. This read causes posting buffers to be flushed.

Device drivers are ultimately responsible for guaranteeing consistency of interrupts and data by assuring that at least one of the three methods described above is performed in the system. This means a device driver must do Method 3 unless it implicitly knows Method 2 is done by its device, or it is informed (by some means outside the scope of this specification) that Method 1 is done by the system hardware.

6.6. System Reset

After system reset, the processor(s) must be able to access boot code and any devices necessary to boot the machine. Depending on the system architecture, bridges may need to come up enabled to pass these accesses through to the remote bus.

Similarly, devices on PCI may need to come up enabled to recognize fixed addresses to support the boot sequence in a system architecture. Such devices are required to support the Command register disabling function described in Section 6.2.2. They should also provide a mechanism (invoked through the Configuration Space) to re-enable the recognition of fixed addresses.

6.7. Capabilities List

Certain capabilities added to PCI after the publication of revision 2.1 are supported by adding a set of registers to a linked list called the *Capabilities List*. This optional data structure is indicated in the PCI Status Register by setting the Capabilities List bit (bit 4) to indicate that the Capabilities Pointer is located at offset 34h. This register points to the first item in the list of capabilities.

Each capability in the list consists of an 8-bit ID field assigned by the PCI SIG, an 8 bit pointer in configuration space to the next capability, and some number of additional registers immediately following the pointer to implement that capability. Each capability must be DWORD aligned. The bottom two bits of all pointers (including the initial pointer at 34h) are reserved and must be implemented as 00b although software must mask them to allow for future uses of these bits. A pointer value of 00h is used to indicate the last capability in the list. Figure 6-8 shows how this list is constructed.

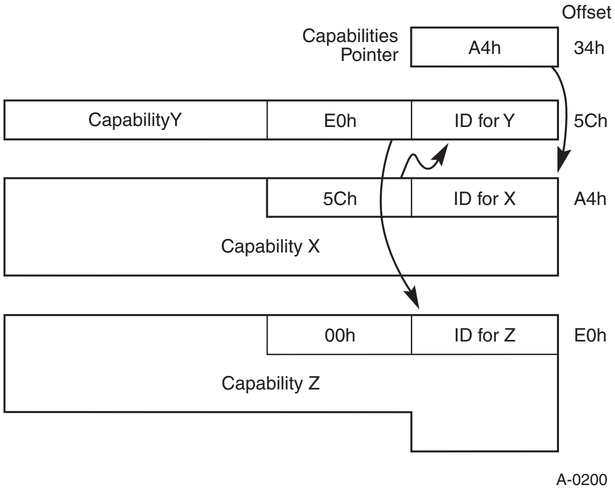


Figure 6-8: Example Capabilities List

Each defined capability must have a SIG assigned ID code. These codes are assigned and handled much like the Class Codes. Refer to Appendix H for a list of currently defined Capabilities. Each Capability must define the detailed register map for that capability. These registers must immediately follow the pointer to the next capability.

6.8. Message Signaled Interrupts

Message Signaled Interrupts (MSI) is an optional feature that enables a device function to request service by writing a system-specified data value to a system-specified address (using a PCI DWORD memory write transaction). System software initializes the message address and message data (from here on referred to as the "vector") during device configuration, allocating one or more vectors to each MSI capable function.

Since the target of the transaction cannot distinguish between an MSI write transaction and any other write transaction, all transaction termination conditions are supported. Therefore, an MSI write transaction can be terminated with a Retry, Master-Abort, Target-Abort, or normal completion (refer to Section 3.3.3.2).

It is recommended that devices implement interrupt pins to provide compatibility in systems that do not support MSI (devices default to interrupt pins). However, it is expected that the need for interrupt pins will diminish over time. Devices that do not support interrupt pins due to pin constraints (rely on polling for device service) may implement messages to increase performance without adding additional pins. Therefore, system configuration software must not assume that a message capable device has an interrupt pin.

Interrupt latency (the time from interrupt signaling to interrupt servicing) is system dependent. Consistent with current interrupt architectures, message signaled interrupts do not provide interrupt latency time guarantees.

MSI-X defines a separate optional extension to basic MSI functionality. Compared to MSI, MSI-X supports a larger maximum number of vectors per function, the ability for software to control aliasing when fewer vectors are allocated than requested, plus the ability for each vector to use an independent address and data value, specified by a table that resides in

Memory Space. However, most of the other characteristics of MSI-X are identical to those of MSI.

MSI and MSI-X each support per-vector masking. Per-vector masking is an optional extension to MSI, and a standard feature with MSI-X. A function that supports the per-vector masking extension to MSI is still backward compatible with system software that is unaware of the extension. MSI-X also supports a Function Mask bit, which when set masks all of the vectors associated with a function.

Per-vector masking is managed through a *Mask* and *Pending* bit pair per MSI vector or MSI-X Table entry. An MSI vector is masked when its associated Mask bit is set. An MSI-X vector is masked when its associated MSI-X Table entry Mask bit or the MSI-X Function Mask bit is set. While a vector is masked, the function is prohibited from sending the associated message, and the function must set the associated Pending bit whenever the function would otherwise send the message. When software unmasks a vector whose associated Pending bit is set, the function must schedule sending the associated message, and clear the Pending bit as soon as the message has been sent.

A function is permitted to implement both MSI and MSI-X, but system software is prohibited from enabling both at the same time. If system software enables both at the same time, the result is undefined.

For the sake of software backward compatibility, MSI and MSI-X use separate and independent capability structures. On functions that support both MSI and MSI-X, system software that supports only MSI can still enable and use MSI without any modification. MSI functionality is managed exclusively through the MSI Capability Structure, and MSI-X functionality is managed exclusively through the MSI-X Capability Structure.

6.8.1. MSI Capability Structure

The capabilities mechanism (refer to Section 6.7) is used to identify and configure an MSI or MSI-X capable device. The MSI capability structure is described in the current section. The MSI-X capability structure is described in Section 6.8.2.

The MSI capability structure is illustrated in Figure 6-9. Each device function that supports MSI (in a multi-function device) must implement its own MSI capability structure. More than one MSI capability structure per function is prohibited, but a function is permitted to have both an MSI and an MSI-X capability structure.

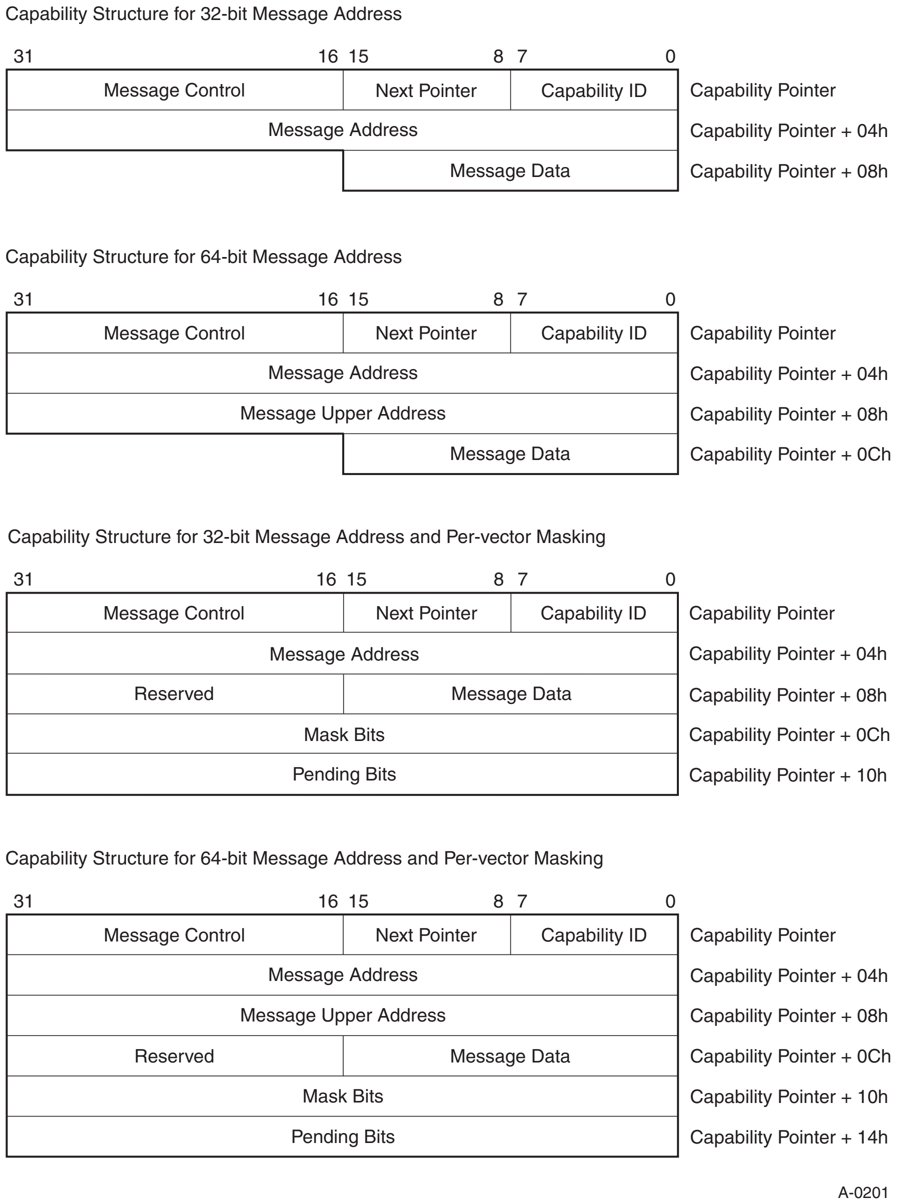


Figure 6-9: MSI Capability Structures

To request service, an MSI function writes the contents of the Message Data register to the address specified by the contents of the Message Address register (and, optionally, the Message Upper Address register for a 64-bit message address). A read of the address specified by the contents of the Message Address register produces undefined results.

A function supporting MSI implements one of four MSI Capability Structure layouts illustrated in Figure 6-9, depending upon which optional features are supported. If a function supports 64-bit addressing (DAC) when acting as a master, the function is required to implement 64-bit addressing.

The message control register indicates the function's capabilities and provides system software control over MSI.

Each field is further described in the following sections. Reserved registers and bits always return 0 when read and write operations have no effect. Read-only registers return valid data when read and write operations have no effect.

6.8.1.1. Capability ID for MSI

Bits	Field	Description
7::0	CAP_ID	The value of 05h in this field identifies the function as being MSI capable. This field is read only.

6.8.1.2. Next Pointer for MSI

Bits	Field	Description
7::0	NXT_PTR	Pointer to the next item in the capabilities list. Must be NULL for the final item in the list. This field is read only.

6.8.1.3. Message Control for MSI

This register provides system software control over MSI. After reset, MSI is disabled. If MSI and MSI-X are both disabled, the function requests servicing via its INTx# pin (if supported). System software can enable MSI by setting bit 0 of this register. System software is permitted to modify the Message Control register's read/write bits and fields. A device driver is not permitted to modify the Message Control register's read/write bits and fields.

Bits	Field	Description
15::09	Reserved	Always returns 0 on a read, and a write operation has no effect.
8	Per-vector masking capable	If 1, the function supports MSI per-vector masking. If 0, the function does not support MSI per-vector masking. This bit is read only.
7	64 bit address capable	If 1, the function is capable of sending a 64-bit message address. If 0, the function is not capable of sending a 64-bit message address. This bit is read only.

Bits	Field	Description
6::4	Multiple Message Enable	software writes to this field to indicate the number of allocated vectors (equal to or less than the number of requested vectors). The number of allocated vectors is aligned to a power of two. If a function requests four vectors (indicated by a Multiple Message Capable encoding of “010”), system software can allocate either four, two, or one vector by writing a “010”, “001, or “000” to this field, respectively. When MSI is enabled, a function will be allocated at least 1 vector. The encoding is defined as: **Encoding** **# of vectors allocated** 000 1 001 2 010 4 011 8 100 16 101 32 110 Reserved 111 Reserved This field’s state after reset is “000”. This field is read/write.
3::1	Multiple Message Capable	System software reads this field to determine the number of requested vectors. The number of requested vectors must be aligned to a power of two (if a function requires three vectors, it requests four by initializing this field to “010”). The encoding is defined as: **Encoding** **# of vectors requested** 000 1 001 2 010 4 011 8 100 16 101 32 110 Reserved 111 Reserved This field is read only.

Bits	Field	Description
0	MSI Enable	If 1 and the MSI-X Enable bit in the MSI-X Message Control register (see Section 6.8.2.3) is 0, the function is permitted to use MSI to request service and is prohibited from using its INTx# pin (if implemented; see Section 6.2.4 Interrupt pin register). System configuration software sets this bit to enable MSI. A device driver is prohibited from writing this bit to mask a function's service request. Refer to Section 6.2.2 for control of INTx#. If 0, the function is prohibited from using MSI to request service. This bit's state after reset is 0 (MSI is disabled). This bit is read/write.

6.8.1.4. Message Address for MSI

Bits	Field	Description
31::02	Message Address	System-specified message address. If the Message Enable bit (bit 0 of the Message Control register) is set, the contents of this register specify the DWORD-aligned address (AD[31::02]) for the MSI memory write transaction. AD[1::0] are driven to zero during the address phase. This field is read/write.
01::00	Reserved	Always returns 0 on read. Write operations have no effect.

6.8.1.5. Message Upper Address for MSI (Optional)

Bits	Field	Description
31::00	Message Upper Address	System-specified message upper address. This register is optional and is implemented only if the function supports a 64-bit message address (bit 7 in Message Control register set).[48] If the Message Enable bit (bit 0 of the Message Control register) is set, the contents of this register (if non-zero) specify the upper 32-bits of a 64-bit message address (AD[63::32]). If the contents of this register are zero, the function uses the 32 bit address specified by the message address register. This field is read/write.

---

[48] This register is required when the device supports 64-bit addressing (DAC) when acting as a master.

6.8.1.6. Message Data for MSI

Bits	Field	Description
15::00	Message Data	System-specified message data. If the Message Enable bit (bit 0 of the Message Control register) is set, the message data is driven onto the lower word (AD[15::00]) of the memory write transaction's data phase. AD[31::16] are driven to zero during the memory write transaction's data phase. C/BE[3::0]# are asserted during the data phase of the memory write transaction. The Multiple Message Enable field (bits 6-4 of the Message Control register) defines the number of low order message data bits the function is permitted to modify to generate its system software allocated vectors. For example, a Multiple Message Enable encoding of "010" indicates the function has been allocated four vectors and is permitted to modify message data bits 1 and 0 (a function modifies the lower message data bits to generate the allocated number of vectors). If the Multiple Message Enable field is "000", the function is not permitted to modify the message data. This field is read/write.

6.8.1.7. Mask Bits for MSI (Optional)

The Mask Bits and Pending Bits registers enable software to disable or defer message sending on a per-vector basis.

MSI vectors are numbered 0 through N-1, where N is the number of vectors allocated by software. Each vector is associated with a correspondingly numbered bit in the Mask Bits and Pending Bits registers.

The Multiple Message Capable field indicates how many vectors (with associated Mask and Pending bits) are implemented. All unimplemented Mask and Pending bits are reserved.

After reset, the state of all implemented Mask and Pending bits is 0 (no vectors are masked and no messages are pending).

Bits	Field	Description
31::00	Mask Bits	For each Mask bit that is set, the function is prohibited from sending the associated message. This field is read/write.

6.8.1.8. Pending Bits for MSI (Optional)

Bits	Field	Description
31::00	Pending Bits	For each Pending bit that is set, the function has a pending associated message. This field is read only.

6.8.2. MSI-X Capability and Table Structures

The MSI-X capability structure is illustrated in Figure 6-10. More than one MSI-X capability structure per function is prohibited, but a function is permitted to have both an MSI and an MSI-X capability structure.

In contrast to the MSI capability structure, which directly contains all of the control/status information for the function's vectors, the MSI-X capability structure instead points to an MSI-X Table structure and a MSI-X Pending Bit Array (PBA) structure, each residing in Memory Space.

Each structure is mapped by a Base Address register (BAR) belonging to the function, located beginning at 10h in Configuration Space. A BAR Indicator register (BIR) indicates which BAR, and a QWORD-aligned Offset indicates where the structure begins relative to the base address associated with the BAR. The BAR is permitted to be either 32-bit or 64-bit, but must map Memory Space. A function is permitted to map both structures with the same BAR, or to map each structure with a different BAR.

The MSI-X Table structure, illustrated in Figure 6-11, typically contains multiple entries, each consisting of several fields: Message Address, Message Upper Address, Message Data, and Vector Control. Each entry is capable of specifying a unique vector.

The Pending Bit Array (PBA) structure, illustrated in Figure 6-12, contains the function's Pending Bits, one per Table entry, organized as a packed array of bits within QWORDs. The last QWORD will not necessarily be fully populated.

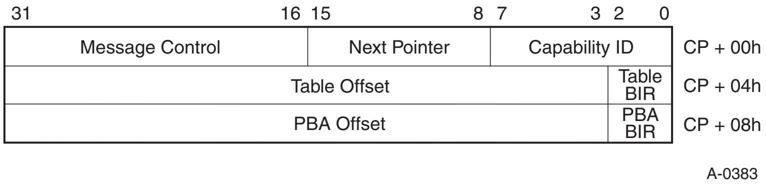


Figure 6-10: MSI-X Capability Structure

DWORD 3	DWORD 2	DWORD 1	DWORD 0		
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 0	Base
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 1	Base + 1*16
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 2	Base + 2*16
...	...	...	...	...	...
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry (N-1)	Base + (N-1)*16

A-0384

Figure 6-11: MSI-X Table Structure

63 0		
Pending Bits 0 through 63	QWORD 0	Base
Pending Bits 64 through 127	QWORD 1	Base + 1*8
...	...	...
Pending Bits ((N-1) div 64)*64 through N-1	QWORD ((N-1) div 64)	Base + ((N-1) div 64)*8

A-0385

Figure 6-12: MSI-X PBA Structure

To request service using a given MSI-X Table entry, a function performs a DWORD memory write transaction using the contents of the Message Data field entry for data, the contents of the Message Upper Address field for the upper 32 bits of address, and the contents of the Message Address field entry for the lower 32 bits of address. A memory read transaction from the address targeted by the MSI-X message produces undefined results.

If a Base Address register that maps address space for the MSI-X Table or MSI-X PBA also maps other usable address space that is not associated with MSI-X structures, locations (e.g., for CSRs) used in the other address space must not share any naturally aligned 4-KB address range with one where either MSI-X structure resides. This allows system software where applicable to use different processor attributes for MSI-X structures and the other address space. (Some processor architectures do not support having different processor attributes associated with the same naturally aligned 4-KB physical address range.) The MSI-X Table and MSI-X PBA are permitted to co-reside within a naturally aligned 4-KB address range, though they must not overlap with each other.

IMPLEMENTATION NOTE

Dedicated BARs and Address Range Isolation

To enable system software to map MSI-X structures onto different processor pages for improved access control, it is recommended that a function dedicate separate Base Address registers for the MSI-X Table and MSI-X PBA, or else provide more than the minimum required isolation with address ranges.

If dedicated separate Base Address registers is not feasible, it is recommended that a function dedicate a single Base Address register for the MSI-X Table and MSI-X PBA.

If a dedicated Base Address register is not feasible, it is recommended that a function isolate the MSI-X structures from the non-MSI-X structures with aligned 8 KB ranges rather than the mandatory aligned 4 KB ranges.

For example, if a Base Address register needs to map 2 KB for an MSI-X Table containing 128 entries, 16 bytes for an MSI-X PBA containing 128 bits, and 64 bytes for registers not related to MSI-X, the following is an *acceptable* implementation. The Base Address register requests 8 KB of total address space, maps the first 64 bytes for the non MSI-X registers, maps the MSI-X Table beginning at an offset of 4 KB, and maps the MSI-X PBA beginning at an offset of 6 KB.

A *preferable* implementation for a shared Base Address register is for it to request 16 KB of total address space, map the first 64 bytes for the non MSI-X registers, map the MSI-X Table beginning at an offset of 8 KB, and map the MSI-X PBA beginning at an offset of 12 KB.

IMPLEMENTATION NOTE

MSI-X Memory Space Structures in Read/Write Memory

The MSI-X Table and MSI-X PBA structures are defined such that they can reside in general purpose read/write memory on a device, for ease of implementation and added flexibility. To achieve this, none of the contained fields are required to be read-only, and there are also restrictions on transaction alignment and sizes.

For all accesses to MSI-X Table and MSI-X PBA fields, software must use aligned full DWORD or aligned full QWORD transactions; otherwise, the result is undefined.

MSI-X Table entries and Pending bits are each numbered 0 through N-1, where N-1 is indicated by the Table Size field in the MSI-X Message Control register. For a given arbitrary MSI-X Table entry *K*, its starting address can be calculated with the formula:

entry starting address = Table base + K*16

For the associated Pending bit K, its address for QWORD access and bit number within that QWORD can be calculated with the formulas:

QWORD address = PBA base + (K div[49] 64)*8

QWORD bit# = K mod[50] 64

Software that chooses to read Pending bit K with DWORD accesses can use these formulas:

DWORD address = PBA base + (K div 32)*4

DWORD bit# = K mod 32

Each field in the MSI-X capability, Table, and PBA structures is further described in the following sections. Within the MSI-X capability structure, reserved registers and bits always return 0 when read, and write operations have no effect. Read-only registers return valid data when read, and write operations have no effect. Within the MSI-X Table and PBA structures, reserved fields have special rules.

6.8.2.1. Capability ID for MSI-X

7::0	CAP_ID	The value of 11h in this field identifies the function as being MSI-X capable. This field is read only.

6.8.2.2. Next Pointer for MSI-X

7::0	NXT_PTR	Pointer to the next item in the capabilities list. Must be NULL for the final item in the list. This field is read only.

---

[49] Div is an integer divide with truncation.

[50] Mod is the remainder from an integer divide.

6.8.2.3. Message Control for MSI-X

After reset, MSI-X is disabled. If MSI and MSI-X are both disabled, the function requests servicing via its INTx# pin (if supported). System software can enable MSI-X by setting bit 15 of this register. System software is permitted to modify the Message Control register's read/write bits and fields. A device driver is not permitted to modify the Message Control register's read/write bits and fields.

Bits	Field	Description
15	MSI-X Enable	If 1 and the MSI Enable bit in the MSI Message Control register (see Section 6.8.1.3) is 0, the function is permitted to use MSI-X to request service and is prohibited from using its INTx# pin (if implemented; see Section 6.2.4). System configuration software sets this bit to enable MSI-X. A device driver is prohibited from writing this bit to mask a function's service request. If 0, the function is prohibited from using MSI-X to request service. This bit's state after reset is 0 (MSI-X is disabled). This bit is read/write.
14	Function Mask	If 1, all of the vectors associated with the function are masked, regardless of their per-vector Mask bit states. If 0, each vector's Mask bit determines whether the vector is masked or not. Setting or clearing the MSI-X Function Mask bit has no effect on the state of the per-vector Mask bits. This bit's state after reset is 0 (unmasked). This bit is read/write.
13::11	Reserved	Always returns 0 on a read, and a write operation has no effect.
10::00	Table Size	System software reads this field to determine the MSI-X Table Size N, which is encoded as N-1. For example, a returned value of "00000000011" indicates a table size of 4. This field is read only.

6.8.2.4. Table Offset/Table BIR for MSI-X

Bits	Field	Description
31::3	Table Offset	Used as an offset from the address contained by one of the function's Base Address registers to point to the base of the MSI-X Table. The lower 3 Table BIR bits are masked off (set to zero) by software to form a 32-bit QWORD-aligned offset. This field is read only.
2::0	Table BIR	Indicates which one of a function's Base Address registers, located beginning at 10h in Configuration Space, is used to map the function's MSI-X Table into Memory Space. **BIR Value** **Base Address register** 0 10h 1 14h 2 18h 3 1Ch 4 20h 5 24h 6 Reserved 7 Reserved For a 64-bit Base Address register, the Table BIR indicates the lower DWORD. With PCI-to-PCI bridges, BIR values 2 through 5 are also reserved. This field is read only.

6.8.2.5. PBA Offset/PBA BIR for MSI-X

Bits	Field	Description
31::3	PBA Offset	Used as an offset from the address contained by one of the function's Base Address registers to point to the base of the MSI-X PBA. The lower 3 PBA BIR bits are masked off (set to zero) by software to form a 32-bit QWORD-aligned offset. This field is read only.
2::0	PBA BIR	Indicates which one of a function's Base Address registers, located beginning at 10h in Configuration Space, is used to map the function's MSI-X PBA into Memory Space. The PBA BIR value definitions are identical to those for the MSI-X Table BIR. This field is read only.

6.8.2.6. Message Address for MSI-X Table Entries

Bits	Field	Description
31::02	Message Address	System-specified message lower address. For MSI-X messages, the contents of this field from an MSI-X Table entry specifies the lower portion of the DWORD-aligned address (AD[31::02]) for the memory write transaction. This field is read/write.
01:00	Message Address	For proper DWORD alignment, software must always write zeroes to these two bits; otherwise the result is undefined. The state of these bits after reset must be 0. These bits are permitted to be read only or read/write.

6.8.2.7. Message Upper Address for MSI-X Table Entries

Bits	Field	Description
31::00	Message Upper Address	System-specified message upper address bits. If this field is zero, Single Address Cycle (SAC) messages are used. If this field is non-zero, Dual Address Cycle (DAC) messages are used. This field is read/write.

6.8.2.8. Message Data for MSI-X Table Entries

Bits	Field	Description
31::00	Message Data	System-specified message data. For MSI-X messages, the contents of this field from an MSI-X Table entry specifies the data driven on AD[31::00] during the memory write transaction's data phase. C/BE[3::0]# are asserted during the data phase of the memory write transaction. In contrast to message data used for MSI messages, the low-order message data bits in MSI-X messages are not modified by the function. This field is read/write.

6.8.2.9. Vector Control for MSI-X Table Entries

Bits	Field	Description
31::01	Reserved	After reset, the state of these bits must be 0. However, for potential future use, software must preserve the value of these reserved bits when modifying the value of other Vector Control bits. If software modifies the value of these reserved bits, the result is undefined.
00	Mask Bit	When this bit is set, the function is prohibited from sending a message using this MSI-X Table entry. However, any other MSI-X Table entries programmed with the same vector will still be capable of sending an equivalent message unless they are also masked. This bit's state after reset is 1 (entry is masked). This bit is read/write.

6.8.2.10. Pending Bits for MSI-X PBA Entries

Bits	Field	Description
63::00	Pending Bits	For each Pending Bit that is set, the function has a pending message for the associated MSI-X Table entry. Pending bits that have no associated MSI-X Table entry are reserved. After reset, the state of reserved Pending bits must be 0. Software should never write, and should only read Pending Bits. If software writes to Pending Bits, the result is undefined. Each Pending Bit's state after reset is 0 (no message pending). These bits are permitted to be read only or read/write.

6.8.3. MSI and MSI-X Operation

At configuration time, system software traverses the function's capability list. If a capability ID of 05h is found, the function implements MSI. If a capability ID of 11h is found, the function implements MSI-X. A given function is permitted to implement MSI alone, MSI-X alone, both, or neither. Within a device, different functions are permitted to implement different sets of these interrupt mechanisms, and system software manages each function's interrupt mechanisms independently.

6.8.3.1. MSI Configuration

In this section, all register and field references are in the context of the MSI capability structure.

System software reads the Message Control register to determine the function's MSI capabilities.

System software reads the Multiple Message Capable field (bits 3-1 of the Message Control register) to determine the number of requested vectors. MSI supports a maximum of 32 vectors per function. System software writes to the Multiple Message Enable field (bits 6-4 of the Message Control register) to allocate either all or a subset of the requested vectors. For example, a function can request four vectors and be allocated either four, two, or one vector. The number of vectors requested and allocated is aligned to a power of two (a function that requires three vectors must request four).

If the Per-vector Masking Capable bit (bit 8 of the Message Control register) is set and system software supports per-vector masking, system software may mask one or more vectors by writing to the Mask Bits register.

If the 64-bit Address Capable bit (bit 7 of the Message Control register) is set, system software initializes the MSI capability structure's Message Address register (specifying the lower 32 bits of the message address) and the Message Upper Address register (specifying the upper 32 bits of the message address) with a system-specified message address. System software may program the Message Upper Address register to zero so that the function uses a 32-bit address for the MSI write transaction. If this bit is clear, system software initializes the MSI capability structure's Message Address register (specifying a 32-bit message address) with a system specified message address.

System software initializes the MSI capability structure's Message Data register with a system specified data value. Care must be taken to initialize only the Message Data register (i.e., a 2-byte value) and not modify the upper two bytes of that DWORD location.

6.8.3.2. MSI-X Configuration

In this section, all register and field references are in the context of the MSI-X capability, MSI-X Table, and MSI-X PBA structures.

System software allocates address space for the function's standard set of Base Address registers and sets the registers accordingly. One of the function's Base Address registers includes address space for the MSI-X Table, though the system software that allocates address space does not need to be aware of which Base Address register this is, or the fact the address space is used for the MSI-X Table. The same or another Base Address register includes address space for the MSI-X PBA, and the same point regarding system software applies.

Depending upon system software policy, system software, device driver software, or each at different times or environments may configure a function's MSI-X capability and table structures with suitable vectors. For example, a booting environment will likely require only a single vector, whereas a normal operating system environment for running applications may benefit from multiple vectors if the function supports an MSI-X Table with multiple entries. For the remainder of this section, "software" refers to either system software or device driver software.

Software reads the Table Size field from the Message Control register to determine the MSI-X Table size. The field encodes the number of table entries as N-1, so software must add 1 to the value read from the field to calculate the number of table entries N. MSI-X supports a maximum table size of 2048 entries.

Software calculates the base address of the MSI-X Table by reading the 32-bit value from the Table Offset/Table BIR register, masking off the lower 3 Table BIR bits, and adding the remaining QWORD-aligned 32-bit Table offset to the address taken from the Base Address register indicated by the Table BIR. Software calculates the base address of the MSI-X PBA using the same process with the PBA Offset/PBA BIR register.

For each MSI-X Table entry that will be used, software fills in the Message Address field, Message Upper Address field, Message Data field, and Vector Control Field. Software must not modify the Address or Data fields of an entry while it is unmasked. Refer to Section 6.8.3.5 for details.

IMPLEMENTATION NOTE

Special Considerations for QWORD Accesses

Software is permitted to fill in MSI-X Table entry DWORD fields individually with DWORD writes, or software in certain cases is permitted to fill in appropriate pairs of DWORDs with a single QWORD write. Specifically, software is always permitted to fill in the Message Address and Message Upper Address fields with a single QWORD write. If a given entry is currently masked (via its Mask bit or the Function Mask bit), software is permitted to fill in the Message Data and Vector Control fields with a single QWORD write, taking advantage of the fact the Message Data field is guaranteed to become visible to hardware no later than the Vector Control field. However, if software wishes to mask a currently unmasked entry (without setting the Function Mask bit), software must set the entry's Mask bit using a DWORD write to the Vector Control field, since performing a QWORD write to the Message Data and Vector Control fields might result in the Message Data field being modified before the Mask bit in the Vector Control field becomes set.

For potential use by future specifications, the Reserved bits in the Vector Control field must have their values after reset preserved by software. If software does not preserve their values, the result is undefined.

For each MSI-X Table entry that software chooses not to configure for generating messages, software can simply leave the entry in its default state of being masked.

Software is permitted to configure multiple MSI-X Table entries with the same vector, and this may indeed be necessary when fewer vectors are allocated than requested.

IMPLEMENTATION NOTE

Handling MSI-X Vector Shortages

For the case where fewer vectors are allocated to a function than desired, software-controlled aliasing as enabled by MSI-X is one approach for handling the situation. For example, if a function supports five queues, each with an associated MSI-X table entry, but only three vectors are allocated, the function could be designed for software still to configure all five table entries, assigning one or more vectors to multiple table entries. Software could assign the three vectors {A,B,C} to the five entries as ABCCC, ABBCC, ABCBA, or other similar combinations.

Alternatively, the function could be designed for software to configure it (using a device-specific mechanism) to use only three queues and three MSI-X table entries. Software could assign the three vectors {A,B,C} to the five entries as ABC--, A-B-C, A--CB, or other similar combinations.

6.8.3.3. *Enabling Operation*

To maintain backward compatibility, the MSI Enable bit in the MSI Message Control register and the MSI-X Enable bit in the MSI-X Message Control register are each cleared after reset (MSI and MSI-X are both disabled). System configuration software sets one of these bits to enable either MSI or MSI-X, but never both simultaneously. Behavior is undefined if both MSI and MSI-X are enabled simultaneously. A device driver is prohibited from writing this bit to mask a function's service request. While enabled for MSI or MSI-X operation, a function is prohibited from using its INTx# pin (if implemented) to request service (MSI, MSI-X, and INTx# are mutually exclusive).

6.8.3.4. *Sending Messages*

Once MSI or MSI-X is enabled (the appropriate bit in one of the Message Control registers is set), and one or more vectors is unmasked, the function is permitted to send messages. To send a message, a function does a DWORD memory write to the appropriate message address with the appropriate message data.

For MSI, the DWORD that is written is made up of the value in the MSI Message Data register in the lower two bytes and zeroes in the upper two bytes.

For MSI, if the Multiple Message Enable field (bits 6-4 of the MSI Message Control register) is non-zero, the function is permitted to modify the low order bits of the message data to generate multiple vectors. For example, a Multiple Message Enable encoding of "010" indicates the function is permitted to modify message data bits 1 and 0 to generate up to four unique vectors. If the Multiple Message Enable field is "000", the function is not permitted to modify the message data.

For MSI-X, the MSI-X Table contains at least one entry for every allocated vector, and the 32-bit Message Data field value from a selected table entry is used in the message without any modification to the low-order bits by the function.

How a function uses multiple vectors (when allocated) is device dependent. A function must handle being allocated fewer vectors than requested.

6.8.3.5. *Per-vector Masking and Function Masking*

Per-vector masking is an optional feature with MSI, and a standard feature in MSI-X.

Function Masking is a standard feature in MSI-X. When the MSI-X Function Mask bit is set, all of the function's entries must behave as being masked, regardless of the per-entry Mask bit states. Function Masking is not supported in MSI, but software can readily achieve a similar effect by setting all MSI Mask bits using a single DWORD write.

"Per-vector masking" in MSI-X is controlled by a Mask bit in each MSI-X Table entry. While more accurately termed "per-entry masking", masking an MSI-X Table entry is still referred to as "vector masking" so similar descriptions can be used for both MSI and MSI-X. However, since software is permitted to program the same vector (a unique Address/Data pair) into multiple MSI-X table entries, all such entries must be masked in order to guarantee the function will not send a message using that Address/Data pair.

For MSI and MSI-X, while a vector is masked, the function is prohibited from sending the associated message, and the function must set the associated Pending bit whenever the function would otherwise send the message. When software unmasks a vector whose associated Pending bit is set, the function must schedule sending the associated message, and clear the Pending bit as soon as the message has been sent. Note that clearing the MSI-X Function Mask bit may result in many messages needing to be sent.

If a masked vector has its Pending bit set, and the associated underlying interrupt events are somehow satisfied (usually by software though the exact manner is function-specific), the function must clear the Pending bit, to avoid sending a spurious interrupt message later when software unmasks the vector. However, if a subsequent interrupt event occurs while the vector is still masked, the function must again set the Pending bit.

Software is permitted to mask one or more vectors indefinitely, and service their associated interrupt events strictly based on polling their Pending bits. A function must set and clear its Pending bits as necessary to support this “pure polling” mode of operation.

For MSI-X, a function is permitted to cache Address and Data values from unmasked MSI-X Table entries. However, anytime software unmasks a currently masked MSI-X Table entry either by clearing its Mask bit or by clearing the Function Mask bit, the function must update any Address or Data values that it cached from that entry. If software changes the Address or Data value of an entry while the entry is unmasked, the result is undefined.

6.8.3.6. Hardware/Software Synchronization

If a function sends messages with the same vector multiple times before being acknowledged by software, only one message is guaranteed to be serviced. If all messages must be serviced, a device driver handshake is required. In other words, once a function sends Vector A, it cannot send Vector A again until it is explicitly enabled to do so by its device driver (provided all messages must be serviced). If some messages can be lost, a device driver handshake is not required. For functions that support multiple vectors, a function can send multiple unique vectors and is guaranteed that each unique message will be serviced. For example, a function can send Vector A followed by Vector B without any device driver handshake (both Vector A and Vector B will be serviced).

IMPLEMENTATION NOTE

Servicing MSI and MSI-X Interrupts

When system software allocates fewer MSI or MSI-X vectors to a function than it requests, multiple interrupt sources within the function, each desiring a unique vector, may be required to share a single vector. Without proper handshakes between hardware and software, hardware may send fewer messages than software expects, or hardware may send what software considers to be extraneous messages.

A rather sophisticated but resource-intensive approach is to associate a dedicated event queue with each allocated vector, with producer and consumer pointers for managing each event queue. Such event queues typically reside in host memory. The function acts as the producer and software acts as the consumer. Multiple interrupt sources within a function may be assigned to each event queue as necessary. Each time an interrupt source needs to signal an interrupt, the function places an entry on the appropriate event queue (assuming there's room), updates a copy of the producer pointer (typically in host memory), and sends an interrupt message with the associated vector when necessary to notify software that the event queue needs servicing. The interrupt service routine for a given event queue processes all entries it finds on its event queue, as indicated by the producer pointer. Each event queue entry identifies the interrupt source and possibly additional information about the nature of the event. The use of event queues and producer/consumer pointers can be used to guarantee that interrupt events won't get dropped when multiple interrupt sources are forced to share a vector. There's no need for additional handshaking between sending multiple messages associated with the same event queue, to guarantee that every message gets serviced. In fact, various standard techniques for "interrupt coalescing" can be used to avoid sending a separate message for every event that occurs, particularly during heavy bursts of events.

In more modest implementations, the hardware design of a function's MSI or MSI-X logic sends a message any time a falling edge would have occurred on the INTx# pin if MSI or MSI-X had not been enabled. For example, consider a scenario in which two interrupt events (possibly from distinct interrupt sources within a function) occur in rapid succession. The first event causes a message to be sent. Before the interrupt service routine has had an opportunity to service the first event, the second event occurs. In this case, only one message is sent, because the first event is still active at the time the second event occurs (a hardware INTx# pin signal would have had only one falling edge).

One handshake approach for implementations like the above is to use standard per-vector masking, and allow multiple interrupt sources to be associated with each vector. A given vector's interrupt service routine sets the vector's Mask bit before it services any associated interrupting events and clears the Mask bit after it has serviced all the events it knows about. (This could be any number of events.) Any occurrence of a new event while the Mask bit is set results in the Pending bit being set. If one or more associated events are still pending at the time the vector's Mask bit is cleared, the function immediately sends another message.

A handshake approach for MSI functions that do not implement per-vector masking is for a vector's interrupt service routine to re-inspect all of the associated interrupt events after clearing what is presumed to be the last pending interrupt event. If another event is found to be active, it is serviced in the same interrupt service routine invocation, and the complete re-inspection is repeated until no pending events are found. This ensures that if an additional interrupting event occurs before a previous interrupt event is cleared, whereby the function does *not* send an additional interrupt message, that the new event is serviced as part of the current interrupt service routine invocation.

This alternative has the potential side effect of one vector's interrupt service routine processing an interrupting event that has already generated a new interrupt message. The interrupt service routine invocation resulting from the new message may find no pending interrupt events. Such occurrences are sometimes referred to as spurious interrupts, and software using this approach must be prepared to tolerate them.

An MSI or MSI-X message, by virtue of being a posted memory write (PMW) transaction, is prohibited by PCI ordering rules from passing PMW transactions sent earlier by the function. The system must guarantee that an interrupt service routine invoked as a result of a given message will observe any updates performed by PMW transactions arriving prior to that message. Thus, the interrupt service routine of a device driver is not required to read from a device register in order to ensure data consistency with previous PMW transactions. However, if multiple MSI-X Table entries share the same vector, the interrupt service routine may need to read from some device specific register to determine which interrupt sources need servicing.

6.8.3.7. Message Transaction Termination

The target of an MSI or MSI-X write transaction cannot distinguish between it and any other memory write transaction. The termination requirements for an MSI or MSI-X write transaction are the same as for any other memory write transaction except as noted below.

If the MSI or MSI-X write transaction is terminated with a Master-Abort or a Target-Abort, the master that originated the MSI or MSI-X memory write transaction is required to report the error by asserting SERR# (if bit 8 in the Command register is set) and to set the appropriate bits in the Status register (refer to Section 3.7.4.2). An MSI or MSI-X memory write transaction is ignored by the target if it is terminated with a Master-Abort or Target-Abort.

Refer to the *PCI-to-PCI Bridge Architecture Specification, Revision* 1.1 (Section 6, Error Support) for PCI-to-PCI bridge SERR# generation in response to error conditions from posted memory writes on the destination bus.

Note that SERR# generation in an MSI-enabled environment containing PCI-to-PCI bridges requires the SERR# reporting enable bits in all devices in the MSI message path to be set. For PCI-to-PCI bridges specifically, refer to Section 6.2.2 and *PCI-to-PCI Bridge Architecture Specification, Revision* 1.1, Sections 3.2.4.3 and 3.2.5.17.).

If the MSI or MSI-X write transaction results in a data parity error, the master that originated the MSI or MSI-X write transaction is required to assert SERR# (if bit 8 in the Command register is set) and to set the appropriated bits in the Status register (refer to Section 3.7.4).

6.8.3.8. Message Transaction Reception and Ordering Requirements

As with all memory write transactions, the device that includes the target of the interrupt message (the interrupt receiver) is required to complete all interrupt message transactions as a target without requiring other transactions to complete first as a master. (Refer to Section 3.3.3.3.4. In general, this means that the message receiver must complete the interrupt message transaction independent of when the CPU services the interrupt. For example, each time the interrupt receiver receives an interrupt message, it could set a bit in an internal register indicating that this message had been received and then complete the transaction on the bus. The appropriate interrupt service routine would later be dispatched because this bit was set. The message receiver would not be allowed to delay the completion of the interrupt message on the bus pending acknowledgement from the processor that the interrupt was being serviced. Such dependencies can lead to deadlock when multiple devices send interrupt messages simultaneously.

Although interrupt messages remain strictly ordered throughout the PCI bus hierarchy, the order of receipt of the interrupt messages does not guarantee any order in which the interrupts will be serviced. Since the message receiver must complete all interrupt message transactions without regard to when the interrupt was actually serviced, the message receiver will generally not maintain any information about the order in which the interrupts were received. This is true both of interrupt messages received from different devices and multiple messages received from the same device. If a device requires one interrupt message to be serviced before another, the device must not send the second interrupt message until the first one has been serviced.

7

7. 66 MHz PCI Specification

7.1. Introduction

The 66 MHz PCI bus is a compatible superset of PCI defined to operate up to a maximum clock speed of 66 MHz. The 66 MHz PCI bus is intended to be used by low latency, high bandwidth bridges, and peripherals. Systems may augment the 66 MHz PCI bus with a separate 33 MHz PCI bus to handle lower speed peripherals.

Differences between 33 MHz PCI and 66 MHz PCI are minimal. Both share the same protocol, signal definitions, and connector layout. To identify 66 MHz PCI devices, one static signal is added by redefining an existing ground pin, and one bit is added to the Configuration Status register. Bus drivers for the 66 MHz PCI bus meet the same DC characteristics and AC drive point limits as 33 MHz PCI bus drivers; however, 66 MHz PCI requires faster timing parameters and redefined measurement conditions. As a result, 66 MHz PCI buses may support smaller loading and trace lengths.

A 66 MHz PCI device operates as a 33 MHz PCI device when it is connected to a 33 MHz PCI bus. Similarly, if any 33 MHz PCI devices are connected to a 66 MHz PCI bus, the 66 MHz PCI bus will operate as a 33 MHz PCI bus.

The programming models for 66 MHz PCI and 33 MHz PCI are the same, including configuration headers and class types. Agents and bridges include a 66 MHz PCI status bit.

7.2. Scope

This chapter defines aspects of 66 MHz PCI that differ from those defined elsewhere in this document, including information on device and bridge support. This chapter will not repeat information defined elsewhere.

7.3. Device Implementation Considerations

7.3.1. Configuration Space

Identification of a 66 MHz PCI-compliant device is accomplished through the use of the read-only 66MHZ_CAPABLE flag located in bit 5 of the PCI Status register (see Figure 6-3). If set, this bit signifies that the device is capable of operating in 66 MHz mode.

7.4. Agent Architecture

A 66 MHz PCI agent is defined as a PCI agent capable of supporting 66 MHz PCI.

All 66 MHz PCI agents must support a read-only 66MHZ_CAPABLE flag located in bit 5 of the PCI Status register for that agent. If set, the 66MHZ_CAPABLE bit signifies that the agent can operate in 66 MHz PCI mode.[51]

7.5. Protocol

7.5.1. 66MHZ_ENABLE (M66EN) Pin Definition

Pin 49B on the PCI connector is designated M66EN. Add-in cards and bus segments not capable of operation in 66 MHz mode must connect this pin to ground. A 66 MHz PCI system board segment must provide a single pullup resistor to V_{CC} on the M66EN pin. Refer to Section 7.7.7 for the appropriate pullup value. M66EN is bused to all 66 MHz PCI connectors and system board-only 66 MHz PCI components that include the M66EN pin. The 66 MHz PCI clock generation circuitry must connect to M66EN to generate the appropriate clock for the segment (33 to 66 MHz if M66EN is asserted, 0 to 33 MHz if M66EN is deasserted).

If a 66 MHz PCI agent requires clock speed information (for example, for a PLL bypass), it is permitted to use M66EN as an input. If a 66 MHz PCI agent can run without any knowledge of the speed, it is permitted to leave M66EN disconnected.

Table 7-1: Bus and Agent Combinations

Bus 66MHZ_CAPABLE[52]	Agent 66MHZ_CAPABLE	Description
0	0	33 MHz PCI agent located on a 33 MHz PCI bus
0	1	66 MHz PCI agent located on a 33 MHz PCI bus[53]
1	0	33 MHz PCI agent located on a 66 MHz PCI bus[52]
1	1	66 MHz PCI agent located on a 66 MHz PCI bus

---

[51] Configuration software identifies agent capabilities by checking the 66MHZ_CAPABLE bit in the Status register. This includes both the primary and secondary Status registers in a PCI-to-PCI bridge. This allows configuration software to detect a 33 MHz PCI agent on a 66 MHz PCI bus or a 66 MHz PCI agent on a 33 MHz PCI bus and issue a warning to the user describing the situation.

[52] The bus 66MHZ_CAPABLE status bit is located in a bridge's Status registers.

[53] This condition may cause the configuration software to generate a warning to the user stating that the add-in card is installed in an inappropriate socket and should be relocated.

7.5.2. Latency

The 66 MHz PCI bus is intended for low latency devices. It is required that the target initial latency not exceed 16 clocks.

7.6. Electrical Specification

7.6.1. Overview

This chapter defines the electrical characteristics and constraints of 66 MHz PCI components, systems, and add-in cards, including connector pin assignments.

All electrical specifications from Chapter 4 of this document apply to 66 MHz PCI except where explicitly superseded. Specifically:

- ☐ The 66 MHz PCI bus uses the 3.3V signaling environment.
- ☐ Timing parameters have been scaled to 66 MHz.
- ☐ AC test loading conditions have been changed.

7.6.2. Transition Roadmap to 66 MHz PCI

The 66 MHz PCI bus utilizes the PCI bus protocol; 66 MHz PCI simply has a higher maximum bus clock frequency. Both 66 MHz and 33 MHz devices can coexist on the same bus segment. In this case, the bus segment will operate as a 33 MHz segment.

To ensure compatibility, 66 MHz PCI devices have the same DC specifications and AC drive point limits as 33 MHz PCI devices. However, 66 MHz PCI requires modified timing parameters as described in the analysis of the timing budget shown in Figure 7-1.

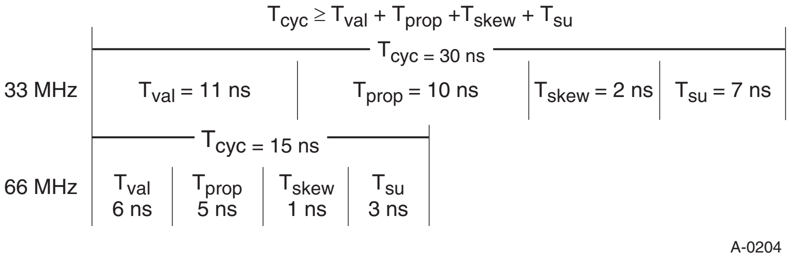


Figure 7-1: 33 MHz PCI vs. 66 MHz PCI Timing

Since AC drive requirements are the same for 66 MHz PCI and 33 MHz PCI, it is expected that 66 MHz PCI devices will function on 33 MHz PCI buses. Therefore, 66 MHz PCI devices must meet both 66 MHz PCI and 33 MHz PCI requirements.

7.6.3. Signaling Environment

A 66 MHz PCI system board segment must use the PCI 3.3V keyed connector. Therefore, 66 MHz PCI system board segments accept either 3.3V or Universal add-in cards; 5V add-in cards are not supported.

While 33 MHz PCI bus drivers are defined by their V/I curves, 66 MHz PCI output buffers are specified in terms of their AC and DC drive points, timing parameters, and slew rate. The minimum AC drive point defines an acceptable first step voltage and must be reached within the maximum T_{val} time. The maximum AC drive point limits the amount of overshoot and undershoot in the system. The DC drive point specifies steady state conditions. The minimum slew rate and the timing parameters guarantee 66 MHz operation. The maximum slew rate minimizes system noise. This method of specification provides a more concise definition for the output buffer.

7.6.3.1. DC Specifications

Refer to Section 4.2.2.1.

7.6.3.2. AC Specifications

Table 7-2: AC Specifications

Symbol	Parameter	Condition	Min	Max	Units	Notes
AC Drive Points						
$I_{oh(AC,min)}$	Switching Current High, minimum	$V_{out} = 0.3V_{cc}$	$-12V_{cc}$	-	mA	1
$I_{oh(AC,max)}$	Switching Current High, maximum	$V_{out} = 0.7V_{cc}$	-	$-32V_{cc}$	mA	
$I_{ol(AC,min)}$	Switching Current Low, minimum	$V_{out} = 0.6V_{cc}$	$16V_{cc}$	-	mA	1
$I_{ol(AC,max)}$	Switching Current Low, maximum	$V_{out} = 0.18V_{cc}$	-	$38V_{cc}$	mA	
DC Drive Points						
V_{OH}	Output high voltage	$I_{out} = -0.5$ mA	$0.9V_{cc}$	-	V	2
V_{OL}	Output low voltage	$I_{out} = 1.5$ mA	-	$0.1V_{cc}$	V	2
Slew Rate						
t_r	Output rise slew rate	$0.3V_{cc}$ to $0.6V_{cc}$	1	4	V/ns	3
t_f	Output fall slew rate	$0.6V_{cc}$ to $0.3V_{cc}$	1	4	V/ns	3

Symbol	Parameter	Condition	Min	Max	Units	Notes
Clamp Current						
I_{ch}	High clamp current	$V_{cc} + 4 > V_{in} \geq V_{cc} + 1$	$25 + (V_{in} - V_{cc} - 1) / 0.015$	-	mA	
I_{cl}	Low clamp current	$-3 < V_{in} \leq -1$	$-25 + (V_{in} + 1) / 0.015$	-	mA	

NOTES:

1. Switching current characteristics for REQ# and GNT# are permitted to be one half of that specified here; i.e., half size drivers may be used on these signals. This specification does not apply to CLK and RST# which are system outputs. "Switching Current High" specifications are not relevant to SERR#, PME#, INTA#, INTB#, INTC#, and INTD# which are open drain outputs.
2. These DC values are duplicated from Section 4.2.2.1 and are included here for completeness.
3. This parameter is to be interpreted as the cumulative edge rate across the specified range rather than the instantaneous rate at any point within the transition range. The specified load (see Figure 7-7) is optional. The designer may elect to meet this parameter with an unloaded output . However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline). Rise slew rate does not apply to open drain outputs.

7.6.3.3. Maximum AC Ratings and Device Protection

Refer to Section 4.2.2.3.

7.6.4. Timing Specification

7.6.4.1. Clock Specification

The clock waveform must be delivered to each 66 MHz PCI component in the system. In the case of add-in cards, compliance with the clock specification is measured at the add-in card component, not at the connector. Figure 7-2 shows the clock waveform and required measurement points for 3.3V signaling environments. Refer to item 8 in Section 3.10 for special considerations when using PCI-to-PCI bridges on add-in cards, or when add-in card slots are located downstream of a PCI-to-PCI bridge. Table 7-3 summarizes the clock specifications.

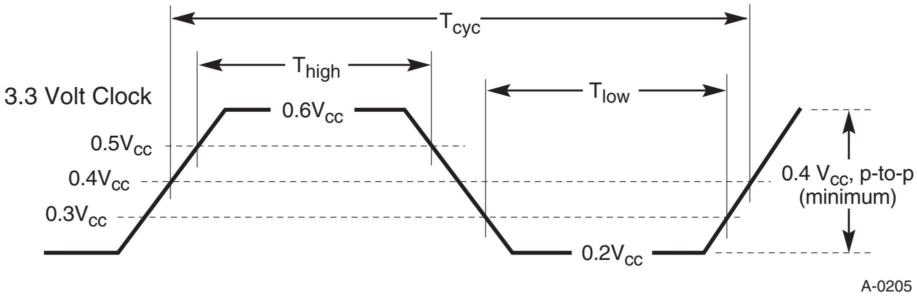


Figure 7-2: 3.3V Clock Waveform

Spread spectrum modulation techniques are permitted within the limits specified in Table 7-3.

Table 7-3: Clock Specifications

		66 MHz		33 MHz[4]			
Symbol	**Parameter**	**Min**	**Max**	**Min**	**Max**	**Units**	**Notes**
T_{cyc}	CLK Cycle Time	15	30	30	∞	ns	1,3
T_{high}	CLK High Time	6		11		ns	
T_{low}	CLK Low Time	6		11		ns	
-	CLK Slew Rate	1.5	4	1	4	V/ns	2
Spread Spectrum Requirements							
f_{mod}	modulation frequency	30	33	-	-	kHz	
f_{spread}	frequency spread	-1	0			%	

Notes:

1. In general, all 66 MHz PCI components must work with any clock frequency up to 66 MHz. CLK requirements vary depending upon whether the clock frequency is above 33 MHz.
    a. Device operational parameters at frequencies at or under 33 MHz will conform to the specifications in Chapter 4. The clock frequency may be changed at any time during the operation of the system so long as the clock edges remain "clean" (monotonic) and the minimum cycle and high and low times are not violated. The clock may only be stopped in a low state. A variance on this specification is allowed for components designed for use on the system board only. Refer to Section 4.2.3.1 for more information.
    b. For clock frequencies between 33 MHz and 66 MHz, the clock frequency may not change except while RST# is asserted or when spread spectrum clocking (SSC) is used to reduce EMI emissions.
2. Rise and fall times are specified in terms of the edge rate measured in V/ns. This slew rate must be met across the minimum peak-to-peak portion of the clock waveform as shown in Figure 7-2. Clock slew rate is measured by the slew rate circuit shown in Figure 7-7.
3. The minimum clock period must not be violated for any single clock cycle; i.e., accounting for all system jitter.
4. These values are duplicated from Section 4.2.3.1 and included here for comparison.

IMPLEMENTATION NOTE

Spread Spectrum Clocking (SSC)

Spreading the frequency is only allowed below the maximum clock frequency that is specified (i.e., the minimum clock period shown in Table 7-3 cannot be violated). In other words, the frequency change can only occur to reduce the frequency of the clock and never to increase it. When PLLs are used to track CLK, they need to track the SSC modulation quickly in order not to accumulate excessive phase difference between the PLL input and output clocks (commonly referred to as SSC tracking skew). The amount of tracking skew depends on the PLL bandwidth, phase angle at 30-33 kHz, and the amount of the spread. It is desirable to maximize bandwidth and/or reduce the phase angle in order to minimize the tracking skew; otherwise, the SSC frequency spread of the system must be reduced, thereby, reducing the SSC EMI reduction capability.

7.6.4.2. Timing Parameters

Table 7-4: 66 MHz and 33 MHz Timing Parameters

		66 MHz		33 MHz[7]			
Symbol	**Parameter**	**Min**	**Max**	**Min**	**Max**	**Units**	**Notes**
T_{val}	CLK to Signal Valid Delay - bused signals	2	6	2	11	ns	1, 2, 3, 8
T_{val}(ptp)	CLK to Signal Valid Delay - point to point signals	2	6	2	12	ns	1, 2, 3, 8
T_{on}	Float to Active Delay	2		2		ns	1, 8, 9
T_{off}	Active to Float Delay		14		28	ns	1, 9
T_{su}	Input Setup Time to CLK - bused signals	3		7		ns	3, 4, 10
T_{su}(ptp)	Input Setup Time to CLK - point to point signals	5		10,12		ns	3, 4
T_h	Input Hold Time from CLK	0		0		ns	4
T_{rst}	Reset Active Time after power stable	1		1		ms	5
$T_{rst\text{-}clk}$	Reset Active Time after CLK stable	100		100		μs	5
$T_{rst\text{-}off}$	Reset Active to output float delay		40		40	ns	5, 6
t_{rrsu}	REQ64# to RST# setup time	$10T_{cyc}$		$10T_{cyc}$		ns	

		66 MHz		33 MHz[7]			
Symbol	**Parameter**	**Min**	**Max**	**Min**	**Max**	**Units**	**Notes**
t_{rrh}	RST# to REQ64# hold time	0	50	0	50	ns	
T_{rhfa}	RST# high to first Configuration access	2^{25}		2^{25}		clocks	
T_{rhff}	RST# high to first FRAME# assertion	5		5		clocks	

Notes:

1. See the timing measurement conditions in Figure 7-3. It is important that all driven signal transitions drive to their V_{oh} or V_{ol} level within one T_{cyc}.
2. Minimum times are measured at the package pin with the load circuit shown in Figure 7-7. Maximum times are measured with the load circuit shown in Figures 7-5 and 7-6.
3. REQ# and GNT# are point-to-point signals and have different input setup times than do bused signals. GNT# and REQ# have a setup of 5 ns at 66 MHz. All other signals are bused.
4. See the timing measurement conditions in Figure 7-4.
5. If M66EN is asserted, CLK is stable when it meets the requirements in Section 7.6.4.1. RST# is asserted and deasserted asynchronously with respect to CLK. Refer to Section 4.3.2 for more information.
6. All output drivers must be floated when RST# is active. Refer to Section 4.3.2 for more information.
7. These values are duplicated from Section 4.2.3.2 and are included here for comparison.
8. When M66EN is asserted, the minimum specification for T_{val}(min), T_{val}(ptp)(min), and T_{on} may be reduced to 1 ns if a mechanism is provided to guarantee a minimum value of 2 ns when M66EN is deasserted.
9. For purposes of Active/Float timing measurements, the Hi-Z or “off” state is defined to be when the total current delivered through the component pin is less than or equal to the leakage current specification.
10. Setup time applies only when the device is not driving the pin. Devices cannot drive and receive signals at the same time. Refer to Section 3.10, item 9 for additional details.

7.6.4.3. Measurement and Test Conditions

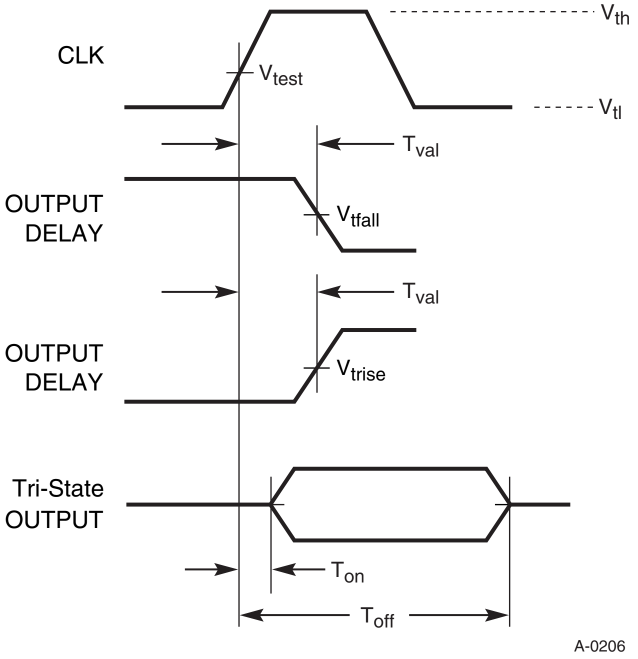


Figure 7-3: Output Timing Measurement Conditions

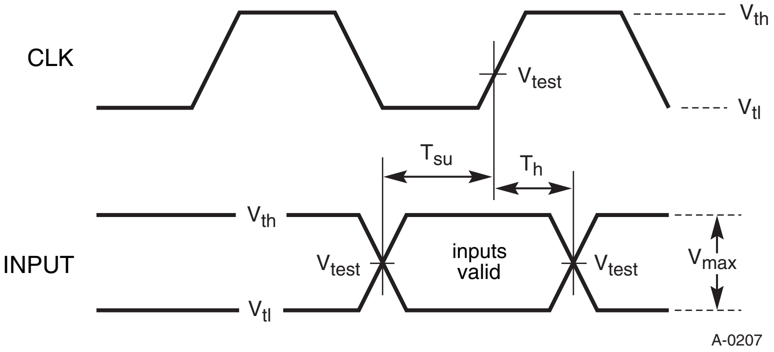


Figure 7-4: Input Timing Measurement Conditions

Table 7-5: Measurement Condition Parameters

Symbol	3.3V Signaling	Units	Notes
V_{th}	$0.6V_{cc}$	V	1
V_{tl}	$0.2V_{cc}$	V	1
V_{test}	$0.4V_{cc}$	V	
V_{trise}	$0.285V_{cc}$	V	2
V_{tfall}	$0.615V_{cc}$	V	2
V_{max}	$0.4V_{cc}$	V	1
Input Signal Slew Rate	1.5	V/ns	3

Notes:

1. The test for the 3.3V environment is done with $0.1 * V_{cc}$ of overdrive. V_{max} specifies the maximum peak-to-peak waveform allowed for measuring input timing. Production testing may use different voltage values but must correlate results back to these parameters.
2. V_{trise} and V_{tfall} are reference voltages for timing measurements only. Developers of 66 MHz PCI systems need to design buffers that launch enough energy into a 25 Ω transmission line so that correct input levels are guaranteed after the first reflection.
3. Outputs will be characterized and measured at the package pin with the load shown in Figure 7-7. Input signal slew rate will be measured between $0.2V_{cc}$ and $0.6V_{cc}$.

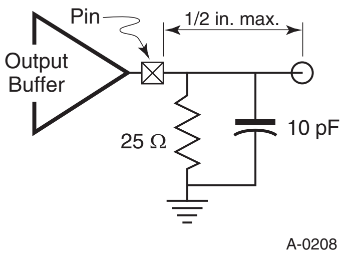


Figure 7-5: T_{val}(max) Rising Edge

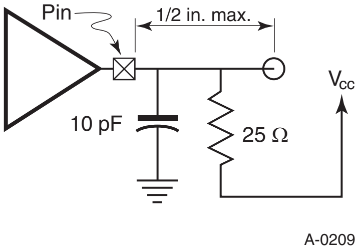


Figure 7-6: T_{val}(max) Falling Edge

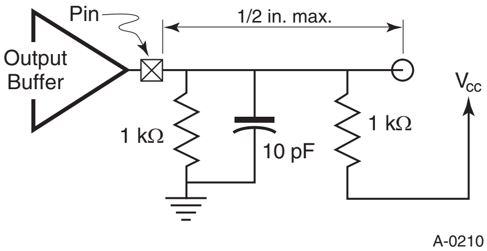


Figure 7-7: T_{val} (min) and Slew Rate

7.6.5. Vendor Provided Specification

Refer to Section 4.2.5.

7.6.6. Recommendations

7.6.6.1. Pinout Recommendations

Refer to Section 4.2.6.

The 66 MHz PCI electrical specification and physical requirements must be met; however, the designer is permitted to modify the suggested pinout shown in Figure 4-9 as required.

7.6.6.2. Clocking Recommendations

This section describes a recommended method for routing the 66 MHz PCI clock signal. Routing the 66 MHz PCI clock as a point-to-point signal from individual low-skew clock drivers to both system board and add-in card components will greatly reduce signal reflection effects and optimize clock signal integrity. This, in addition to observing the physical requirements outlined in Section 4.4.3.1, will minimize clock skew.

Developers must pay careful attention to the clock trace length limits stated in Section 4.4.3.1 and the velocity limits in Section 4.4.3.3. Figure 7-8 illustrates the recommended method for routing the 66 MHz PCI clock signal.

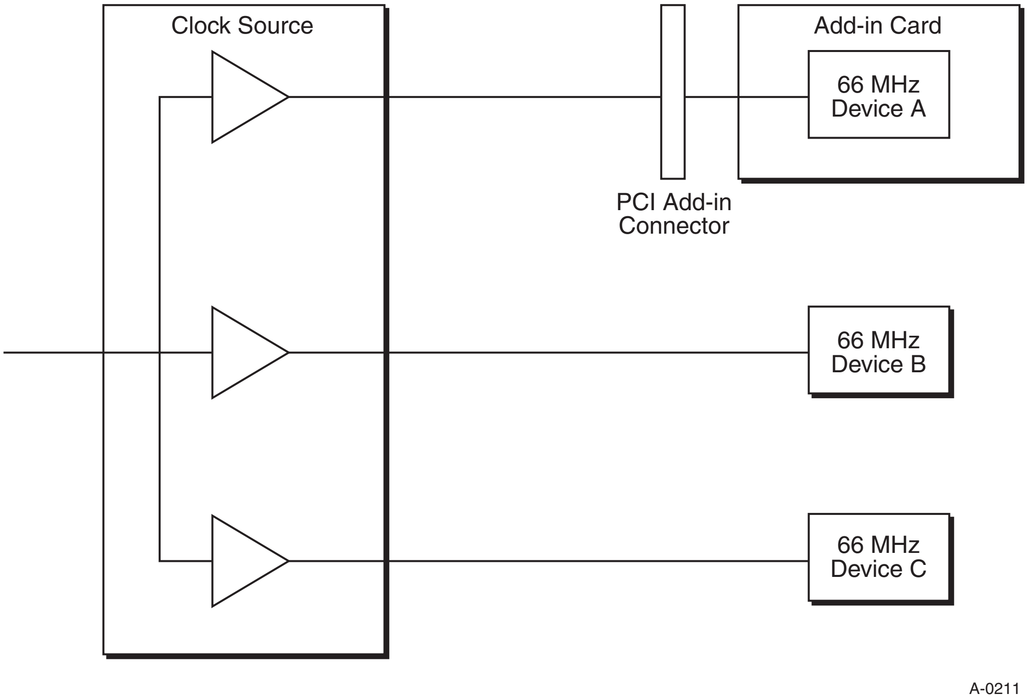


Figure 7-8: Recommended Clock Routing

7.7. System Board Specification

7.7.1. Clock Uncertainty

The maximum allowable clock skew including jitter is 1 ns. This specification applies not only at a single threshold point, but at all points on the clock edge that fall in the switching range defined in Table 7-6 and Figure 7-9. The maximum skew is measured between any two components,[54] not between connectors. To correctly evaluate clock skew, the system designer must take into account clock distribution on the add-in card as specified in Section 4.4.

---

[54] The system designer must address an additional source of clock skew. This clock skew occurs between two components that have clock input trip points at opposite ends of the V_{il} - V_{ih} range. In certain circumstances, this can add to the clock skew measurement as described here. In all cases, total clock skew must be limited to the specified number.

Developers must pay careful attention to the clock trace length limits stated in Section 4.4.3.1 and the velocity limits in Section 4.4.3.3.

Table 7-6: Clock Skew Parameters

Symbol	66 MHz 3.3V Signaling	33 MHz 3.3V Signaling	Units
V_{test}	$0.4V_{CC}$	$0.4V_{CC}$	V
T_{skew}	1 (max)	2 (max)	ns

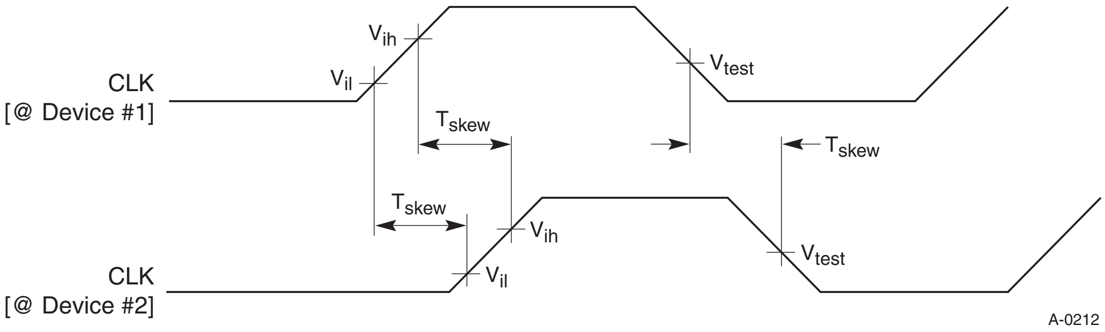


Figure 7-9: Clock Skew Diagram

7.7.2. Reset

Refer to Section 4.3.2.

7.7.3. Pullups

The 66 MHz PCI bus requires a single pullup resistor, supplied by the system board, on the M66EN pin. Refer to Section 7.7.7 for the resistor value.

7.7.4. Power

7.7.4.1. Power Requirements

Refer to Section 4.3.4.1.

7.7.4.2. Sequencing

Refer to Section 4.3.4.2.

7.7.4.3. Decoupling

Refer to Section 4.3.4.3.

7.7.5. System Timing Budget

Refer to Section 4.3.5.

The total clock period can be divided into four segments. Valid output delay (T_{val}) and input setup times (T_{su}) are specified by the component specification. Total clock skew (T_{skew}) and bus propagation times (T_{prop}) are system parameters. T_{prop} is a system parameter that is indirectly specified by subtracting the other timing budget components from the cycle time. Table 7-7 lists timing budgets for several bus frequencies.

Table 7-7: Timing Budgets

Timing Element	33 MHz	66 MHz	50 MHz[1]	Units	Notes
T_{cyc}	30	15	20	ns	
T_{val}	11	6	6	ns	
T_{prop}	10	5	10	ns	2
T_{su}	7	3	3	ns	
T_{skew}	2	1	1	ns	3

Notes:

1. The 50 MHz example is shown for example purposes only.
2. These times are computed. The other times are fixed. Thus, slowing down the bus clock enables the system manufacturer to gain additional distance or add additional loads. The component specifications are required to guarantee operation at 66 MHz.
3. Clock skew specified here includes all sources of skew. If spread spectrum clocking (SSC) is used on the system board, the maximum clock skew at the input of the device on an add-in card includes SSC tracking skew.

7.7.6. Physical Requirements

7.7.6.1. Routing and Layout Recommendations for Four-Layer System Boards

Refer to Section 4.3.6.1.

7.7.6.2. System Board Impedance

Refer to Section 4.3.6.2. Timing numbers are changed according to the tables in this chapter.

7.7.7. Connector Pin Assignments

The **M66EN** pin (pin 49B) is the only connector difference between bus segments capable of 66 MHz operation and those limited to 33 MHz. Refer to Section 4.3.7 for system board connectors. This pin is a normal ground pin in implementations that are not capable of 66 MHz operation.

In implementations that are 66-MHz-capable, the **M66EN** pin is bused between all connectors[55] within the single logical bus segment that is 66-MHz-capable, and this net is pulled up with a 5 kΩ resistor to V_{cc}. Also, this net is connected to the **M66EN** input pin of components located on the same logical bus segment of the system board. This signal is static; there is no stub length restriction.

To complete an AC return path, a 0.01 μF capacitor must be located within 0.25 inches of the **M66EN** pin, of each such add-in card connector and must decouple the **M66EN** signal to ground. Any attached component or installed add-in card, that is not 66-MHz-capable, must pull the **M66EN** net to the V_{il} input level. The remaining components, add-in cards, and the logical bus segment clock resource are, thereby, signaled to operate in 33-MHz mode.

7.8. Add-in Card Specifications

The 66 MHz add-in card specifications are the same as the 33 MHz add-in card specifications (refer to Section 4.4) except as noted in this section. The **M66EN** pin (pin 49B) is the only connector difference between add-in cards capable of 66 MHz operation and those limited to 33 MHz. Refer to Section 4.4.1 for the add-in card edge-connector. The **M66EN** pin is a normal ground pin in implementations that are not capable of 66 MHz operation.

In implementations that are 66-MHz-capable, the **M66EN** pin must be decoupled to ground with a 0.01 μF capacitor, which must be located within 0.25 inches of the edge contact to complete an AC return path. If the **M66EN** pin is pulled to the V_{il} input level, it indicates that the add-in card must operate in the 33-MHz mode.

---

[55] As a general rule, there will be only one such connector, but more than one are possible in certain cases.

8. System Support for SMBus

The SMBus interface is based upon the *System Management Bus Specification*[56] (SMBus 2.0 Specification). This two-wire serial interface has low power and low software overhead characteristics that make it well suited for low-bandwidth system management functions. The capabilities enabled by the SMBus interface include, but are not limited to, the following:

- ☐ Support for client management technologies
- ☐ Support for server management technologies
- ☐ Support for thermal sensors and other instrumentation devices on add-in cards
- ☐ Add-in card identification when the bus is in the B3 state or when the PCI device is in the $D3_{hot}$ or $D3_{cold}$ states as defined in the *PCI Power Management Interface Specification*[57]

8.1. SMBus System Requirements

SMBus device interfaces must meet the electrical and protocol requirements in the SMBus 2.0 Specification.

The SMBus interface connections on the PCI connector are optional. However, if the SMBus interface is supported, then all of the following requirements must be met.

8.1.1. Power

It is recommended that the system board provide 3.3V auxiliary power to the PCI connector (pin 14A). This allows an add-in card to maintain SMBus functionality when the system is in a low power state.

8.1.2. Physical and Logical SMBus

A physical SMBus segment is defined as a set of SMBus device interfaces whose SMBCLK and SMBDAT lines are directly connected to one another (i.e., not connected through an SMBus repeater or bridge). Each physical bus segment must meet the electrical requirements of the SMBus 2.0 Specification.

---

[56] *System Management Bus (SMBus) Specification, Version 2.0*, available from the SMBus website at http://www.smbus.org.

[57] *PCI Bus Power Management Interface Specification, Rev. 1.1*.

A logical SMBus is defined as one or more physical SMBuses having a single SMBus address space and a single SMBus arbitration domain. Multiple physical SMBus segments are connected into a single logical SMBus segment by means of bus repeaters. An SMBus address space is a logical connection of SMBus devices such that two devices with the same slave address will conflict with one another. An arbitration domain is a collection of one or more physical buses connected such that bus signal assertions caused by any device on any physical bus within the arbitration domain are seen by all devices on all the physical buses within the arbitration domain.

8.1.3. Bus Connectivity

Connection of SMBus to system board add-in card slot connectors is optional. However, if SMBus is connected to one slot in a chassis, it must be connected to all slots in that chassis. SMBus-connected slots in a chassis must be in the same logical SMBus. This logical SMBus may optionally be broken during system initialization or hot remove to allow the system to determine which SMBus device is in which PCI slot, but must be restored before the end of system initialization and during normal operation.

A typical implementation of a single physical SMBus is illustrated in Figure 8-1. The SMBus host bus controller is provided by the system.

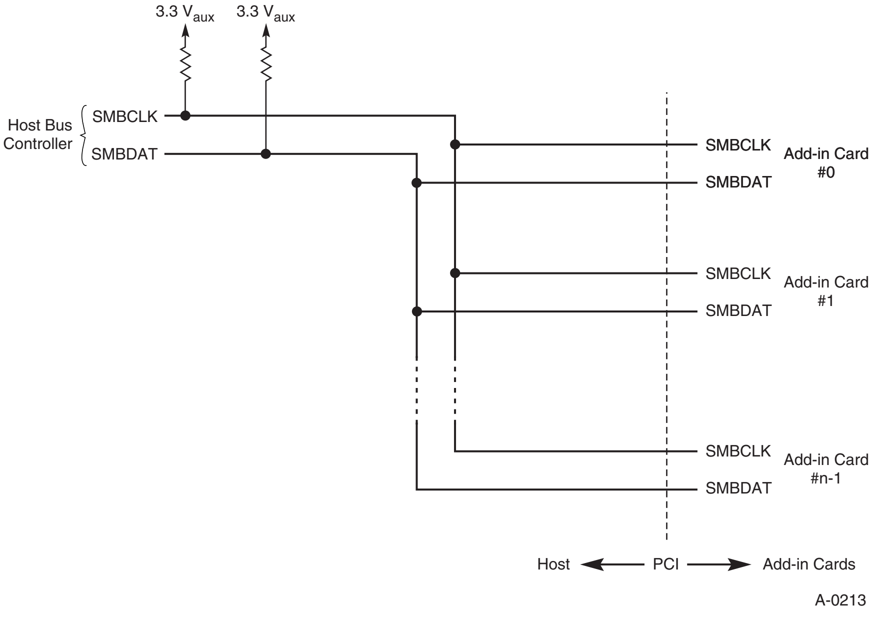


Figure 8-1: A Typical Single Physical SMBus

An SMBus in any one chassis is not required to support connection to an SMBus in another chassis. This does not preclude connection to another chassis, but the specification of any such connection is outside the scope of this document.

System board configurations with one or more SMBus-connected PCI bus segments must contain a single logical SMBus between all the PCI bus segments. There is no correlation between PCI bus segments and SMBus physical buses. As such, the SMBus is wired "around" a PCI bus bridge and not "through" it.

8.1.4. Master and Slave Support

As defined in the SMBus 2.0 Specification, an SMBus transaction occurs between a master and a slave. A master is a device that issues commands, generates the clock signal, and terminates the bus transaction. A slave is a device that receives or responds to the bus transaction.

The system board supporting the SMBus interface must have both slave and master capability and support the multi-master arbitration mechanism as defined in the SMBus 2.0 Specification.

8.1.5. Addressing and Configuration

An address resolution protocol (ARP) is defined in the SMBus 2.0 Specification that is used to assign slave addresses to SMBus devices. Although optional in the SMBus 2.0 Specification, it is required that systems that connect the SMBus to PCI slots implement the ARP for assignment of SMBus slave addresses to SMBus interface devices on PCI add-in cards. The system must execute the ARP on a logical SMBus whenever any PCI bus segment associated with the logical SMBus exits the B3 state or a device in an individual slot associated with the logical SMBus exits the $D3_{cold}$ state. Prior to executing the ARP, the system must insure that all ARP-capable SMBus interface devices are returned to their default address state.[58]

The system may optionally support in a chassis inclusion of a single add-in card with an SMBus interface device with a fixed address in the range C6h – C9h. Such a system must not contain SMBus interface devices with a fixed address in this range and must not assign addresses in this range to other SMBus interface devices on PCI add-in cards. Such systems are not required to provide mechanisms to resolve conflicts if more than one such add-in card is installed.

---

[58] The *System Management Bus (SMBus) Specification, Version 2.0* allows SMBus devices that are not associated with PCI add-in cards to have fixed SMBus addresses that are not assigned by the ARP. Such devices include, for example, system board temperature sensors. The SMBus slave addresses of such devices must be known to the system prior to the execution of the ARP and are not assigned to any ARP-capable SMBus devices.

IMPLEMENTATION NOTE

Fixed Address SMBus Interface Devices on an Add-in Card

Versions of the SMBus Specification prior to 2.0 did not include the ARP and SMBus interface devices compliant with those earlier specifications had fixed slave addresses. Many systems used such devices on the system board only. With the advent of the need for SMBus-enabled add-in cards and prior to this specification, a few add-in cards were introduced for special-purpose applications that included the available fixed-address SMBus interface devices with slave addresses fixed in the range C6h – C9h. This specification supports use of such add-in cards for backward compatibility. However, it is expected that all new designs will use the ARP and thereby avoid address-conflict problems.

The system may optionally implement a mechanism by which system configuration software performs a mapping of the associations between SMBus devices and physical add-in card slots. This mechanism determines the slot number in which an add-in card resides and associates this number with the address(s) of the SMBus interface device(s) on that add-in card. The system is permitted to isolate individual slots during the mapping process but must restore isolated slot connection states once the mapping process is complete.

8.1.6. Electrical

SMBus physical segments that connect to a PCI connector must use the high-power DC electrical specifications as defined in the SMBus 2.0 Specification.

The maximum capacitive load for a physical SMBus segment is 400 pF. If an SMBus physical segment includes PCI add-in card slots, a maximum capacitance per add-in card of 40 pF must be used in calculations. The absolute value of the total leakage current for an SMBus physical segment, source, and/or sink, must be less than 200 μA measured at 0.1 * Vcc and 0.9 * Vcc.

8.1.7. SMBus Behavior on PCI Reset

When power is applied to an SMBus device, it must perform default initialization of internal state as specified in the SMBus 2.0 Specification. SMBus device interface logic is not affected by RST#. This normally allows the SMBus to support communications when the PCI bus cannot.

8.2. Add-in Card SMBus Requirements

8.2.1. Connection

Only PCI add-in cards that support the SMBus interface as described in the SMBus 2.0 Specification, including the ARP, are permitted to connect to the SMBCLK and SMBDAT pins. Any PCI add-in card implementing SMBus interface functionality via the PCI connector must connect to the SMBCLK and SMBDAT pins.

8.2.2. Master and Slave Support

An add-in card SMBus implementation that is intended to respond to commands or requests for data from a master must implement SMBus slave functionality and must support the ARP.

Master capability is optional. However, if an add-in card implementation supports master capability, it must support multi-master arbitration.

8.2.3. Addressing and Configuration

Add-in card SMBus interface devices must implement the ARP for establishing their slave addresses as defined in the SMBus 2.0 Specification. Although the ARP is optional in the SMBus 2.0 Specification, it is required by this specification.

8.2.4. Power

An add-in card is permitted to power its SMBus interface logic from the 3.3Vaux pin on the PCI connector if the add-in card vendor intends that the add-in card's SMBus functionality be available while the system is at a low power consumption state. Such an add-in card must support PME# from $D3_{cold}$, as defined in the *PCI Power Management Interface Specification*. Alternatively, the SMBus interface logic may be powered from the standard 3.3V supply.

8.2.5. Electrical

Add-in card SMBus interface devices must comply with the high-power electrical requirements stated in the SMBus 2.0 Specification.

Add-in card designers must meet the maximum-capacitance-per-add-in card requirement of 40 pF per SMBus signal pin. The 40 pF limit includes:

- ☐ The sum of device capacitance loads
- ☐ The capacitance of trace length from the add-in card's PCI connector

The absolute value of the sum of the leakage current, source, and/or sink, for all SMBus interface devices on the add-in card must be less than 20 μA measured at 0.1* V_{CC} and 0.9 * V_{CC}.

There is no limitation on the number of SMBus devices on an add-in card provided these requirements are met.

IMPLEMENTATION NOTE

SMBus Loading in a PCI Low-Power State

When/if power is removed from SMBus interface logic when an associated PCI function is transitioned to a low-power state, the SMBus clock and data lines must not be loaded so as to make the SMBus inoperative. Other SMBus interfaces on the bus may still be powered and operational. The designer is reminded to observe the leakage current parameters in the SMBus 2.0 Specification for unpowered SMBus interfaces.

A

A. Special Cycle Messages

Special Cycle message encodings are defined in this appendix. Reserved encodings should not be used. PCI-SIG member companies that require special encodings outside the range of currently defined encodings should send a written request to the PCI-SIG Board of Directors. The Board of Directors will allocate and define special cycle encodings based upon information provided by the requester specifying usage needs and future product or application direction.

A.1. Message Encodings

AD[15::0]	**Message Type**
0000h	SHUTDOWN
0001h	HALT
0002h	x86 architecture-specific
0003h	Reserved
through	
FFFFh	Reserved

SHUTDOWN is a broadcast message indicating the processor is entering into a shutdown mode.

HALT is a broadcast message from the processor indicating it has executed a halt instruction.

The x86 architecture-specific encoding is a generic encoding for use by x86 processors and chipsets. AD[31::16] determine the specific meaning of the Special Cycle message. Specific meanings are defined by Intel Corporation and are found in product specific documentation.

A.2. Use of Specific Encodings

Use or generation of architecture-specific encodings is not limited to the requester of the encoding. Specific encodings may be used by any vendor in any system. These encodings allow system specific communication links between cooperating PCI devices for purposes which cannot be handled with the standard data transfer cycle types.

B

B. State Machines

This appendix describes master and target state machines. These state machines are for illustrative purposes only and are included to help illustrate PCI protocol. Actual implementations should not directly use these state machines. The machines are believed to be correct; however, if a conflict exists between the specification and the state machines, the specification has precedence.

The state machines use three types of variables: states, PCI signals, and internal signals. They can be distinguished from each other by:

State in a state machine = STATE
PCI signal = SIGNAL
Internal signal = Signal

The state machines assume no delays from entering a state until signals are generated and available for use in the machine. All PCI signals are latched on the rising edge of CLK.

The state machines support some options (but not all) discussed in the PCI specification. A discussion about each state and the options illustrated follows the definition of each state machine. The target state machine assumes medium decode and, therefore, does not describe fast decode. If fast decode is implemented, the state diagrams (and their associated equations) will need to be changed to support fast decode. Caution needs to be taken when supporting fast decode (refer to Section 3.4.2).

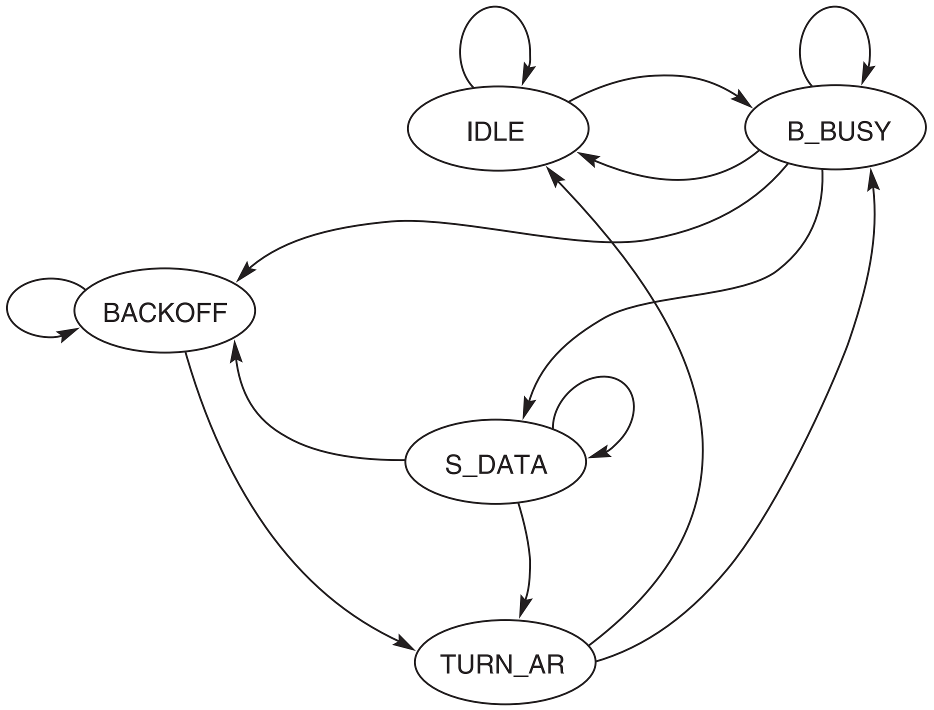
The bus interface consists of two parts. The first is the bus sequencer that performs the actual bus operation. The second part is the backend or hardware application. In a master, the backend generates the transaction and provides the address, data, command, Byte Enables, and the length of the transfer. It is also responsible for the address when a transaction is retried. In a target, the backend determines when a transaction is terminated. The sequencer performs the bus operation as requested and guarantees the PCI protocol is not violated. Note that the target implements a resource lock.

The state machine equations assume a logical operation where "*" is an AND function and has precedence over "+" which is an OR function. Parentheses have precedence over both. The "!" character is used to indicate the NOT of the variable. In the state machine equations, the PCI SIGNALs represent the actual state of the signal on the PCI bus. Low true signals will be true or asserted when they appear as !SIGNAL# and will be false or deasserted when they appear as SIGNAL#. High true signals will be true or asserted when they appear as SIGNAL and will be false or deasserted when they appear as !SIGNAL. Internal signals will be true when they appear as Signal and false when they appear as !Signal. A few of the output enable equations use the "==" symbol to refer to the previous state. For example:

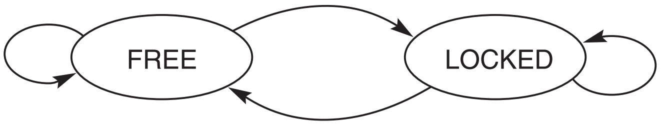
OE[PAR] == [S_DATA * !TRDY# * (cmd=read)]

This indicates the output buffer for PAR is enabled when the previous state is S_DATA, TRDY# is asserted, the transaction is a read. The first state machine presented is for the target, the second is the master. Caution needs to be taken when an agent is both a master and a target. Each must have its own state machine that can operate independently of the other to avoid deadlocks. This means that the target state machine cannot be affected by the master state machine. Although they have similar states, they cannot be built into a single machine.

Note: LOCK# can only be implemented by a bridge; refer to Appendix F for details about the use of LOCK#. For a non-bridge device, the use of LOCK# is prohibited.



Target Sequencer Machine



Target LOCK Machine

A-0214

IDLE or TURN_AR -- Idle condition or completed transaction on bus.

goto IDLE if FRAME#
goto B_BUSY if !FRAME# * !Hit

B_BUSY -- Not involved in current transaction.

```
goto B_BUSY     if (!FRAME# + !D_done) * !Hit
goto IDLE       if FRAME# * D_done + FRAME# * !D_done * !DEVSEL#
goto S_DATA     if (!FRAME# + !IRDY#) * Hit * (!Term + Term * Ready)
goto BACKOFF    *if (!FRAME# + !IRDY#) * Hit
```

S_DATA -- Agent has accepted request and will respond.

```
goto S_DATA     if !FRAME# * !STOP# * !TRDY# * IRDY#
                + !FRAME# * STOP# + FRAME# * TRDY# * STOP#
goto BACKOFF    if !FRAME# * !STOP# * (TRDY# + !IRDY#)
goto TURN_AR    if FRAME# * (!TRDY# + !STOP#)
```

BACKOFF -- Agent busy unable to respond at this time.

```
goto BACKOFF    if !FRAME#
goto TURN_AR    if FRAME#
```

B.1. Target LOCK Machine

FREE -- Agent is free to respond to all transactions.

```
goto LOCKED     if !FRAME# * LOCK# * Hit * (IDLE + TURN_AR)
                + L_lock# * Hit * B_BUSY)
goto FREE       if ELSE
```

LOCKED -- Agent will not respond unless LOCK# is deasserted during the address phase.

```
goto FREE       if FRAME# * LOCK#
goto LOCKED     if ELSE
```

Target of a transaction is responsible to drive the following signals:[59]

OE[AD[31::00]] = (S_DATA + BACKOFF) * Tar_dly * (cmd = read)
OE[TRDY#] = BACKOFF + S_DATA + TURN_AR (See Note.)
OE[STOP#] = BACKOFF + S_DATA + TURN_AR (See Note.)
OE[DEVSEL#] = BACKOFF + S_DATA + TURN_AR (See Note.)
OE[PAR] = OE[AD[31::00]] delayed by one clock
OE[PERR#] = R_perr + R_perr (delayed by one clock)

Note: If the device does fast decode, OE[PERR#] must be delayed one clock to avoid contention.

---

[59] When the target supports the Special Cycle command, an additional term must be included to ensure these signals are not enabled during a Special Cycle transaction.

TRDY# = !(Ready * !T_abort * S_DATA * (cmd=write + cmd=read *
STOP# = ![BACKOFF + S_DATA * (T_abort + Term) * (cmd=write + cmd=read * Tar_dly)]
DEVSEL# = ![(BACKOFF + S_DATA) * !T_abort]
PAR = even parity across AD[31::00] and C/BE#[3::0] lines.
PERR# = R_perr

Definitions

These signals are between the target bus sequencer and the backend. They indicate how the bus sequencer should respond to the current bus operation.

Hit = Hit on address decode.
D_done = Decode done. Device has completed the address decode.
T_abort = Target is in an error condition and requires the current transaction to
Term = Terminate the transaction. (Internal conflict or > n wait states.)
Ready = Ready to transfer data.
L_lock# = Latched (during address phase) version of LOCK#.
Tar_dly = Turn around delay only required for zero wait state decode.
R_perr = Report parity error is a pulse of one PCI clock in duration.
Last_targe = Device was the target of the last (prior) PCI transaction.

The following paragraphs discuss each state and describe which equations can be removed if some of the PCI options are not implemented.

The IDLE and TURN_AR are two separate states in the state machine, but are combined here because the state transitions are the same from both states. They are implemented as separate states because active signals need to be deasserted before the target tri-states them.

If the target cannot do single cycle address decode, the path from IDLE to S_DATA can be removed. The reason the target requires the path from the TURN_AR state to S_DATA and B_BUSY is for back-to-back bus operations. The target must be able to decode back-to-back transactions.

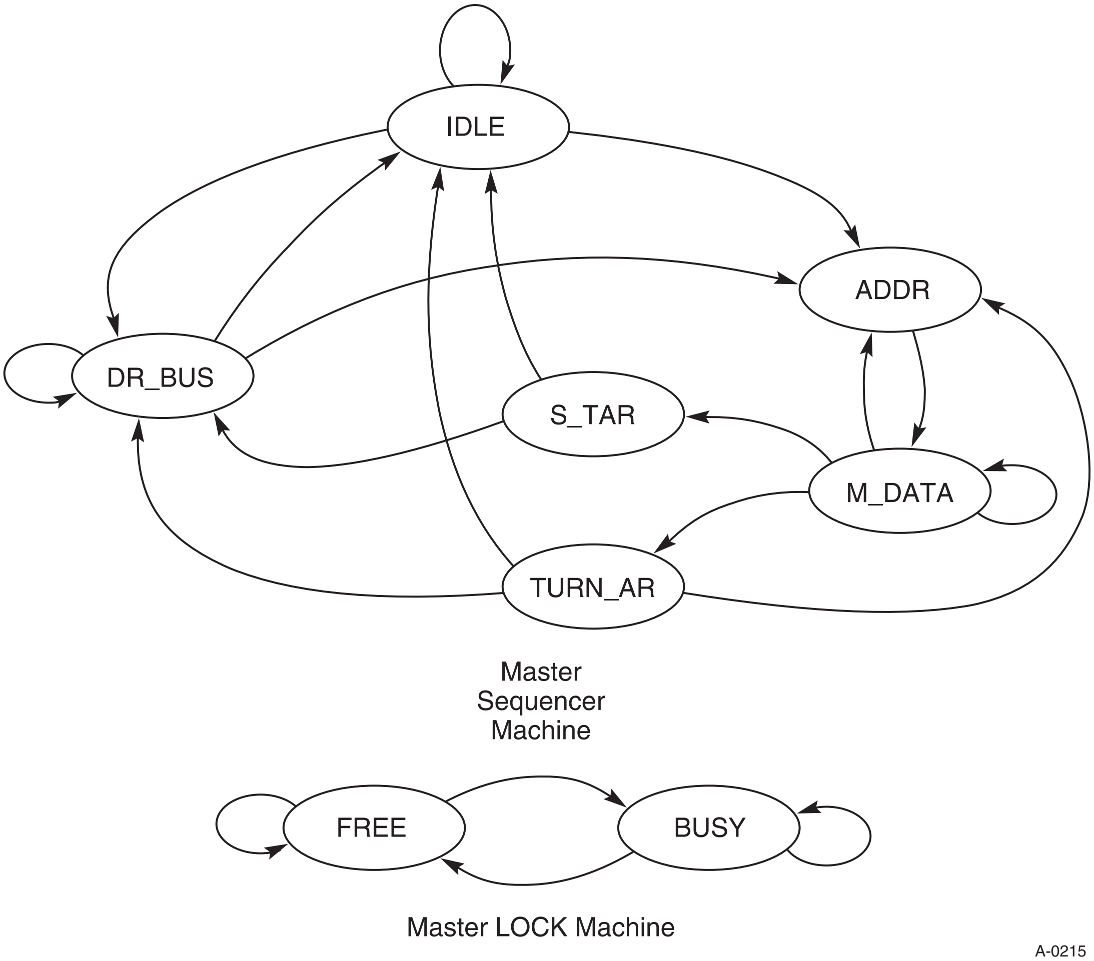
B_BUSY is a state where the agent waits for the current transaction to complete and the bus to return to the Idle state. B_BUSY is useful for devices that do slow address decode or perform subtractive decode. If the target does neither of these two options, the path to S_DATA and BACKOFF may be removed. The term "!Hit" may be removed from the B_BUSY equation also. This reduces the state to waiting for the current bus transaction to complete.

S_DATA is a state where the target transfers data and there are no optional equations.

BACKOFF is where the target goes after it asserts STOP# and waits for the master to deassert FRAME#.

FREE and LOCKED refer to the state of the target with respect to a lock operation. If the target does not implement LOCK#, then these states are not required. FREE indicates when the agent may accept any request when it is the target. If LOCKED, the target will retry any request when it is the target unless LOCK# is deasserted during the address phase. The agent marks itself locked whenever it is the target of a transaction and LOCK# is deasserted during the address phase. It is a little confusing for the target to lock itself on a transaction that is not locked. However, from an implementation point of view, it is a simple mechanism that uses combinatorial logic and always works. The device will unlock itself at the end of the transaction when it detects FRAME# and LOCK# both deasserted.

The second equation in the goto LOCKED in the FREE state can be removed if fast decode is done. The first equation can be removed if medium or slow decode is done. L_lock# is LOCK# latched during the address phase and is used when the agent's decode completes.



B.2. Master Sequencer Machine

IDLE -- Idle condition on bus.

goto ADDR	if (Request * !Step) * !GNT# * FRAME# * IRDY#
goto DR_BUS	if (Request * Step + !Request) * !GNT# * FRAME# * IRDY#
goto IDLE	if ELSE

ADDR -- Master starts a transaction.

goto M_DATA	on the next rising edge of CLK.

M_DATA -- Master transfers data.

```
goto M_DATA     if !FRAME# + FRAME# * TRDY# * STOP#
                * !Dev_to
goto ADDR       if (Request *!Step) * !GNT# * FRAME# * !TRDY# *
                STOP# * L-cycle * (Sa +FB2B_Ena)
goto S_TAR      if FRAME# * !STOP# + FRAME# * Dev_to
goto TURN_AR    if ELSE
```

TURN_AR -- Transaction complete, do housekeeping.

```
goto ADDR       if (Request * !Step) * !GNT#
goto DR_BUS     if (Request * Step + !Request) * !GNT#
goto IDLE       if GNT#
```

S_TAR -- Stop was asserted, do turn around cycle.

```
goto DR_BUS     if !GNT#
goto IDLE       if GNT#
```

DR_BUS -- Bus parked at this agent or agent is using address stepping.

```
goto DR_BUS     if (Request * Step + !Request)* !GNT#
goto ADDR       if (Request * !Step) * !GNT#
goto IDLE       if GNT#
```

B.3. Master LOCK Machine

FREE -- LOCK# is not in use (not owned).

```
goto FREE       if LOCK# + !LOCK# * Own_lock
goto BUSY       if !LOCK# * !Own_lock
```

BUSY -- LOCK# is currently being used (owned).

```
goto FREE       if LOCK# * FRAME#
goto BUSY       if !LOCK# + !FRAME#
```

The master of the transaction is responsible to drive the following signals:

Enable the output buffers:

OE[FRAME#] = ADDR + M_DATA
OE[C/BE#[3::0]] = ADDR + M_DATA + DR_BUS
if ADDR drive command
if M_DATA drive byte enables
if DR_BUS if (Step * Request) drive command else drive lines to a valid state

OE[AD[31::00] = ADDR + M_DATA * (cmd=write) + DR_BUS
if ADDR drive address
if M_DATA drive data
if DR_BUS if (Step * Request) drive address else drive lines to a valid state

OE[LOCK#] = Own_lock * M_DATA + OE [LOCK#] * (!FRAME# + !LOCK#)
OE[IRDY#] == [M_DATA + ADDR]
OE[PAR] = OE[AD[31::00]] delayed by one clock
OE[PERR#] = R_perr + R_perr (delayed by one clock)

The following signals are generated from state and sampled (not asynchronous) bus signals.

FRAME# = !(ADDR + M_DATA * !Dev_to * {[!Comp
* (!To + !GNT#) * STOP#] + !Ready })

IRDY# = ![M_DATA * (Ready + Dev_to)]
REQ# = ![(Request * !Lock_a + Request * Lock_a * FREE)
*!(S_TAR * Last State was S_TAR)]

LOCK# = Own_lock * ADDR + Target_abort
+ Master_abort + M_DATA * !STOP# * TRDY# * !Ldt
+ Own_lock * !Lock_a * Comp * M_DATA * FRAME# * !TRDY#
PAR = even parity across AD[31::00] and C/BE#[3::0] lines.
PERR# = R_perr

Master_abort = (M_DATA * Dev_to)
Target_abort = (!STOP# * DEVSEL# * M_DATA * FRAME# * !IRDY#)
Own_lock = LOCK# * FRAME# * IRDY# * Request * !GNT# * Lock_a
+ Own_lock * (!FRAME# + !LOCK#)

Definitions

These signals go between the bus sequencer and the backend. They provide information to the sequencer when to perform a transaction and provide information to the backend on how the transaction is proceeding. If a cycle is retried, the backend will make the correct modification to the affected registers and then indicate to the sequencer to perform another transaction. The bus sequencer does not remember that a transaction was retried or aborted but takes requests from the backend and performs the PCI transaction.

Master_abort = The transaction was aborted by the master. (No DEVSEL#.)
Target_abort = The transaction was aborted by the target.
Step = Agent using address stepping (wait in the state until !Step).
Request = Request pending.
Comp = Current transaction in last data phase.
L-cycle = Last cycle was a write.
To = Master timeout has expired.
Dev_to = Devsel timer has expired without DEVSEL# being asserted.
Sa = Next transaction to same agent as previous transaction.
Lock_a = Request is a locked operation.
Ready = Ready to transfer data.
Sp_cyc = Special Cycle command.
Own_lock = This agent currently owns LOCK#.
Ldt = Data was transferred during a LOCK operation.
R_perr = Report parity error is a pulse one PCI clock in duration.
FB2B_Ena = Fast Back-to-Back Enable (Configuration register bit).

The master state machine has many options built in that may not be of interest to some implementations. Each state will be discussed indicating what affect certain options have on the equations.

IDLE is where the master waits for a request to do a bus operation. The only option in this state is the term "Step." It may be removed from the equations if address stepping is not supported. All paths must be implemented. The path to DR_BUS is required to insure that the bus is not left floating for long periods. The master whose GNT# is asserted must go to the drive bus if its Request is not asserted.

ADDR has no options and is used to drive the address and command on the bus.

M_DATA is where data is transferred. If the master does not support fast back-to-back transactions, the path to the ADDR state is not required.

The equations are correct from the protocol point of view. However, compilers may give errors when they check all possible combinations. For example, because of protocol, Comp cannot be asserted when FRAME# is deasserted. Comp indicates the master is in the last data phase, and FRAME# must be deasserted for this to be true.

TURN_AR is where the master deasserts signals in preparation for tri-stating them. The path to ADDR may be removed if the master does not do back-to-back transactions.

S_TAR could be implemented a number of ways. The state was chosen to clarify that "state" needs to be remembered when the target asserts STOP#.

DR_BUS is used when GNT# has been asserted, and the master either is not prepared to start a transaction (for address stepping) or has none pending. If address stepping is not implemented, then the equation in goto DR_BUS that has "Step" may be removed and the goto ADDR equation may also remove "Step."

If LOCK# is not supported by the master, the FREE and BUSY states may be removed. These states are for the master to know the state of LOCK# when it desires to do a locked transaction. The state machine simply checks for LOCK# being asserted. Once asserted, it stays BUSY until FRAME# and LOCK# are both deasserted signifying that LOCK# is now free.

C. Operating Rules

This appendix is not a complete list of rules of the specification and should not be used as a replacement for the specification. This appendix only covers the basic protocol and requirements contained in Chapter 3. It is meant to be used as an aid or quick reference to the basic rules and relationships of the protocol.

C.1. When Signals are Stable

1. The following signals are guaranteed to be stable on all rising edges of CLK once reset has completed: LOCK#, IRDY#, TRDY#, FRAME#, DEVSEL#, STOP#, REQ#, GNT#, REQ64#, ACK64#, SERR# (on falling edge only), and PERR#.
2. Address/Data lines are guaranteed to be stable at the specified clock edge as follows:
    a. Address -- AD[31::00] are stable regardless of whether some are logical don't cares on the first clock that samples FRAME# asserted.
    b. Address -- AD[63::32] are stable and valid during the first clock after REQ64# assertion when 32-bit addressing is being used (SAC), or the first two clocks after REQ64# assertion when 64-bit addressing is used (DAC). When REQ64# is deasserted, AD[63::32] are pulled up by the central resource.
    c. Data -- AD[31::00] are stable and valid regardless which byte lanes are involved in the transaction on reads when TRDY# is asserted and on writes when IRDY# is asserted. At any other time, they may be indeterminate. The AD lines cannot change until the current data phase completes once IRDY# is asserted on a write transaction or TRDY# is asserted on a read transaction.
    d. Data -- AD[63::32] are stable and valid regardless which byte lanes are involved in the transaction when ACK64# is asserted and either TRDY# is asserted on reads, or IRDY# is asserted on writes. At any other time, they may be indeterminate.
    e. Data -- Special cycle command -- AD[31::00] are stable and valid regardless which byte lanes are involved in the transaction when IRDY# is asserted.
    f. Do not gate asynchronous data directly onto PCI while IRDY# is asserted on a write transaction and while TRDY# is asserted on a read transaction.

3. Command/Byte enables are guaranteed to be stable at the specified clock edge as follows:
   a. Command -- C/BE[3::0]# are stable and valid the first time FRAME# is sampled asserted and contain the command codes. C/BE[7::4]# are stable and valid during the first clock after REQ64# assertion when 32-bit addressing is being used (SAC) and are reserved. C/BE[7::4]# are stable and valid during the first two clocks after REQ64# assertion when 64-bit addressing is used (DAC) and contain the actual bus command. When REQ64# is deasserted, the C/BE[7::4]# are pulled up by the central resource.
   b Byte Enables -- C/BE[3::0]# are stable and valid the clock following the address phase and each completed data phase and remain valid every clock during the entire data phase regardless if wait states are inserted and indicate which byte lanes contain valid data. C/BE[7::4]# have the same meaning as C/BE[3::0]# except they cover the upper 4 bytes when REQ64# is asserted.
4. PAR is stable and valid one clock following the valid time of AD[31::00]. PAR64 is stable and valid one clock following the valid time of AD[63::32].
5. IDSEL is only stable and valid the first clock FRAME# is asserted when the access is a configuration command. IDSEL is indeterminate at any other time.
6. RST#, INTA#, INTB#, INTC#, and INTD# are not qualified or synchronous.

C.2. Master Signals

7. A transaction starts when FRAME# is asserted for the first time.
8. The following govern FRAME# and IRDY# in all PCI transactions.
   a. FRAME# and its corresponding IRDY# define the Busy/Idle state of the bus; when either is asserted, the bus is busy; when both are deasserted, the bus is in the Idle state.
   b. Once FRAME# has been deasserted, it cannot be reasserted during the same transaction.
   c. FRAME# cannot be deasserted unless IRDY# is asserted. (IRDY# must always be asserted on the first clock edge that FRAME# is deasserted.)
   d. Once a master has asserted IRDY#, it cannot change IRDY# or FRAME# until the current data phase completes.
   e. The master must deassert IRDY# the clock after the completion of the last data phase.
9. When FRAME# and IRDY# are deasserted, the transaction has ended.

10. When the current transaction is terminated by the target (**STOP#** asserted), the master must deassert its **REQ#** signal before repeating the transaction. A device containing a single source of master activity must deassert **REQ#** for a minimum of two clocks; one being when the bus goes to the Idle state (at the end of the transaction where **STOP#** was asserted) and either the clock before or the clock after the Idle state. A device containing multiple sources of master activity is permitted to allow each source to use the bus without deasserting **REQ#** even if one or more sources are target terminated. However, the device must deassert **REQ#** for two clocks, one of which while the bus is Idle before any transaction that was target terminated can be repeated.
11. A master that is target terminated with Retry must unconditionally repeat the same request until it completes; however, it is not required to repeat the transaction when terminated with Disconnect.

C.3. Target Signals

12. The following general rules govern **FRAME#**, **IRDY#**, **TRDY#**, and **STOP#** while terminating transactions.
    a. A data phase completes on any rising clock edge on which **IRDY#** is asserted and either **STOP#** or **TRDY#** is asserted.
    b. Independent of the state of **STOP#**, a data transfer takes place on every rising edge of clock where both **IRDY#** and **TRDY#** are asserted.
    c. Once the target asserts **STOP#**, it must keep **STOP#** asserted until **FRAME#** is deasserted, whereupon it must deassert **STOP#**.
    d. Once a target has asserted **TRDY#** or **STOP#**, it cannot change **DEVSEL#**, **TRDY#**, or **STOP#** until the current data phase completes.
    e. Whenever **STOP#** is asserted, the master must deassert **FRAME#** as soon as **IRDY#** can be asserted.
    f. If not already deasserted, **TRDY#**, **STOP#**, and **DEVSEL#** must be deasserted the clock following the completion of the last data phase and must be tri-stated the next clock.
13. An agent claims to be the target of the access by asserting **DEVSEL#**.
14. **DEVSEL#** must be asserted with, or prior to, the edge at which the target enables its outputs (**TRDY#**, **STOP#**, or (on a read) **AD** lines).
15. Once **DEVSEL#** has been asserted, it cannot be deasserted until the last data phase has completed, except to signal Target-Abort.

C.4. Data Phases

16. The source of the data is required to assert its *x*RDY# signal unconditionally when data is valid (IRDY# on a write transaction, TRDY# on a read transaction).
17. Data is transferred between master and target on each clock edge for which both IRDY# and TRDY# are asserted.
18. Last data phase completes when:
    a. FRAME# is deasserted and TRDY# is asserted (normal termination) or
    b. FRAME# is deasserted and STOP# is asserted (target termination) or
    c. FRAME# is deasserted and the device select timer has expired (Master-Abort) or
    d. DEVSEL# is deasserted and STOP# is asserted (Target-Abort).
19. Committing to complete a data phase occurs when the target asserts either TRDY# or STOP#. The target commits to:
    a. Transfer data in the current data phase and continue the transaction (if a burst) by asserting TRDY# and not asserting STOP#.
    b. Transfer data in the current data phase and terminate the transaction by asserting both TRDY# and STOP#.
    c. Not transfer data in the current data phase and terminate the transaction by asserting STOP# and deasserting TRDY#.
    d. Not transfer data in the current data phase and terminate the transaction with an error condition (Target-Abort) by asserting STOP# and deasserting TRDY# and DEVSEL#.
20. The target has not committed to complete the current data phase while TRDY# and STOP# are both deasserted. The target is simply inserting wait states.

C.5. Arbitration

21. The agent is permitted to start a transaction only in the following two cases:
    a. GNT# is asserted and the bus is idle (FRAME# and IRDY# are deasserted).
    b. GNT# is asserted in the last data phase of a transaction and the agent is starting a new transaction using fast back-to-back timing (FRAME# is deasserted and TRDY# or STOP# is asserted or the transaction terminates with Master-Abort).
22. The arbiter may deassert an agent's GNT# on any clock.

23. Once asserted, GNT# may be deasserted according to the following rules.
    a. If GNT# is deasserted and FRAME# is asserted on the same clock, the bus transaction is valid and will continue.
    b. One GNT# can be deasserted coincident with another GNT# being asserted if the bus is not in the Idle state. Otherwise, a one clock delay is required between the deassertion of a GNT# and the assertion of the next GNT# or else there may be contention on the AD lines and PAR.
    c. While FRAME# is deasserted, GNT# may be deasserted at any time in order to service a higher priority[60] master or in response to the associated REQ# being deasserted.
24. When the arbiter asserts an agent's GNT# and the bus is in the Idle state, that agent must enable its AD[31::00], C/BE[3::0]#, and (one clock later) PAR output buffers within eight PCI clocks (required), while two-three clocks is recommended.

C.6. Latency

25. All targets are required to complete the initial data phase of a transaction (read or write) within 16 clocks from the assertion of FRAME#. Host bus bridges have an exception (refer to Section 3.5.1.1).
26. The target is required to complete a subsequent data phase within eight clocks from the completion of the previous data phase.
27. A master is required to assert its IRDY# within eight clocks for any given data phase (initial and subsequent).

C.7. Device Selection

28. A target must do a full decode before driving/asserting DEVSEL# or any other target response signal.
29. A target must assert DEVSEL# (claim the transaction) before it is allowed to issue any other target response.
30. In all cases except Target-Abort, once a target asserts DEVSEL# it must not deassert DEVSEL# until FRAME# is deasserted (IRDY# is asserted) and the last data phase has completed.
31. A PCI device is a target of a Type 0 configuration transaction (read or write) only if its IDSEL is asserted, and AD[1::0] are "00" during the address phase of the command.

---

[60] Higher priority here does not imply a fixed priority arbitration, but refers to the agent that would win arbitration at a given instant in time.

C.8. Parity

32. Parity is generated according to the following rules:
    a. Parity is calculated the same on all PCI transactions regardless of the type or form.
    b. The number of "1"s on AD[31::00], C/BE[3::0]#, and PAR equals an even number.
    c. The number of "1"s on AD[63::32], C/BE[7::4]#, and PAR64 equals an even number.
    d. Generating parity is not optional; it must be done by all PCI-compliant devices.
33. Only the master of a corrupted data transfer is allowed to report parity errors to software using mechanisms other than PERR# (i.e., requesting an interrupt or asserting SERR#). In some cases, the master delegates this responsibility to a PCI-to-PCI bridge handling posted memory write data. See the *PCI-to-PCI Bridge Architecture Specification* for details.

D

D. Class Codes

This appendix describes the current Class Code encodings. This list may be enhanced at any time. The PCI-SIG web pages contain the latest version. Companies wishing to define a new encoding should contact the PCI-SIG. All unspecified values are reserved for PCI-SIG assignment.

Base Class	Meaning
00h	Device was built before Class Code definitions were finalized
01h	Mass storage controller
02h	Network controller
03h	Display controller
04h	Multimedia device
05h	Memory controller
06h	Bridge device
07h	Simple communication controllers
08h	Base system peripherals
09h	Input devices
0Ah	Docking stations
0Bh	Processors
0Ch	Serial bus controllers
0Dh	Wireless controller
0Eh	Intelligent I/O controllers
0Fh	Satellite communication controllers
10h	Encryption/Decryption controllers
11h	Data acquisition and signal processing controllers
12h - FEh	Reserved
FFh	Device does not fit in any defined classes

D.1. Base Class 00h

This base class is defined to provide backward compatibility for devices that were built before the Class Code field was defined. No new devices should use this value and existing devices should switch to a more appropriate value if possible.

For class codes with this base class value, there are two defined values for the remaining fields as shown in the table below. All other values are reserved.

Base Class	Sub-Class	Interface	Meaning
00h	00h	00h	All currently implemented devices except VGA-compatible devices
	01h	00h	VGA-compatible device

D.2. Base Class 01h

This base class is defined for all types of mass storage controllers. Several sub-class values are defined. The IDE controller class is the only one that has a specific register-level programming interface defined.

Base Class	Sub-Class	Interface	Meaning
01h	00h	00h	SCSI bus controller
	01h	xxh	IDE controller (see Note 1)
	02h	00h	Floppy disk controller
	03h	00h	IPI bus controller
	04h	00h	RAID controller
	05h	20h	ATA controller with ADMA interface–single stepping (see Note 2)
		30h	ATA controller with ADMA interface–continuous operation (see Note 2)
	06h	00h	Serial ATA controller–vendor specific interface
		01h	Serial ATA controller–AHCI 1.0 interface
	07h	00h	Serial Attached SCSI (SAS) controller
	80h	00h	Other mass storage controller

Notes:

1. Register interface conforms to the PCI Compatibility and PCI-Native Mode Bus interface defined in *ANSI INCITS.370:2003: ATA Host Adapters Standard* (see http://www.incits.org and http://www.t13.org).
2. Register interface conforms to the ADMA interface defined in *ANSI INCITS.370:2003: ATA Host Adapters Standard* (see http://www.incits.org and http://www.t13.org).

D.3. Base Class 02h

This base class is defined for all types of network controllers. Several sub-class values are defined. There are no register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
02h	00h	00h	Ethernet controller
	01h	00h	Token Ring controller
	02h	00h	FDDI controller
	03h	00h	ATM controller
	04h	00h	ISDN controller
	05h	00h	WorldFip controller
	06h	xxh (see below)	PICMG 2.14 Multi Computing
	80h	00h	Other network controller

For information on the use of this field see the PICMG 2.14 Multi Computing Specification (http://www.picmg.com).

D.4. Base Class 03h

This base class is defined for all types of display controllers. For VGA devices (Sub-Class 00h), the programming interface byte is divided into a bit field that identifies additional video controller compatibilities. A device can support multiple interfaces by using the bit map to indicate which interfaces are supported. For the XGA devices (Sub-Class 01h), only the standard XGA interface is defined. Sub-Class 02h is for controllers that have hardware support for 3D operations and are not VGA compatible.

Base Class	Sub-Class	Interface	Meaning
03h	00h	0000 0000b	VGA-compatible controller. Memory addresses 0A 0000h through 0B FFFFh. I/O addresses 3B0h to 3BBh and 3C0h to 3DFh and all aliases of these addresses.
		0000 0001b	8514-compatible controller -- 2E8h and its aliases, 2EAh-2EFh
	01h	00h	XGA controller
	02h	00h	3D controller
	80h	00h	Other display controller

D.5. Base Class 04h

This base class is defined for all types of multimedia devices. Several sub-class values are defined. There are no register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
04h	00h	00h	Video device
	01h	00h	Audio device
	02h	00h	Computer telephony device
	80h	00h	Other multimedia device

D.6. Base Class 05h

This base class is defined for all types of memory controllers. Several sub-class values are defined. There are no register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
05h	00h	00h	RAM
	01h	00h	Flash
	80h	00h	Other memory controller

D.7. Base Class 06h

This base class is defined for all types of bridge devices. A PCI bridge is any PCI device that maps PCI resources (memory or I/O) from one side of the device to the other. Several sub-class values are defined.

Base Class	Sub-Class	Interface	Meaning
06h	00h	00h	Host bridge
	01h	00h	ISA bridge
	02h	00h	EISA bridge
	03h	00h	MCA bridge
	04h	00h	PCI-to-PCI bridge
		01h	Subtractive Decode PCI-to-PCI bridge. This interface code identifies the PCI-to-PCI bridge as a device that supports subtractive decoding in addition to all the currently defined functions of a PCI-to-PCI bridge.
	05h	00h	PCMCIA bridge
	06h	00h	NuBus bridge
	07h	00h	CardBus bridge
	08h	xxh	RACEway bridge (see below)
	09h	40h	Semi-transparent PCI-to-PCI bridge with the primary PCI bus side facing the system host processor
		80h	Semi-transparent PCI-to-PCI bridge with the secondary PCI bus side facing the system host processor
	0Ah	00h	InfiniBand-to-PCI host bridge
	80h	00h	Other bridge device

RACEway is an ANSI standard (ANSI/VITA 5-1994) switching fabric. For the Programming Interface bits, [7:1] are reserved, read-only, and return zeros. Bit 0 defines the operation mode and is read-only:

0 - Transparent mode

1 - End-point mode

D.8. Base Class 07h

This base class is defined for all types of simple communications controllers. Several sub-class values are defined, some of these having specific well-known register-level programming interfaces.

Base Class	Sub-Class	Interface	Meaning
07h	00h	00h	Generic XT-compatible serial controller
		01h	16450-compatible serial controller
		02h	16550-compatible serial controller
		03h	16650-compatible serial controller
		04h	16750-compatible serial controller
		05h	16850-compatible serial controller
		06h	16950-compatible serial controller
	01h	00h	Parallel port
		01h	Bi-directional parallel port
		02h	ECP 1.X compliant parallel port
		03h	IEEE1284 controller
		FEh	IEEE1284 target device (not a controller)
	02h	00h	Multiport serial controller
	03h	00h	Generic modem
		01h	Hayes compatible modem, 16450-compatible interface (see below)
		02h	Hayes compatible modem, 16550-compatible interface (see below)
		03h	Hayes compatible modem, 16650-compatible interface (see below)
		04h	Hayes compatible modem, 16750-compatible interface (see below)
	04h	00h	GPIB (IEEE 488.1/2) controller
	05h	00h	Smart Card
	80h	00h	Other communications device

For Hayes-compatible modems, the first base address register (at offset 10h) maps the appropriate compatible (i.e., 16450, 16550, etc.) register set for the serial controller at the beginning of the mapped space. Note that these registers can be either memory or I/O mapped depending what kind of BAR is used.

D.9. Base Class 08h

This base class is defined for all types of generic system peripherals. Several sub-class values are defined, most of these having a specific well-known register-level programming interface.

Base Class	Sub-Class	Interface	Meaning
08h	00h	00h	Generic 8259 PIC
		01h	ISA PIC
		02h	EISA PIC
		10h	I/O APIC interrupt controller (see below)
		20h	I/O(x) APIC interrupt controller
	01h	00h	Generic 8237 DMA controller
		01h	ISA DMA controller
		02h	EISA DMA controller
	02h	00h	Generic 8254 system timer
		01h	ISA system timer
		02h	EISA system timers (two timers)
	03h	00h	Generic RTC controller
		01h	ISA RTC controller
	04h	00h	Generic PCI Hot-Plug controller
	05h	00h	SD Host controller
	80h	00h	Other system peripheral

For I/O APIC Interrupt Controller, the Base Address Register at offset 10h is used to request a minimum of 32 bytes of non-prefetchable memory. Two registers within that space are located at Base+00h (I/O Select Register) and Base+10h (I/O Window Register). For a full description of the use of these registers, refer to the data sheet for the Intel 8237EB in the 82420/82430 PCIset EISA Bridge Databook #290483-003.

D.10. Base Class 09h

This base class is defined for all types of input devices. Several sub-class values are defined. A register-level programming interface is defined for gameport controllers.

Base Class	Sub-Class	Interface	Meaning
09h	00h	00h	Keyboard controller
	01h	00h	Digitizer (pen)
	02h	00h	Mouse controller
	03h	00h	Scanner controller
	04h	00h	Gameport controller (generic)
		10h	Gameport controller (see below)
	80h	00h	Other input controller

A gameport controller with a Programming Interface == 10h indicates that any Base Address registers in this function that request/assign I/O address space, the registers in that I/O space conform to the standard 'legacy' game ports. The byte at offset 00h in an I/O region behaves as a legacy gameport interface where reads to the byte return joystick/gamepad information, and writes to the byte start the RC timer. The byte at offset 01h is an alias of the byte at offset 00h. All other bytes in an I/O region are unspecified and can be used in vendor unique ways.

D.11. Base Class 0Ah

This base class is defined for all types of docking stations. No specific register-level programming interfaces are defined.

Base Class	Sub-Class	Interface	Meaning
0Ah	00h	00h	Generic docking station
	80h	00h	Other type of docking station

D.12. Base Class 0Bh

This base class is defined for all types of processors. Several sub-class values are defined corresponding to different processor types or instruction sets. There are no specific register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
0Bh	00h	00h	386
	01h	00h	486
	02h	00h	Pentium
	10h	00h	Alpha
	20h	00h	PowerPC
	30h	00h	MIPS
	40h	00h	Co-processor

D.13. Base Class 0Ch

This base class is defined for all types of serial bus controllers. Several sub-class values are defined. There are specific register-level programming interfaces defined for Universal Serial Bus controllers and IEEE 1394 controllers.

Base Class	Sub-Class	Interface	Meaning
0Ch	00	00h	IEEE 1394 (FireWire)
		10h	IEEE 1394 following the 1394 OpenHCI specification
	01h	00h	ACCESS.bus
	02h	00h	SSA
	03h	00h	Universal Serial Bus (USB) following the Universal Host Controller Specification
		10h	Universal Serial Bus (USB) following the Open Host Controller Specification
		20h	USB2 host controller following the Intel Enhanced Host Controller Interface
		80h	Universal Serial Bus with no specific programming interface
		FEh	USB device (not host controller)
	04h	00h	Fibre Channel
	05h	00h	SMBus (System Management Bus)
	06h	00h	InfiniBand
	07h (see Note 1 below)	00h	IPMI SMIC Interface
		01h	IPMI Kybd Controller Style Interface
		02h	IPMI Block Transfer Interface
	08h (see Note 2 below	00h	SERCOS Interface Standard (IEC 61491)
	09h	00h	CANbus

Notes:

1. The register interface definitions for the Intelligent Platform Management Interface (Sub-Class 07h) are in the IPMI specification.
2. There is no register level definition for the SERCOS Interface standard. For more information see IEC 61491.

D.14. Base Class 0Dh

This base class is defined for all types of wireless controllers. Several sub-class values are defined. There are no specific register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
0Dh	00	00h	iRDA compatible controller
	01h	00h	Consumer IR controller
	10h	00h	RF controller
	11h	00h	Bluetooth
	12h	00h	Broadband
	20h	00h	Ethernet (802.11a – 5 GHz)
	21h	00h	Ethernet (802.11b – 2.4 GHz)
	80h	00h	Other type of wireless controller

D.15. Base Class 0Eh

This base class is defined for intelligent I/O controllers. The primary characteristic of this base class is that the I/O function provided follows some sort of generic definition for an I/O controller.

Base Class	Sub-Class	Interface	Meaning
0Eh	00	xxh	Intelligent I/O (I2O) Architecture Specification 1.0
		00h	Message FIFO at offset 040h

The specification for Intelligent I/O Architecture I/O can be downloaded from: ftp.intel.com/pub/IAL/i2o/.

D.16. Base Class 0Fh

This base class is defined for satellite communication controllers. Controllers of this type are used to communicate with satellites.

Base Class	Sub-Class	Interface	Meaning
0Fh	01h	00h	TV
	02h	00h	Audio
	03h	00h	Voice
	04h	00h	Data

D.17. Base Class 10h

This base class is defined for all types of encryption and decryption controllers. Several sub-class values are defined. There are no register-level interfaces defined.

Base Class	Sub-Class	Interface	Meaning
10h	00h	00h	Network and computing en/decryption
	10h	00h	Entertainment en/decryption
	80h	00h	Other en/decryption

D.18. Base Class 11h

This base class is defined for all types of data acquisition and signal processing controllers. Several sub-class values are defined. There are no register-level interfaces defined.

Base Class	Sub-Class	Interface	Meaning
11h	00h	00h	DPIO modules
	01h	00h	Performance counters
	10h	00h	Communications synchronization plus time and frequency test/measurement
	20h	00h	Management card
	80h	00h	Other data acquisition/signal processing controllers

E

E. System Transaction Ordering

Many programming tasks, especially those controlling intelligent peripheral devices common in PCI systems, require specific events to occur in a specific order. If the events generated by the program do not occur in the hardware in the order intended by the software, a peripheral device may behave in a totally unexpected way. PCI transaction ordering rules are written to give hardware the flexibility to optimize performance by rearranging certain events that do not affect device operation, yet strictly enforce the order of events that do affect device operation.

One performance optimization that PCI systems are allowed to do is the posting of memory write transactions. Posting means the transaction is captured by an intermediate agent; e.g., a bridge from one bus to another, so that the transaction completes at the source before it actually completes at the intended destination. This allows the source to proceed with the next operation while the transaction is still making its way through the system to its ultimate destination.

While posting improves system performance, it complicates event ordering. Since the source of a write transaction proceeds before the write actually reaches its destination, other events that the programmer intended to happen after the write may happen before the write. Many of the PCI ordering rules focus on posting buffers requiring them to be flushed to keep this situation from causing problems.

If the buffer flushing rules are not written carefully, however, deadlock can occur. The rest of the PCI transaction ordering rules prevent the system buses from deadlocking when posting buffers must be flushed.

Simple devices do not post outbound transactions. Therefore, their requirements are much simpler than those presented here for bridges. Refer to Section 3.2.5.1 for the requirements for simple devices.

The focus of the remainder of this appendix is on a PCI-to-PCI bridge. This allows the same terminology to be used to describe a transaction initiated on either interface and is easier to understand. To apply these rules to other bridges, replace a PCI transaction type with its equivalent transaction type of the host bus (or other specific bus). While the discussion focuses on a PCI-to-PCI bridge, the concepts can be applied to all bridges.

The ordering rules for a specific implementation may vary. This appendix covers the rules for all accesses traversing a bridge assuming that the bridge can handle multiple transactions at the same time in each direction. Simpler implementations are possible but are not discussed here.

E.1. Producer - Consumer Ordering Model

The Producer - Consumer model for data movement between two masters is an example of a system that would require this kind of ordering. In this model, one agent, the Producer, produces or creates the data and another agent, the Consumer, consumes or uses the data. The Producer and Consumer communicate between each other via a flag and a status element. The Producer sets the flag when all the data has been written and then waits for a completion status code. The Consumer waits until it finds the flag set, then it resets the flag, consumes the data, and writes the completion status code. When the Producer finds the completion status code, it clears it and the sequence repeats. Obviously, the order in which the flag and data are written is important. If some of the Producer's data writes were posted, then without buffer-flushing rules it might be possible for the Consumer to see the flag set before the data writes had completed. The PCI ordering rules are written such that no matter which writes are posted, the Consumer can never see the flag set and read the data until the data writes are finished. This specification refers to this condition as "having a consistent view of data." Notice that if the Consumer were to pass information back to the Producer in addition to the status code, the order of writing this additional information and the status code becomes important, just as it was for the data and flag.

In practice, the flag might be a doorbell register in a device or it might be a main-memory pointer to data located somewhere else in memory. And the Consumer might signal the Producer using an interrupt or another doorbell register, rather than having the Producer poll the status element. But in all cases, the basic need remains the same; the Producer's writes to the data area must complete before the Consumer observes that the flag has been set and reads the data.

This model allows the data, the flag, the status element, the Producer, and the Consumer to reside anywhere in the system. Each of these can reside on different buses and the ordering rules maintain a consistent view of the data. For example, in Figure E-1, the agent producing the data, the flag, and the status element reside on Bus 1, while the actual data and the Consumer of the data both reside on Bus 0. The Producer writes the last data and the PCI-to-PCI bridge between Bus 0 and 1 completes the access by posting the data. The Producer of the data then writes the flag changing its status to indicate that the data is now valid for the Consumer to use. In this case, the flag has been set before the final datum has actually been written (to the final destination). PCI ordering rules require that when the Consumer of the data reads the flag (to determine if the data is valid), the read will cause the PCI-to-PCI bridge to flush the posted write data to the final destination before completing the read. When the Consumer determines the data is valid by checking the flag, the data is actually at the final destination.

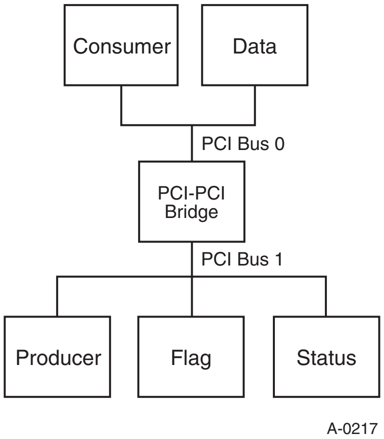


Figure E-1: Example Producer - Consumer Model

The ordering rules lead to the same results regardless of where the Producer, the Consumer, the data, the flag, and the status element actually reside. The data is always at the final destination before the Consumer can read the flag. This is true even when all five reside on different bus segments of the system. In one configuration, the data will be forced to the final destination when the Consumer reads the flag. In another configuration, the read of the flag occurs without forcing the data to its final destination; however, the read request of the actual data pushes the final datum to the final destination before completing the read.

A system may have multiple Producer-Consumer pairs operating simultaneously, with different data - flag-status sets located all around the system. But since only one Producer can write to a single data-flag set, there are no ordering requirements between different masters. Writes from one master on one bus may occur in one order on one bus, with respect to another master's writes, and occur in another order on another bus. In this case, the rules allow for some writes to be rearranged; for example, an agent on Bus 1 may see Transaction A from a master on Bus 1 complete first, followed by Transaction B from another master on Bus 0. An agent on Bus 0 may see Transaction B complete first followed by Transaction A. Even though the actual transactions complete in a different order, this causes no problem since the different masters must be addressing different data-flag sets.

E.2. Summary of PCI Ordering Requirements

Following is a summary of the general PCI ordering requirements presented in Section 3.2.5. These requirements apply to all PCI transactions, whether they are using Delayed Transactions or not.

General Requirements

1. The order of a transaction is determined when it completes. Transactions terminated with Retry are only requests and can be handled by the system in any order.
2. Memory writes can be posted in both directions in a bridge. I/O and Configuration writes are not posted. (I/O writes can be posted in the Host Bridge, but some restrictions apply.) Read transactions (Memory, I/O, or Configuration) are not posted.
3. Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus.
4. Write transactions crossing a bridge in opposite directions have no ordering relationship.
5. A read transaction must push ahead of it through the bridge any posted writes originating on the *same* side of the bridge and posted *before* the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the *opposite* side and were posted *before* the read command completes on the read-destination bus.
6. A bridge can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. Otherwise, a deadlock may occur. Bridges are allowed to refuse to accept a memory write for temporary conditions which are guaranteed to be resolved with time. A bridge can make the acceptance of a memory write transaction as a target contingent on the prior completion of locked transaction as a master only if the bridge has already established a locked operation with its intended target.

The following is a summary of the PCI ordering requirements specific to Delayed Transactions, presented in Section 3.3.3.3.

Delayed Transaction Requirements

1. A target that uses Delayed Transactions may be designed to have any number of Delayed Transactions outstanding at one time.
2. Only non-posted transactions can be handled as Delayed Transactions.
3. A master must repeat any transaction terminated with Retry since the target may be using a Delayed Transaction.
4. Once a Delayed Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, it is only a request and may be discarded at any time.
5. A Delayed Completion can only be discarded when it is a read from a prefetchable region, or if the master has not repeated the transaction in 2^{15} clocks.

6. A target must accept all memory writes addressed to it, even while completing a request using Delayed Transaction termination.
7. Delayed Requests and Delayed Completions are not required to be kept in their original order with respect to themselves or each other.
8. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything except another Posted Memory Write.
9. A single master may have any number of outstanding requests terminated with Retry. However, if a master requires one transaction to be completed before another, it cannot attempt the second one on PCI until the first one has completed.

E.3. Ordering of Requests

A transaction is considered to be a *request* when it is presented on the bus. When the transaction is terminated with Retry, it is still considered a request. A transaction becomes *complete* or a *completion* when data actually transfers (or is terminated with Master-Abort or Target-Abort). The following discussion will refer to transactions as being a request or completion depending on the success of the transaction.

A transaction that is terminated with Retry has no ordering relationship with any other access. Ordering of accesses is only determined when an access completes (transfers data). For example, four masters A, B, C, and D reside on the same bus segment and all desire to generate an access on the bus. For this example, each agent can only request a single transaction at a time and will not request another until the current access completes. The order in which transactions complete are based on the algorithm of the arbiter and the response of the target, not the order in which each agent's REQ# signal was asserted. Assuming that some requests are terminated with Retry, the order in which they complete is independent of the order they were first requested. By changing the arbiter's algorithm, the completion of the transactions can be any sequence (i.e., A, B, C, and then D or B, D, C, and then A, and so on). Because the arbiter can change the order in which transactions are requested on the bus, and, therefore, the completion of such transactions, the system is allowed to complete them in any order it desires. This means that a request from any agent has no relationship with a request from any other agent. The only exception to this rule is when LOCK# is used, which is described later.

Take the same four masters (A, B, C, and D) used in the previous paragraph and integrate them onto a single piece of silicon (a multi-function device). For a multi-function device, the four masters operate independent of each other, and each function only presents a single request on the bus for this discussion. The order their requests complete is the same as if they where separate agents and not a multi-function device, which is based on the arbitration algorithm. Therefore, multiple requests from a single agent may complete in any order, since they have no relationship to each other.

Another device, not a multi-function device, has multiple internal resources that can generate transactions on the bus. If these different sources have some ordering relationship, then the device must ensure that only a single request is presented on the bus at any one time. The agent must not attempt a subsequent transaction until the previous transaction completes. For example, a device has two transactions to complete on the bus, Transaction A and Transaction B and A must complete before B to preserve internal ordering requirements. In this case, the master cannot attempt B until A has completed.

The following example would produce inconsistent results if it were allowed to occur. Transaction A is to a flag that covers data, and Transaction B accesses the actual data covered by the flag. Transaction A is terminated with Retry, because the addressed target is currently busy or resides behind a bridge. Transaction B is to a target that is ready and will complete the request immediately. Consider what happens when these two transactions are allowed to complete in the wrong order. If the master allows Transaction B to be presented on the bus after Transaction A was terminated with Retry, Transaction B can complete before Transaction A. In this case, the data may be accessed before it is actually valid. The responsibility to prevent this from occurring rests with the master, which must block Transaction B from being attempted on the bus until Transaction A completes. A master presenting multiple transactions on the bus must ensure that subsequent requests (that have some relationship to a previous request) are not presented on the bus until the previous request has completed. The system is allowed to complete multiple requests from the same agent in any order. When a master allows multiple requests to be presented on the bus without completing, it must repeat each request independent of how any of the other requests complete.

E.4. Ordering of Delayed Transactions

A Delayed Transaction progresses to completion in three phases:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first phase, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. During the second phase, the target independently completes the request on the destination bus using the latched information from the Delayed Request. The result of completing the Delayed Request on the destination bus produces a Delayed Completion, which consists of the latched information of the Delayed Request and the completion status (and data if a read request). During the third phase, the master successfully re-arbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed.

The number of simultaneous Delayed Transactions a bridge is capable of handling is limited by the implementation and not by the architecture. Table E-1 represents the ordering rules when a bridge in the system is capable of allowing multiple transactions to proceed in each direction at the same time. Each column of the table represents an access that was accepted by the bridge earlier, while each row represents a transaction just accepted. The contents of the box indicate what ordering relationship the second transaction must have to the first.

PMW - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

DRR - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple commands. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus the DRR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

DWR - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write command. Note: Memory Write and Memory Write and Invalidate commands must be posted (PMW) and not be completed as DWR. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes. Before it is attempted on the destination bus, the DWR is only a request and may be discarded at any time to prevent deadlock or improve performance, since the master must repeat the request later.

DRC - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The DRC contains the data requested by the master and the status of the target (normal, Master-Abort, Target-Abort, parity error, etc.).

DWC - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The DWC does not contain the data of the access, but only status of how it completed (Normal, Master-Abort, Target-Abort, parity error, etc.). The write data has been written to the specified target.

No - indicates that the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes found in column 2 prevent PMW data from being passed by other accesses and thereby maintain a consistent view of data in the system.

Yes - indicates that the subsequent transaction must be allowed to complete before the previous one or a deadlock can occur.

When blocking occurs, the PMW is required to pass the Delayed Transaction. If the master continues attempting to complete Delayed Requests, it must be fair in attempting to complete the PMW. There is no ordering violation when these subsequent transactions complete before a prior transaction.

Yes/No - indicates that the bridge designer may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid. How a bridge designer chooses to implement these boxes may have a cost impact on the bridge implementation or performance impact on the system.

Table E-1: Ordering Rules for a Bridge

Row pass Col.?	**PMW** (Col 2)	**DRR** (Col 3)	**DWR** (Col 4)	**DRC** (Col 5)	**DWC** (Col 6)
PMW (Row 1)	No[1]	Yes[5]	Yes[5]	Yes[7]	Yes[7]
DRR (Row 2)	No[2]	Yes/No	Yes/No	Yes/No	Yes/No
DWR (Row 3)	No[3]	Yes/No	Yes/No	Yes/No	Yes/No
DRC (Row 4)	No[4]	Yes[6]	Yes[6]	Yes/No	Yes/No
DWC (Row 5)	Yes/No	Yes[6]	Yes[6]	Yes/No	Yes/No

Rule 1: A subsequent PMW cannot pass a previously accepted PMW. (Col 2, Row 1)

Posted Memory write transactions must complete in the order they are received. If the subsequent write is to the flag that covers the data, the Consumer may use stale data if write transactions are allowed to pass each other.

Rule 2: A read transaction must push posted write data to maintain ordering. (Col 2, Row 2)

For example, a memory write to a location followed by an immediate memory read of the same location returns the new value (refer to Section 3.10, item 6, for possible exceptions). Therefore, a memory read cannot pass posted write data. An I/O read cannot pass a PMW, because the read may be ensuring the write data arrives at the final destination.

Rule 3: A non-postable write transaction must push posted write data to maintain ordering. (Col 2, Row 3)

A Delayed Write Request may be the flag that covers the data previously written (PMW), and, therefore, the write flag cannot pass the data that it potentially covers.

Rule 4: A read transaction must pull write data back to the originating bus of the read transaction. (Col 2, Row 4)

For example, the read of a status register of the device writing data to memory must not complete before the data is pulled back to the originating bus; otherwise, stale data may be used.

Rule 5: A Posted Memory Write must be allowed to pass a Delayed Request (read or write) to avoid deadlocks. (Col 3 and Col 4, Row 1)

A deadlock can occur when bridges that support Delayed Transactions are used with bridges that do not support Delayed Transactions. Referring to Figure E-2, a deadlock can occur when Bridge Y (using Delayed Transactions) is between Bridges X and Z (designed to a previous version of this specification and not using Delayed Transactions). Master 1 initiates a read to Target 1 that is forwarded through Bridge X and is queued as a Delayed Request in

Bridge Y. Master 3 initiates a read to Target 3 that is forwarded through Bridge Z and is queued as a Delayed Request in Bridge Y. After Masters 1 and 3 are terminated with Retry, Masters 2 and 4 begin memory write transactions of a long duration addressing Targets 2 and 4 respectively, which are posted in the write buffers of Bridges X and Z respectively. When Bridge Y attempts to complete the read in either direction, Bridges X and Z must flush their posted write buffers before allowing the Read Request to pass through it.

If the posted write buffers of Bridges X and Z are larger than those of Bridge Y, Bridge Y's buffers will fill. If posted write data is not allowed to pass the DRR, the system will deadlock. Bridge Y cannot discard the read request since it has been attempted, and it cannot accept any more write data until the read in the opposite direction is completed. Since this condition exists in both directions, neither DRR can complete because the other is blocking the path. Therefore, the PMW data is required to pass the DRR when the DRR blocks forward progress of PMW data.

The same condition exists when a DWR sits at the head of both queues, since some old bridges also require the posting buffers to be flushed on a non-posted write cycle.

Rule 6: A Delayed Completion (read or write) must be allowed to pass a Delayed Request (read or write) to avoid deadlocks. (Cols 3 and 4, Rows 4 and 5)

A deadlock can occur when two bridges that support Delayed Transactions are requesting accesses to each other. The common PCI bus segment is on the secondary bus of Bridge A and the primary bus for Bridge B. If neither bridge allows Delayed Completions to pass the Delayed Requests, neither can make progress.

For example, suppose Bridge A's request to Bridge B completes on Bridge B's secondary bus, and Bridge B's request completes on Bridge A's primary bus. Bridge A's completion is now behind Bridge B's request and Bridge B's completion is behind Bridge A's request. If neither bridge allows completions to pass the requests, then a deadlock occurs because neither master can make progress.

Rule 7: A Posted Memory Write must be allowed to pass a Delayed Completion (read or write) to avoid deadlocks. (Col 5 and Col 6, Row 1)

As in the example for Rule 5, another deadlock can occur in the system configuration in Figure E-2. In this case, however, a DRC sits at the head of the queues in both directions of Bridge Y at the same time. Again the old bridges (X and Z) contain posted write data from another master. The problem in this case, however, is that the read transaction cannot be *repeated* until all the posted write data is flushed out of the old bridge and the master is allowed to repeat its original request. Eventually, the new bridge cannot accept any more posted data because its internal buffers are full and it cannot drain them until the DRC at the other end completes. When this condition exists in both directions, neither DRC can complete because the other is blocking the path. Therefore, the PMW data is required to pass the DRC when the DRC blocks forward progress of PMW data.

The same condition exists when a DWC sits at the head of both queues.

Transactions that have no ordering constraints

Some transactions enqueued as Delayed Requests or Delayed Completions have no ordering relationship with any other Delayed Requests or Delayed Completions. The designer can (for performance or cost reasons) allow or disallow Delayed Requests to pass other Delayed Requests and Delayed Completions that were previously enqueued.

☐ Delayed Requests can pass other Delayed Requests. (Cols 3 and 4, Rows 2 and 3)

Since Delayed Requests have no ordering relationship with other Delayed Requests, these four boxes are don't cares.

☐ Delayed Requests can pass Delayed Completions. (Col 5 and 6, Rows 2 and 3)

Since Delayed Requests have no ordering relationship with Delayed Completions, these four boxes are don't cares.

☐ Delayed Completions can pass other Delayed Completions. (Col 5 and 6, Rows 4 and 5)

Since Delayed Completions have no ordering relationship with other Delayed Completions, these four boxes are don't cares.

☐ Delayed Write Completions can pass posted memory writes or be blocked by them. (Col 2, Row 5)

If the DWC is allowed to pass a PMW or if it remains in the same order, there is no deadlock or data inconsistencies in either case. The DWC data and the PMW data are moving in opposite directions, initiated by masters residing on different buses accessing targets on different buses.

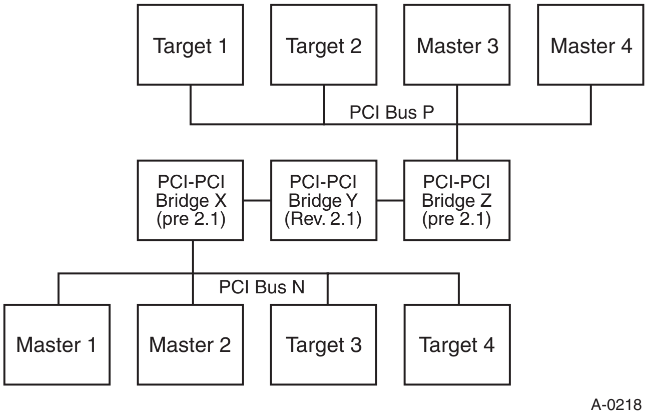


Figure E-2: Example System with PCI-to-PCI Bridges

E.5. Delayed Transactions and LOCK#

The bridge is required to support LOCK# when a transaction is initiated on its primary bus (and is using the lock protocol), but is not required to support LOCK# on transactions that are initiated on its secondary bus. If a locked transaction is initiated on the primary bus and the bridge is the target, the bridge must adhere to the lock semantics defined by this specification. The bridge is required to complete (push) all PMWs (accepted from the primary bus) onto the secondary bus before attempting the lock on the secondary bus. The bridge may discard any requests enqueued, allow the locked transaction to pass the enqueued requests, or simply complete all enqueued transactions before attempting the locked transaction on the secondary interface. Once a locked transaction has been enqueued by the bridge, the bridge cannot accept any other transaction from the primary interface until the lock has completed except for a continuation of the lock itself by the lock master. Until the lock is established on the secondary interface, the bridge is allowed to continue enqueuing transactions from the secondary interface, but not the primary interface. Once lock has been established on the secondary interface, the bridge cannot accept any posted write data moving toward the primary interface until LOCK# has been released (FRAME# and LOCK# deasserted on the same rising clock edge). (In the simplest implementation, the bridge does not accept any other transactions in either direction once lock is established on the secondary bus, except for locked transactions from the lock master.) The bridge must complete PMW, DRC, and DWC transactions moving toward the primary bus before allowing the locked access to complete on the originating bus. The preceding rules are sufficient for deadlock free operation. However, an implementation may be more or less restrictive, but, in all cases must ensure deadlock-free operation.

E.6. Error Conditions

A bridge is free to discard data or status of a transaction that was completed using Delayed Transaction termination when the master has not repeated the request within 2^{10} PCI clocks (about 30 µs at 33 MHz). However, it is recommended that the bridge not discard the transaction until 2^{15} PCI clocks (about 983 µs at 33 MHz) after it acquired the data or status. The shorter number is useful in a system where a master designed to a previous version of this specification frequently fails to repeat a transaction exactly as first requested. In this case, the bridge may be programmed to discard the abandoned Delayed Completion early and allow other transactions to proceed. Normally, however, the bridge would wait the longer time, in case the repeat of the transaction is being delayed by another bridge or bridges designed to a previous version of this specification that did not support Delayed Transactions.

When this timer (referred to as the Discard Timer) expires, the device is required to discard the data; otherwise, a deadlock may occur.

> Note: When the transaction is discarded, data may be destroyed. This occurs when the discarded Delayed Completion is a read to a non-prefetchable region.

When the Discard Timer expires, the device may choose to report or ignore the error. When the data is prefetchable, it is recommended that the device ignore the error since system integrity is not affected. However, when the data is not prefetchable, it is recommended that the device report the error to its device driver since system integrity is affected. A bridge may assert SERR# since it typically does not have a device driver.

F. Exclusive Accesses

The use of LOCK# is only allowed to be supported by a host bus bridge, a PCI-to-PCI bridge, or an expansion bus bridge. In earlier versions of this specification, other devices were allowed to initiate and respond to exclusive accesses using LOCK#. However, the usefulness of a hardware-based lock mechanism has diminished and is only useful to prevent a deadlock or to provide backward compatibility. Therefore, all other devices are required to ignore LOCK#.

IMPLEMENTATION NOTE

Restricted LOCK# Usage

The use of LOCK# by a host bridge is permitted but strongly discouraged. A non-bridge device that uses LOCK# is not compliant with this specification. The use of LOCK# may have significant negative impacts on bus performance.

☐ PCI-to-PCI Bridges must not accept any new requests while they are in a locked condition except from the owner of LOCK# (see Section F.1).

☐ Arbiters are permitted to grant exclusive access of the bus to the agent that owns LOCK# (see section F.1).

These two characteristics of LOCK# may result in data overruns for audio, streaming video, and communications devices (plus other less real time sensitive devices). The goal is to drive the use of LOCK# to zero and then delete it from all PCI specifications.

A host bus bridge can only initiate an exclusive access to prevent a deadlock as described in Section 3.10, item 5, or to provide backward compatibility to an expansion bus bridge that supports exclusive access. A host bus bridge can only honor an exclusive access as a target when providing compatibility to an access initiated by an expansion bus bridge that supports exclusive accesses. (No other agent can initiate a locked access to the Host Bus bridge.)

A PCI-to-PCI bridge is only allowed to propagate an exclusive access from its primary bus to its secondary bus and is never allowed to initiate an exclusive access of its own initiative. A PCI-to-PCI bridge is required to ignore LOCK# when acting as a target on its secondary interface.

An expansion bus bridge is only allowed to initiate an exclusive access to provide backward compatibility. This means that the expansion bus supports a hardware based exclusive access mechanism (i.e., EISA and not ISA). The expansion bus bridge can honor an exclusive access as a target when supported by the expansion bus; otherwise, LOCK# has no meaning to the bridge.

The remainder of this chapter is only applicable to a device that is allowed to support LOCK#. Note: existing software that does not support the PCI lock usage rules has the potential of not working correctly. Software is not allowed to use an exclusive access to determine if a device is present.

F.1. Exclusive Accesses on PCI

PCI provides an exclusive access mechanism, which allows non-exclusive accesses to proceed in the face of exclusive accesses. This allows a master to hold a hardware lock across several accesses without interfering with non-exclusive data transfer, such as real-time video between two devices on the same bus segment. The mechanism is based on locking only the PCI resource to which the original locked access was targeted and is called a resource lock.

LOCK# indicates whether the master desires the current transaction to complete as an exclusive access or not. Control of LOCK# is obtained under its own protocol in conjunction with GNT#. Refer to Section F.2 for details. Masters and targets not involved in the exclusive access are allowed to proceed with non-exclusive accesses while another master retains ownership of LOCK#. However, when compatibility dictates, the arbiter can optionally grant the agent that owns LOCK# exclusive access to the bus until LOCK# is released. This is referred to as complete bus lock and is described in Section F.7. For a resource lock, the target of the access guarantees exclusivity.

The following paragraphs describe the behavior of a master and a target for a locked operation. The rules of LOCK# will be stated for both the master and target. A detailed discussion of how to start, continue, and complete an exclusive access operation follows the discussion of the rules. A discussion of how a target behaves when it supports a resource lock will follow the description of the basic lock mechanism. The concluding section will discuss how to implement a complete bus lock.

Master rules for supporting LOCK#:

1. A master can access only a single resource[61] during a lock operation.
2. The first transaction of a lock operation must be a memory read transaction.
3. LOCK# must be asserted the clock[62] following the address phase and kept asserted to maintain control.
4. LOCK# must be released if the initial transaction of the lock request is terminated with Retry.[63] (Lock was not established.)
5. LOCK# must be released whenever an access is terminated by Target-Abort or Master-Abort.
6. LOCK# must be deasserted between consecutive[64] lock operations for a minimum of one clock while the bus is in the Idle state.

Target Rules for supporting LOCK#:

1. A bridge acting as a target of an access locks itself when LOCK# is deasserted during the address phase and is asserted on the following clock.
2. Once lock is established,[65] a bridge remains locked until both FRAME# and LOCK# are sampled deasserted regardless of how the transaction is terminated.
3. The bridge is not allowed to accept any new requests (from either interface) while it is in a locked condition except from the owner of LOCK#.

F.2. Starting an Exclusive Access

When an agent needs to do an exclusive operation, it checks the internally tracked state of LOCK# before asserting REQ#. The master marks LOCK# busy anytime LOCK# is asserted (unless it is the master that owns LOCK#) and not busy when both FRAME# and LOCK# are deasserted. If LOCK# is busy (and the master does not own LOCK#), the agent must delay the assertion of REQ# until LOCK# is available.

While waiting for GNT#, the master continues to monitor LOCK#. If LOCK# is ever busy, the master deasserts REQ# because another agent has gained control of LOCK#.

---

[61] In previous versions of this specification, a minimum of 16 bytes (naturally aligned) was considered the lock resource. A device was permitted to lock its entire memory address space. This definition still applies for an upstream locked access to main memory. For downstream locked access, a resource is the PCI-to-PCI bridge or the Expansion Bus bridge that is addressed by the locked operation.

[62] For a SAC, this is the clock after the address phase. For a DAC, this occurs the clock after the first address phase.

[63] Once lock has been established, the master retains ownership of LOCK# when terminated with Retry or Disconnect.

[64] Consecutive refers to back-to-back locked operations and not a continuation of the current locked operation.

[65] A locked operation is established when LOCK# is deasserted during the address phase, asserted the following clock, and data is transferred during the current transaction.

When the master is granted access to the bus and LOCK# is not busy, ownership of LOCK# has been obtained. The master is free to perform an exclusive operation when the current transaction completes and is the only agent on the bus that can drive LOCK#. All other agents must not drive LOCK#, even when they are the current master.

Figure F-1 illustrates starting an exclusive access. LOCK# is deasserted during the address phase (one clock for SAC or DAC) to request a lock operation, which must be initiated with a memory read command. LOCK# must be asserted the clock following the first address phase, which occurs on clock 3 to keep the target in the locked state. This allows the current master to retain ownership of LOCK# beyond the end of the current transaction.

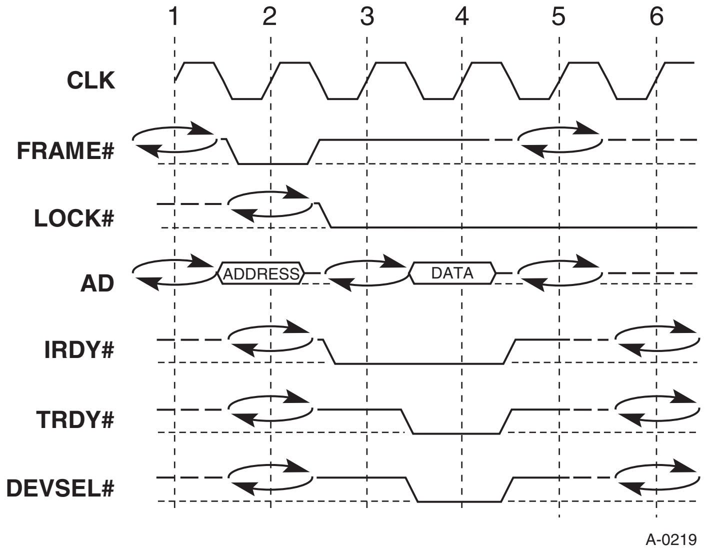


Figure F-1: Starting an Exclusive Access

A locked operation is not established on the bus until completion of the first data phase of the first transaction (IRDY# and TRDY# asserted). If the target terminates the first transaction with Retry, the master terminates the transaction and releases LOCK#. Once the first data phase completes with both TRDY# and IRDY# asserted, the exclusive operation is established and the master keeps LOCK# asserted until either the lock operation completes or an error (Master-Abort or Target-Abort) causes an early termination. Target termination of Retry and Disconnect is normal termination even when a lock operation is established. When a master is terminated by the target with Disconnect or Retry after the lock has been established, the target is indicating it is currently busy and unable to complete the requested data phase. The target will accept the access when it is not busy and continues to honor the lock by excluding all other accesses. The master continues to control LOCK# if this condition occurs. Non-exclusive accesses to unlocked targets on the same PCI bus segment are allowed to occur while LOCK# is asserted. However, transactions to other bus segments are not allowed to cross a locked bridge.

When a bridge is locked, it may only accept requests when LOCK# is deasserted during the address phase (which indicates that the transaction is a continuation of the exclusive access sequence by the master that established the lock). If LOCK# is asserted during the address phase, a locked bridge will terminate all accesses by asserting STOP# with TRDY#

deasserted (Retry). A locked target remains in the locked state until both FRAME# and LOCK# are deasserted.

Delayed Transactions and Lock

A locked transaction can be completed using Delayed Transaction termination. All the rules of LOCK# still apply except the bridge must consider itself locked when it enqueues the request even though no data has transferred. This condition is referred to as a *target-lock*. While in target-lock, the bridge enqueues no new requests on the primary interface and terminates all new requests with Retry. The bridge locks its secondary interface when lock is established on the secondary bus and starts checking for the repeat of the original lock request on the primary interface. A target-lock becomes a *full-lock* when the master repeats the locked request and the bridge transfers data. At this point, the master has established the lock.

A bridge acting as a target that supports exclusive accesses must sample LOCK# with the address and on a subsequent clock. If the bridge performs medium or slow decode, it must latch LOCK# during the first address phase. Otherwise, the bridge cannot determine if the access is a lock operation when decode completes. A bridge marks itself as target-locked if LOCK# is deasserted during the first address phase and is asserted on the next clock. A bridge does not mark itself target-locked if LOCK# is deasserted the clock following the first address phase and is free to respond to other requests.

F.3. Continuing an Exclusive Access

Figure F-2 shows a master continuing an exclusive access. However, this access may or may not complete the exclusive operation. When the master is granted access to the bus, it starts another exclusive access to the target it previously locked. LOCK# is deasserted during the address phase to continue the lock. The locked device accepts and responds to the request. LOCK# is asserted on clock 3 to keep the target in the locked state and allow the current master to retain ownership of LOCK# beyond the end of the current transaction.

When the master is continuing the lock operation, it continues to assert LOCK#. When the master completes the lock operation, it deasserts LOCK# after the completion of the last data phase which occurs on clock 5. Refer to Section F.5 for more information on completing an exclusive access.

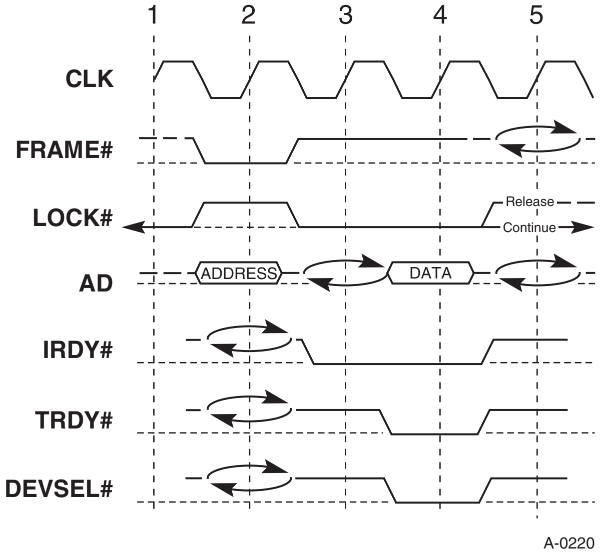


Figure F-2: Continuing an Exclusive Access

F.4. Accessing a Locked Agent

Figure F-3 shows a master trying a non-exclusive access to a locked agent. When LOCK# is asserted during the address phase, and if the target is locked (full-lock or target-lock), it terminates the transaction with Retry and no data is transferred.

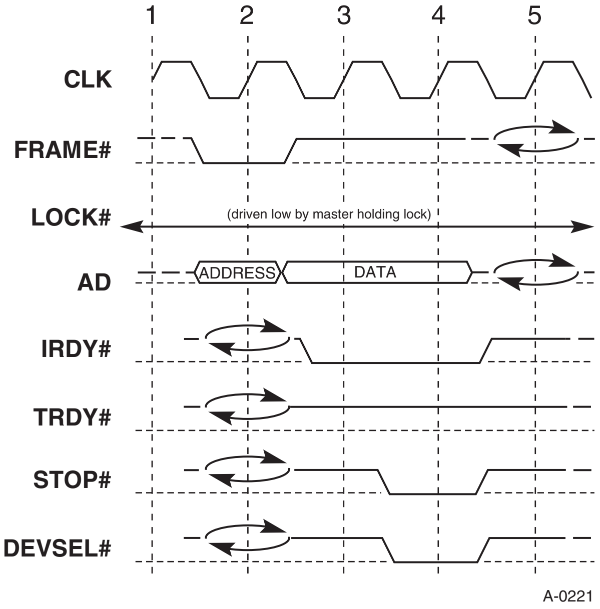


Figure F-3: Accessing a Locked Agent

F.5. Completing an Exclusive Access

During the final transaction of an exclusive operation, LOCK# is deasserted so the target will accept the request, and then reasserted until the exclusive access terminates successfully. The master may deassert LOCK# at any time when the exclusive operation has completed. However, it is recommended (but not required) that LOCK# be deasserted with the deassertion of IRDY# following the completion of the last data phase of the locked operation. Releasing LOCK# at any other time may result in a subsequent transaction being terminated with Retry unnecessarily. A locked agent unlocks itself whenever LOCK# and FRAME# are deasserted.

If a master wants to execute two independent exclusive operations on the bus, it must ensure a minimum of one clock between operations where both FRAME# and LOCK# are deasserted. (For example, the fast back-to-back case depicted in Figure 3-16 (clock 3) cannot lock both transactions.) This ensures any target locked by the first operation is released prior to starting the second operation. (An agent must unlock itself when FRAME# and LOCK# are both deasserted on the same clock.)

F.6. Complete Bus Lock

The PCI resource lock can be converted into a complete bus lock by having the arbiter not grant the bus to any other agent while LOCK# is asserted. If the first access of the locked sequence is terminated with Retry, the master must deassert both REQ# and LOCK#. If the first access completes normally, the complete bus lock has been established and the arbiter will not grant the bus to any other agent. If the arbiter granted the bus to another agent when the complete bus lock was being established, the arbiter must remove the other grant to ensure that complete bus lock semantics are observed. A complete bus lock may have a significant impact on the performance of the system, particularly the video subsystem. All non-exclusive accesses will not proceed while a complete bus lock operation is in progress

G. I/O Space Address Decoding for Legacy Devices

A function that supports a PC legacy function (IDE, VGA, etc.) is allowed to claim those addresses associated with the specific function when the I/O Space (see Figure 6-2) enable bit is set.

These addresses are not requested using a Base Address register but are assigned by initialization software. If a device identifies itself as a legacy function (class code), the initialization software grants the device permission to claim the I/O legacy addresses by setting the device's I/O Space enable bit.

If the device does not own all bytes within a DWORD of a legacy I/O range, it is required to use AD[1::0] to complete the decode before claiming the access by asserting DEVSEL#. If a legacy function is addressed by an I/O transaction, but does not own all bytes being accessed in the DWORD, it is required to terminate the transaction with Target-Abort. An expansion bus bridge is granted an exception to this requirement when performing subtractive decode. The bridge is permitted to assume that all bytes within the DWORD being addressed reside on the expansion bus. This means that the bridge is not required to check the encoding of AD[1::0], and the byte enables before passing the request to the expansion bus to complete.

H

H. Capability IDs

This appendix describes the current Capability IDs. Each defined capability must have a PCI SIG-assigned ID code. These codes are assigned and handled much like the Class Codes.

Section 6.7 of this specification provides a full description of the Extended Capabilities mechanism for PCI devices.

Table H-1: Capability IDs

ID	Capability
00h	Reserved
01h	PCI Power Management Interface – This capability structure provides a standard interface to control power management features in a PCI device. It is fully documented in the *PCI Power Management Interface Specification*. This document is available from the PCI SIG as described in Chapter 1 of this specification.
02h	AGP – This capability structure identifies a controller that is capable of using Accelerated Graphics Port features. Full documentation can be found in the *Accelerated Graphics Port Interface Specification.* This is available at http://www.agpforum.org.
03h	VPD – This capability structure identifies a device that supports Vital Product Data. Full documentation of this feature can be found in Section 6.4 and Appendix I of this specification.
04h	Slot Identification – This capability structure identifies a bridge that provides external expansion capabilities. Full documentation of this feature can be found in the *PCI to PCI Bridge Architecture Specification*. This document is available from the PCI SIG as described in Chapter 1 of this specification.
05h	Message Signaled Interrupts – This capability structure identifies a PCI function that can do message signaled interrupt delivery as defined in Section 6.8 of this specification.
06h	CompactPCI Hot Swap – This capability structure provides a standard interface to control and sense status within a device that supports Hot Swap insertion and extraction in a CompactPCI system. This capability is documented in the *CompactPCI Hot Swap Specification PICMG 2.1, R1.0* available at http://www.picmg.org.
07h	PCI-X – Refer to the *PCI-X Addendum to the PCI Local Bus Specification* for details.
08h	HyperTransport – This capability structure provides control and status for devices that implement HyperTransport Technology links. For details, refer to the HyperTransport I/O Link Specification available at http://www.hypertransport.org.

ID	Capability
09h	Vendor Specific – This ID allows device vendors to use the capability mechanism for vendor specific information. The layout of the information is vendor specific, except that the byte immediately following the “Next” pointer in the capability structure is defined to be a length field. This length field provides the number of bytes in the capability structure (including the ID and Next pointer bytes). An example vendor specific usage is a device that is configured in the final manufacturing steps as either a 32-bit or 64-bit PCI agent and the Vendor Specific capability structure tells the device driver which features the device supports.
0Ah	Debug port
0Bh	CompactPCI central resource control – Definition of this capability can be found in the *PICMG 2.13 Specification* (http://www.picmg.com).
0Ch	PCI Hot-Plug – This ID indicates that the associated device conforms to the Standard Hot-Plug Controller model.
0Dh	PCI Bridge Subsystem Vendor ID
0Eh	AGP 8x
0Fh	Secure Device
10h	PCI Express
11h	MSI-X – This ID identifies an optional extension to the basic MSI functionality.
12h-0FFh	Reserved

I. Vital Product Data

Vital Product Data (VPD) is information that uniquely identifies hardware and, potentially, software elements of a system. The VPD can provide the system with information on various Field Replaceable Units such as part number, serial number, and other detailed information. The objective from a system point of view is to make this information available to the system owner and service personnel. Support of VPD is optional.

VPD resides in a storage device (for example, a serial EEPROM) in a PCI device. Access to the VPD is provided using the Capabilities List in Configuration Space. The VPD capability structure has the following format.

31	30 16	15 8	7 0
F	VPD Address	Pointer to Next ID	ID = 03h
VPD Data			

Figure I-1: VPD Capability Structure

Register Field Descriptions:

ID—Capability structure ID 03h, which is a read-only field.

Pointer to Next ID—Pointer to the next capability structure, or 00h if this is the last structure in the Capability List. This is a read-only field.

VPD Address—DWORD-aligned byte address of the VPD to be accessed. The register is read/write, and the initial value at power-up is indeterminate.

F—A flag used to indicate when the transfer of data between the VPD Data register and the storage component is completed. The flag register is written when the VPD Address register is written. To read VPD information, a zero is written to the flag register when the address is written to the VPD Address register. The hardware device will set the flag to a one when 4 bytes of data from the storage component have been transferred to the VPD Data register. Software can monitor the flag and, after it is set to a one, read the VPD information from the VPD Data register. If either the VPD Address or VPD Data register is written, prior to the flag bit being set to a one, the results of the original read operation are unpredictable. To write VPD information, to the read/write portion of the VPD space, write the data to the VPD Data register. Then write the address of where the VPD data is to be stored into the VPD Address register and write the flag bit to a one (at the time the address is written). The software then monitors the flag bit and when it is set to zero (by device hardware), the VPD data (all 4 bytes) has been transferred from the VPD Data register to the storage component. If either the VPD Address or VPD Data register is written, prior to the flag bit being set to a zero, the results of the write operation to the storage component are unpredictable.

VPD Data—VPD data can be read through this register. The least significant byte of this register (at offset 4 in this capability structure) corresponds to the byte of VPD at the address specified by the VPD Address register. The data read from or written to this register uses the normal PCI byte transfer capabilities. Four bytes are always transferred between this register and the VPD storage component. Reading or writing data outside of the VPD space in the storage component is not allowed. The VPD (both the read only items and the read/write fields) is stored information and will have no direct control of any device operations. The initial value of this register at power up is indeterminate.

The VPD Address field is a byte address but must specify a DWORD-aligned location (i.e., the bottom two bits must be zero).

Every add-in card may contain VPD. When a PCI add-in card contains multiple devices, VPD, if provided, is required on only one of them but may be included in each. PCI devices designed exclusively for use on the system board may also support the optional VPD registers.

VPD in a PCI add-in card uses two of the predefined tag item names previously defined in the *Plug and Play ISA Specification* and two new ones defined specifically for PCI VPD. The PnP ISA tag item names that are used are: Identifier String (02h) for a Large Resource Data Type and End Tag (0Fh) for a Small Resource Data Type. The new large resource item names for VPD are VPD-R with a value of 10h for read only data and VPD-W with a value of 11h for read/write data.

Vital Product Data is made up of Small and Large Resource Data Types as described in the *Plug and Play ISA Specification,* Version 1.0a. Use of these data structures allows leveraging of data types already familiar to the industry and minimizes the amount of additional resources needed for support. This data format consists of a series of "tagged" data structures. The data types from the *Plug and Play ISA Specification*, Version 1.0a, are reproduced in the following figures.

Offset	Field Name
Byte 0	Value = 0xxx xyyyb (Type = Small(0), Small Item Name = xxxx, length = yy bytes
Bytes 1 to n	Actual information

Figure I-2: Small Resource Data Type Tag Bit Definitions

Offset	Field Name
Byte 0	Value = 1xxx xxxxb (Type = Large(1), Large item name = xxxxxxx)
Byte 1	Length of data items bits[7:0] (lsb)
Byte 2	Length of data items bits[15:8] (msb)
Byte 3 to n	Actual data items

Figure I-3: Large Resource Data Type Tag Bit Definitions

The Identifier String (02h) tag is the first VPD tag and provides the product name of the device. One VPD-R (10h) tag is used as a header for the read-only keywords, and one VPD-W (11h) tag is used as a header for the read-write keywords. The VPD-R list (including tag and length) must checksum to zero. The storage component containing the read/write data is a non-volatile device that will retain the data when powered off. Attempts to write the read-only data will be executed as a no-op. The last tag must be the End Tag (0Fh). A small example of the resource data type tags used in a typical VPD is shown in Figure I-4.

TAG	Identifier String
TAG	VPD-R list containing one or more VPD keywords
TAG	VPD-W list containing one or more VPD keywords
TAG	End Tag

Figure I-4: Resource Data Type Flags for a Typical VPD

I.1. VPD Format

Information fields within a VPD resource type consist of a three-byte header followed by some amount of data (see Figure I-5). The three-byte header contains a two-byte keyword and a one-byte length.

A keyword is a two-character (ASCII) mnemonic that uniquely identifies the information in the field. The last byte of the header is binary and represents the length value (in bytes) of the data that follows.

Keyword		Length	Data
Byte 0	Byte 1	Byte 2	Bytes 3 through n

Figure I-5: VPD Format

VPD keywords are listed in two categories: read-only fields and read/write fields. Unless otherwise noted, keyword data fields are provided as ASCII characters. Use of ASCII allows keyword data to be moved across different enterprise computer systems without translation difficulty. An example of the "add-in card serial number" VPD item is as follows:

Keyword:	SN The S is in byte 0 (in Figure I-5) and the N is in byte 1.
Length:	08h
Data:	"00000194"

I.2. Compatibility

Optional VPD was supported in prior versions of this specification. For information on the previous definition of VPD, see *PCI Local Bus Specification*, *Revision 2.1*.

I.3. VPD Definitions

This section describes the current VPD large and small resource data tags plus the VPD keywords. This list may be enhanced at any time. Companies wishing to define a new keyword should contact the PCI SIG. All unspecified values are reserved for SIG assignment.

I.3.1. VPD Large and Small Resource Data Tags

VPD is contained in four types of Large and Small Resource Data Tags. The following tags and VPD keyword fields may be provided in PCI devices.

Large resource type Identifier String Tag (02h)	This tag is the first item in the VPD storage component. It contains the name of the add-in card in alphanumeric characters.
Large resource type VPD-R Tag (010h)	This tag contains the read only VPD keywords for an add-in card.
Large resource type VPD-W Tag (011h)	This tag contains the read/write VPD keywords for an add-in card.
Small resource type End Tag (0Fh)	This tag identifies the end of VPD in the storage component.

I.3.1.1. Read-Only Fields

PN	Add-in Card Part Number	This keyword is provided as an extension to the Device ID (or Subsystem ID) in the Configuration Space header in Figure 6-1.
EC	EC Level of the Add-in Card	The characters are alphanumeric and represent the engineering change level for this add-in card.
FG	Fabric Geography	This keyword provides a standard interface for inventorying the fabric devices on a CompactPCI board and is applicable to multiple switch fabric architectures. This keyword is defined and used in specifications maintained by the PCI Industrial Computer Manufacturers Group (PICMG) and available at *www.picmg.org*, but also follows the architecture requirements defined in Appendix I of this specification. This keyword is intended to be used only in designs based on PICMG specifications.
LC	Location	This keyword provides a standard interface for determining the location of a CompactPCI board. For example, the keyword field can provide the physical slot number and chassis or shelf number where the board is installed. This keyword is defined and used in specifications maintained by the PICMG and available at *www.picmg.org*, but also follows the architecture requirements defined in Appendix I of this specification. This keyword is intended to be used only in designs based on PICMG specifications.
MN	Manufacture ID	This keyword is provided as an extension to the Vendor ID (or Subsystem Vendor ID) in the Configuration Space header in Figure 6-1. This allows vendors the flexibility to identify an additional level of detail pertaining to the sourcing of this device.
PG	PCI Geography	This keyword provides a standard interface for determining the PCI slot geography (the mapping between physical slot numbers and PCI logical addresses) of a segment of peripheral slots created by a CompactPCI Specification board. This keyword is defined and used in specifications maintained by the PICMG and available at *www.picmg.org*, but also follows the architecture requirements defined in Appendix I of this specification. This keyword is intended to be used only in designs based on PICMG specifications.
SN	Serial Number	The characters are alphanumeric and represent the unique add-in card Serial Number.

Vx	Vendor Specific	This is a vendor specific item and the characters are alphanumeric. The second character (x) of the keyword can be 0 through Z.
CP	Extended Capability	This field allows a new capability to be identified in the VPD area. Since dynamic control/status cannot be placed in VPD, the data for this field identifies where, in the device's memory or I/O address space, the control/status registers for the capability can be found. Location of the control/status registers is identified by providing the index (a value between 0 and 5) of the Base Address register that defines the address range that contains the registers, and the offset within that Base Address register range where the control/status registers reside. The data area for this field is four bytes long. The first byte contains the ID of the extended capability. The second byte contains the index (zero based) of the Base Address register used. The next two bytes contain the offset (in little-endian order) within that address range where the control/status registers defined for that capability reside.
RV	Checksum and Reserved	The first byte of this item is a checksum byte. The checksum is correct if the sum of all bytes in VPD (from VPD address 0 up to and including this byte) is zero. The remainder of this item is reserved space (as needed) to identify the last byte of read-only space. The read-write area does not have a checksum. This field is required.

I.3.1.2. Read/Write Fields

Vx	Vendor Specific	This is a vendor specific item and the characters are alphanumeric. The second character (x) of the keyword can be 0 through Z.
Yx	System Specific	This is a system specific item and the characters are alphanumeric. The second character (x) of the keyword can be 0 through 9 and B through Z.
YA	Asset Tag Identifier	This is a system specific item and the characters are alphanumeric. This keyword contains the system asset identifier provided by the system owner.
RW	Remaining Read/Write Area	This descriptor is used to identify the unused portion of the read/write space. The product vendor initializes this parameter based on the size of the read/write space or the space remaining following the Vx VPD items. One or more of the Vx, Yx, and RW items are required.

I.3.2. VPD Example

The following is an example of a typical VPD.

Offset	**Item**	**Value**
0	Large Resource Type ID String Tag (02h)	82h “Product Name”
1	Length	0021h
3	Data	“ABCD Super-Fast Widget Controller”
36	Large Resource Type VPD-R Tag (10h)	90h
37	Length	0059h
39	VPD Keyword	“PN”
41	Length	08h
42	Data	“6181682A”
50	VPD Keyword	“EC”
52	Length	0Ah
53	Data	“4950262536”
63	VPD Keyword	“SN”
65	Length	08h
66	Data	“00000194”
74	VPD Keyword	“MN”

Offset	Item	Value
76	Length	04h
77	Data	“1037”
81	VPD Keyword	“RV”
83	Length	2Ch
84	Data	Checksum
85	Data	Reserved (00h)
128	Large Resource Type VPD-W Tag (11h)	91h
129	Length	007Ch
131	VPD Keyword	“V1”
133	Length	05h
134	Data	“65A01”
139	VPD Keyword	“Y1”
141	Length	0Dh
142	Data	“Error Code 26”
155	VPD Keyword	“RW”
157	Length	61h
158	Data	Reserved (00h)
255	Small Resource Type End Tag (0Fh)	78h

Glossary

add-in card	A circuit board that plugs into a system board and adds functionality.
64-bit extension	A group of PCI signals that support a 64-bit data path.
Address Spaces	A reference to the three separate physical address regions of PCI: Memory, I/O, and Configuration.
agent	An entity that operates on a computer bus.
arbitration latency	The time that the master waits after having asserted REQ# until it receives GNT#, and the bus returns to the idle state after the previous master's transaction.
backplate	The metal plate used to fasten an add-in card to the system chassis.
BIST register	An optional register in the header region used for control and status of built-in self tests.
bridge	The logic that connects one computer bus to another, allowing an agent on one bus to access an agent on the other.
burst transfer	The basic bus transfer mechanism of PCI. A burst is comprised of an address phase and one or more data phases.
bus commands	Signals used to indicate to a target the type of transaction the master is requesting.
bus device	A bus device can be either a bus master or target: ❑ master -- drives the address phase and transaction boundary (FRAME#). The master initiates a transaction and drives data handshaking (IRDY#) with the target. ❑ target -- claims the transaction by asserting DEVSEL# and handshakes the transaction (TRDY#) with the initiator.
catastrophic error	An error that affects the integrity of system operation such as a detected address parity error or an invalid PWR_GOOD signal.
central resources	Bus support functions supplied by the host system, typically in a PCI compliant bridge or standard chipset.

command	*See* bus command.
Configuration Address Space	A set of 64 registers (DWORDs) used for configuration, initialization, and catastrophic error handling. This address space consists of two regions: a header region and a device-dependent region.
configuration transaction	Bus transaction used for system initialization and configuration via the configuration address space.
DAC	Dual address cycle. A PCI transaction where a 64-bit address is transferred across a 32-bit data path in two clock cycles. *See also* SAC.
deadlock	When two devices (one a master, the other a target) require the other device to respond first during a single bus transaction. For example, a master requires the addressed target to assert TRDY# on a write transaction before the master will assert IRDY#. (This behavior is a violation of this specification.)
Delayed Transaction	The process of a target latching a request and completing it after the master was terminated with Retry.
device	See PCI device.
device dependent region	The last 48 DWORDs of the PCI configuration space. The contents of this region are not described in this document.
Discard Timer	When this timer expires, a device is permitted to discard unclaimed Delayed Completions (refer to Section 3.3.3.3.3 and Appendix E).
DWORD	A 32-bit block of data.
EISA	Extended Industry Standard Architecture expansion bus, based on the IBM PC AT bus, but extended to 32 bits of address and data.
expansion bus bridge	A bridge that has PCI as its primary interface and ISA, EISA, or Micro Channel as its secondary interface. This specification does not preclude the use of bridges to other buses, although deadlock and other system issues for those buses have not been considered.
Function	A set of logic that is represented by a single Configuration Space.

header region	The first 16 DWORDs of a device's Configuration Space. The header region consists of fields that uniquely identify a PCI device and allow the device to be generically controlled. *See also* device dependent region.
hidden arbitration	Arbitration that occurs during a previous access so that no PCI bus cycles are consumed by arbitration, except when the bus is idle.
host bus bridge	A low latency path through which the processor may directly access PCI devices mapped anywhere in the memory, I/O, or configuration address spaces.
Idle state	Any clock period that the bus is idle (FRAME# and IRDY# deasserted).
Initialization Time	The period of time that begins when RST# is deasserted and completes 2^{25} PCI clocks later.
ISA	Industry Standard Architecture expansion bus built into the IBM PC AT computer.
keepers	Pull-up resistors or active components that are only used to sustain a signal state.
latency	*See* arbitration latency, master data latency, target initial latency, and target subsequent latency.
Latency Timer	A mechanism for ensuring that a bus master does not extend the access latency of other masters beyond a specified value.
master	An agent that initiates a bus transaction.
Master-Abort	A termination mechanism that allows a master to terminate a transaction when no target responds.
master data latency	The number of PCI clocks until IRDY# is asserted from FRAME# being asserted for the first data phase or from the end of the previous data phase.
MC	The Micro Channel architecture expansion bus as defined by IBM for its PS/2 line of personal computers.
multi-function device	A device that implements from two to eight functions. Each function has its own Configuration Space that is addressed by a different encoding of AD[10::08] during the address phase of a configuration transaction.
multi-master device	A single-function device that contains more than one source of bus master activity. For example, a device that has a receiver and transmitter that operate independently.

NMI	Non-maskable interrupt.
operation	A logical sequence of transactions, e.g., Lock.
output driver	An electrical drive element (transistor) for a single signal on a PCI device.
PCI connector	An expansion connector that conforms to the electrical and mechanical requirements of the PCI local bus standard.
PCI device	A device that (electrical component) conforms to the PCI specification for operation in a PCI local bus environment.
PGA	Pin grid array component package.
phase	One or more clocks in which a single unit of information is transferred, consisting of: ☐ an *address phase* (a single address transfer in one clock for a single address cycle and two clocks for a dual address cycle) ☐ a *data phase* (one transfer state plus zero or more wait states)
positive decoding	A method of address decoding in which a device responds to accesses only within an assigned address range. *See also* subtractive decoding.
POST	Power-on self test. A series of diagnostic routines performed when a system is powered up.
pullups	Resistors used to insure that signals maintain stable values when no agent is actively driving the bus.
run time	The time that follows Initialization Time.
SAC	Single address cycle. A PCI transaction where a 32-bit address is transferred across a 32-bit data path in a single clock cycle. *See also* DAC.
single-function device	A device that contains only one function.
sideband signals	Any signal not part of the PCI specification that connects two or more PCI-compliant agents and has meaning only to those agents.
Special Cycle	A message broadcast mechanism used for communicating processor status and/or (optionally) logical sideband signaling between PCI agents.

stale data	Data in a cache-based system that is no longer valid and, therefore, must be discarded.
stepping	The ability of an agent to spread assertion of qualified signals over several clocks.
subtractive decoding	A method of address decoding in which a device accepts all accesses not positively decoded by another agent. *See also* positive decoding.
system board	A circuit board containing the basic functions (e.g., CPU, memory, I/O, and add-in card connectors) of a computer.
target	An agent that responds (with a positive acknowledgment by asserting DEVSEL#) to a bus transaction initiated by a master.
Target-Abort	A termination mechanism that allows a target to terminate a transaction in which a fatal error has occurred, or to which the target will never be able to respond.
target initial latency	The number of PCI clocks that the target takes to assert TRDY# for the first data transfer.
target subsequent latency	The number of PCI clocks that the target takes to assert TRDY# from the end of the previous data phase of a burst.
termination	A transaction termination brings bus transactions to an orderly and systematic conclusion. All transactions are concluded when FRAME# and IRDY# are deasserted (an idle cycle). Termination may be initiated by the master or the target.
transaction	An address phase plus one or more data phases.
turnaround cycle	A bus cycle used to prevent contention when one agent stops driving a signal and another agent begins driving it. A turnaround cycle must last one clock and is required on all signals that may be driven by more than one agent.
wait state	A bus clock in which no transfer occurs.