PCI Local Bus Specification

Production Version

Revision 2.1

June 1, 1995

REVISION	REVISION HISTORY	DATE
1.0	Original issue	6/22/92
2.0	Incorporated connector and expansion board specification	4/30/93
2.1	Incorporated clarifications and added 66 MHz chapter	6/1/95

The PCI Special Interest Group 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 the PCI Special Interest Group make a commitment to update the information contained herein.

Contact the PCI Special Interest Group office to obtain the latest revision of the specification.

Questions regarding the PCI specification or membership in the PCI Special Interest Group may be forwarded to:

PCI Special Interest Group
P.O. Box 14070
Portland, OR 97214
(800)433-5177 (U.S.)
(503)797-4207 (International)
(503)234-6762 (FAX)

Contents

Chapter 1 Introduction

Chapter 2 Signal Definition

Chapter 3 Bus Operation

Chapter 4 Electrical Specification

Chapter 5 Mechanical Specification

Chapter 6 Configuration Space

Chapter 7 66 MHz PCI Specification

Figures

Preface

Specification Supersedes Earlier Documents

This document contains the formal specifications of the protocol, electrical, and mechanical features of the PCI Local Bus, Revision 2.1, as the production version effective June 1, 1995. The *PCI Local Bus Specification, Revision 2.0,* issued April 30, 1993 is superceded by this specification.

Incorporation of Engineering Change Requests (ECRs)

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

ECR #	Date	Title
6	8/16/93	Miscellaneous Mechanical Corrections
7	10/12/93	Standard Definition of Device Specific User Selectable Configuration Items
8	2/24/94	Cache Line Toggle Mode Deletion
11	3/21/94	PCI Subsystem Identification
12	3/24/94	Miscellaneous Mechanical Clarifications/Corrections
13	4/13/94	ISA Retainer Correction
14	2/7/94	Cardbus Data Structure Pointer
15	2/7/94	Clock Run
16	4/25/94	CLKRUN# Support Status Bit
17	4/24/94	PCI Mini-card Specification

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 active state occurs when it is at a low voltage. The absence of a # symbol indicates that the signal is active 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 Special Interest Group. 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 bold** font. At the first mention of a signal, the full signal name appears with its abbreviation in parentheses. After its first mention, the signal is referred to by its abbreviation.
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 enclosed in a box. They are not part of the PCI specification and are included for clarification and illustration only.

Chapter 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 boards, and processor/memory systems.

The *PCI Local Bus Specification, Rev. 2.1* includes the protocol, electrical, mechanical, and configuration specification for PCI Local Bus components and expansion boards. The electrical definition provides for 5.0V and 3.3V signaling environments.

The *PCI Local Bus Specification* defines the PCI hardware environment. Contact the PCI SIG for more information on the available PCI design guides and the *PCI BIOS Specification*. For information on how to join the PCI SIG or to obtain these documents, refer to Section 1.6.

1.2. Motivation

Graphics-oriented operating systems such as Windows and OS/2 have 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.

The advantages offered by local bus designs have motivated several versions of local bus implementations. The benefits of establishing an open standard for system I/O buses have been clearly demonstrated in the PC industry. It is important that a new standard for local buses be established to simplify designs, reduce costs, and increase the selection of local bus components and add-in cards.

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.

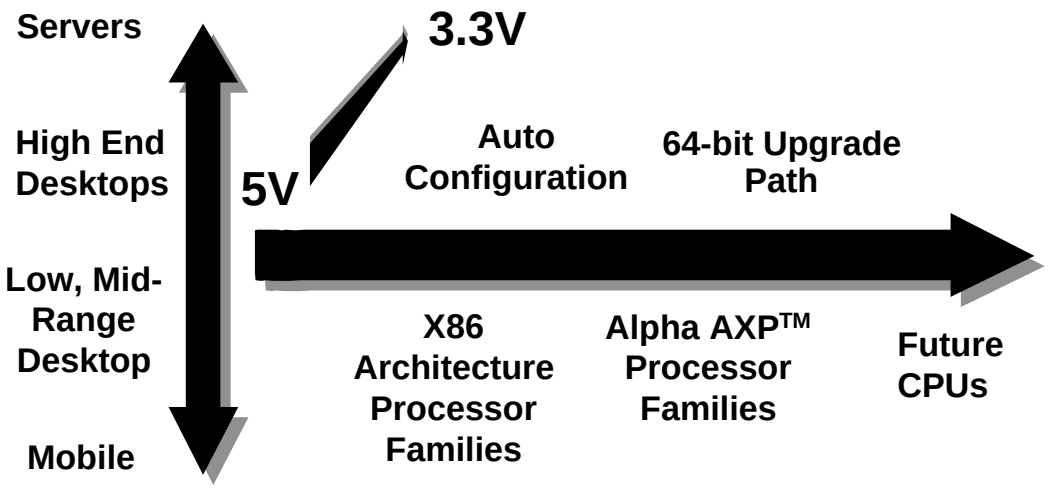


Figure 1-1: PCI Local Bus Applications

While the initial focus of local bus applications has been on low to high end desktop systems, the PCI Local Bus also comprehends the requirements from mobile applications up through departmental servers. The 3.3-volt requirements of the mobile environment and the imminent move from 5 volts to 3.3 volts in desktop applications must be accounted for in a new standard. The PCI Local Bus specifies both voltages and describes a clear migration path between them.

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 66 MHz specification is also defined, doubling the bandwidth capabilities of the 33 MHz definition.

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 optionally include such functions as data buffering/posting and PCI central functions (e.g., arbitration).

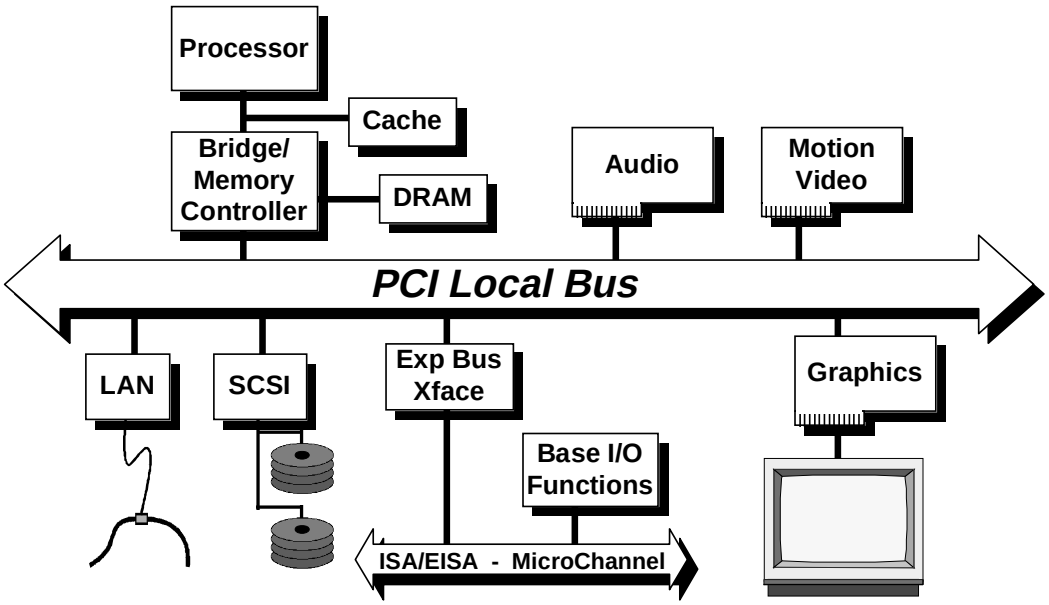


Figure 1-2: PCI System Block Diagram

Typical PCI Local Bus implementations will support up to four *add-in board connectors*, although expansion capability is not required. The PCI add-in board connector is a Micro Channel (MC)-style connector. The same PCI expansion board can be used in ISA-, EISA-, and MC-based systems. PCI expansion cards use an edge connector and motherboards that allow a female connector to be mounted parallel to the system bus connectors. To provide a quick and easy transition from 5V to 3.3V component technology, PCI defines two add-in board connectors: one for the 5V signaling environment and one for the 3.3V signaling environment.

Three sizes of PCI *add-in boards* are defined: long, short, and variable short length. Systems are not required to support all board types. The long boards include an ISA/EISA extender to enable them to use ISA/EISA card guides in ISA/EISA systems. To accommodate the 5V and 3.3V signaling environments and to facilitate a smooth migration path between the voltages, three add-in board electrical types are specified: a "5 volt" board which plugs into only the 5V connector, a "universal" board which plugs into both 5V and 3.3V connectors, and a "3.3 volt" board which plugs into only the 3.3V connector.

Two types of PCI *backplates* are currently defined: ISA/EISA- and MC-compatible. The interchangeable backplates should both be supplied with each PCI Local Bus add-in board shipped to accommodate usage of the board in all three system types.

It is assumed that typical low bandwidth, after-market add-ins will remain on the standard I/O expansion buses such as ISA, EISA, or MC. One component (or set of components) on PCI may generate the standard I/O expansion bus used in the system. In some mobile or client systems, a standard expansion bus may not be required.

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) and from 32-bit data path at 66 MHz (264 MB/s peak) to 64-bit data path at 66 MHz (528 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 or 30-ns for 66 MHz PCI to slave registers from master parked on bus).
- Capable of full concurrency with processor/memory subsystem.
- Synchronous bus with operation up to 33 MHz or 66 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.
- Single PCI add-in card works in ISA-, EISA-, or MC-based systems (with minimal change to existing chassis designs), reducing inventory cost and end user confusion.

Ease of Use

- Enables full auto configuration support of PCI Local Bus add-in boards 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. • Both 5-volt and 3.3-volt signaling environments are specified. Voltage migration path enables smooth industry transition from 5 volts to 3.3 volts.
Interoperability/ Reliability	• Small form factor add-in boards. • Present signals allow power supplies to be optimized for the expected system usage by monitoring add-in boards 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 boards and components. • Forward and backward compatibility with 33 MHz and 66 MHz add-in boards 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. • MC-style expansion connectors.
Flexibility	• Full multi-master capability allowing any PCI master peer-to-peer access to any PCI master/target. • A shared slot accommodates either a standard ISA, EISA, or MC board or a PCI add-in board (refer to Chapter 5, "Mechanical Specification" for connector layout details).
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 the PCI SIG. The PCI SIG, an unincorporated association of members of the microcomputer industry, was established to monitor and enhance the development of the PCI Local Bus in three ways. The 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.

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 Steering Committee membership
- Vendor ID number assignment
- PCI technical support
- PCI support documentation and materials
- Participation in SIG sponsored trade show suites and events, conferences, and other PCI Local Bus promotional activities

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

PCI Special Interest Group
P.O. Box 14070
Portland, OR 97214

Phone (800) 433-5177 (U.S.)
(503) 797-4207 (International)
FAX (503) 234-6762

Chapter 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.

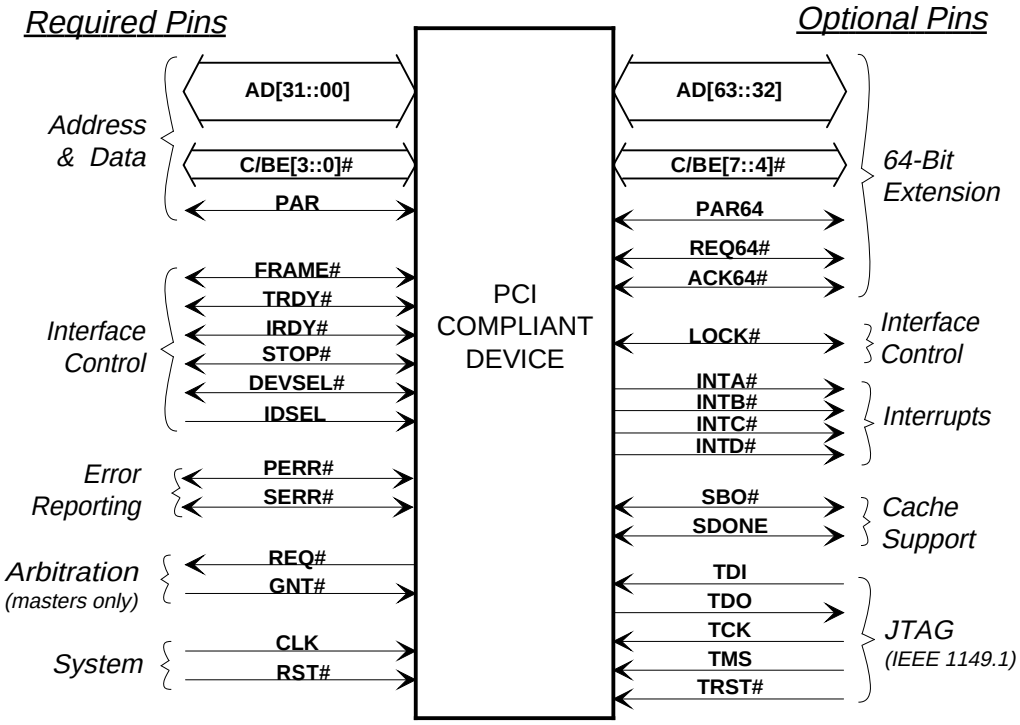


Figure 2-1: PCI Pin List

[1] The minimum number of pins for a planar-only device is 45 for a target-only and 47 for a master (**PERR#** and **SERR#** are optional for planar-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 pullup 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 active state occurs when the signal is at a low voltage. When the # symbol is absent, the signal is active 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#**, and **INTD#**, 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 (refer to Chapter 4) or 66 MHz (refer to Chapter 7) and, in general, the minimum frequency is DC (0 Hz); however, component-specific permissions are described in Chapter 4 (refer to Section 4.2.3.1).

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. 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. **SERR#** (open drain) is floated. **SBO#** and **SDONE**[2] may optionally be driven to a logic low level if tri-state outputs are not provided here. **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. **REQ64#** has meaning at the end of reset and is described in Section 4.3.2.

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[3] phase followed by one or more data phases. PCI supports both read and write bursts.

The address phase is the 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 and 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).

---

[2] **SDONE** and **SBO#** have no meaning until **FRAME#** is asserted indicating the start of a transaction.

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

PAR	t/s	*Parity* is even[4] 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 the 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.

2.2.3. 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 sampled 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 sampled 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.

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

LOCK#	s/t/s	*Lock* indicates an atomic operation that may require multiple transactions to complete. When **LOCK#** is asserted, non-exclusive transactions may proceed to an address 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#**. If a device implements Executable Memory, it should also implement **LOCK#** and guarantee complete access exclusion in that memory. A target of an access that supports **LOCK#** must provide exclusion to a minimum of 16 bytes (aligned). Host bridges that have system memory behind them should implement **LOCK#** as a target from the PCI bus point of view and optionally as a master. Refer to Section 3.6 for details on the requirements of **LOCK#**.
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#**[5] 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 when 3.3V or 5.0V only add-in boards are used with add-in boards that use a universal I/O buffer.

[5] **REQ#** is an input to the arbiter, while **GNT#** is an output.

2.2.5. Error Reporting Pins

The error reporting pins are required[6] by all devices and maybe 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 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. There are no special conditions when a data parity error may be lost or when reporting of an error may be delayed. An agent cannot report a **PERR#** until it has claimed the access by asserting **DEVSEL#** (for a target) and completed a data phase or is the master of the current transaction.

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 pullup (same value as used for s/t/s) which is provided by the system designer and not by the signaling agent or central resource. This pullup may take two to three clock periods to fully restore **SERR#**. The agent that reports **SERR#**s to the operating system does so anytime **SERR#** is sampled asserted.

[6] Some planar devices are granted exceptions (refer to Section 3.8.2. for details).

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. 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*[7] 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.

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 *all* **INTx#** signals from each connector are 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.

[7] 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.

Implementation Note: Interrupt Routing

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

MB = (D + I) MOD 4

MB = MotherBoard 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 Motherboard	Interrupt Pin on device	Interrupt Pin on Motherboard
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. Cache Support Pins (Optional)

A cacheable PCI memory should implement both cache support pins as inputs, to allow it to work with either write-through or write-back caches. If cacheable memory is located on PCI, a bridge connecting a write-back cache to PCI must implement both pins

as outputs; a bridge connecting a write-through cache may only implement one pin as described in Section 3.9..

It is recommended in systems that do not support PCI cacheable memory to pull-up **SBO#** and **SDONE** (on the connector). PCI cacheable memory agents must come out of reset ignoring **SBO#** and **SDONE** to work in systems that do not support PCI cacheable memory and will be configured at initialization time to operate in a cacheable mode when supported.

SBO#	in/out	*Snoop Backoff* indicates a hit to a modified line when asserted. When **SBO#** is deasserted and **SDONE** is asserted, it indicates a "CLEAN" snoop result.
SDONE	in/out	*Snoop Done* indicates the status of the snoop for the current access. When deasserted, it indicates the result of the snoop is still pending. When asserted, it indicates the snoop is complete.

2.2.8. Additional Signals

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

Implementation Note: PRSNT# Pins

At a minimum, the add-in board must ground one of the two **PRSNT[1:2]#** pins to indicate to the motherboard that a board is physically in the connector. The signal level of **PRSNT1#** and **PRSNT2#** inform the motherboard of the power requirements of the add-in board. The add-in board may simply tie **PRSNT1#** and/or **PRSNT2#** to ground to signal the appropriate power requirements of the board. (Refer to Section 4.4.1 for details.) The motherboard provides pull-ups on these signals to indicate when no board 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**. At the end of reset, **CLKRUN#** is asserted. The central resource must provide the pullup 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 if the bus segment is operating at 66 or 33 MHz. Refer to Section 7.5.1. for details of this signal's operation.

2.2.9. 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*[8] but are stable and indeterminate. During a data phase, an additional 32-bits of data are transferred when **REQ64#** and **ACK64#** are both asserted.
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 **REQ64#** and **ACK64#** are both asserted. **C/BE[4]#** applies to byte 4 and **C/BE[7]#** applies to byte 7.
REQ64#	s/t/s	*Request 64-bit Transfer*, when actively driven by the current bus master, indicates it desires to transfer data using 64 bits. **REQ64#** has the same timing as **FRAME#**. **REQ64#** has meaning at the end of reset and is described in Section 4.3.2..
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#**.

[8] Reserved means reserved for future use by the PCI SIG Steering Committee. Reserved bits must not be used by any device.

PAR64	t/s	*Parity Upper DWORD* is the even parity bit that protects **AD[63::32]** and **C/BE[7::4]#**. **PAR64** is valid one clock after the initial address phase when **REQ64#** is asserted and the DAC command is indicated on **C/BE[3::0]#**. **PAR64** is valid the clock after the second address phase of a DAC command when **REQ64#** is asserted. **PAR64** is stable and valid for data phases when both **REQ64#** and **ACK64#** are asserted and one clock after either **IRDY#** is asserted on a write transaction or **TRDY#** is asserted on a read transaction. Once **PAR64** is valid, it remains valid for one clock after the completion of the data phase. (**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.

2.2.10. 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 board 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 should operate in the same electrical environment (5V or 3.3V) 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 motherboard test connector for test purposes or to a resident 1149.1 Controller IC.

The PCI specification supports expansion boards with a connector that includes 1149.1 Boundary Scan signals. The devices on the expansion board could be connected to a

1149.1 chain on the motherboard. Methods of connecting and using the 1149.1 test rings in a system with expansion boards include:

- Only use the 1149.1 ring on the expansion board during manufacturing test of the expansion board. In this case, the 1149.1 ring on the motherboard would not be connected to the 1149.1 signals for the expansion boards. The motherboard would be tested by itself during manufacturing.
- For each expansion board connector in a system, create a separate 1149.1 ring on the motherboard. For example, with two expansion board connectors there would be three 1149.1 rings on the motherboard.
- 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.

Expansion boards that do not support the IEEE Standard 1149.1 interface should hardwire the board's **TDI** pin to its **TDO** pin.

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 may be used 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 planar 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 may include, but are not limited to, the following:

- Central Arbitration. (**REQ#** is an input and **GNT#** is an output.)
- Required signal pullups or "keepers," as described in Section 4.3.3.
- 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 3.2.2).
- Convert processor transaction into a configuration transaction.
- Generation of the individual **IDSEL** signals to each device for system configuration.
- Driving **REQ64#** during reset.

Chapter 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 as 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 3.7.2..

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 cycle, **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 3.7.4). **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. An agent is selected when its **IDSEL** signal is asserted and **AD[1::0]**are 00. During the address phase of a configuration cycle, the **AD[7::2]** lines address the 64 DWORD (where byte enables address the byte(s) within each DWORD) Configuration Space of each device and **AD[31::11]** are logical don’t cares. **AD[10::08]** indicate which device of a multi-function agent is being addressed.

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 should continue 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, pre-fetched 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.). 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.

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. The simplest way for the bridge to correctly handle latent data is to simply mark it invalid at the end of the current transaction. Otherwise, it must participate in the cache coherency.

The preferred use of the read commands is:

Memory Read command	When reading data in an address range that has side-effects (not prefetchable) or a reading a single DWORD.
Memory Read Line command	Reading more than a DWORD up to the next cacheline boundary in a prefetchable address space.
Memory Read Multiple command	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 obtain only the data the master requested and no more, since a side-effect may exists. 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.

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.

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#**, **SBO#**, **SDONE**, **SERR#** (on the falling edge of **SERR#** only), and **PERR#**. **C/BE[3::0]#**, **C/BE[7::4]#** (as bus commands) are qualified on the clock edge that **FRAME#** is first asserted. **C/BE[3::0]#**, **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.

3.2.1. Basic Transfer Control

The fundamentals of all PCI data transfers are controlled with three signals (see Figure 3-1).

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 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 **xRDY#** when it is not ready to accept data. When delaying the assertion of its **xRDY#**, 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.

---

[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 **AD** and **IDSEL** signals is fully defined in Section 3.7.3..

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.

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 3.7.4..

PCI targets (except host bus bridges) are required to implement Base Address register(s) to access 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 requests (via Base Address register(s)) that its internal registers be mapped into Memory Space and not I/O Space. In PC systems, I/O Space is limited and highly fragmented and will become more difficult to allocate in the future, however the use of I/O Space is allowed. 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 point at the same registers internally. Note: A Base Address register does not contain a valid address when it is equal to "0".

Address decoding on PCI is distributed; i.e., done on every device. This obviates the need for central decode logic or for device select signals beyond the one used for configuration. Each agent is responsible for its own address decode. PCI supports two styles of address decoding: positive and subtractive. *Positive decoding* is faster since each device is looking for accesses in the address range(s) that it has been assigned. *Subtractive decoding* can be implemented by only one device on the bus since it accepts all accesses not positively decoded by some other agent. This decode mechanism is slower since it must give all other bus agents a "first right of refusal" on the access.

However, it is very useful for an agent, such as a standard expansion bus, that must respond to a highly fragmented address space. Targets that perform either positive or subtractive decode must not respond (assert **DEVSEL#**) to reserved bus commands

The information contained in the two low order address bits (**AD[1::0]**) varies by address space. In the I/O Address Space, all 32 **AD** lines are used to provide a full byte address. This allows an agent requiring byte level address resolution to complete address decode and claim the cycle[12] without waiting an extra cycle for the byte enables (thus delaying all subtractive decode cycles by an extra clock). **AD[1::0]** are used for the generation of **DEVSEL#** only and indicate the least significant valid byte involved in the transfer. For example, if **BE0#** were asserted then **AD[1::0]** would be "00"; if only **BE3#** were asserted, then **AD[1::0]** would be "11". (Note: An access with no byte enables asserted is valid, in this case **AD[1:0]** can be "xx".) Once a target has claimed an I/O access (using **AD[1::0]**) by asserting **DEVSEL#**, it then determines if it can complete the entire access as indicated in the byte enables. Note: A device may restrict what type of access(es) it supports in I/O or Memory Space but not in Configuration 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.

The device decodes the **AD** lines and asserts **DEVSEL#** if it “owns” the starting address. The **AD** lines that are used for decode purposes are dependent on the size of the space the device claims. For example, a device that claims four bytes (which are within a single DWORD) is not required to decode **AD[1::0]**. Once the access is claimed, the agent completes the access, if possible, as indicated by the byte enables. Even though the device “owns” the entire DWORD, it may terminate the access with Target-Abort when it is accessed with a byte enable combination not supported by the device.

For a device that may share bytes within a single DWORD with another device, the device addressed by **AD[31::00]** asserts **DEVSEL#** and claims the access. If the byte enables request data not owned by the device, it is required not to transfer any data and must terminate the access with Target-Abort. Only a device that may share bytes within a DWORD should use the following table. The table indicates valid address/byte enable combinations that a master may use when accessing a device in I/O Space. Any other combination is illegal and must be terminated with Target-Abort if the byte enables access an address outside the range of bytes inclusive in the device. Only a standard expansion bridge (a bridge that does subtractive decoding) is exempt from terminating the transaction with Target-Abort, because it does not know what devices reside behind it. This condition can only occur for devices behind a standard expansion bus bridge. Devices on the PCI bus are only assigned address space aligned on DWORD boundaries and in quantities of 2^N DWORDs.

AD1	AD0	C/BE3#	C/BE2#	C/BE1#	C/BE0#
0	0	X	X	X	0
0	1	X	X	0	1
1	0	X	0	1	1
1	1	0	1	1	1
X	X	1	1	1	1

Note: X = either 1 or 0

[12] Standard PC address assignments in the I/O Space are such that separate physical devices may share the same DWORD address. This means that in certain cases, a full byte address is required for the device to claim the access (assert **DEVSEL#**).

All targets are required to check **AD[1::0]** during a memory command transaction, and either provide the requested burst order or execute a target Disconnect after or with the first data phase. Implementation of linear burst ordering is required by all devices that can support bursting. Implementing cacheline wrap is not required. In the Memory Address Space, accesses are decoded to a DWORD address using **AD[31::02]**. In linear incrementing mode, the address is assumed to increment by one DWORD (four bytes) for 32-bit transfers and two DWORDs (eight bytes) for 64-bit transfers after each data phase until the transaction is terminated (an exception is described in Section 3.10.). The Memory Write and Invalidate command can only use the linear incrementing burst mode.

During Memory commands, **AD[1::0]** have the following meaning:

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)

Cacheline wrap burst may begin anywhere in 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 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 8, the sequence for a 32-bit transfer would be:

First data phase is to DWORD 8

Second data phase is to DWORD C (which is the end of the current cacheline)

Third data phase is to DWORD 0 (which is the beginning of the addressed cacheline)

Last data phase is to DWORD 4 (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 location of the next cacheline. Continuing the burst of the previous example would be:

Fifth data phase is to DWORD 18

Sixth data phase is to DWORD 1C (which is the end of the second cacheline)

Seventh data phase is to DWORD 10 (which is the beginning of the second cacheline)

Last data phase is to DWORD 14 (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 when Cacheline Wrap or a reserved mode is used. If an alternate burst sequence is desired after the first cacheline has completed, the master must stop the access after the first line has been transferred and begin a new access in another 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.

In the Configuration Address Spaces, accesses are decoded to a DWORD address using **AD[7::2]**. An agent determines it is the target of the access (asserts **DEVSEL#**) when a configuration command is decoded, **IDSEL** is asserted, and **AD[1::0]** are "00". Otherwise, the agent ignores the current transaction. A bridge determines a configuration access is for a device behind it by decoding a configuration command, its bridge number and **AD[1::0]** are "01".

3.2.3. Byte Alignment

Byte lane swapping is not done on PCI since all PCI compliant devices must connect to all 32 address/data bits for address decode purposes. However, DWORD swapping is done by masters that support 64-bit data paths when transferring data to a 32-bit device. This means that bytes will always appear in their natural byte lane based upon byte address.

Furthermore, PCI does not support automatic bus sizing. In general, software is very aware of the characteristics of the target device and only issues appropriate length accesses; 64-bit data path transfers are one particular exception.

The byte enables alone are used to determine which bytes carry meaningful data. The byte enables are free to change between data phases but must be valid on the edge of the clock that starts each data phase and must stay valid for the entire data phase. In Figure 3-1, 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#**.

When the current read transaction is to cacheable memory, all bytes must be returned regardless of which byte enables are asserted. This requires the agent that determines cacheability to guarantee the target returns all bytes. If cacheability is determined by the request initiator, it must ensure that all byte enables are asserted so the target will return all required data. If cacheability is determined by the target, it must ignore the byte enables (except for **PAR** generation) and return the entire DWORD for a 32-bit transfer (two DWORDs for a 64-bit transfer). The cacheable target must either return the entire cacheline or only the first data requested.

If a target does not support caching but does support prefetching (bit set in the Memory Base Address register -- refer to Section 6.2.5.1.), it must also return all data 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 permanent change. On a read transaction, this means that data or 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. On a read transaction, the target must provide parity across **AD[31::0]** and **C/BE[3::0]#** for a 32-bit transfer and **AD[63::32]** and **C/BE[7::4]#** for a 64-bit transfer regardless of the state of the byte enables. On a write

transaction, the data is not stored and **PAR** is valid for a 32-bit transfer and **PAR64** is valid for a 64-bit transfer.

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 slaves). If this occurs, a target (expansion bus bridge) may optionally report an illegal pattern as an asynchronous error (**SERR#**) or, if capable, break the transaction into two 16-bit transactions that are legal for the intended agent. On an I/O access, the target 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 may be 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 3-1).

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 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

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 systems 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 transfered. Transactions which terminate with Retry have not completed since no data was transfered, and, therefore, have no ordering requirements relative to each other. Transactions that teminate with Master-Abort or Traget-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, then 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 3.3.3.3.6. for further discussion of request ordering.

Transactions can be divided into two general groups based on how they cross a bridge in a multi-bus system, posted and non-posted. Posted transactions complete on the orginating bus before they complete on the destination bus. In essence, the intermediate target (bridge) of the access 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 complete on the destination bus before completing on the originating bus. Memory read transactions (Memory Read, Memory Read Line, 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).

Host bus bridges may 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.

Bridges may 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.

1. 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 that their order on the originating bus.
2. 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.
3. 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. And before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes which originated on the opposite side and were posted before the read command completes on the read-destination bus.

4. A device can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-posted transaction as a master. Otherwise, a deadlock may occur.

Implementation Note: Deadlock When Posted Write Data is Not Accepted

As an example of a case where a device must accept posted write data to avoid a deadlock, suppose a PCI-to-PCI bridge contains posted memory write data addressed to a downstream agent. But before the bridge can acquire the downstream bus to do the write transaction, the downstream agent initiates a read from host memory. Since requirement 3 above 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 agent 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 version of this spec require their posting buffer to be flushed before starting any non-posted transaction, the same deadlock could occur if the agent makes the acceptance of a posted write contingent on the prior completion of any non-posted transaction.

Bridges are not the only devices permitted to post memory write data. Non-bridge devices are strongly encouraged to post memory write data to speed the transaction on the PCI bus. How such a device deals with ordering of posted write data is strictly implementation dependent and beyond the scope of this specification. However, non-bridges must follow the same rules for avoiding deadlock as bridges. For example, non-bridges can never make the acceptance of a posted write transaction contingent on the prior completion of a non-posted transaction as a master.

Since memory write transactions may be posted 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 a intermediate device), it must complete a read to the same device that the write was 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.. See also Section 3.11, item 6, for other cases where a read may be necessary.

Interrupt requests 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.

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 mulitple 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 transaction (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 may only be done 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 where 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 cycle accesses which were combined into a burst. (Note: If a bridge does combining, a disable bit in device specific Configuration Space is recommended.)

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 may only be done 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 would have been 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.

Cacheline Merging -- is when a sequence of memory writes are merged into a single cacheline and may only occur when the address space has been defined to be cacheable or when byte merging and/or combining are used to create a cacheline transfer. This means that memory write accesses to a cacheline can be merged as bytes, words or DWORDs in any order as long as each byte is only written once. An agent can do cacheline merging when it either participates in the cache coherency of the memory or has explicit knowledge of the address range of cacheable memory.

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. Since the typical frame buffer is not supported as PCI cacheable, the processor will generate DWORD accesses to this region. 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 merging. With the addition of combining sequential DWORDs, the number of transactions on PCI can be reduced further. Merging data (DWORDs) within a single cacheline appears to have minimal performance gains since PCI cacheable memory appears to be rare (if in existence at all).

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 3-1 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.

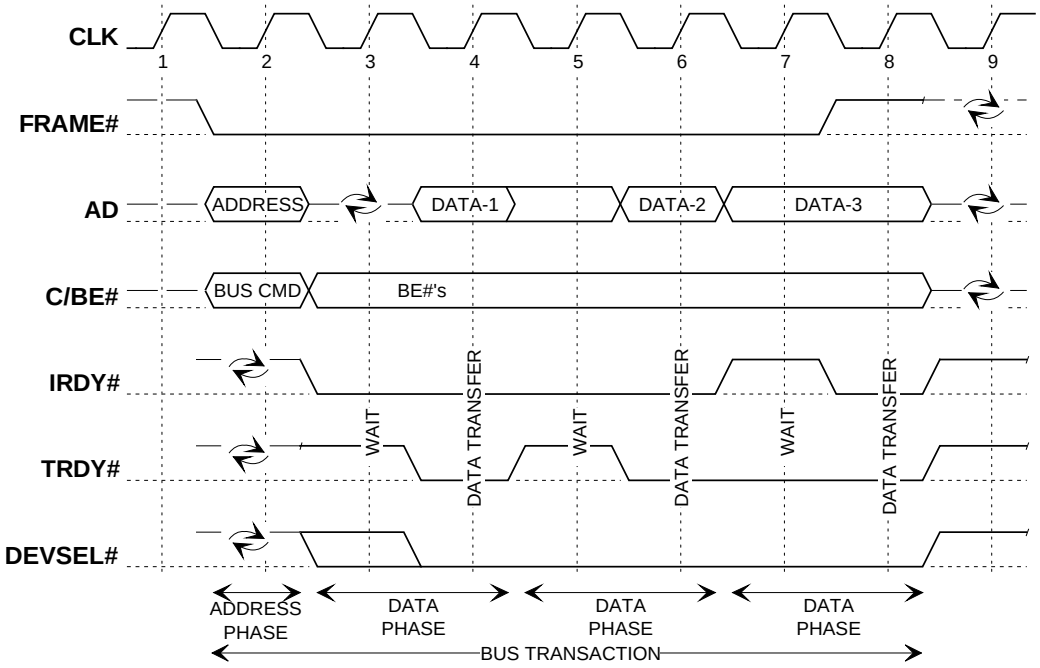


Figure 3-1: Basic Read Operation

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 3-1 because a burst read transaction typically has all byte enables asserted; however, it is shown in Figure 3-2. Notice on clock 5 in Figure 3-2, 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 3-1, 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 3-2 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.

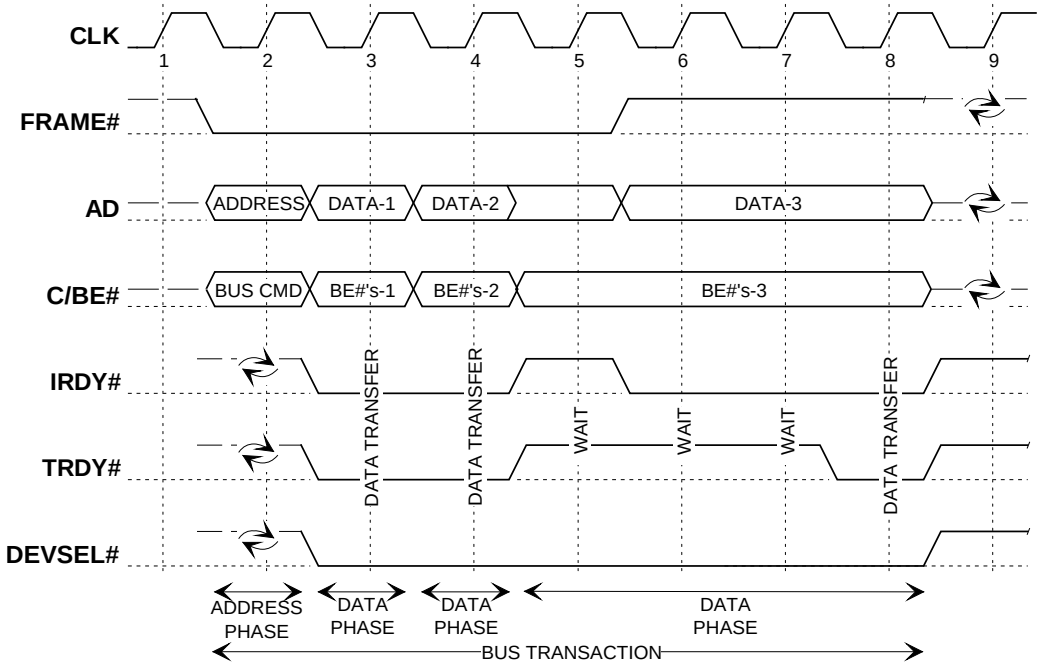


Figure 3-2: Basic Write Operation

In Figure 3-2, 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 3-2).

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 3.5.3. 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 3-3. 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.

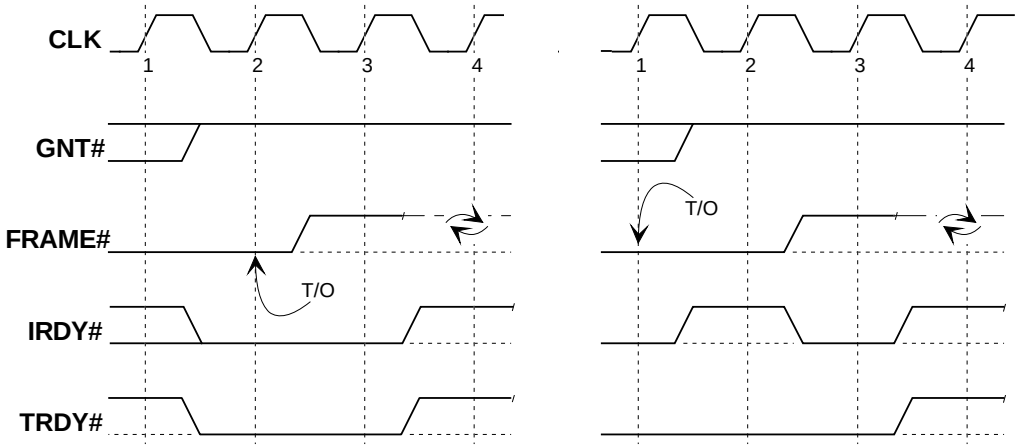


Figure 3-3: Master Initiated Termination

Both sides of Figure 3-3 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 3.5.3.), 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 3-4, 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 3.7.1..) 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 3.8.2.2..) 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.

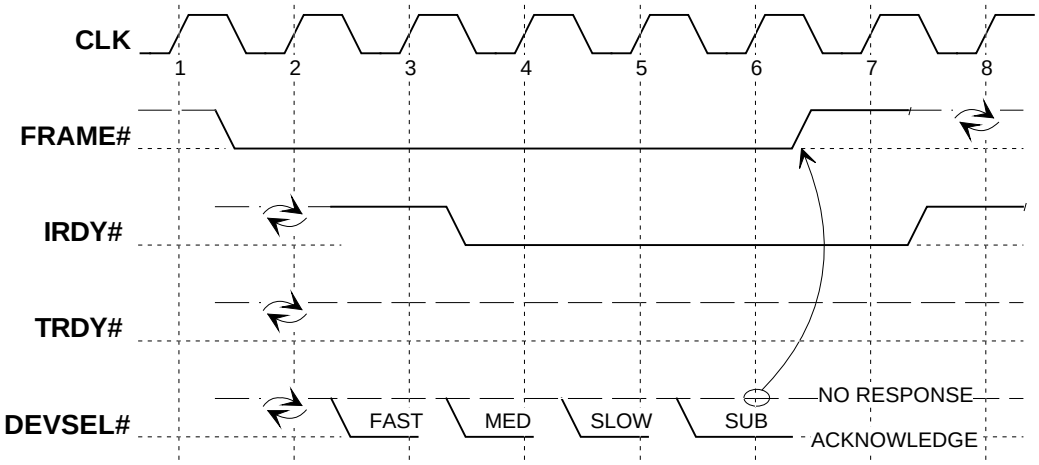


Figure 3-4: Master-Abort Termination

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.

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. 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 addressed DWORD to be read, but the target owns only the lower two bytes of the addressed DWORD. Since the target cannot complete the entire 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 1) do not support exclusive (locked) accesses, 2) do not have a posted memory write buffer which needs to be flushed to meet the PCI ordering rules, 3) cannot get into a state where they may need to reject an access, and 4) 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.

Cacheable targets must not disconnect a Memory Write and Invalidate command except at cacheline boundaries, whether caching is currently enabled or not. Therefore, a "snooping" agent may always assume a Memory Write and Invalidate command will complete without being "disconnected" when the access is to a cacheable memory range.

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 3-5 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 also 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. The master is required to deassert **FRAME#** as soon as **IRDY#** can be asserted. In this case, **FRAME#** is deaserted 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.

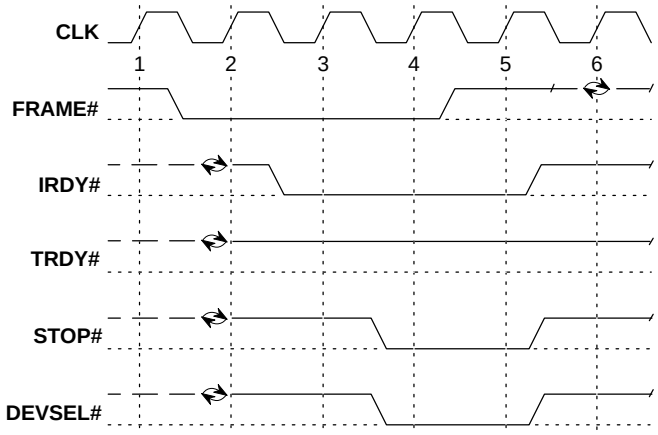


Figure 3-5: Retry

Disconnect With Data

Disconnect - A, in Figure 3-6, 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 3-6, 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.

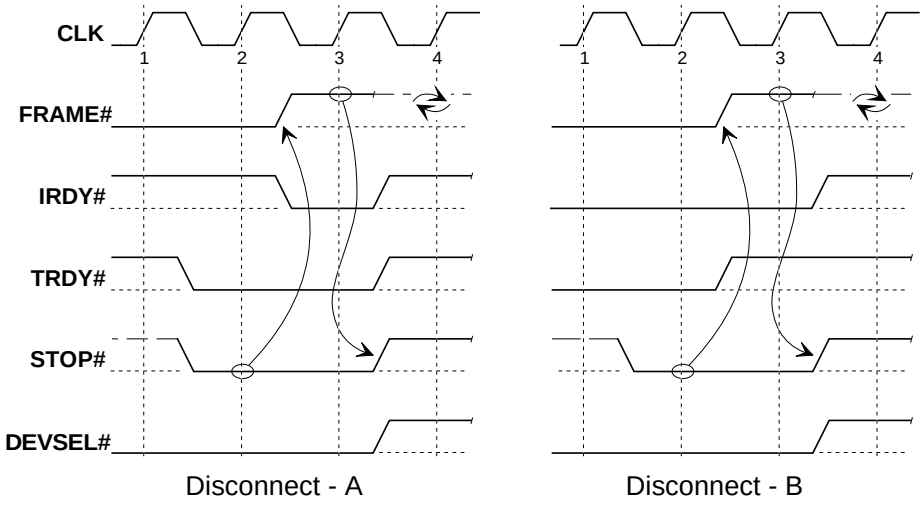


Figure 3-6: Disconnect With Data

Figure 3-7 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.

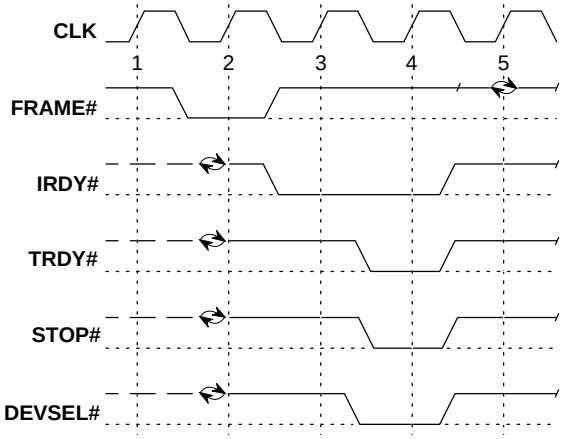


Figure 3-7: Master Completion Termination

Disconnect Without Data

Figure 3-8 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 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 where 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.

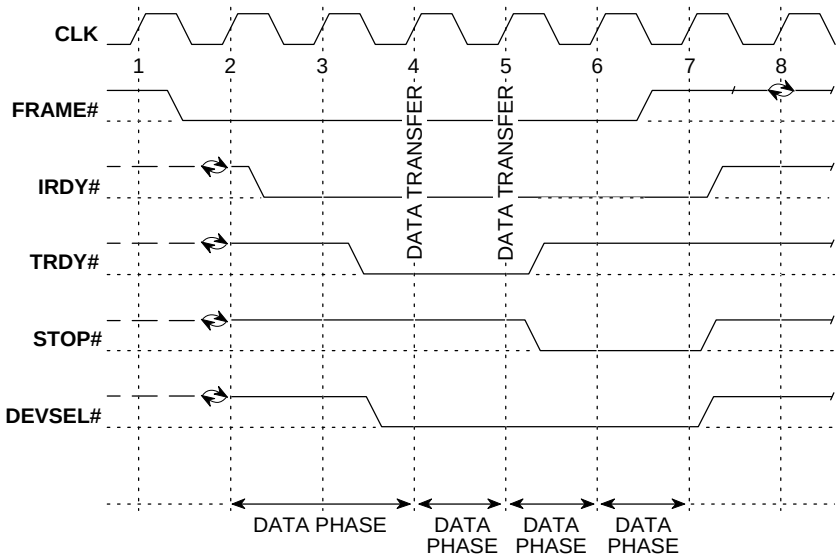


Figure 3-8: Disconnect-1 Without Data Termination

Figure 3-9 shows the same transaction as described in Figure 3-8 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.

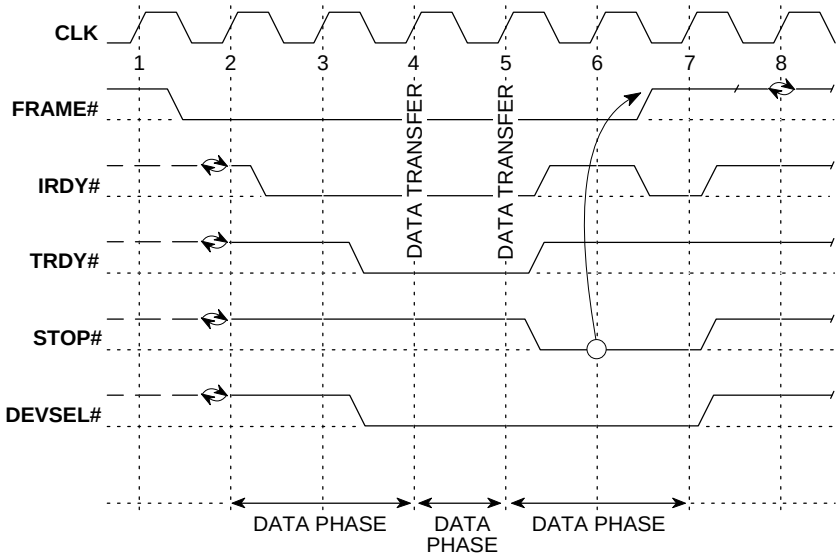


Figure 3-9: Disconnect-2 Without Data Termination

Target-Abort

Figure 3-10 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.

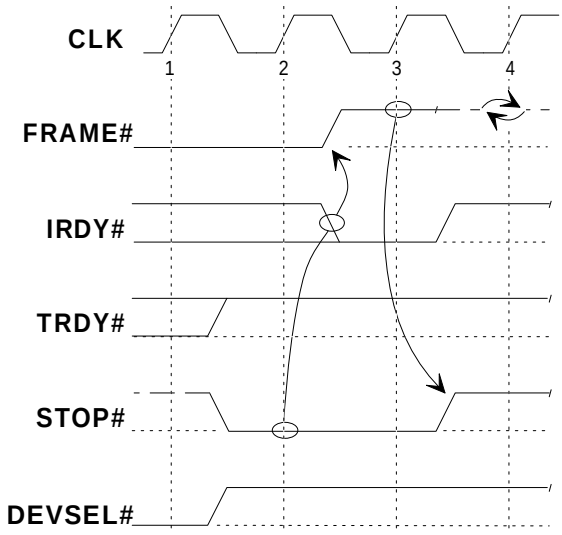


Figure 3-10: Target-Abort

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. The master 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. If another master is waiting to use the bus, the arbiter is required to grant a different master access to the bus to prevent deadlocks. It also allows other masters to use the bus which would normally be wasted since the master would attempt to complete the transaction that was terminated by the target. 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 3-7 which is really Master Completion termination and not target termination.

Repeat Request Terminated With Retry

A master which 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 transaction" means that the same address, same command, same byte enables, 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 may 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#**, **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.

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 non-postable transaction completes and still maintains the system ordering rules.

The following discussion will focus on the basic operation and requirements of a device that supports a single Delayed Transaction at a time. The next section 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). These commands are not completed using Delayed Transaction termination and are normally posted. 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 orginates 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 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. 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 3.3.3.2.2.).

During the second phase, 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 phase, 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

The information that must be latched by the target to complete the access includes the address, command and byte enables (parity bits may also be used if parity checking is enabled), data (if a write transaction), and **REQ64#** (if a 64-bit transfer). **LOCK#** must also be used if the target supports locked transactions. On a read transaction the address and command are available during the address phase and the byte enables during the following clock. (Byte enables are valid the entire data phase and are independent of **IRDY#**.) On a write transaction, all information is valid the same time as a read, except for the actual data which is valid only when **IRDY#** is asserted.

> Note: Write data is only valid when **IRDY#** is asserted while 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) or Delayed Completion(s)). The byte enables are not required to be used as part of the compare when the target returns all bytes when doing a read transaction regardless of which byte enables are asserted. A target can only do this when there are no read side-effects (pre-fetchable) when accessing the data. When 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. When the compare matches a Delayed Completion, the target responds by signaling the status and provides the data if a read transaction.

The master must repeat the transaction exactly as the original request; otherwise, the target will assume it is a new transaction. If the original transaction is never completed, a deadlock may occur. Two masters could request the exact same transaction and the target cannot and need not distinguish between them and will simply complete the access.

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 pre-fetchable region (or the command was Memory Read Line or Memory Read Multiple). The second case is for all Delayed Completions (read or write, pre-fetchable 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.

When the Discard Timer expires, the device may choose to report or ignore the error. When the data is prefetchable (case 1), it is recommended that the device ignore the error since the system integrity is not affected. However, when the data is not prefetchable (case 2), it is recommended that the device report the error to its device driver[14] 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. See Section 3.11, 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[15] 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 Delayed Transaction to meet the initial latency requirements is required to accept memory write even though the Delayed Transaction has not completed.

3.3.3.3.5. Delayed Transactions and LOCK#

A locked transaction can be completed using Delayed Transaction termination. All the rules of **LOCK#** still apply except the target must consider itself locked when it enqueues the request even though no data has transferred. While in this state, the target enqueues no new requests. After lock has been established, the target can only accept requests from the lock master. Note: **LOCK#** must be latched and used in the compare to determine when the request is being repeated. Note: The device cannot complete any

[14] A bridge may assert **SERR#** since it does not have a device driver.

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

access, including memory writes, to the locked resource except if initiated by the lock master.

3.3.3.3.6. 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 a PCI-to-PCI bridge) 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 be avoided.

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 a 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. Until that time, the DRR is only a request and may be discarded at anytime 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 anytime 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 3-1 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. 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 4 are required No's, 4 are required Yes's, and the remaining 19 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. A more detailed discussion of the following table is contained in Appendix E.

Table 3-1: 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/No	Yes/No	Yes/No	Yes/No
DWC (Row 5)	Yes/No	Yes/No	Yes/No	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 - indicates the PMW must be allowed to complete before Delayed Requests or Delayed Completions moving in the same direction or a deadlock can occur. The four Yes boxes are found in row 1 and prevent 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.

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, there are no ordering requirements 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, they do have ordering relationship with memory write transactions which is described in Table 3-1.

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 3.5.3. 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) or when cacheable memory is located on PCI. 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)

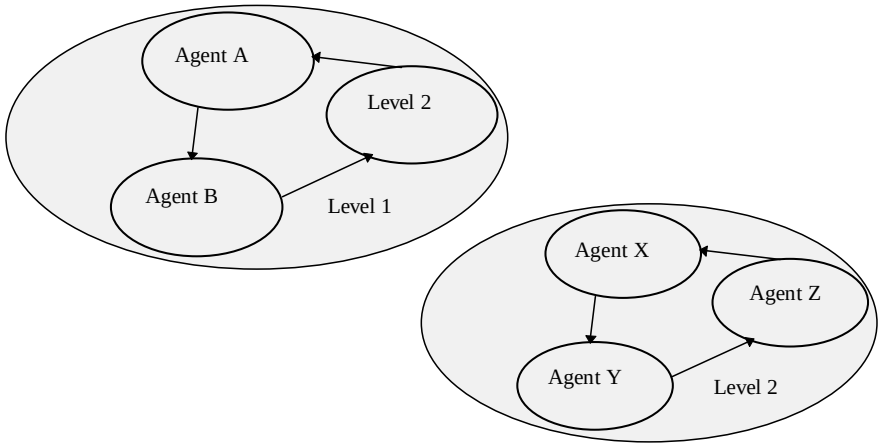
and so forth.

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 address stepping.
3. While **FRAME#** is deasserted, **GNT#** may be deasserted at any time in order to service a higher priority[16] master, or in response to the associated **REQ#** being deasserted.

Figure 3-11 illustrates basic arbitration. Two agents are used to illustrate how an arbiter may alternate bus accesses.

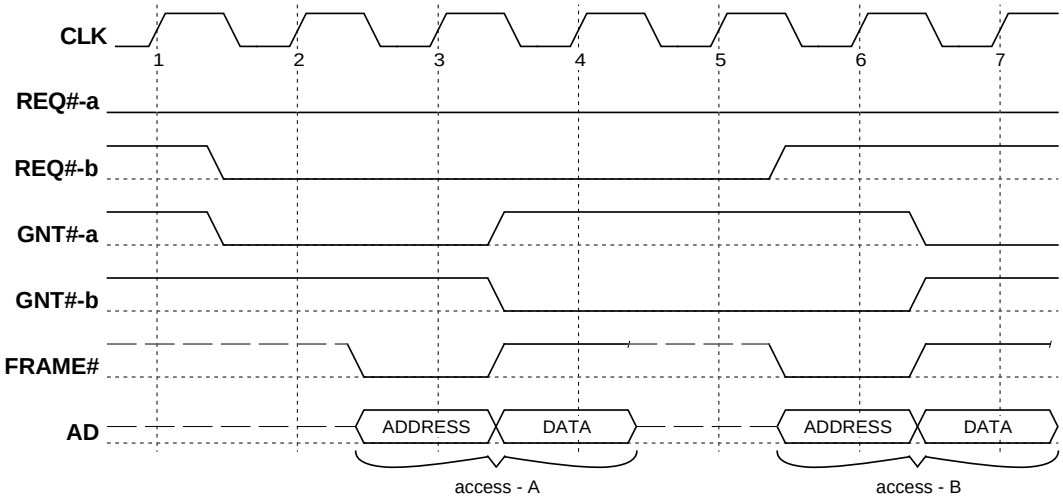


Figure 3-11: Basic Arbitration

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

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

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 enter 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 3.3.3.2.1.. 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. The preceding discussion does not apply to a configuration transaction since address stepping may be used.

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.[17]

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

[17] 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).

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. 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 presumably 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 all new targets, and those masters that can improve bus utilization should implement this feature, particularly since the implementation cost is negligible. However, it is not recommended that existing parts be put through a redesign cycle solely for this feature, as it will likely not benefit the utility of the part.

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: Multi-ported targets should only lock themselves when they are truly locked during fast back-to-back transactions (refer to Section 3.6. for more information).

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, 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 3-12, 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.

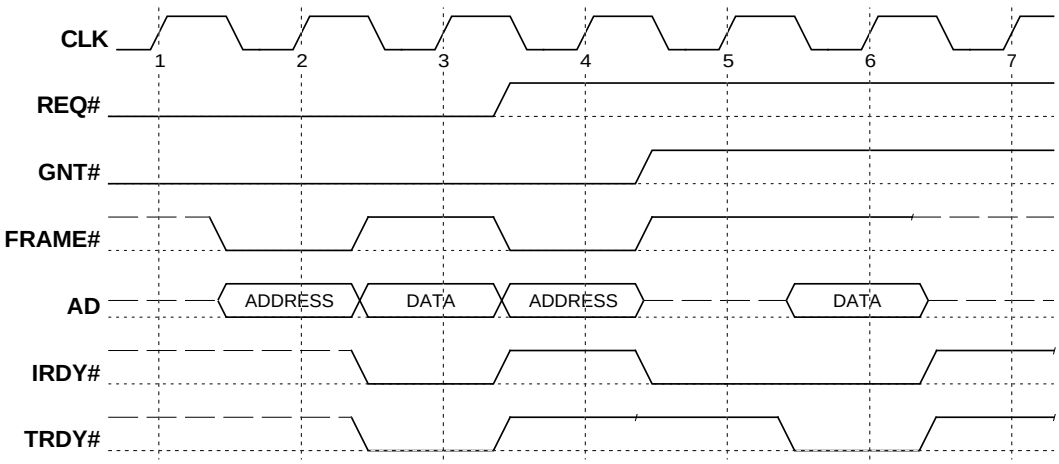


Figure 3-12: Arbitration for Back-to-Back Access

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. (Refer to Section 3.8.1. 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, 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.

All targets are required to 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. All devices are granted two exceptions to the initial latency rule during initialization time. Initialization time begins when **RST#** is deasserted and completes when the POST code has initialized the system. The time following the completion of the POST code is considered Run-time. The following two exceptions have no upper bound on initial latency and are granted during initialization time only. The target being accessed after initialization time must adhere to the 16 clock initial latency requirements.

- POST code accessing the device's configuration registers.
- POST code copying the expansion ROM image to memory.

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.

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#**.

The majority of I/O controllers built before this revision will meet the initial latency requirements using Option 1 and will require no modifications. 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 may only do this 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. For example, an agent that is currently locked by the lock master should use Option 2 to terminate the request with minimum delay but always before 16 clocks for all accesses not initiated by the lock master.

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 meet 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 8 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, there is no reason for a master to delay the assertion of **IRDY#** more than one or two clocks for a write transaction and 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. 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 assserted, 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 it's **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 which 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 cache line 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 cache line 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#)**.

3.5.3.1. 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 masters **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 masters behavior. The bus command used, target latency, and target subsequent latency are the primary parameters which control the targets 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 cache line 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 cache line 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. The target can use the target Disconnect protocol to terminate the burst transaction early when its buffers fill during the transaction. Using these more realistic considerations, the worst case latency equations can be modified to give a typical latency (assuming that the targets initial data phase latency is 8 clocks) again ignoring the effects of the Latency Timer.

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

If a master were allowed to burst indefinitely to a target which could absorb 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 3-2 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 cache line 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 7th data phase. The transaction completes with the 8th data phase.

Table 3-2: 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

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 3-2 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.

3.5.3.2. 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 two 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, all master moves an entire 64-byte cacheline before the 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 affect is that the Latency Timer expires. The second affect 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 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 stayed at the same 3 μs as 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 3-3 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 3-3: 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 3-3 with the appropriate row in Table 3-4. In either case, the total bandwidth required on PCI is reduced.

Table 3-4: 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 3-3 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 maybe 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 Arbitration Implementation Note, 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 or level two. The table above it 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.)

3.5.3.3. 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. Exclusive Access

PCI provides an exclusive access mechanism which allows non-exclusive accesses to proceed in the face of exclusive accesses. This is referred to as a resource lock. This allows future processors to hold a hardware lock across several accesses without interfering with non-exclusive, real-time data transfer, such as video. The mechanism is based on locking only the PCI resource to which the original locked access was targeted. This mechanism is fully compatible with existing software use of exclusion.

In general, **LOCK#** is to be used by bridges (host bus, PCI-to-PCI and standard expansion bus) to provide backward compatibility for existing devices and to prevent deadlocks. A host bus bridge may need to support **LOCK#** as a target to provide backward compatibility with some existing add-in cards that reside behind standard expansion bus bridges. A host bus bridge (as a master) may support **LOCK#** (or software mechanism) to prevent deadlocks (i.e., 8-byte read) with PCI-to-PCI bridges (refer to Section 3.11, item 5) and to provide compatibility with standard expansion bus bridges that require it. PCI-to-PCI bridges support **LOCK#** on transactions that originate on the primary bus of the bridge and have destination on the secondary bus of the bridge. The bridge may optionally support **LOCK#** as a target on the secondary bus. The use of **LOCK#** is not recommended for devices other than bridges or memory controllers that support system memory.

LOCK# is recommended on any device providing system memory. Specifically, if the device implements executable memory, then it should also implement **LOCK#**, and guarantee complete access exclusion in that memory (i.e., if there is a master local to that memory, it must also honor the lock). However, in some system architectures the host

bus bridge cannot guarantee exclusivity of the locked resource to a PCI master that uses **LOCK#**. In these systems, device drivers that require exclusivity use a software mechanism to guarantee exclusion and do not rely on **LOCK#**. To ensure exclusive access to system memory in multiple host or processor architectures, it is recommended that a PCI master utilize a software protocol and not **LOCK#**. Agents other than host bus bridges that support executable memory must guarantee exclusivity of the locked resource when **LOCK#** is used since this type of agent is independent of the host bus and processor architecture. This type of agent (if multi-ported), must guarantee exclusivity to memory from all agents whether PCI or non-PCI. A potential deadlock exists when using PCI-to-PCI bridges in a system (refer to Section 3.11., item 5, for more details).

The **LOCK#** signal indicates an exclusive access is underway. The assertion of **GNT#** does not guarantee control of **LOCK#**. Control of **LOCK#** is obtained under its own protocol in conjunction with **GNT#**. When using resource lock, agents performing non-exclusive accesses are free to proceed even while another master retains ownership of **LOCK#**. However, when compatibility dictates, the arbiter can optionally convert a resource lock into a "complete bus" lock by granting the agent that owns **LOCK#** exclusive access of the bus until **LOCK#** is released. Refer to Section 3.6.6. for more details about complete bus locks.

In a resource lock, exclusivity of an access is guaranteed by the target of the access, not by excluding all other agents from accessing the bus. The granularity of the lock is defined to be 16 bytes aligned. An exclusive access to any byte in the 16 byte block will lock the entire 16 byte block. The master cannot rely on any addresses outside the 16 bytes to be locked. A target is required to lock a minimum of 16 bytes (aligned) and up to a maximum of the entire resource. With this in mind, the following paragraphs describe the behavior of master and target.

The rules of **LOCK#** will be stated for both the master and target. Following the rules, a detailed description of how to start, continue, and complete an exclusive operation on PCI will be discussed. A discussion of how a target behaves when it supports both resource lock and write-back cacheable memory will follow the discussion of how exclusive operations work on PCI. The concluding section will discuss how to implement a complete bus lock on PCI.

A target that supports **LOCK#** on PCI must adhere to the following rules:

1. The target of an access locks itself when **LOCK#** is deasserted during the address phase.
2. Once lock is established, the target remains locked until both **FRAME#** and **LOCK#** are sampled deasserted or the target signals Target-Abort
3. Guarantee exclusivity to the owner of **LOCK#** (once lock is established) of a minimum of 16 bytes (aligned) of the resource.[18] This includes accesses that do not originate on PCI for multiport devices.

[18] The maximum is the complete resource.

Implementation Note: Multiport Devices and LOCK#

A multiport device is an agent that provides access to the same memory via different buses or ports. For a host bus bridge, the use of **LOCK#** only guarantees exclusivity to the owner of **LOCK#** for accesses that originate on the PCI bus. Any exclusivity beyond PCI must be guaranteed by the architecture of the host bus bridge or by software. In some system architectures, Rule 3 cannot be guaranteed by a host bus bridge because the host bus does not use hardware based locks to ensure exclusivity, but relies on a software mechanism. For a host bus bridge that uses a software mechanism to provide exclusivity, Rule 3 is not required. Note: A device driver that requires **LOCK#** for exclusivity may not work properly in some system architectures. A multi-function device that is not a host bus bridge and implements **LOCK#** must adhere to Rule 3. This type of multiport device must guarantee exclusivity from other agents (PCI or local masters) when **LOCK#** is implemented by the agent.

All PCI targets that support exclusive accesses must sample **LOCK#** with address. If the target of the access performs medium or slow decode, it must latch **LOCK#** during the address phase to determine if the access is a lock operation when decode completes. The target of a transaction marks itself locked if **LOCK#** is deasserted during the address phase. If a target waits to sample **LOCK#** until it asserts **DEVSEL#**, it cannot distinguish if the current access is a locked transaction or one that occurs concurrently with a locked access. An agent may store "state" to determine if the access is locked but this requires latching **LOCK#** on consecutive clocks and comparing to determine if the access is locked. A simpler way is for the target to mark itself locked on any access it claims where **LOCK#** is deasserted during the address phase. A locked target remains in the locked state until both **FRAME#** and **LOCK#** are deasserted.

To allow a non-PCI agent to access the memory of a multiport device, the target may sample **LOCK#** the clock following the address phase to determine if the device is really locked. Note: Sampling **LOCK#** during the address phase is still required to determine if the current transaction is a locked access or not. The status of **LOCK#** on the clock after the address phase provides additional information to a multiported device, identifying if the current transaction is really locked (this requires additional logic not required for a single ported device). When **LOCK#** is deasserted during the address phase and is asserted (the clock following the address phase), the multiport device is locked and must ensure exclusivity to the PCI master. When **LOCK#** is deasserted during the address phase and the clock following the address phase, the target is free to respond to other requests and is not locked. A currently locked target may only accept requests when **LOCK#** is deasserted during the address phase. A currently locked target will respond by asserting **STOP#** with **TRDY#** deasserted (Retry) to all transactions that access the locked address space when **LOCK#** is asserted during the address phase.

To summarize, a target of an access locks itself on any access it claims when **LOCK#** is deasserted during the address phase. It unlocks itself anytime **FRAME#** and **LOCK#** are both deasserted. 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. A target can also remember state (which is useful for a multiport device) to determine if it is truly locked or not. (The target is truly locked when **LOCK#** is deasserted during the address phase and asserted on the following clock.)

Existing software that does not support the PCI lock usage rules has the potential of not working correctly. PCI resident memory (primarily system memory) that supports

LOCK# and desires to be backward compatible to existing software is recommended to implement complete resource lock. Refer to Section 3.6.5. for details of how to avoid the deadlock.

A master that uses **LOCK#** on PCI must adhere to the following rules:

1. A master can access only a single resource during a lock operation.
2. A lock cannot straddle a device boundary.
3. Sixteen bytes (aligned) is the maximum resource size a master can count on as being exclusive during a lock operation. An access to any part of the 16 bytes locks the entire 16 bytes.
4. The first transaction of a lock operation must be a read transaction.
5. **LOCK#** must be asserted the clock following the address phase and kept asserted to maintain control.
6. **LOCK#** must be released if Retry is signaled before a data phase has completed and the lock has not been established.[19]
7. **LOCK#** must be released whenever an access is terminated by Target-Abort or Master-Abort.
8. **LOCK#** must be deasserted for a minimum of one Idle state between consecutive lock operations.

3.6.1. 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 grant, the master continues to monitor **LOCK#**. If **LOCK#** is ever busy, the master deasserts **REQ#** because another agent has gained control of **LOCK#**.

When the master is granted access to the bus and **LOCK#** is not busy, ownership of **LOCK#** has occurred. 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 3-13 illustrates starting an exclusive access. **LOCK#** is deasserted during the address phase to request a lock operation which must be initiated with a read command. **LOCK#** must be asserted the clock following the address phase, which is on clock 3, or the clock after **FRAME#** is asserted (for either SAC or DAC) to keep the target in the locked state which allows the current master to retain ownership of **LOCK#** beyond the end of the current transaction.

[19] Once lock has been established, the master retains ownership of **LOCK#** when terminated with Retry or Disconnect.

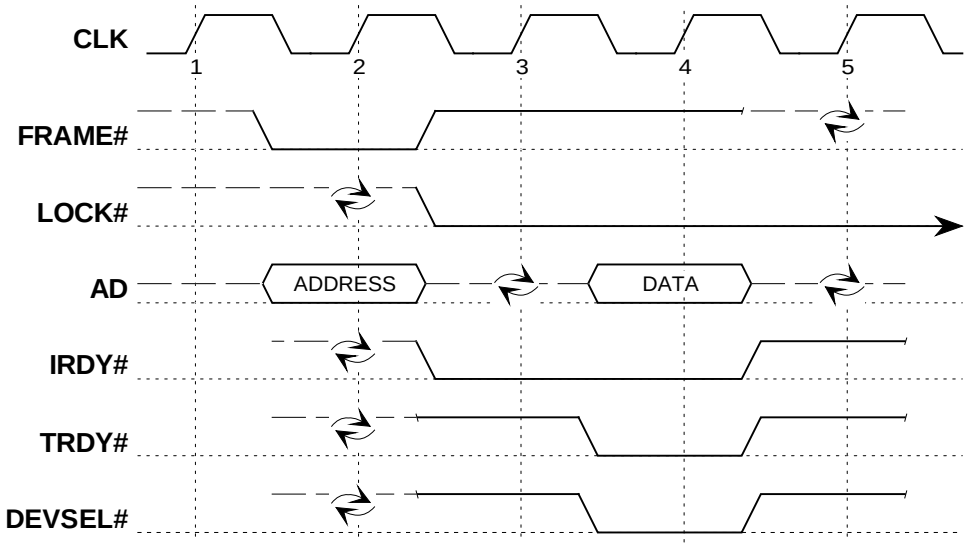


Figure 3-13: Starting an Exclusive Access

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 retries the first transaction without a data phase completing, not only must the master terminate the transaction but it must also release **LOCK#**. Once the first data phase completes, the exclusive operation is established and the master keeps **LOCK#** asserted until either the lock operation completes or an error (Master- or Target-Abort) causes an early termination. Target termination of Retry and Disconnect are 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#**. Non-exclusive accesses to unlocked targets on PCI are allowed to occur while **LOCK#** is asserted. When the exclusive access is complete, **LOCK#** is deasserted and other masters may vie for ownership.

3.6.2. Continuing an Exclusive Access

Figure 3-14 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 re-establish 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 last data phase which occurs on clock 5 (refer to Section 3.6.4. for more information on completing an exclusive access).

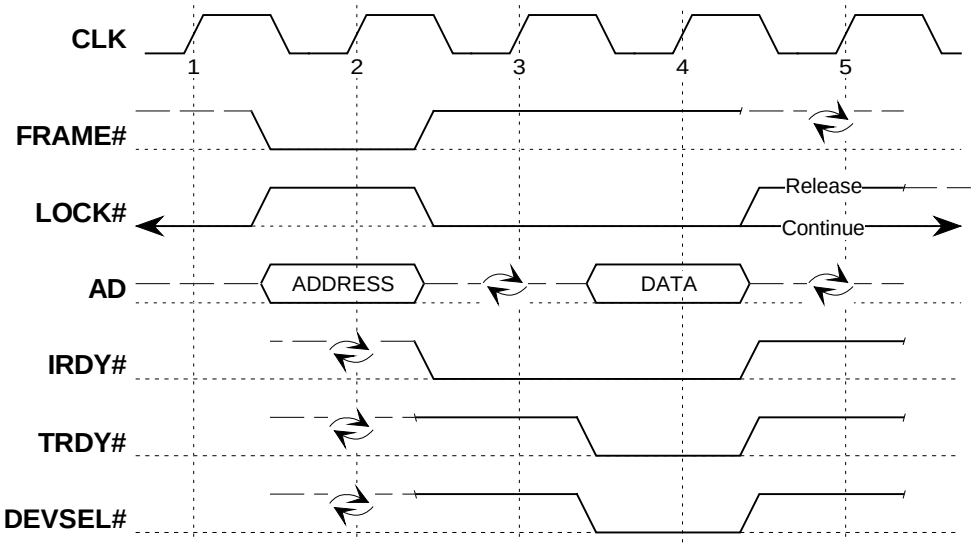


Figure 3-14: Continuing an Exclusive Access

3.6.3. Accessing a Locked Agent

Figure 3-15 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, it signals Retry and no data is transferred. An unlocked target ignores **LOCK#** when deciding if it should respond. Also, since **LOCK#** and **FRAME#** are asserted during the address phase, an unlocked target does not go into a locked state.

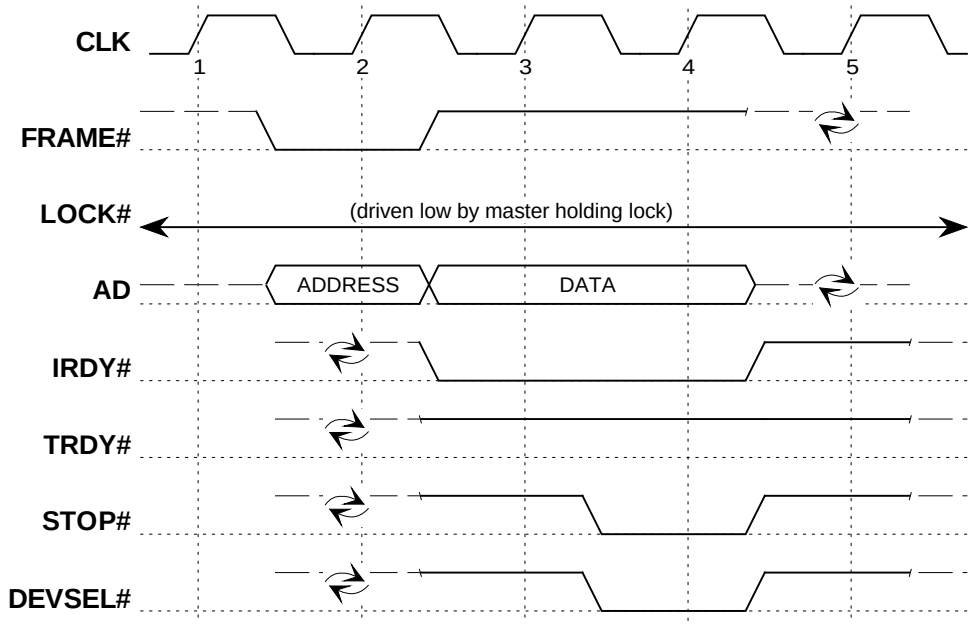


Figure 3-15: Accessing a Locked Agent

3.6.4. Completing an Exclusive Access

During the final transfer of an exclusive operation, **LOCK#** is deasserted so the target will accept the request, and then re-asserted 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 3-12 (clock 3) would be illegal.) 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 one clock edge.)

3.6.5. Supporting LOCK# and Write-back Cache Coherency

The resource lock, as described earlier, has a potential deadlock when using write-back cache coherency. A deadlock may occur if software allows locks to cross a cacheline boundary when write-back caching is supported and a complete resource lock is used. An example of this potential deadlock is where the lock spans cachelines n and n+1. Cacheline n+1 has been modified by the cache. A master establishes a lock by reading cacheline n. The lock operation continues by reading cacheline n+1. The snoop (of n+1) results in HITM which indicates that a modified line was detected. The writeback of the modified line fails because the target only accepts accesses from the owner of **LOCK#**. This results in a deadlock because the read cannot occur until the modified line is written back and the writeback cannot occur until **LOCK#** ownership is released.

This deadlock is avoided by requiring targets that support cacheable writeback memory (and complete resource lock) to allow writebacks even when locked.

The target can distinguish between a writeback and other write transactions by the state of **SDONE** and **SBO#** during the address phase (or the clock after the assertion of **SDONE** when addresses are queued). When CLEAN is indicated during the address phase, the current transaction is either CLEAN or a writeback. A transition from STANDBY to CLEAN during the address phase indicates a line replacement, while a transition from HITM to CLEAN during the address phase indicates a writeback caused by a snoop. The target of a writeback caused by a snoop (HITM to CLEAN during the address phase) must accept the writeback even when locked. The target may optionally accept line replacements (STANDBY to CLEAN during the address phase) but is not required when locked. All other transactions are terminated by the target with Retry when locked. Note: The target of a writeback caused by a snoop cannot terminate the transaction until the cacheline has been transferred. This means that Retry and Disconnect are not allowed for writebacks caused by a snoop.

3.6.6. 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. When the first access of the locked sequence is retried, the master must deassert both its **REQ#** and **LOCK#**. When 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 locked operation is in progress.

As with the complete resource lock and write-back cacheable memory, a potential deadlock exists for complete bus lock. The arbiter that supports complete bus lock must grant the bus to the cache to perform a writeback due to a snoop to a modified line when a lock is in progress. (The target is required to accept a writeback when locked because it cannot tell if complete bus or resource lock is being used.)

3.7. Other Bus Operations

3.7.1. Device Selection

DEVSEL# is driven by the target of the current transaction as shown in Figure 3-16 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) issue 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.

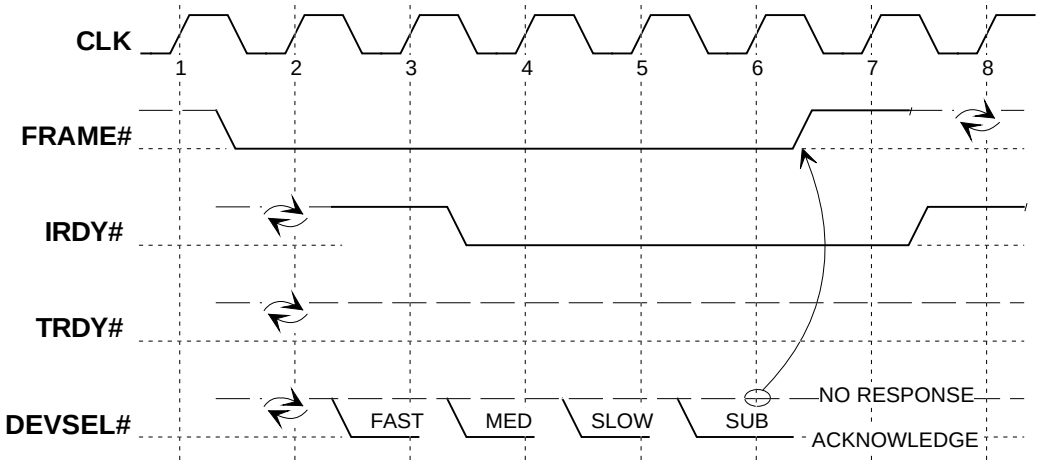


Figure 3-16: DEVSEL# Assertion

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 samples **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 3.2.2. 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.)

3.7.2. Special Cycle

The Special Cycle command provides a simple message broadcast mechanism on PCI. In addition to communicating processor status (as is done on Intel processor buses), 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 command to do so. Type 1 configuration commands can traverse PCI-to-PCI bridges in both directions for the purpose of generating Special Cycles 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 3.7.4. 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 is message dependent. When the master inserts a wait state or transfers 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 (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 64K messages. The message encodings are defined and described in Appendix A.

3.7.3. Address/Data Stepping

The ability of an agent to spread assertion of qualified signals over several clocks is referred to as *stepping*. This notion allows an agent with "weak" output buffers to drive a set of signals to a valid state over several clocks (*continuous stepping*), thereby reducing the ground current load generated by each buffer. An alternative approach allows an agent with "strong" output buffers to drive a subset of them on each of several clock edges until they are all driven (*discrete stepping*), thereby reducing the number of signals that must be switched simultaneously. All agents must be able to handle address and data 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**.

Either continuous or discrete stepping allows an agent to trade off performance for cost (fewer power/ground pins). When using the continuous stepping approach, care must be taken to avoid mutual coupling between critical control signals that must be sampled on each clock edge and the stepped signals that may be transitioning on a clock edge. Performance critical peripherals should apply this "permission" sparingly.

Stepping is only permitted on **AD[31::00]**, **AD[63::32]**, **PAR**, **PAR64#** (for 64-bit data transfers but not for the DAC command), and **IDSEL** pins, because they are always qualified by control signals; i.e., these signals are only considered valid on clock edges for which they are qualified. **AD**s are qualified by **FRAME#** in address phases and by **IRDY#** or **TRDY#** in data phases (depending on which direction data is being transferred). **PAR** is implicitly qualified on each clock after which **AD** was qualified. **IDSEL** is qualified by the combination of **FRAME#** and a decoded configuration command.

Figure 3-17 illustrates a master delaying the assertion of **FRAME#** until it has successfully driven all **AD** lines. 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. But it may

take multiple clocks to drive a valid address before asserting **FRAME#**. 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.

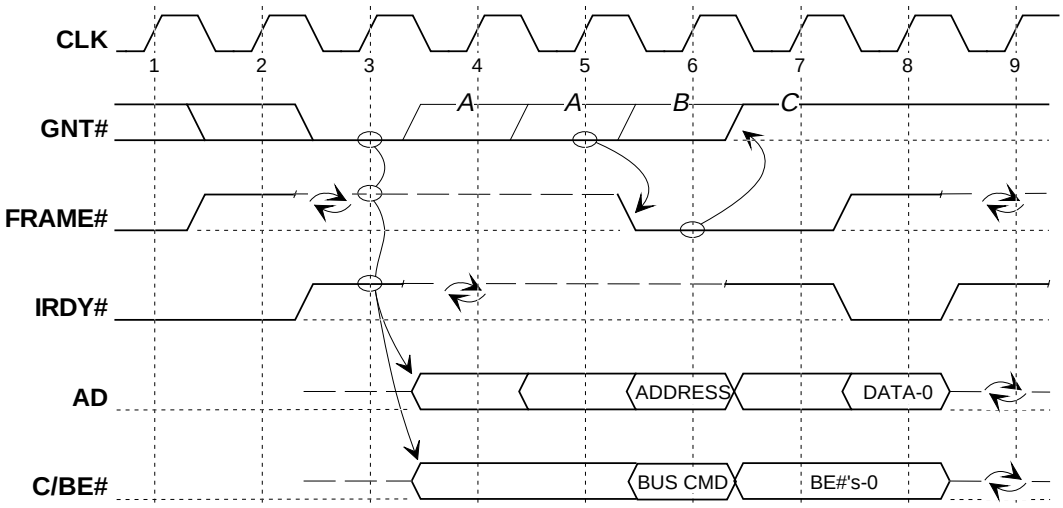


Figure 3-17: Address Stepping

3.7.4. Configuration Cycle

The PCI definition provides for totally software driven initialization and configuration via a separate Configuration Address Space. PCI devices are required to provide 256 bytes of configuration registers for this purpose. Register descriptions are provided in Chapter 6. This section describes the PCI bus commands for accessing PCI Configuration Space.

As previously discussed, each device decodes its own addresses for normal accesses. However, accesses in the Configuration Address Space require device selection decoding to be done externally, and to be signaled to the PCI device via the **IDSEL** pin, which functions as a classical "chip select" signal. A PCI device is a target of a configuration command (read or write) only if its **IDSEL** is asserted and **AD[1::0]** are "00" (indicating a Type 0 configuration transaction) during the address phase of the command. Internal addressing of the 64-DWORD register space is done by **AD[7::2]** and the byte enables. 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.

When a configuration command has multiple data phases (burst) linear burst ordering is the only addressing mode allowed, since **AD[1::0]** convey configuration access 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 access). The rest of the transaction is the same as other commands, including all termination semantics. If no agent responds, the request is terminated via Master-Abort (Section 3.3.3.1.). A standard expansion bus bridge must not forward a configuration transaction to an expansion bus. Note: The *PCI-to-PCI Bridge Architecture Specification* restricts Type 1 configuration cycles that are converted into a Special Cycle command to a single data phase (no Special Cycle bursts).

Implementation Note: System Generation of IDSEL

How a system generates **IDSEL**s 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 **AD16**, **IDSEL** of Device Number 1 is connected to **AD17,** and so forth until **IDSEL** of Device Number 16 is connected to **AD31**. 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.

The binding between a device number in the CONFIG_ADDRESS register and the generation of an **IDSEL** is not specified. Therefore, BIOS 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 accesses 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.

Exactly how the **IDSEL** pin is driven is left to the discretion of the host/memory bridge or system designer. IDSEL generation behind a PCI-to-PCI bridge is specified in the *PCI-to-PCI Bridge Architecture Specification.* However, this select 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. There is no known or standard way of determining **IDSEL** from the upper 21 address bits; therefore, the **IDSEL** pin MUST be supported. 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 primary bus bridge, since it defines how **IDSEL**s are mapped. **AD[31::00]** lines must be actively driven during the address phase. By connecting a different address line to each device, and by asserting one of the **AD[31::11]** lines at a time, 21 different devices can be uniquely selected for configuration accesses.

The issue with this approach (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- 18 with the "XXXX" marks. However, since it is only used on the address phase of a configuration cycle, the address bus can be pre-driven a few clocks before **FRAME#**[20], thus guaranteeing **IDSEL** to be stable when it needs to be sampled. For all other cycles, **IDSEL** is undefined and may be at a non-deterministic level during the address phase. Pre-driving the address bus is equivalent to address stepping as discussed in the previous section.

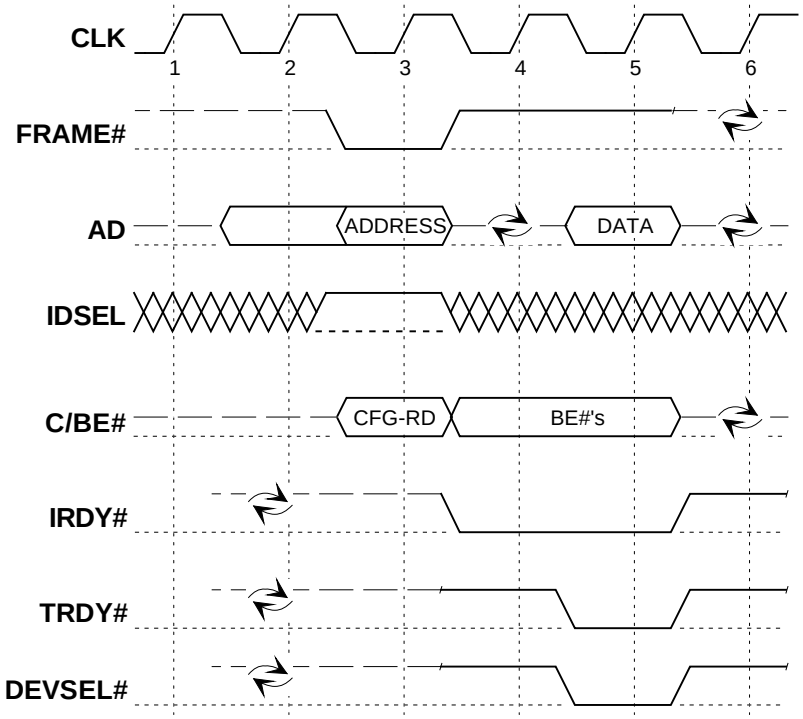


Figure 3-18: Configuration Read

To support hierarchical PCI buses, two types of configuration access are used. They have the formats illustrated in Figure 3-19 which show the interpretation of **AD** lines during the address phase of a configuration access.

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

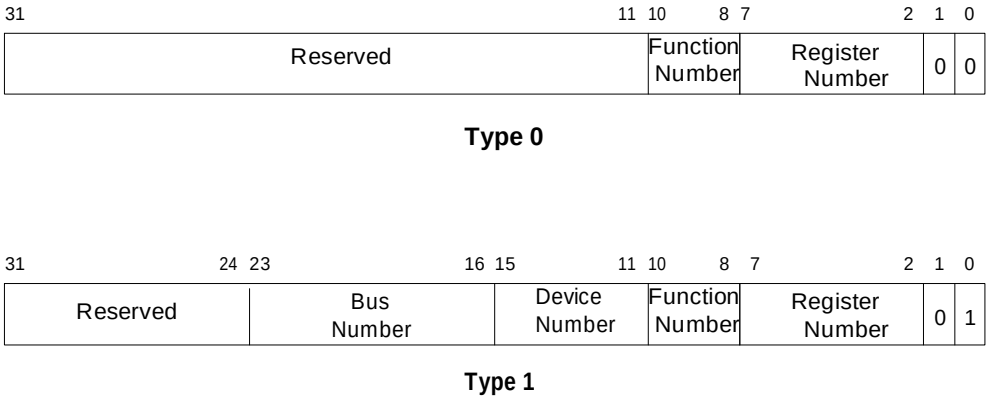


Figure 3-19: Configuration Access Formats

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

The Register Number and Function Number fields have the same meaning for both configuration types, while Device Number and Bus Number are used only in Type 1 accesses. Reserved fields must be ignored by targets.

Register Number	is an encoded value used to index a DWORD in 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. (There are only 21 devices that can be selected by tying the **IDSEL** to an **AD** (**AD[31::11]**) line.)
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 cycle 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 access.

Type 0 configuration accesses are 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 access resides on another bus (not the local PCI bus), a Type 1 configuration access must be used. Type 1 accesses are ignored by all targets except PCI-to-PCI bridges. These devices decode the Bus Number field to determine if the destination of the configuration access is residing behind the bridge. If the Bus Number is not for a bus behind the bridge, the access is ignored. The bridge claims the access if the access is to a bus behind the bridge. If the Bus Number is not to the secondary bus of the bridge, the access is simply passed through unchanged. If the Bus Number matches the secondary bus number, the bridge converts the access into a Type 0 configuration access. 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 configuration access.

Note: PCI-to-PCI bridges can also forward transactions upstream (refer to the *PCI-to-PCI Bridge Architecture Specification* for more information).

Devices that respond to Type 0 configuration cycles are separated into two classes. The first class (single function device) is defined for backward compatibility, and only uses its **IDSEL** pin and **AD[1::0]** ("00") to determine whether or not to respond. The second class of device (multi-function device) understands the Function Number field and uses its **IDSEL** pin, **AD[1::0]** ("00") as well as the encoded value on **AD[10::08]** to determine whether or not to respond. The two classes are differentiated by an encoding in the Configuration Space header.

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 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).

Configuration code will probe the bus in Device Number order (i.e. Function Number will be 0). If a single function device is detected, no more functions for that Device Number will be checked. If a multi-function device is detected, then all remaining Function Numbers will be checked.

Generating Configuration Cycles

Systems must provide a mechanism that allows PCI configuration cycles to be generated by software. This mechanism is typically located in the host bridge. For PC-AT compatible systems, the mechanism for generating configuration cycles is defined and specified below. 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 accesses is not defined in this specification.

For PC-AT compatible machines, there are two distinct mechanisms defined to allow software to generate configuration cycles. These are referred to as Configuration Mechanism #1 and Configuration Mechanism #2. Configuration Mechanism #1 is the preferred implementation and must be provided by all future host bridges (and existing bridges should convert if possible). Configuration Mechanism #2 is defined for backward compatibility and must not be used by new designs[21]. Host bridges to be used in PC-AT compatible systems must implement at least one of these mechanisms.

[21] This mechanism adds a significant software burden and impacts performance when used in a multi-processor system. The operating system and its device drivers must cooperate in order to guarantee mutually exclusive access to the I/O address range of C000h-CFFFh for both configuration space and device I/O accesses. A suitable synchronization mechanism is difficult to add into existing multi-processor operating systems/drivers where drivers currently perform direct access to their allocated I/O space.

3.7.4.1. Configuration Mechanism #1

Two DWORD I/O locations are used in this mechanism. The first DWORD location (CF8h) references a read/write register that is named CONFIG_ADDRESS. The second DWORD address (CFCh) references a register named CONFIG_DATA. The general mechanism for accessing Configuration Space is to write a value (must be a DWORD operation) into CONFIG_ADDRESS that specifies the PCI bus, the device on that bus, and the configuration register in that device being accessed. Note: The host bridge determines the configuration access type (1 or 0) based on the value of the bus number in the CONFIG_ADDRESS register. The Enable bit (bit 31) in the CONFIG_ADDRESS register must be set to access the CONFIG_DATA register; otherwise, the access is passed through the bridge as an I/O transaction. A read or write to CONFIG_DATA will then cause the bridge to translate that CONFIG_ADDRESS value to the requested configuration cycle on the PCI bus. Accesses to CONFIG_DATA determine the size of the access to the configuration register addressed by CONFIG_ADDRESS and can be performed as a byte, word, or DWORD operation. For example, a host processor does a word (host byte enables 1 and 0 asserted) access to CONFIG_DATA. The host bus bridge drives the data stored in CONFIG_ADDRESS onto the AD lines during the address phase. If a Type 0 access (**AD0** = 0), then CONFIG_ADDRESS[10::00] are passed unmodified onto **AD[10::00]** but **AD[31::11]** are decoded from CONFIG_DATA[15::11]. If a Type 1 access (**AD0** = 1), then CONFIG_ADDRESS**[23::00]** are passed unmodified onto **AD[23::01]** but **AD[31::24]** are driven to 0000 0000b. During the data phase, the bridge passes the byte enables from the host bus to PCI by asserting **C/BE#[3::0]** = 1100. This will access the lower two bytes of the configuration register in the device on the specific bus addressed by the CONFIG_ADDRESS register.

The CONFIG_ADDRESS register is 32 bits with the format shown in Figure 3-20. Bit 31 is an enable flag for determining when accesses to CONFIG_DATA should be translated to configuration cycles 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.

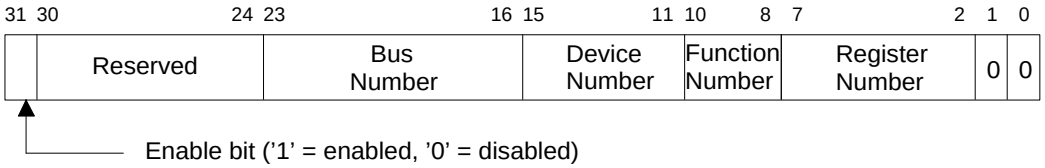


Figure 3-20: Layout of CONFIG_ADDRESS Register

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) must be treated like a normal I/O access and no special action should be taken. Therefore, the only I/O Space consumed by this register is a DWORD at the given address. I/O devices using BYTE or WORD registers are not affected because they will be passed on unchanged.

When a 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 configuration cycle translation is enabled and the Bus Number matches the bridge's Bus Number or any Bus Number behind the bridge, a configuration cycle translation must be done.

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 bridge. The second, Type 1, occurs when the device is on another bus somewhere behind this bridge.

For Type 0 translations (see Figure 3-21), the bridge does a decode of the Device Number field to assert the appropriate **IDSEL** line[22] and performs a configuration cycle 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 may be used by components that contain multiple functions. **AD[7::2]** are also copied from the CONFIG_ADDRESS register. Figure 3-21 shows the translation from the CONFIG_ADDRESS register to **AD** lines on the PCI bus.

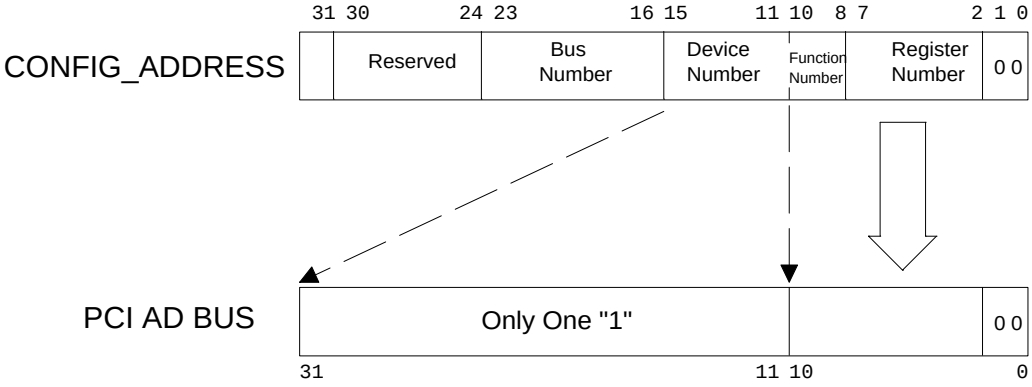


Figure 3-21: Bridge Translation for Type 0 Configuration Cycles

For Type 1 translations, the bridge directly copies the contents of the CONFIG_ADDRESS register onto the PCI **AD** lines during the address phase of a configuration cycle 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.

For systems with peer bridges on the processor bus, one peer bridge would typically be designated to always acknowledge accesses to the CONFIG_ADDRESS register. Other 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 typically require two Configuration Space registers whose contents are used to determine when the bridge does configuration cycle 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

[22] 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. This is easily implemented 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.

behind the bridge.[23] A PCI-to-PCI bridge requires an additional register which is its Primary Bus Number. POST code is responsible for initializing these registers to appropriate values.

Generating Special Cycles with Configuration Mechanism #1

This section defines how host bridges that implement Configuration Mechanism #1 for accessing Configuration Space should allow software to generate Special Cycles. Host bridges are not required to provide a mechanism for allowing software to generate Special Cycles.

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 do a Special Cycle command the next time the CONFIG_DATA register is written. When the CONFIG_DATA register is written, the bridge generates a Special Cycle command encoding (rather than configuration write) on the **C/BE[3::0]#** pins during the address cycle, and drives the data from the I/O write onto **AD[31::00]** during the first data cycle. Reads to CONFIG_DATA, after CONFIG_ADDRESS has been set up this way, have undefined results. The bridge can treat it as a normal configuration cycle operation (i.e., generate a Type 0 configuration cycle 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 cycle just like anytime the bus numbers do not match.

3.7.4.2. Configuration Mechanism #2

This mechanism for accessing PCI Configuration Space provides a mode that maps PCI Configuration Space into 4K bytes of the CPU I/O Space. When the mode is set to enable PCI Configuration Space mapping, any CPU access within I/O address range C000h-CFFFh will be translated to a PCI configuration cycle. When the mode is set to disable PCI Configuration Space mapping, all CPU I/O accesses in that range will be routed to the appropriate I/O port in the system. This mechanism does not support peer bridges on the processor bus.

Two registers are used in this mechanism. These registers are described below.

Configuration Space Enable Register

Configuration Space is mapped into I/O Space by writing to the Configuration Space Enable (CSE) register located at I/O location CF8h. The fields in the CSE register are shown in Figure 3-22.

[23] Host bridges that do not allow peer bridges do not need either of these registers since the bus behind the bridge is, by definition, bus 0 and all other PCI buses are subordinate to bus 0.

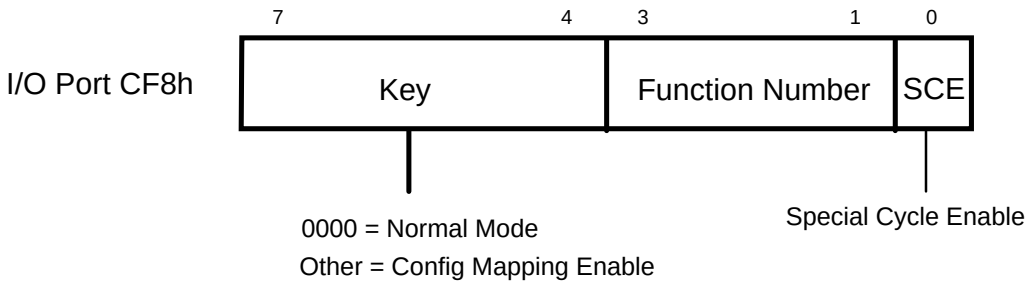


Figure 3-22: Configuration Space Enable Register Layout

This register is a read/write I/O port that logically resides in the host bridge. Bit 0 is an enable bit for generating PCI Special Cycle commands. This bit must be set to zero to generate configuration cycles and set to one to generate Special Cycle commands. Bits 1 to 3 provide the function number for the configuration cycle. These three bits are transferred to **AD[10::08]** when a configuration cycle is generated. The key field is used to enable the mapping function that maps reads/writes of I/O Space to reads/writes in the PCI Configuration Space. The host bridge responds to a read of the CSE register by returning the last data written to that register. All accesses of the CSE register must be single byte operations.

After reset, the CSE register is cleared and the host bridge comes up in the default state where it treats all I/O accesses normally.

Forward Register

The Forward register, located at I/O address CFAh, is used to specify which PCI bus is being accessed. This register is read/write, initialized to 0 at reset, and returns the last value written when read. When the Forward register is 00h, then the bus immediately behind the bridge is being accessed and Type 0 configuration accesses are generated. When the Forward register is non-zero, then Type 1 configuration accesses are generated and the contents of the Forward register are mapped to **AD[23::16]** during the address phase of the configuration cycle.

Configuration Space Mapping

When the bridge is enabled to do Configuration Space mapping (i.e. the Key field of the CSE register is non-zero), the bridge must convert all I/O accesses in the range C000h-CFFFh to PCI configuration cycles. Sixteen PCI devices (per bus) are addressable using bits 11::8 of the I/O address. Bits 7::2 of the I/O address select a particular DWORD within the device's Configuration Space.

Figure 3-23 shows the translation made when the Forward register is zero. This indicates that the device being accessed is on PCI bus 0 which is directly behind the bridge. The translation produces a Type 0 configuration cycle.

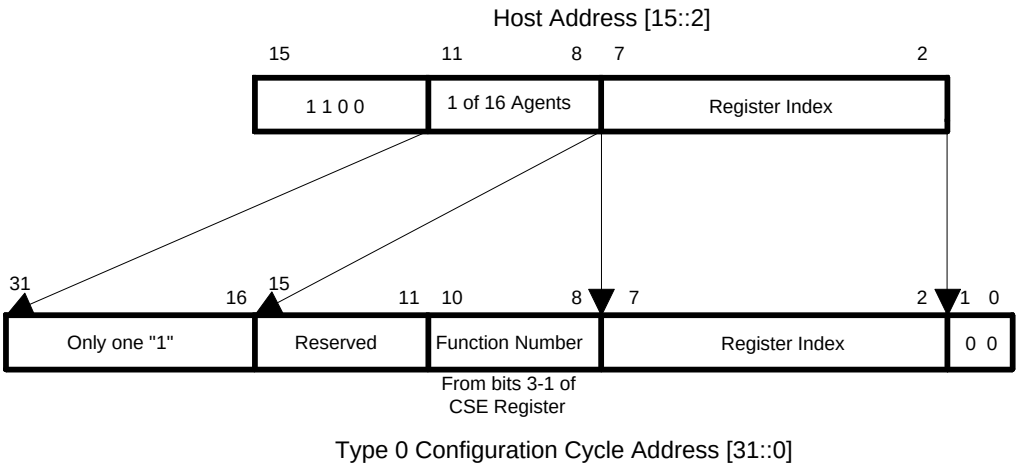


Figure 3-23: Translation to Type 0 Configuration Cycle

Figure 3-24 shows the translation made when the Forward register is non-zero. This indicates that the device being accessed is on a PCI bus other than the one directly behind the bridge. The bridge must generate a Type 1 configuration cycle and map the Forward register onto **AD[23::16]** of the PCI bus.

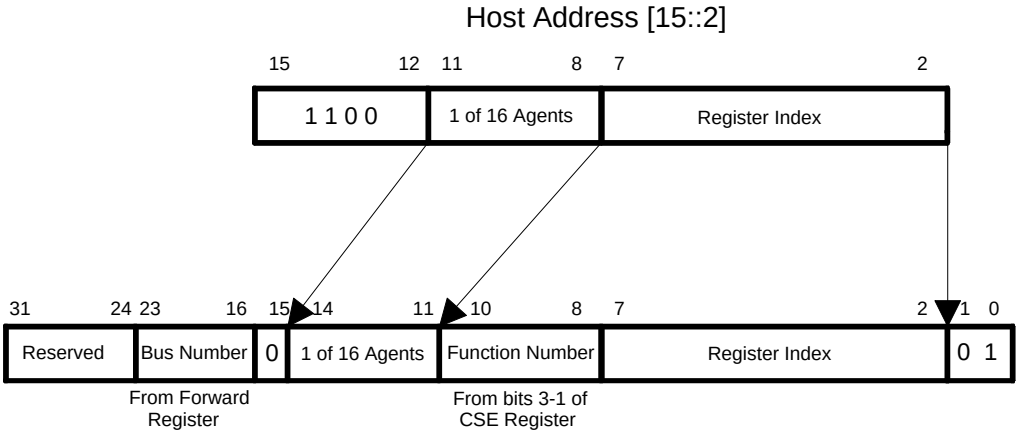


Figure 3-24: Translation to Type 1 Configuration Cycle

Generating Special Cycles with Configuration Mechanism #2

This section defines how host bridges that implement Configuration Mechanism #2 for accessing Configuration Space should allow software to generate Special Cycles. Host bridges are not required to provide a mechanism for allowing software to generate Special Cycles.

When the CSE register is setup so that bit 0 is a "1", the Function Number field is all ones, and the key field is non-zero, then the bridge is enabled to do a Special Cycle or a Type 1 configuration cycle on the PCI bus the next time a CPU I/O write access is made to I/O location CF00h.

When Special Cycle generation is enabled and the CPU does a write access to I/O address CF00h, the bridge compares the contents of the Forward register to 00h. If the contents of Forward register are 00h, then the host bridge generates a PCI Special Cycle on the PCI bus. During the address phase, the bridge generates the Special Cycle

encoding on **C/BE[3::0]#** and drives the data from the I/O write (to CF00h) on **AD[31::00]** during the first data phase of the Special Cycle. If the contents of the Forward register are not equal to "0", then the host bridge generates a Type 1 configuration cycle on the PCI bus with the Device Number and Function Number fields as all 1's (**AD[15::08]** being all 1's) and the Register Number being 00h (**AD[7::2]** being all 0's) during the address phase of PCI configuration write cycle.

Read accesses to I/O addresses CXXXh while the CSE register is enabled for Special Cycles will have undefined results. Write accesses to I/O addresses CXXXh (except for CF00h) while the CSE register is enabled for Special Cycles will have undefined results. In the CSE register, whenever the Special Cycle Enable (SCE) bit is set and the Function Number field is not all 1's, I/O accesses in the CXXXh range will have undefined results.

3.7.5. Interrupt Acknowledge

The PCI bus supports an Interrupt Acknowledge cycle as shown in Figure 3-25. 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.

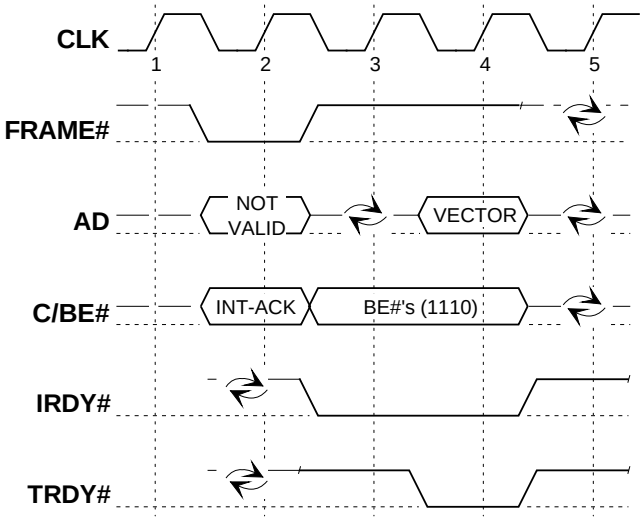


Figure 3-25: Interrupt Acknowledge Cycle

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.

3.8. Error Functions

PCI provides for parity and other system errors to be detected and reported. PCI error coverage may range from devices that have no interest in errors (particularly parity errors) to agents that detect, signal, and recover from errors. This 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 3.8.2.

The discussion of errors is divided into the following two sections covering parity generation and detection, and error reporting. Each section explains what is optional and what is required for each function.

3.8.1. Parity

Parity on PCI provides a mechanism to determine transaction by 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 (see Section 3.10 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 commands 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 3-26 illustrates a read and write transaction 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) while 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.

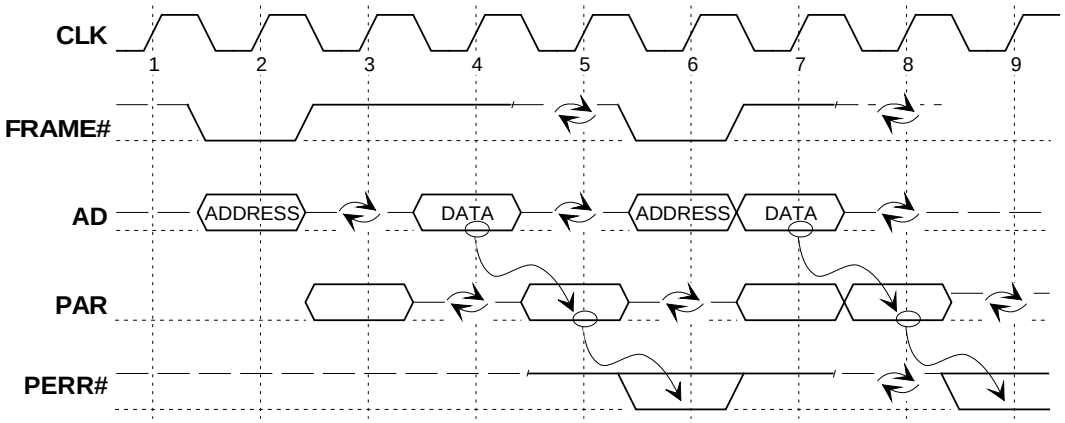


Figure 3-26: Parity Operation

Parity must be checked to determine if the master successfully addressed the desired target and if data transferred correctly. Checking of parity on PCI is required except in two classes of devices listed in Section 3.8.2. Agents that support parity checking must always set the Detected Parity Error bit in the Configuration Space Status register (refer to Section 6.2.3.) when a parity error is detected. Any additional action beyond setting this bit is conditioned upon the Parity Error Response bit in the Configuration Space Command register and is discussed in the error reporting section.

Any agent may check and signal an address parity error on **SERR#**. Only the master may report a read data parity error and only the selected target may signal a write data parity error.

3.8.2. Error Reporting

As previously mentioned, PCI provides for the detection and signaling of both parity and other system errors. It is intended that 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 directly to the system level.

Two signals (pins) are used in the PCI error reporting scheme. **PERR#** is used exclusively for reporting data parity errors on all transactions except Special Cycle commands. **PERR#** is a sustained tri-state signal that is bused to all PCI agents. Bus protocol assures that **PERR#** will never be simultaneously driven by multiple bus agents and that proper signal turn around times are observed to avoid any driver contention.

SERR# is used for other error signaling, including address parity and data parity on Special Cycle commands, and may optionally be used on any other non-parity or system errors. **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.

Both **PERR#** and **SERR#** are required pins since parity error signaling on PCI is required. This requirement is waived, however, for these two classes of devices:

- Devices that are designed exclusively for use on the motherboard or planar; e.g., chip sets. System vendors have control over the use of these devices since they will never appear on add-in boards.
- 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; and, therefore, are not prone to create system integrity problems in the event of undetected failure.

Note: All agents are required to generate parity (no exclusions on this requirement). Use of **SERR#** to signal non-parity errors is optional. It must be assumed, however, that signaling on **SERR#** will generate an NMI and is, therefore, fatal. Consequently, care should be taken in using **SERR#**.

The following sections cover the responsibility placed on each bus agent regarding signaling on the **PERR#** and **SERR#** pins.

3.8.2.1. Parity Error Response and Reporting on PERR#

This section describes proper response to, and reporting of, data parity errors in all bus operations except Special Cycle commands. All address parity errors, as well as Special Cycle command data parity errors are reported on the **SERR#** signal, and are described in the next section. All references to parity errors in this section are, by implication, limited strictly to data parity (except Special Cycle commands).

PCI uses the **PERR#** pin to signal a data parity error between the current master and target on PCI (except on Special Cycle commands). 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#**). 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, this gives the originator of the access, at each hardware or software level, the prerogative of recovery.

Implementation Note: Reporting of Data Parity Errors

PCI allows for data parity error recovery but does not require it. Recovery can occur at the device (lowest level), the device driver, or the operating system (highest level). This means when a data parity is detected by a target and is reported to the master by **PERR#** or the error is detected by the master, it is allowed to attempt recovery from the error. It is recommended that the error be recovered at the lowest level possible and, if not recoverable, the error must be reported (if enabled to do so) to the operating system. The following are examples of how recovery may occur:

- **Recovery by the master.** If the master (of the transaction in which the parity error was detected) has sufficient knowledge that the access can be repeated without side-effects, then the master may simply repeat the access. If no error occurs on the repeated access, reporting of the parity error (to the operating system or device driver) is not required. When the master cannot or is not capable of recovering from the data parity error, it must inform its device driver by generating an interrupt (or modifying a status register or flag to mention a few options). When the master does not have a device driver, it may report the error by asserting **SERR#**.
- **Recovery by the device driver.** The device driver may support an error recovery mechanism such that the data parity error can be corrected and reporting the error is not required. The driver may be able to repeat the 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 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.

Note: FIFOs and most registers (I/O or memory mapped) have side-effects when accessed and, therefore, are not prefetchable and error recovery (by repeating the transaction) is not possible by the master or low level software. The master cannot recover from an error when a target reports a parity error by another mechanism, for example, by asserting an interrupt or **SERR#**, other than **PERR#**.

The intent of the error signals is to provide a way for data parity errors to be detected and, if allowed by the system, to recover from them; otherwise (if enabled), they are simply reported to the operating system.

Except for setting the Detected Parity Error bit, all parity error signaling and response is controlled by the Parity Error Response bit. This bit is required except in the previously listed (excluded) devices. If the bit is cleared, the agent ignores all parity errors and completes the transaction as though parity was correct. If the bit is set, the agent is required to assert **PERR#** when a parity error is detected; additional error response is device dependent. In all cases, the Detected Parity Error bit must be set.

An agent must always assert **PERR#** two clocks after a data transfer in which an error occurred, as shown in Figure 3-26. The agent receiving data is free to assert **PERR#** when a parity error is detected (which may occur before data is transferred).[24] Once

[24] On a write transaction, this can occur when **IRDY#** is asserted and the target is inserting wait states. On a read transaction, this occurs when **TRDY#** is asserted and the master is inserting wait states.

PERR# is asserted, it must remain asserted until two clocks following the actual transfer. A master knows a data parity error occurred anytime **PERR#** is asserted but only knows the transfer was error free two clocks following the transfer.

In the case of multiple data transfers without intervening wait states, **PERR#** will be qualified on multiple consecutive clocks accordingly, and may be asserted in any or all of them. Since **PERR#** is a sustained tri-state signal, it must be actively driven to the correct value on each qualified clock edge. To return it to nominal state at the end of each bus operation, it must be actively driven high for one clock period, starting two clocks after the **AD** bus turnaround cycle (e.g., clock 7 in Figure 3-26). The **PERR#** turnaround cycle occurs one clock later (clock 8 in Figure 3-26). **PERR#** may never be driven (enabled) for the current cycle until at least three clocks after the address phase (SAC or DAC).

When a master detects a data parity error and asserts **PERR#** (on a read transaction) or samples **PERR#** asserted (on a write transaction), it must set the Data Parity Error Detected bit (Status register, bit 8), and can either continue the transaction or terminate it. A target of a transaction that detects a parity error can either continue the operation or cause it to be stopped via target termination. Targets never set the Data Parity Error Detected bit. When **PERR#** is asserted, it is recommended that both the master and target complete the transaction. **PERR#** is only an output signal for targets while masters use **PERR#** as both an input and output.

When the master of the access becomes aware that a parity error has occurred on its transaction, it is required to inform the system. It is recommended that the master inform its device driver of the error by generating an interrupt (or modifying a status register or flag to mention a few options). If none of these options is available to the device, it may, as a last recourse, pass responsibility of the error to the operating system by asserting **SERR#**. Note: The system designer may elect to report all parity errors to the operating system by converting all **PERR#** error signals into **SERR#** error signals in the central resource.

3.8.2.2. Error Response and Reporting on SERR#

SERR# is used to signal all address parity errors, data parity errors on Special Cycle commands (since these are broadcast writes), and all errors other than parity errors. Any agent can check and signal address parity errors on **SERR#**, regardless of the intended master and target. **SERR#** may only be asserted when the SERR# Enable bit (bit 8) in the Command register is set to a logical one (high), regardless of the error type. When an agent asserts **SERR#** it is required to set the Signaled System Error bit (bit 14) in the Configuration Space Status register, regardless of the error type. In addition, if the error type is parity (e.g., address parity), the Detected Parity Error bit (bit 15) in the Status register must be set in all cases, but reporting on **SERR#** is conditioned on the Parity Error Response bit (bit 6) in the Command register.

A selected agent that detects an address parity error should do one of the following: claim the cycle and terminate as though the address was correct, claim the cycle and terminate with Target-Abort, or not claim the cycle and let it terminate with Master-Abort. The target is not allowed to terminate with Retry or Disconnect because an address parity error was detected.

SERR# has no timing relationship to any PCI transaction.[25] However, errors should be signaled as quickly as possible; preferably within two clocks of detection. The only agent interested in **SERR#** (as an input) is the central resource that converts a low pulse into a signal to the processor. How the central resource signals the processor is system dependent, but could include generating an NMI, a high priority interrupt, or setting a status bit or flag. However, the agent that asserts **SERR#** must be willing for the central resource to generate an NMI; otherwise, the error should be reported by a different mechanism (e.g., an interrupt, status register, or flag).

When the Parity Error Response bit is enabled, and the SERR# Enable bit is enabled, an agent may assert **SERR#** under the following conditions:

- Address parity error or data parity error on Special Cycles detected.
- The detection of a parity error that is not reported by some other mechanism (current bus master only).

When the SERR# Enable bit is enabled, an agent may assert **SERR#** under the following conditions:

- The master (which does not have a driver) was involved in a transaction that was abnormally terminated.
- A catastrophic error left the agent questioning its ability to operate correctly.

Note: Master-Abort is not an abnormal condition for bridges for configuration and Special Cycle commands. **SERR#** should not be used for these conditions or for normally recoverable cases. The assertion of **SERR#** should be done with deliberation and care since the result may be an NMI. Target-Abort is generally an abnormal target termination and may be reported (only by the master) as an error by signaling **SERR#** when the master cannot report the error through its device driver.

3.9. Cache Support

In entry level or mobile systems, part or all of the system memory may be on PCI. This may include read only program modules as well as DRAM, both of which must be cacheable by the processor. The PCI cache support option provides a standard interface between PCI memory agent(s) and the bridge (or caching agent), which allows the use of a snooping cache coherency mechanism. This caching option assumes a flat address space (i.e., a single address has a unique destination regardless of access origin) and a single level bridge topology. Note: This support is optimized for simple, entry level systems, rather than maximum processor/cache/memory performance.

Caching support for shared memory is implemented by two optional pins called **SDONE** and **SBO#**. They transfer cache status information between the bridge/cache and the target of the memory request. The bridge/cache snoops memory accesses on PCI, and determines what response is required by the target to ensure system memory coherency. To avoid a "snooping overload," the bridge may be programmed to signal a Clean Snoop immediately on frequently accessed address ranges that are configured as noncacheable (e.g., frame buffer).

Any PCI target that supports cacheable memory must monitor the PCI cache support pins and respond appropriately. Targets configured to be noncacheable may ignore **SDONE** and **SBO#**, as this may save a little access latency, depending on configuration.

---

[25] Except for **CLK**.

Since PCI allows bursts of unlimited length, the cacheable target of the request must disconnect accesses in a memory range that attempt to cross a cacheline boundary. This means that any cacheable target must be aware of the cacheline size, either by implementing the Cacheline Size register (refer to Section 6.2.4.)or by hardwiring this parameter. If a burst is allowed to cross a cacheline boundary, the cache coherency may "break." (Alternatively, the bridge/cache can monitor the transaction and generate the next cacheline address for snooping.)

To enable efficient use of the PCI bus, the cacheable memory controller[26] and the cache/bridge are required to track bus operation. (When only a single address is latched, a condition may occur when a cacheable transaction will be delayed. This occurs when a noncacheable transaction is initiated when the cache is ready to snoop an address. When the address is latched, the cache will start the snoop. Since the transaction is noncacheable, it will complete regardless of the state of **SDONE**. When the next transaction is initiated while the first snoop is still pending, a cacheable transaction is required to be retried; otherwise, the cacheable address will not be snooped. If noncacheable and cacheable transactions are interleaved, the cacheable transaction may never complete.) To minimize Retry termination of cacheable transactions, due to another snoop in progress, the agents involved in a cacheable transaction are required to accept two addresses. This means while the first address is being snooped, the next address presented on the bus will also be latched. When the first snoop completes, the snoop of the second address will begin immediately. The maximum number of addresses required to be latched is two. A third address can never appear on the bus without either the completion of the snoop of the first address or the termination of the second transaction. When either the snoop or the second transaction completes, the cache and memory controller are ready to accept a new address. Hence, only two addresses are ever required to be latched.

If the second transaction is cacheable, the memory controller is required to insert wait states until the first snoop completes. When the snoop of the first transaction completes, the memory controller continues with the transaction. If the second transaction is to a noncacheable address, the target may complete the transaction since **SDONE** and **SBO#** are not monitored. If the target of the second transaction asserts **TRDY#** (or **STOP#**) before or with the assertion of **SDONE**, it implies a noncacheable transaction and the cache will not snoop the address when **TRDY#** is asserted. Therefore, the maximum number of addresses that can be outstanding at any time is two.

[26] For the remainder of this section, cacheable memory controllers are simply referred to as memory controllers.

3.9.1. Definition of Cache States

The PCI specification defines **SDONE** and **SBO#** to provide information between agents that participate in the cache protocol. When **SDONE** is asserted, it indicates that the snoop has completed. When **SBO#** is asserted, it indicates a hit to a modified line. When **SBO#** is deasserted and **SDONE** is asserted, it indicates a "CLEAN" snoop result.

There are three cache states that appear on PCI. The meaning of each state when driven by the cache/bridge (which will be referred to hereafter as cache) and how a cacheable memory controller should interpret them will be discussed next. The PCI cache signals **SDONE** and **SBO#** will signal one of the following three states:

STANDBY	0x
CLEAN	11
HITM	10

3.9.1.1. Cache - Cacheable Memory Controller

When the cache drives the three states on the bus, the following is implied:

STANDBY -- indicates the cache is in one of three conditions. The first condition is when the cache is not currently snooping an address but is ready to do so. The second condition is when an address has been latched and the cache is currently snooping the address and is ready to accept (latch) a second address if presented on the bus. The last condition is when the cache is currently snooping and has latched a second address. The cache will start the snoop of the second address (if still valid) when the snoop completes. (Note: This state is signaled when **SDONE** is deasserted.) The memory controller must track the PCI control signals to know which condition the cache is in. The memory controller responds to a request as it chooses when an address is not being snooped. If a snoop is in progress and the memory controller is the target of the second transaction, it must insert wait states until the first address snoop completes. The memory controller continues the second transaction when the snoop of the first address completes. If the memory controller is not the target, it must monitor the bus to determine if the second address is snooped or is discarded.

CLEAN -- indicates no cache conflict and the memory access may complete normally. This implies a miss to the cache, or a hit to an unmodified line during a write transaction, or a hit to a modified line during a Memory Write and Invalidate command. The writebacks caused by the Memory Write and Invalidate command or a Memory Write to an unmodified cacheline are not required. (This is permissible since the master of the transaction guarantees that every byte will be modified and the target will not terminate the transaction until the entire line is transferred.) The cache will signal CLEAN during the address phase when it is the current bus master and it is writing back a modified line.

The cache may signal CLEAN on two consecutive clocks when two addresses have been latched. The first clock that **SDONE** is asserted indicates the first snoop has completed. If the first snoop is CLEAN and the second transaction was initiated by the cache, it may continue to assert **SDONE** indicating that a snoop of this (second) address will result in CLEAN. (The second consecutive CLEAN has the same meaning as during an address phase -- a writeback operation or CLEAN snoop.) Otherwise, the cache signals STANDBY after CLEAN to indicate the (second) snoop is in progress. In this case,

STANDBY (**SDONE** deasserted) would appear on the bus the clock following the assertion of **SDONE**.

HITM -- indicates the snoop hit a modified line and the cache is required to writeback the snooped line as its next operation. The cache will stay in this state until the writeback occurs. All other cacheable transactions will be terminated with Retry by the memory controller while HITM is signaled on the bus. (If any other potentially cacheable transaction is required to complete before the writeback of the modified line, a livelock will occur.) During a writeback of a modified line, the cache will transition from HITM to CLEAN during the address phase.

The memory controller will "typically" terminate the transaction with Retry, allowing the writeback to occur, and then the agent that was terminated with Retry will re-request the transaction. All cacheable memory controllers will terminate all subsequent cacheable transactions with Retry while HITM is signaled.

3.9.2. Supported State Transitions

[1] STANDBY --> CLEAN --> [CLEAN] --> STANDBY

[2] STANDBY --> HITM --> CLEAN --> [CLEAN] --> STANDBY

Sequence [1] is the normal case where the cache stays in STANDBY until the snoop completes and then signals CLEAN to indicate the transaction should complete normally. The cache transitions to STANDBY if a second address is not pending when the snoop of the first transaction completes. If a second address has been latched and the cache is not the master, it transitions to STANDBY indicating it is snooping. If the cache is the master of the second transaction, it may continue to signal CLEAN (as shown as an optional state) when the transaction is a writeback of a cacheline or knows the snoop is CLEAN; otherwise, it will transition to STANDBY.

Sequence [2] is when a modified line is detected during the snoop. Once HITM is signaled by the cache, it will continue in this state until the modified line is written back. The cache will transition to CLEAN indicating it is performing the writeback. Following CLEAN, the cache will signal STANDBY indicating the cache is ready to snoop a new address. If the cache is the master of the second transaction, it may continue to signal CLEAN (as shown as an optional state) when the transaction is a writeback of a cacheline or knows the snoop is CLEAN; otherwise, it will transition to STANDBY.

3.9.3. Timing Diagrams

In the timing diagrams in this section, it is assumed that the bus starts in the Idle state in clock 1.

The transaction in Figure 3-27 starts when an address is latched on clock 2. The target keeps **TRDY#** deasserted (inserting wait states) until the snoop completes. The snoop completes on clock 5 when **SDONE** is sampled asserted. Since **SBO#** was not asserted, the snoop result indicates CLEAN.

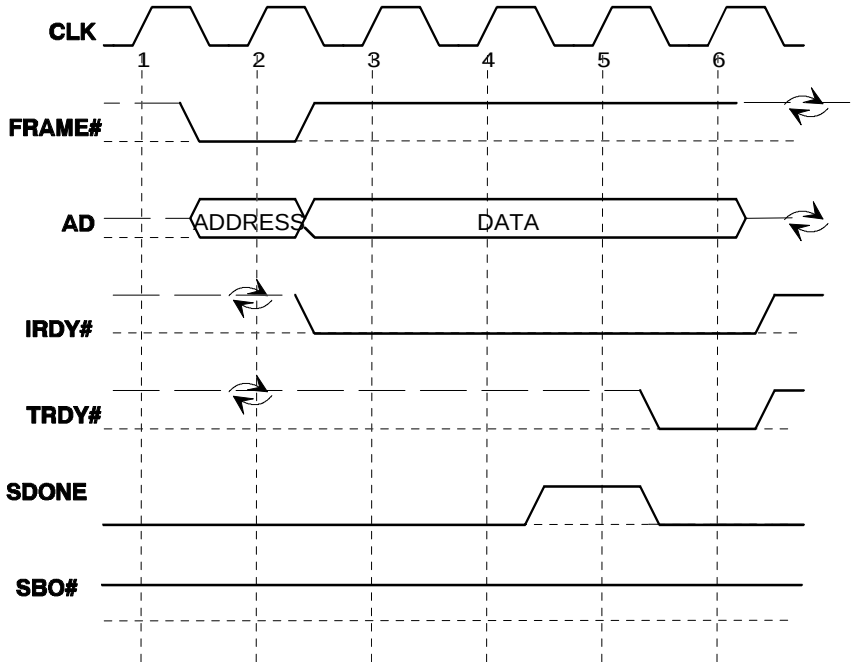


Figure 3-27: Wait States Inserted Until Snoop Completes

In Figure 3-28, the initial transaction starts on clock 2 with the address being latched. The target of this transaction inserts wait states until **SDONE** is asserted. In this example, the cache indicates that the snoop hit a modified line on clock 4 by asserting **SBO#** and **SDONE**. (Once **SBO#** is asserted, it must remain asserted until **SDONE** is asserted.) Since the target of the transaction is cacheable, it asserts **STOP#** to terminate the transaction on clock 5. This allows the cache that is signaling HITM to write the modified line back to memory. For read transactions, the memory controller must tri-state the **AD** lines when HTIM is indicated. All transactions to cacheable targets are terminated with Retry while HITM is signaled on the bus.

The slashed line indicates some amount of time has occurred since the snoop was signaled on the first transaction. During this time, noncacheable transactions may complete and cacheable transactions may start but are required to be terminated with Retry since HITM is signaled. The writeback transaction starts on clock A. Notice the cache transitions from HITM to CLEAN during the address phase. This indicates to the memory controller that the snoop writeback is starting and it is required to accept the entire line. (If the memory controller is not able to complete the transaction, it must insert wait states until it is capable. This condition should only occur when the cacheable target has an internal conflict; e.g., the refresh operation of the array.) (If the target is locked, it accepts writebacks that are caused by snoops to modified lines; otherwise, a deadlock occurs.) Both the cache and the memory controller may insert wait states during a writeback. The memory controller is required to accept the entire line in a single transaction and the cache will provide the entire line. Notice that the cache transitions from CLEAN to STANDBY on clock B. The cache is now ready to accept another address to snoop. Once the writeback completes, the bus returns to normal operation where cacheable transactions will progress. The order of the writeback is independent of the transaction that caused the writeback. In the figure, DATA-1 only indicates the first data transfer and not the DWORD number.

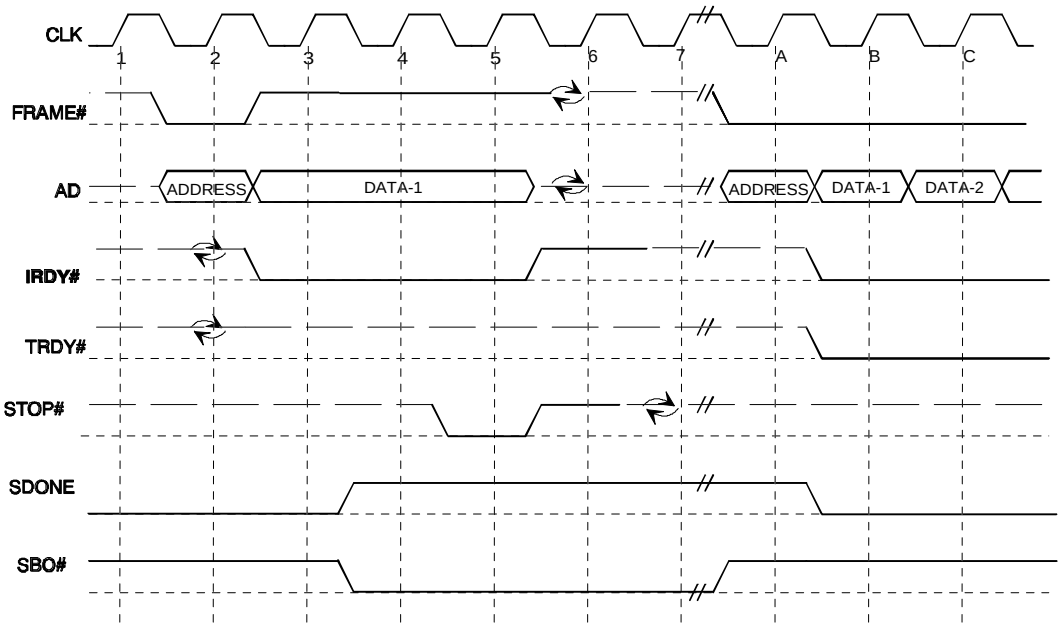


Figure 3-28: Hit to a Modified Line Followed by the Writeback

Figure 3-29 is an example of a Memory Write and Invalidate command. The cache has several options of how to handle this command. Since the master guarantees that every byte in the cacheline will be modified, the cache could simply signal CLEAN even if the line hits a modified line. In this example, the cache signals CLEAN on clock 5 indicating the snoop either resulted in a miss or a hit to a modified line that was invalidated. Once the cache indicates CLEAN, it is ready to snoop the next address presented on the bus. Therefore, the cache is required to wait until it is ready before asserting **SDONE**.

If **SBO#** were asserted on clock 5, the snoop resulted in a hit to modified line and will be written back. The cache may treat the Memory Write and Invalidate command like any other command by allowing the HITM condition to appear on the bus. (The writeback causes an extra transaction on the bus that is not required.) The cache could wait a fixed number of clocks for **TRDY#** to be asserted before indicating the snoop result. If **TRDY#** is asserted before the result, it is recommended that the cache discard the line and signal CLEAN. If **TRDY#** has not been asserted, the cache continues by providing the snoop result. However, the time waiting for **TRDY#** must be fixed because the memory controller may always wait for **SDONE** to be asserted before continuing the transaction.

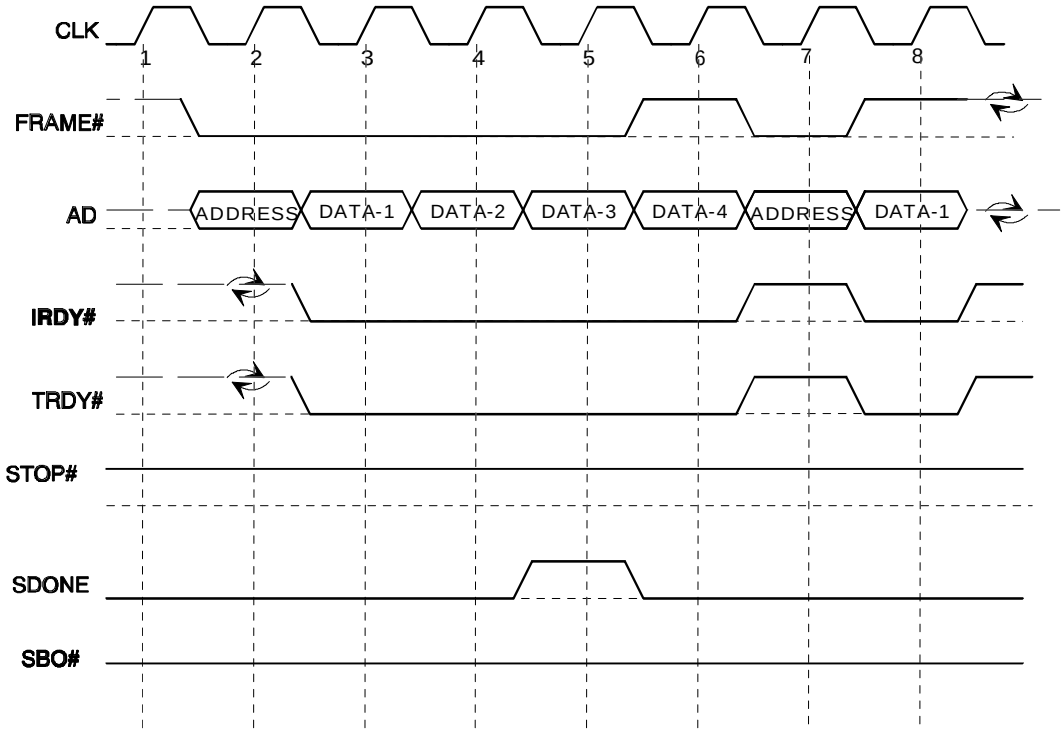


Figure 3-29: Memory Write and Invalidate Command

In Figure 3-30, the initial transaction starts on clock 2 and completes on clock 3. While the snoop of the first transaction is in progress, another transaction starts on clock 5. The second transaction is also short and completes on clock 6. The second transaction is noncacheable if it completes while the snoop of the first transaction is still in progress. On clock 7, the snoop of the first transaction completes. Once **FRAME#** has been asserted, and a snoop is in progress, the state of **SDONE** and **SBO#** only has meaning for the first address until **SDONE** is asserted. Once **SDONE** is asserted, the next time it is asserted it applies to the second transaction. If in Figure 3-30, **SDONE** is asserted on clock 5 instead of clock 7, the snoop result has no effect on the second transaction even though it is signaled during the second transaction.

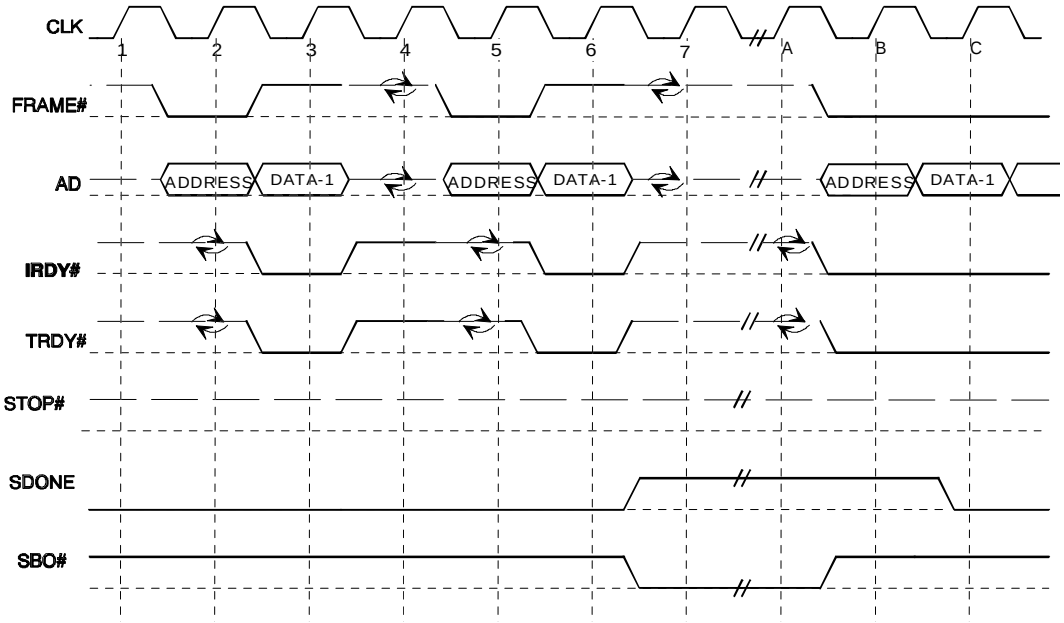


Figure 3-30: Data Transfers - Hit to a Modified Line Signaled Followed by a Writeback

3.9.4. Write-through Cache Support

Support of a write-through cache is the same as a writeback cache except **SBO#** is not used. The memory controller monitors the bus and tracks how many outstanding addresses there are. Amaximum of two addresses can be outstanding. Each time **SDONE** is asserted, the memory controller can allow another cacheable transaction to complete.

The only state transitions supported in write-through mode are:

STANDBY --> CLEAN --> [CLEAN] --> STANDBY

Since **SBO#** is not used in the write-through mode, it can be tied high by the system designer. Therefore, the only state transitions are between STANDBY and CLEAN. If the cache is the master of the second transaction, it may continue to signal CLEAN (as shown as an optional state) when the transaction is a writeback of a cacheline or knows the snoop is CLEAN; otherwise, it will transition to STANDBY. It is recommended that cacheable targets implement both **SDONE** and **SBO#**.

For each **FRAME#** that is asserted on the bus, the cache will assert **SDONE** when it has snooped the address. If two assertions of **FRAME#** occur without an assertion of **SDONE**, the second transaction cannot complete if cacheable. If the second access is cacheable, the memory controller must insert wait states until the previous snoop completes (**SDONE** asserted). If the second access is noncacheable, the access is complete and the cache will not snoop the address. At this point, only a single address is outstanding.

3.9.5. Arbitration Note

The arbiter is required to drop into some sort of fairness algorithm when HITM is indicated on the bus, otherwise, a livelock can occur. The livelock occurs when the cache that has the modified line is unable to perform the writeback because two higher priority agents are accessing cacheable memory. When HITM is signaled on the bus, all cacheable transactions are terminated with Retry.

It is recommended when a cache is present in a system, the arbiter may choose to connect its **REQ#** to a fixed input so its priority level can be raised when HITM is signaled on the bus. This insures that while the writeback is pending, the number of cacheable transactions that are terminated with Retry is kept to a minimum and latency is also minimized.

When a cache is used in the system (in particular a writeback cache), the latency attributed to the target must be increased to account for the time it takes for a writeback of a modified line. This value (addition) is dependent on the arbitration algorithm of when the writeback can access the bus.

3.10. 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 2.2.9. 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: A 64-bit data path is not required to do 64-bit addressing (refer to Section 3.10.1).

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 4.3.3. for information on pull-ups.

At the end of reset, the central resource controls the state of **REQ64#** to inform the 64-bit device that it is connected to a 64-bit bus. If **REQ64#** is deasserted when **RST#** is deasserted, the device is not connected to a 64-bit bus. If **REQ64#** is asserted when **RST#** is deasserted, the device is connected to a 64-bit bus. Refer to Section 4.3.2. for information on how a device behaves when **RST#** is deasserted.

During a 64-bit transaction, all PCI protocol and timing remain intact. Only Memory commands make sense when doing 64-bit data transfers. Interrupt Acknowledge and Special Cycle[27] commands are basically 32-bit transactions and must not be used with a **REQ64#**. The bandwidth requirements for I/O and Configuration commands cannot justify the added complexity and, therefore, only Memory commands support 64-bit data transfers.

---

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

All Memory commands and bus transfers are 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 (**AD2** 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 eight 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. 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 a 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.

PAR64 must be additionally qualified with **REQ64#** and **ACK64#** for data phases by the 64-bit parity checking device. **PAR64** is required for 64-bit data phases; it is not an option 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 3-31, is a read where the target responds with **ACK64#** asserted and the data is transferred in 64-bit data phases. The second, Figure 3-32, 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 Figures 3-1 and 3-2 except that 64-bit signals have been added and in Figure 3-31 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 bus command. **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 commands. **AD[63::32]** and **C/BE[7::4]#** are defined during the two address phases of a dual address command (DAC) and during the 64-bit data phases with a defined bus command (see Section 3.10.1 for details).

Figure 3-31 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.

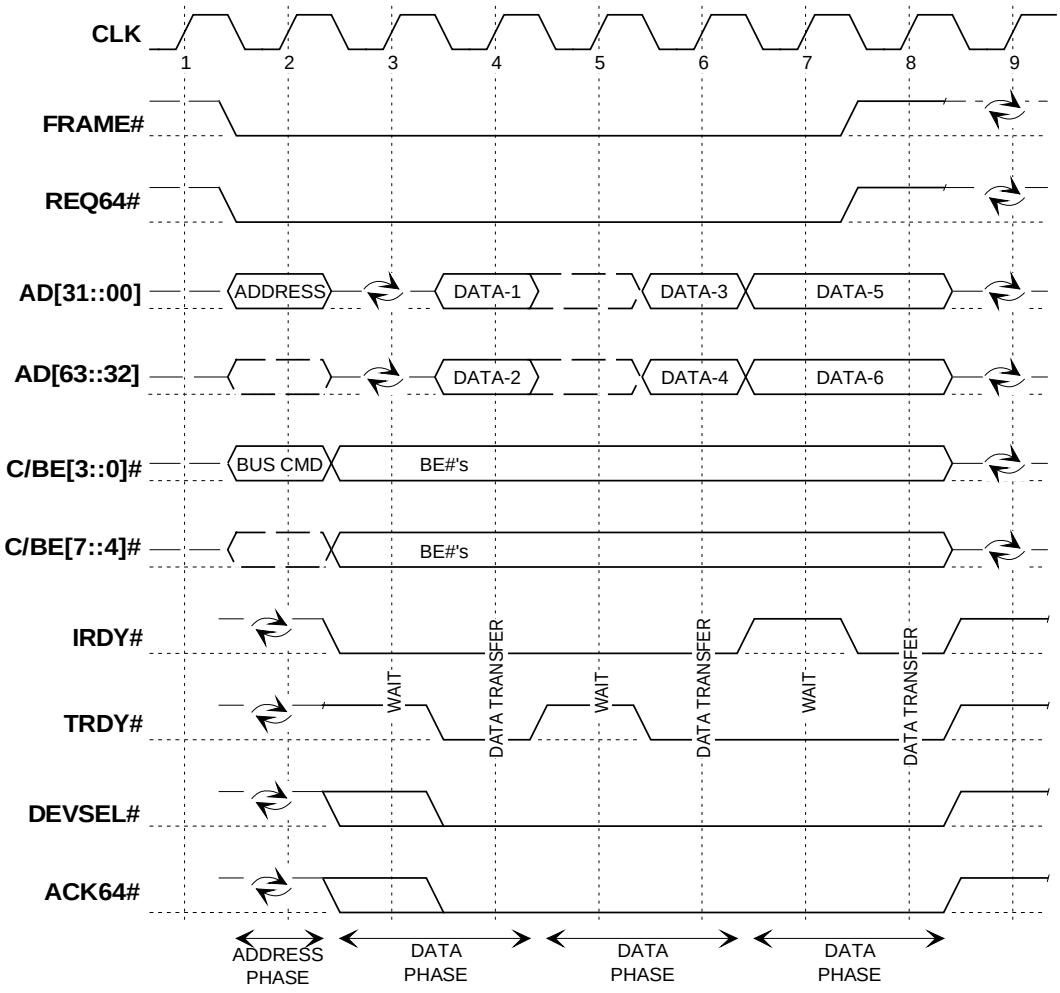


Figure 3-31: 64-bit Read Request with 64-bit Transfer

Figure 3-32 illustrates a master requesting a 64-bit transfer. The target does not comprehend **REQ64#** 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 you remove the 64-bit signals, Figure 3-32 and Figure 3-2 are identical.)

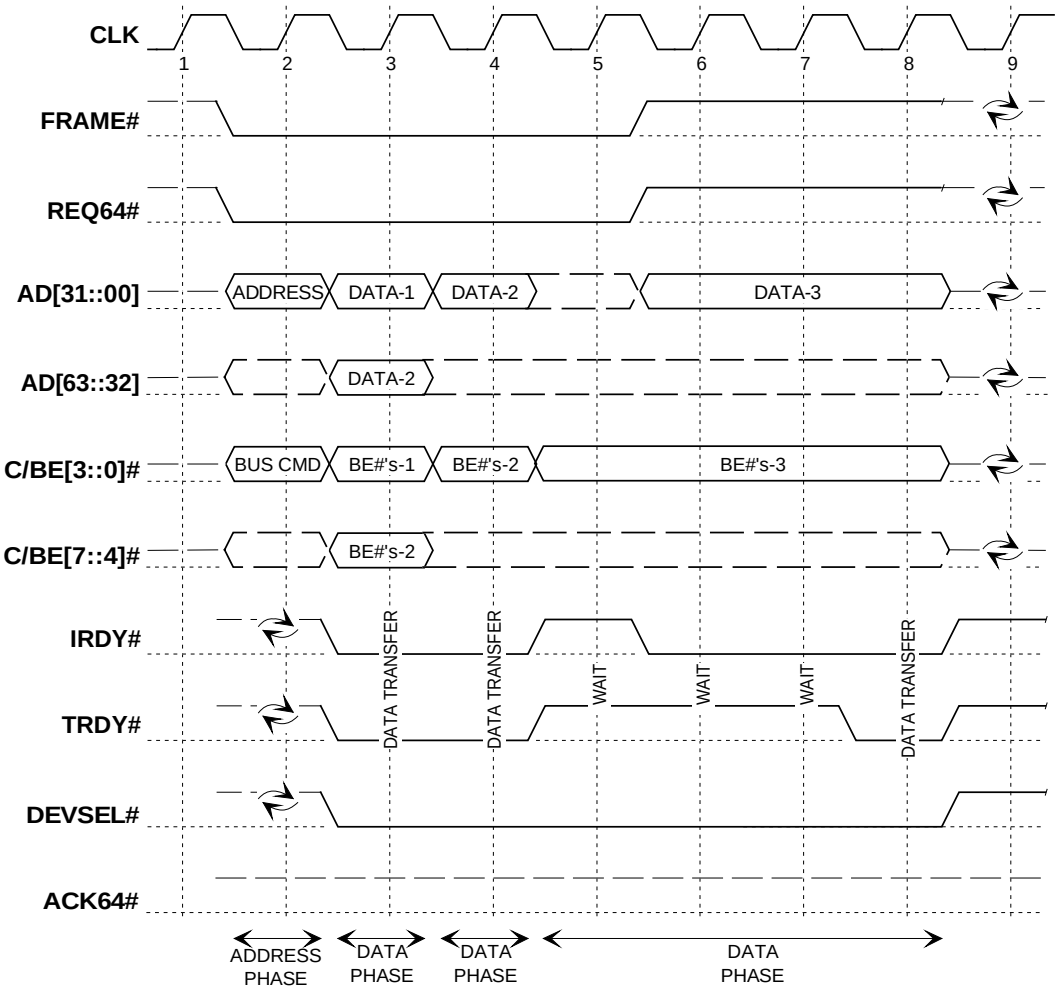


Figure 3-32: 64-bit Write Request with 32-bit Transfer

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.

3.10.1. 64-bit Addressing on PCI

PCI supports 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. A master may generate a 64-bit address independent of the 64-bit extensions. No additional pins are required for a 32- or 64-bit device to support 64-bit addressing. If both the master and target support a 64-bit data path, the entire 64-bit address can be provided in a single clock. Devices that support only 32-bit addresses will work transparently with devices that can generate 64-bit addresses when mapped into the low 4 GB of the address space.

The standard PCI bus data command 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. Masters that use address stepping cannot implement 64-bit addressing since there is no mechanism for delaying or extending the second address phase. 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 data 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 3-33 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 non bursting 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 the 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]#**. 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 which is one clock delayed from SAC decoding. There is no problem with this since a bridge implementing subtractive decode will ignore the entire transaction if it does not support 64-bit addressing. If the bridge does support 64-bit addressing, it will delay asserting its **DEVSEL#** by one clock. 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 (either SAC or DAC aware) 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.

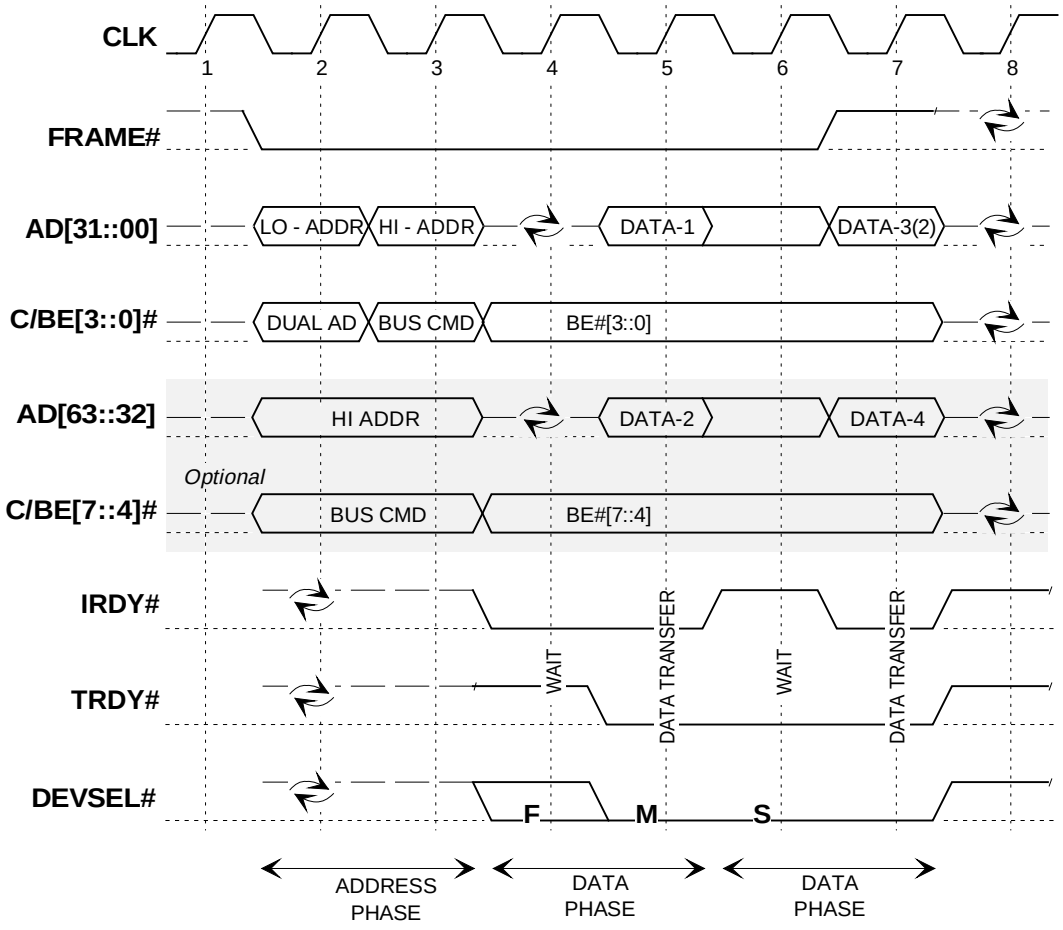


Figure 3-33. 64-Bit Dual Address Read Cycle

The shaded area in Figure 3-33 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 was supplied in the first address phase. The second address phase provides the additional time so that the target can perform a fast **DEVSEL#** decode for DAC commands and has no performance impact. 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 can generate 64-bit addresses to communicate with 32-bit addressable targets via SAC. The type of addressing (SAC or DAC) is determined by the address (greater than 4 GB), 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. A 32-bit master must

support 64-bit addressing using DAC to access targets mapped above the lower 4 GB address space.

3.11. 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 BIOS when it detects both a VGA device and a graphics accelerator device that are on separate boards 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, such as an ISA, EISA, or Micro Channel bridge, 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 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 a PCI add-in connector is supported, PCI-to-PCI bridges may be present in the system.)
2. A read access (that originates on PCI or a different bus) 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.)
3. Any device which blocks access as the target until it completes a read (or any other transaction as a master) will cause the deadlock.

The deadlock occurs when the following steps are met:

A burst read is initiated on PCI and only the first data phase completes. (This occurs because either the target or the bridge in the path terminates the request with Disconnect.)

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.

The agent 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 agent 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 three conditions are not met, the deadlock does not occur. If the system allows all three conditions to exist, then the agent initiating the read request must use **LOCK#** to guarantee that the 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 deserted when **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 a 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. 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 which was actually requested by the other master.

If no intervening write occurs between the two master's reads, 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.
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 master's 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.

Chapter 4
Electrical Specification

4.1. Overview

This chapter defines all the electrical characteristics and constraints of PCI components, systems, and expansion boards, including pin assignment on the expansion board connector. It is divided into major sections covering integrated circuit components (Section 4.2.), systems or motherboards (Section 4.3.), and expansion boards (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 5V and 3.3V signaling environments. These should not be confused with 5V and 3.3V component technologies. A "5V component" can be designed to work in a 3.3V 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 5V or 3.3V.

4.1.1. 5V to 3.3V Transition Road Map

One goal of the PCI electrical specification is to provide a quick and easy transition from 5V to 3.3V component technology. In order to facilitate this transition, PCI defines two expansion board connectors–one for the 5V signaling environment and one for the 3.3V signaling environment–and three board electrical types, as shown in Figure 4-1. A connector keying system prevents a board from being inserted into an inappropriate slot.

The motherboard (including connectors) defines the signaling environment for the bus, whether it be 5V or 3.3V. The 5V board is designed to work only in a 5V signaling environment and, therefore, can only be plugged into the 5V connector. Similarly, the 3.3V board is designed to work only in the 3.3V signaling environment. However, the Universal board is capable of detecting the signaling environment in use, and adapting itself to that environment. It can, therefore, be plugged into either connector type. All three board types define connections to both 5V and 3.3V power supplies, and may contain either 5V and/or 3.3V components. The distinction between board types is the signaling protocol they use, not the power rails they connect to nor the component technology they contain.

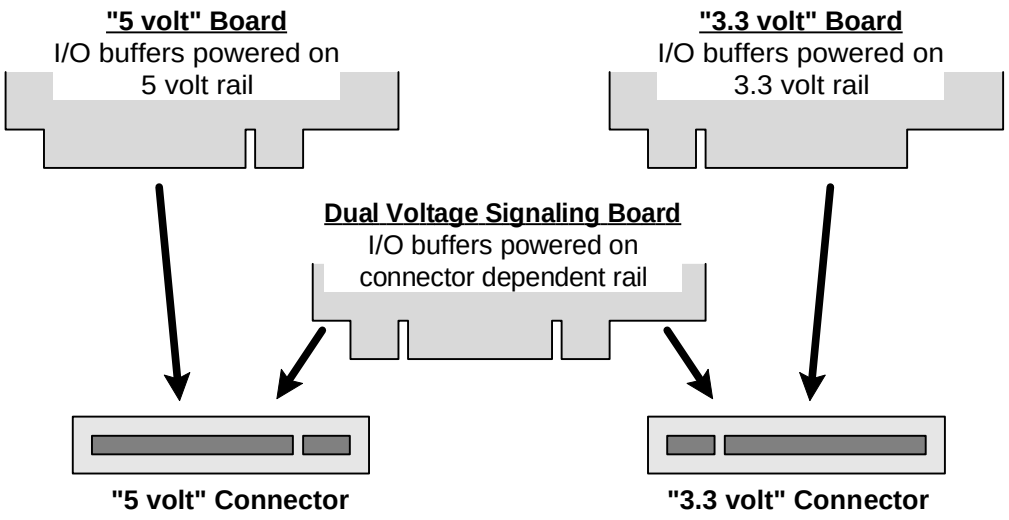


Figure 4-1: PCI Board Connectors

PCI components on the Universal board must use I/O buffers that can be compliant with either the 5V or 3.3V signaling environment. While there are multiple buffer implementations that can achieve this dual environment compliance, it is intended that they be dual voltage buffers - i.e., capable of operating from either power rail. They should be powered from "I/O" designated power pins[28] on PCI connectors that will always be connected to the power rail associated with the signaling environment in use. This means that in the 5V signaling environment, these buffers are powered on the 5V rail. When the same board is plugged into a 3.3V connector, these buffers are powered on the 3.3V rail. This enables the Universal board to be compliant with either signaling environment.

The intent of this transition approach is to move 5V component technology into the 3.3V signaling environment, rather than forcing 3.3V component technology to operate in a 5V signaling environment. While the latter can be done, it is more difficult and more expensive, especially in an unterminated, modular bus environment. The preferred alternative - moving 5V components into a 3.3V signaling environment - can be done without any incremental cost, and has, in addition, some signal performance benefits.

Since the first PCI components will have only 5V I/O buffers, the 5V board is initially necessary. However, all new component designs on 5V technology should use the dual voltage buffers (which will become available in most ASIC libraries) and should be designed into the Universal board. This allows expansions based on 5V component technology to be used in both 5V and 3.3V systems, thus enabling the move to 3.3V systems. By quickly getting to the point where most new PCI systems use the 3.3V signaling environment, expansion components moving to 3.3V technology for green

[28] While the primary goal of the PCI 5V to 3.3V transition strategy is to spare vendors the burden and expense of implementing 3.3V parts that are "5V tolerant," such parts are not excluded. If a PCI component of this type is used on the Universal Board, 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 must 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 extra "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).

machine or functional density reasons will be spared the cost and problems of 5V tolerance.

As shown in Figure 4-2, this results in a short term positioning using only the 5V board and 5V connector, and a long term positioning based on the 3.3V connector, with 5V components on the universal board (using dual voltage buffers) and 3.3V components on the 3.3V board. The transition between these uses primarily the universal board in both 5V and 3.3V connectors. The important step of this transition is moving quickly from the 5V board to the Universal board. If the critical mass of 5V add-in technology is capable of spanning 5V and 3.3V systems, it will not impede the move to the 3.3V connector. After this step of the transition is accomplished (to the Universal board), the rest of the transition should happen transparently, on a market-driven timetable.

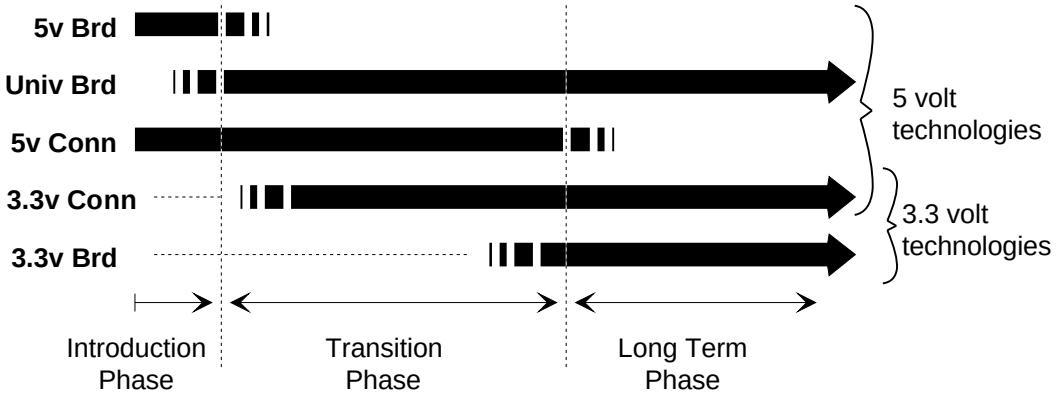


Figure 4-2: 5V and 3.3V Technology Phases

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 very 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 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 motherboard and two expansion connectors or two loads on the motherboard and four expansion connectors. However, it is possible to achieve different or larger configurations depending on the actual equipment practice, layout arrangement, loaded impedance of the motherboard, etc.

4.2. Component Specification

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

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[29] nor at 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.

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 Figures 4-3 and 4-5 define the allowable range for output characteristics.

It is possible to use weaker output drivers that do not comply with the V/I curves or meet the timing parameters in this chapter if they are set up to do continuous stepping as described in Section 3.7.3.. However, this practice is strongly discouraged as it creates violations of the input setup time at all inputs, as well as having significant negative performance impacts. In any case, all output drivers must meet the turn off (float) timing specification.

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.

---

[29] 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.

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	
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

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#**, 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**) with an exception granted to motherboard-only devices, which could be up to 16 pF, in order to accommodate PGA packaging. This would mean, in general, that components for expansion boards would need to use alternatives to ceramic PGA packaging (i.e., PQFP, SGA, etc.).
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.

The pins used for the extended data path, **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 motherboard central resource (not the expansion board) 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 card plugged into a 32-bit PCI slot), that PCI component is responsible to insure that its inputs do not oscillate and that there is not a significant power drain through the input buffer. This can be done in a variety of

ways: e.g., biasing the input buffer; or actively driving the outputs continually (since they are not connected to anything). External resistors on an expansion board or any solution that violates the input leakage specification are prohibited. The **REQ64#** signal is used during reset to distinguish between parts that are connected to a 64-bit data path and those that are not (refer to Section 4.3.2.).

4.2.1.2. AC Specifications

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

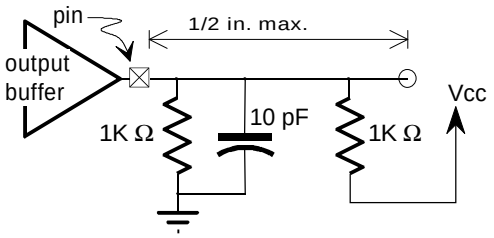
Table 4-2: AC Specifications for 5V Signaling

Symbol	Parameter	Condition	Min	Max	Units	Notes
$I_{oh(AC)}$	Switching	$0<V_{out}\leq 1.4$	-44		mA	1
	Current High	$1.4<V_{out}<2.4$	$-44+(V_{out}-1.4)/0.024$		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	$V_{out} \geq 2.2$	95		mA	1
	Current Low	$2.2>V_{out}>0.55$	$V_{out}/0.023$		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	0.4V to 2.4V load	1	5	V / ns	4
$slew_f$	Output Fall Slew Rate	2.4V to 0.4V load	1	5	V / ns	4

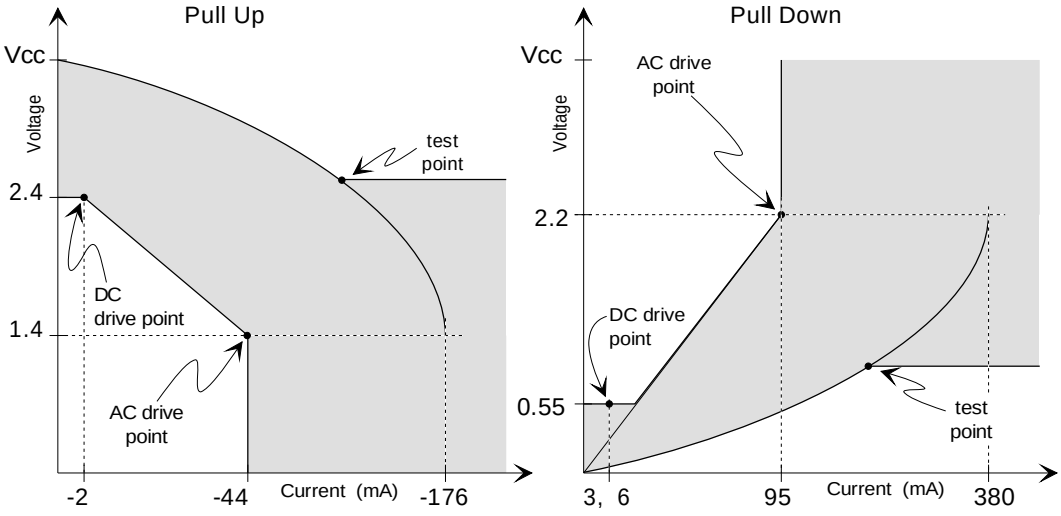
NOTES:

1. Refer to the V/I curves in Figure 4-3. 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#**, **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 4-3. The equation defined maxima 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 per revision 2.0 of the *PCI Local Bus Specification*. However, adherence to both maximum and minimum parameters is now required (the maximum is no longer simply a guideline). Since adherence to the maximum slew rate was not required prior to revision 2.1 of the specification, there may be components in the market for some time that have faster edge rates; therefore, motherboard designers must bear in mind that rise and fall times faster than this specification could occur, and should ensure that signal integrity modeling accounts for this. 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 max 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 22Ω load line.



Equation A:

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

for Vcc > Vout > 3.1v

Equation B:

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

for 0v < Vout < 0.71v

Figure 4-3: V/I Curves for 5V Signaling

Adherence to these curves should be evaluated at worst case conditions. The minimum pull up curve should be evaluated at minimum V_{CC} and high temperature. The minimum pull down curve should be evaluated at maximum V_{CC} and high temperature. The maximum curve test points should be 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 (see "Maximum AC Ratings" below). 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 spec 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 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, board impedance, etc., the "open circuit" voltage at the pins of PCI devices will exceed the ground to V_{CC} voltage range expected 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 should 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 4-4. The open circuit waveform is a composite of simulated worst cases[30]; 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.

[30] 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[31] 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, motherboard vendors may wish to use layout techniques to lower circuit impedance.

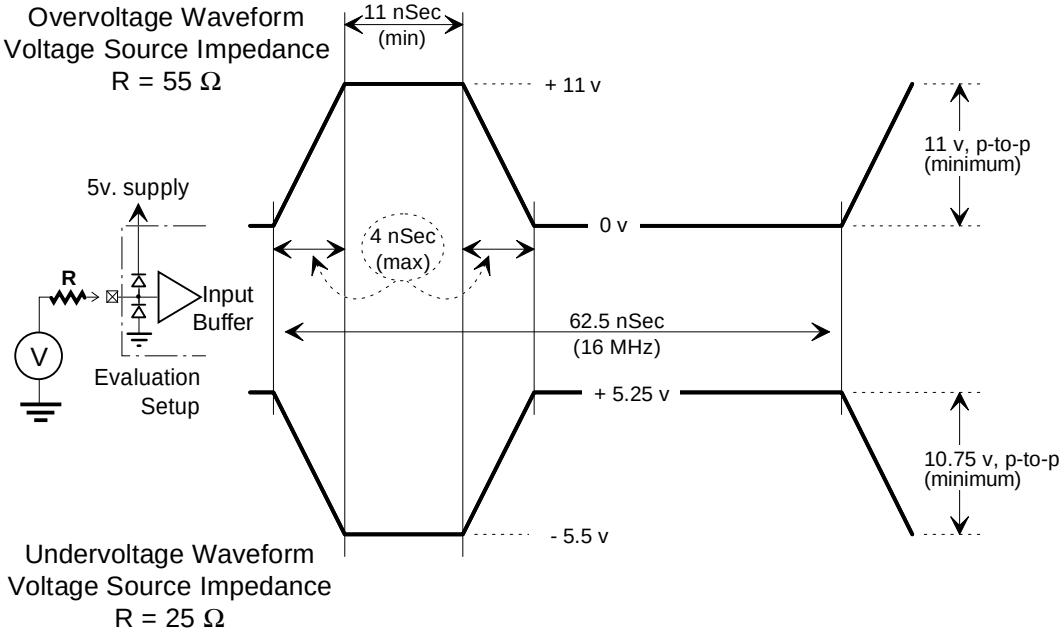


Figure 4-4: Maximum AC Waveforms for 5V Signaling

[31] 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	$0 < V_{in} < V_{cc}$		$\pm$10	µA	2
V_{oh}	Output High Voltage	I_{out} = -500 µA	$0.9V_{cc}$		V	
V_{ol}	Output Low Voltage	I_{out} = 1500 µA		$0.1V_{cc}$	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

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 should 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**) with an exception granted to motherboard-only devices, which could be up to 16 pF, in order to accommodate PGA packaging. This would mean in general that components for expansion boards would need to use alternatives to ceramic PGA packaging - i.e., PQFP, SGA, etc.
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.

The pins used for the extended data path, **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 motherboard central resource (not the expansion board) 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 card plugged into a 32-bit PCI slot), that PCI component is responsible to insure that its inputs do not oscillate and that there is not a significant power drain through the input buffer. This can be done in a variety of ways, e.g., biasing the input buffer or actively driving the outputs continually (since they

are not connected to anything). External resistors on an expansion board or any solution that violates the input leakage specification are prohibited. The **REQ64#** signal is used during reset to distinguish between parts that are connected to a 64-bit data path, and those that are not (refer to Section 4.3.2.).

4.2.2.2. AC Specifications

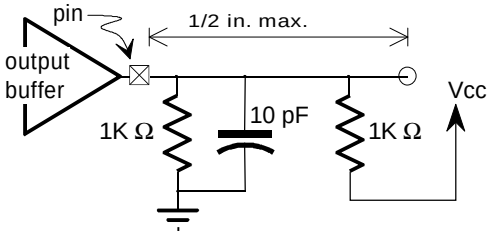
Table 4-4 summarizes the AC specifications for 3.3V signaling.

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

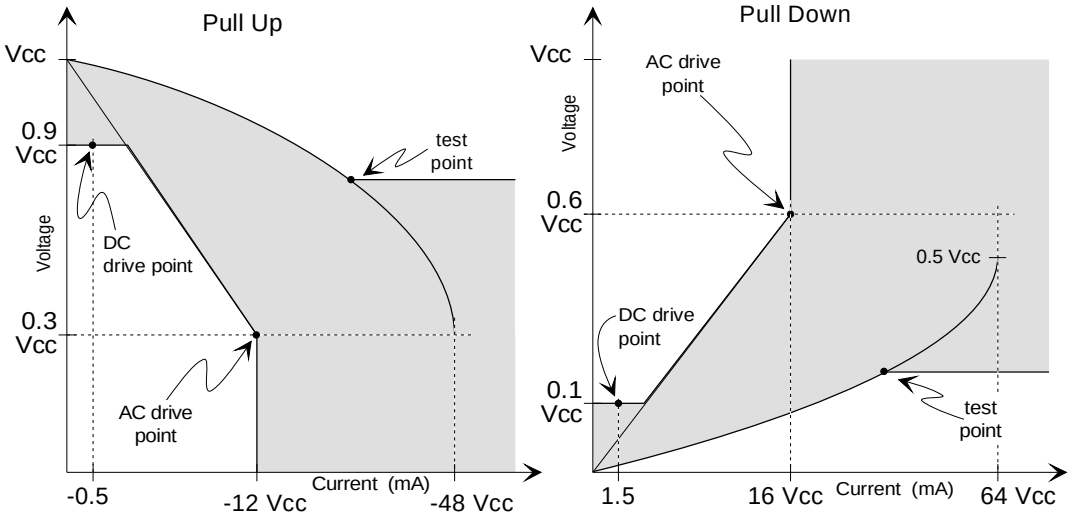
NOTES:

1. Refer to the V/I curves in Figure 4-5. 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#**, **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 4-5. The equation defined maxima 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 per the last revision of the PCI specification. 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 max 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 22Ω load line.

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



Equation C:

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

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

Equation D:

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

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

Figure 4-5: 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 spec 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 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 should 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 4-6. 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, motherboard vendors may wish to use layout techniques to lower circuit impedance.

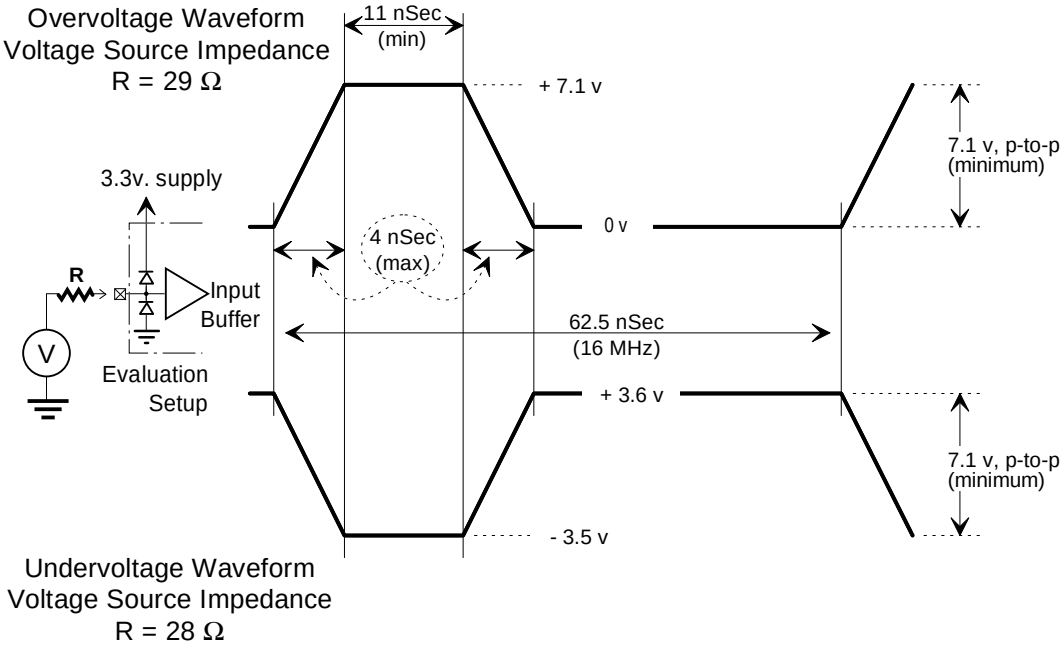


Figure 4-6: Maximum AC Waveforms for 3.3V Signaling

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 expansion boards, compliance with the clock specification is measured at the expansion board component, not at the connector slot. Figure 4-7 shows the clock waveform and required measurement points for both 5V and 3.3V signaling environments. Table 4-5 summarizes the clock specifications.

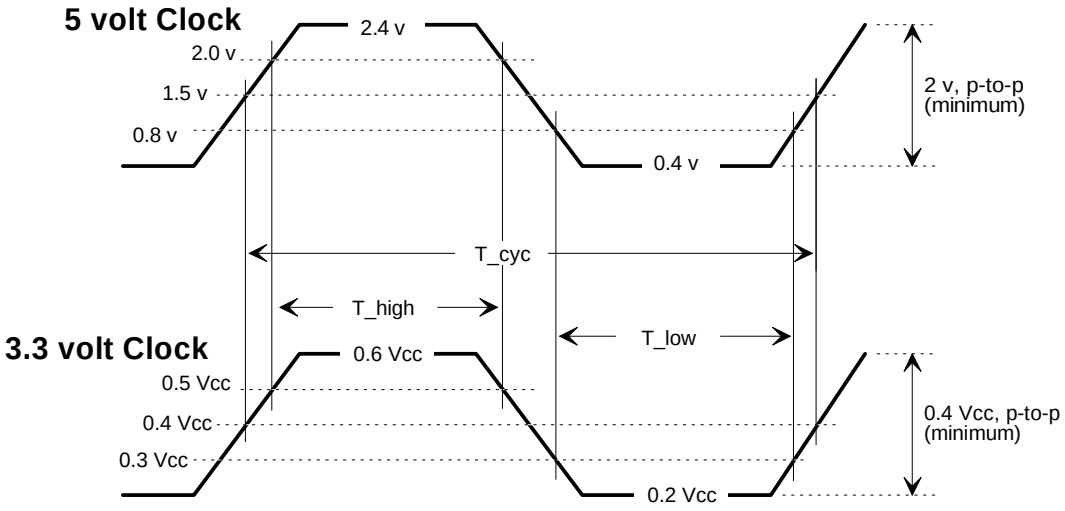


Figure 4-7: Clock Waveforms

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. 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 motherboard 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

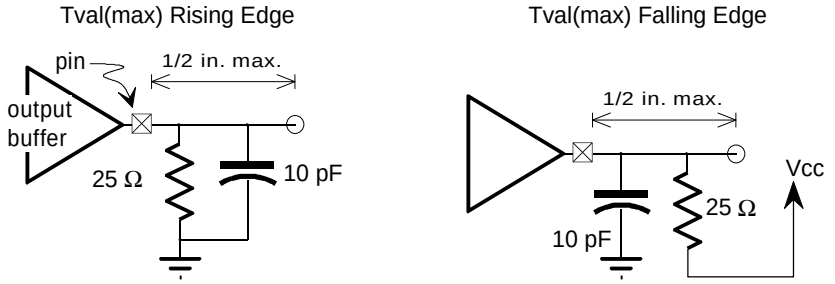
Table 4-6 provides the timing parameters for 5V and 3.3V signaling environments.

Table 4-6: 5V and 3.3V 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 Set up Time to **CLK** - bused signals	7		ns	3, 4
$t_{su}(ptp)$	Input Set up 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
$t_{rst-clk}$	Reset Active Time After **CLK** Stable	100		µs	5
$t_{rst-off}$	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	

NOTES:

1. See the timing measurement conditions in Figure 48.
2. 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 should 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.
   For parts compliant to the 3.3V signaling environment:
   Minimum times are evaluated with 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.



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 49.
5. **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.
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.

4.2.3.3. Measurement and Test Conditions

Figures 4-8 and 4-9 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.

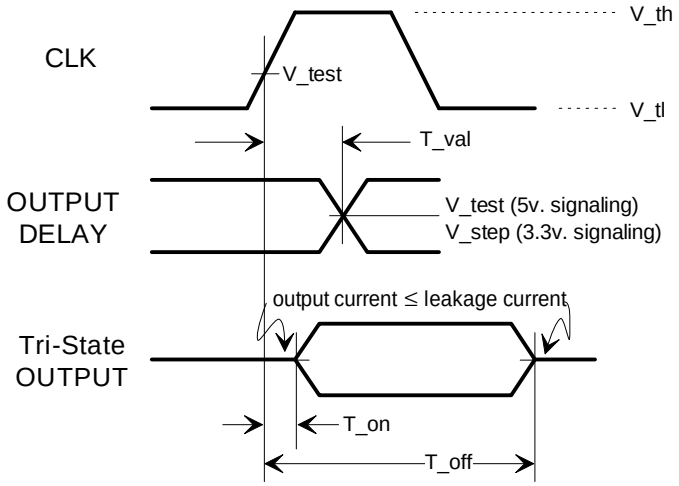


Figure 4-8: Output Timing Measurement Conditions

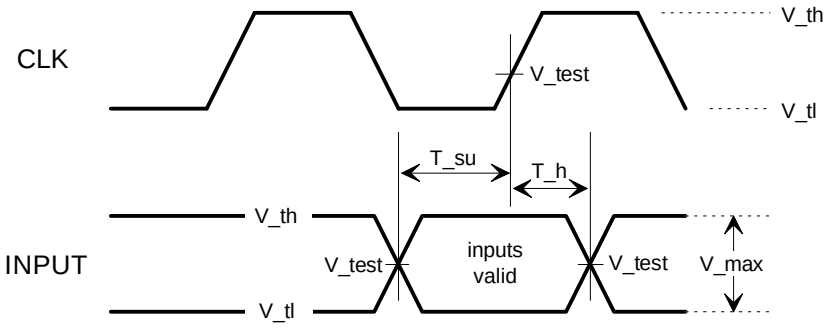


Figure 4-9: Input Timing Measurement Conditions

Table 4-7: Measure Condition Parameters

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

NOTE:

The input test for the 5V environment is done with 400 mV of overdrive (over V_{ih} and V_{il}); the test for the 3.3V environment is done with $0.1V_{cc}$ of overdrive. 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 should avoid logical operational errors and metastability by sampling inputs ONLY on "qualified" clock edges. In general, synchronous signals may only be assumed to be valid and determinate at the clock edge on which they are “qualified” (refer to Section 3.2.).

System designs must assure that floating inputs get 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 should 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 should 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 may 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 **SERR#** pin should be considered indeterminate for a number of cycles after it has 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 should eventually resolve high.

4.2.5. Vendor Provided Specification

In the time frame of PCI, many system vendors will do board-level electrical simulation of PCI components. This will ensure that system implementations are manufacturable and that components are used correctly. To help facilitate this effort, as well as provide complete information, component vendors should 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 -5V to 10V for 5V signaling and -3V to 7V for 3.3V 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 should make available accurate simulation models of PCI connectors.

4.2.6. Pinout Recommendation

This section provides a recommended pinout for PCI components. Since expansion board stubs are so limited, layout issues are greatly minimized if the component pinout aligns exactly with the board (connector) pinout. Components for use on motherboards only should also follow this same signal ordering, to allow layouts with minimum stubs. Figure 4-10 shows the recommended pinout for a typical PQFP PCI component. Note that the pinout is exactly aligned with the signal order on the board connector (**SDONE** and **SBO#** are not shown). Placement and number of power and ground pins is device-dependent.

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

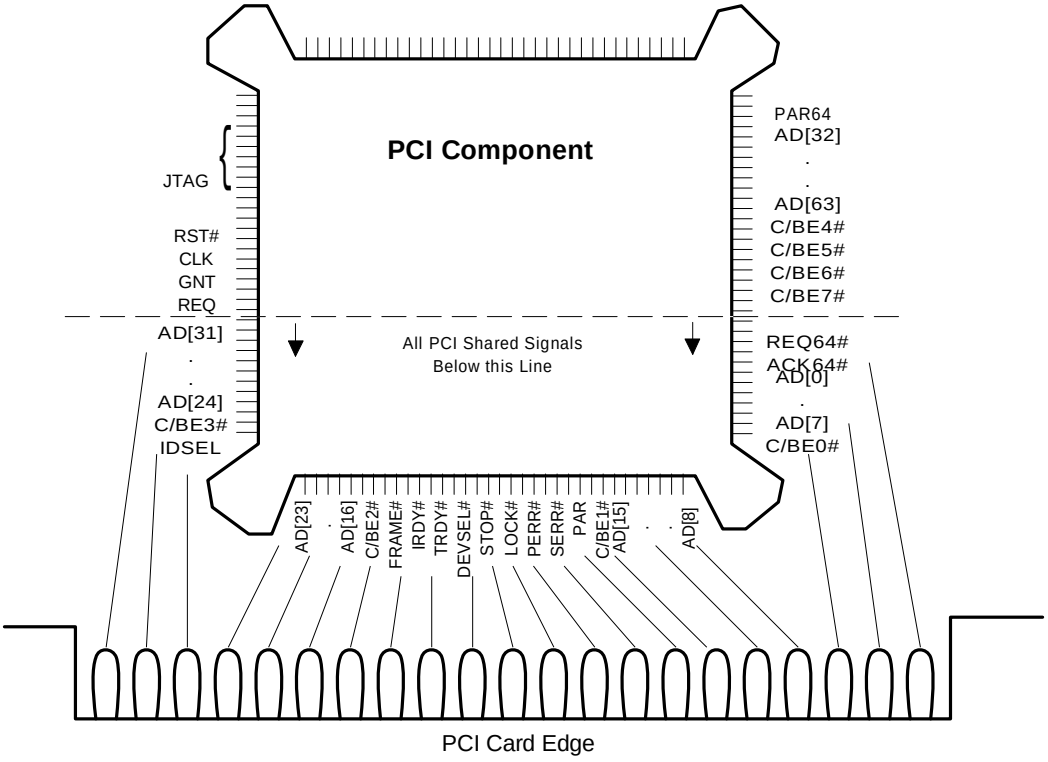


Figure 4-10: Suggested Pinout for PQFP PCI Component

Placing the **IDSEL** input as close as possible to **AD[31::11]** allows the option for a non-resistive[32] connection of **IDSEL** to the appropriate address line with a small additional load. Note that this pin has a lower capacitance specification that could constrain its placement in the package.

4.3. System (Motherboard) 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 4-11. The maximum skew is measured between any two components[33], not between connectors. To correctly evaluate clock skew, the system designer must take into account clock distribution on the expansion board, which is specified in Section 4.4..

---

[32] Non-resistive connections of **IDSEL** to one of the **AD[xx]** lines create a technical violation of the single load per add-in board 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.

[33] There may be an additional source of clock skew that the system designer may need to address. 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.

Table 4-8: Clock Skew Parameters

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

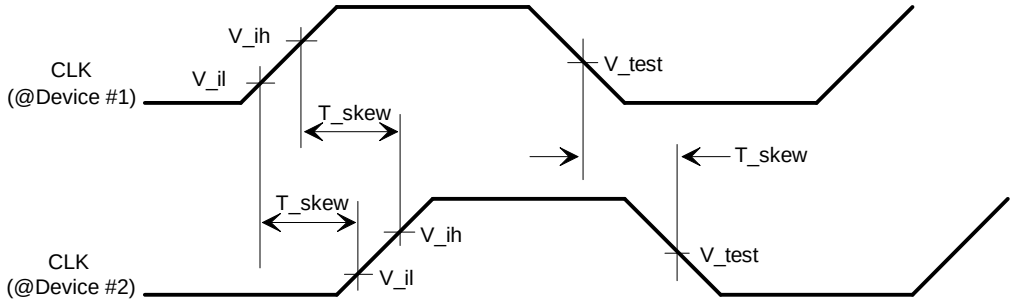


Figure 4-11: Clock Skew Diagram

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 contained 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 spec. If this occurs, parasitic diode paths could short circuit 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#** should be asserted as soon as possible during the power up sequence. Figure 4-12 shows a worst case assertion of **RST#** asynchronously following the "power good" signal.[34] 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. Figure 4-12 shows **RST#** signal timing.

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

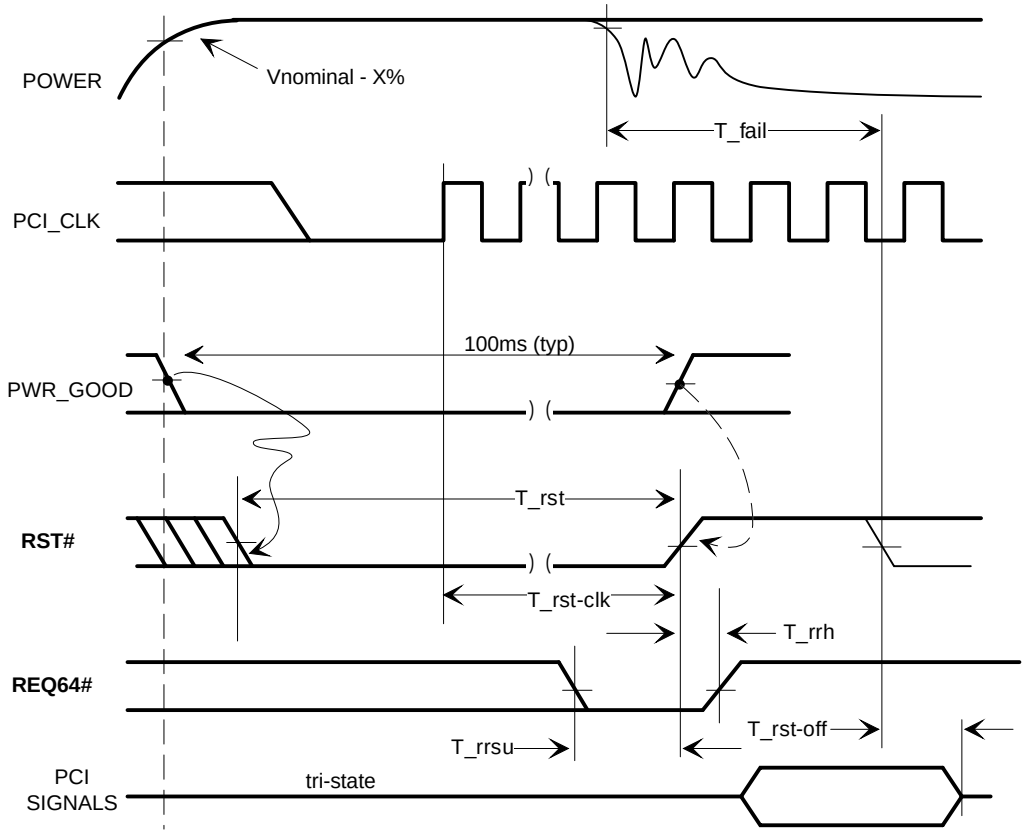


Figure 4-12: Reset Timing[35]

The **REQ64#** signal is used during reset to distinguish between parts that are connected to a 64-bit data path, and those that are not. The **REQ64#** signal is bused to all devices on the motherboard (including PCI connector slots) that support a 64-bit data path. This signal has a single pull-up resistor on the motherboard. On PCI expansion slots that do not support the 64-bit data path, the **REQ64#** signal is NOT bused or connected, but has its own, separate pull-up resistor. The central resource must drive **REQ64#** low (asserted) during the time that **RST#** is asserted, according to the timing specification. Devices that see **REQ64#** asserted during reset are connected to the 64-bit data path, while those that do not see the **REQ64#** assertion are not connected. This information may be used by the component to stabilize floating inputs during runtime, as described in Sections 4.2.1.1. and 4.2.2.1..

During reset, **REQ64#** has setup and hold time requirements with respect to the deasserting (high going) edge[36] of **RST#**. **REQ64#** is asynchronous with respect to the clock during reset.

[35] 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.

[36] This allows **REQ64#** to be sampled on the deassertion edge of **RST#**.

4.3.3. Pull-ups

PCI control signals always require pull-up resistors on the motherboard (NOT the expansion board) 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#**, and, when used, **REQ64#,** and **ACK64#**. The point-to-point and shared 32-bit signals do not require pull-ups; bus parking ensures their stability. The 64-bit data path expansion signals, **AD[63::32]**, **C/BE[7::4]#**, and **PAR64**, must also be pulled up when they are connected. When they are left unconnected (as with a 64-bit board in a 32-bit connector) the component itself must deal with the floating input as described in Sections 4.2.1.1. and 4.2.2.1..

In addition, on any connector for which they are not connected, **SBO#** and **SDONE** should be separately pulled up with an ~5 KΩ resistor. Also, if boundary scan is not implemented on the planar, **TMS** and **TDI** should be independently bused and pulled up each with ~5 KΩ resistors, and **TRST#** and **TCK** should be independently bused and pulled down, each with ~5 KΩ resistors. **TDO** should 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.

$$R_{\min} = [Vcc_{(\max)} - V_{ol}] / [I_{ol} + (16 \cdot I_{il})], \quad \text{where 16 = max number of loads}$$

$$R_{\max} = [Vcc_{(\min)} - Vx] / [num\_loads \times I_{ih}],$$

where: $V_x = 2.7v$ for 5V signaling, and $V_x = 0.7\,Vcc$ for 3.3V signaling.

Table 4-9 provides minimum and typical values for both 5V and 3.3V 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}
5V	963 Ω	2.7 KΩ @ 10%	see formula
3.3V	2.42 KΩ	8.2 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 add-in slot in order to insure that these signals do not float. Values for this pull-up shall be specified by the central resource vendor.

4.3.4. Power

4.3.4.1. Power Requirements

All PCI connectors require four power rails: +5V, +3.3V, +12V, and -12V. Systems implementing the 3.3V signaling environment are always required to provide all four rails in every system, with the current budget specified in Table 4-10. Systems implementing the 5V signaling environment may either ship the 3.3V supply with the system, or provide a means to add it afterward (i.e., bus and decouple all 3.3V power pins) to support expansion boards that require it, but must provide the other three power rails with each system. System vendors are nonetheless encouraged to provide 3.3V power for PCI slots since boards with 3.3V parts are expected to appear soon. However, in the 5V signaling environment, an expansion card may not currently depend on 3.3V power being already available in the system. If a board used in the 5V signaling environment requires 3.3V power, it must provide its own power from a source on or associated with the expansion board, such as a regulator powered by either the 5V or 12V supplies.

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 5V and 3.3V rails; this is system dependent. The system provides a total power budget for PCI expansion boards 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 board 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	Expansion Cards (Short and Long)
5V ±5%	5A max. (system dependent)
3.3V ±0.3V	7.6A max. (system dependent)
12V ±5%	500 mA
-12V ±10%	100 mA

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 5V or 3.3V rails go out of spec (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 in such a manner as to provide for reasonable management of the switching currents (*d* I/*d t*) to which the plane and its supply path are subjected. This is platform dependent and not detailed in the specification.

The +3.3V pins in PCI connectors (even if they are not actually delivering power) provide an AC return path and must be bused together on the motherboard, preferably on a 3.3V power plane, and decoupled to ground in a manner consistent with high speed signaling techniques. To ensure an adequate AC return path, it is recommended that 12 high-speed 0.01 μF capacitors be evenly spaced on the 3.3V plane.

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 expansion boards, as specified in Section 4.4.3.. Also, the maximum pin capacitance of 10 pF must be assumed for expansion boards, whereas the actual pin capacitance may be used for planar 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 systems (motherboards) only; all add-in board designs must assume 33 MHz operation.

In 5V signaling environments, T_{prop} is measured as shown in Figure 4-13. It begins at the time the output buffer would have crossed the threshold point (V_{test} in Figure 4-13), had it been driving the 50 pF lump load specified for T_{val} measurement. It ends when the slowest input (probably the closest one) reaches V_{th} and does not ring back across V_{ih} (high going) or reaches V_{tl} and does not ring back across V_{il} (low going). Note that input buffer timing is specified with a certain amount of overdrive (past V_{ih} and V_{il}), which may be needed to guarantee input buffer timings. This means the input is not valid (and consequently T_{prop} time is still running) unless it goes up to V_{th} and does not ring back across V_{ih}.

For 3.3V signaling environments (not shown in Figure 4-13), T_{prop} is measured in a similar way, except that the output buffer driving the 25Ω lump load specified for T_{val} measurement never moves beyond the first step voltage, and, therefore, never reaches the threshold point. In this case, the measurement of T_{prop} begins at the time the output buffer would have reached the T_{val} measurement specification (V_{step}). The end of the T_{prop} period is marked the same as in the 5V case.

Refer to Table 4-7 and the 5V and 3.3V DC parameter tables for the values of parameters in Figure 4-13.

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 PCI unpopulated 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.

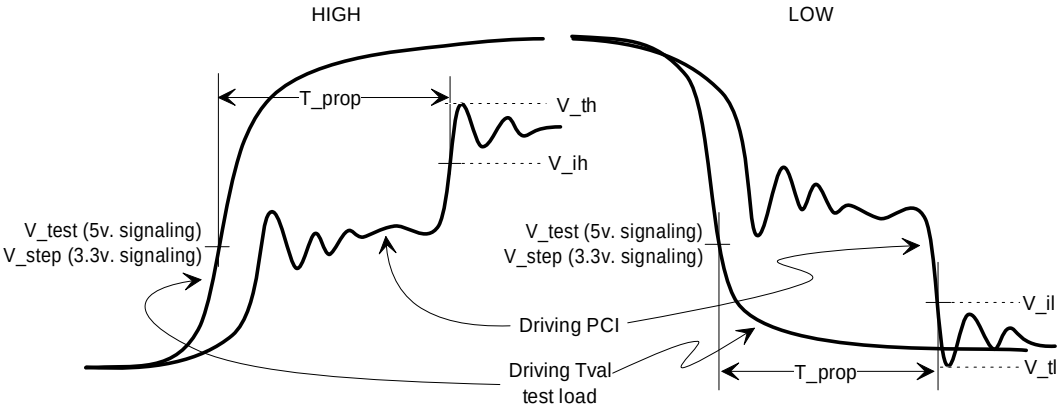


Figure 4-13: Measurement of T_{prop}

4.3.6. Physical Requirements

4.3.6.1. Routing and Layout of Four Layer Boards

The power pins have been arranged on the connector to facilitate layouts on four layer motherboards. A "split power plane" may be used - 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 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 that 0.25 inches from the point the signals cross the split.

This recommendation does not apply to slower speed signals such as ISA bus signals.

4.3.6.2. Motherboard Impedance

There is no bare board impedance specification for motherboards. 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 expansion boards.

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.. Motherboards 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 motherboard. Further use of these pins on the motherboard is optional. If the motherboard design accesses these pins to obtain board information, each pin must have an appropriate pull-up resistor (of approximately 5 KΩ) on the motherboard. The connector pin assignments are shown in Table 4-11. Pins labeled “Reserved” must be left unconnected on all connectors.

Pin B49 is a special purpose pin that has logical significance in 66 MHz capable systems, and in such 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 edge finger must be plated and connected to the ground plane.

Table 4-11: PCI Connector Pinout

Pin	5V System Environment		3.3V System Environment		Comments
	Side B	Side A	Side B	Side A	
1	-12V	TRST#	-12V	TRST#	32-bit connector 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	+5V (I/O)	Reserved	+3.3V (I/O)	
11	PRSNT2#	Reserved	PRSNT2#	Reserved	
12	Ground	Ground	CONNECTOR KEY		3.3 volt key
13	Ground	Ground	CONNECTOR KEY		3.3 volt key
14	Reserved	Reserved	Reserved	Reserved	
15	Ground	RST#	Ground	RST#	
16	CLK	+5V (I/O)	CLK	+3.3V (I/O)	
17	Ground	GNT#	Ground	GNT#	
18	REQ#	Ground	REQ#	Ground	
19	+5V (I/O)	Reserved	+3.3V (I/O)	Reserved	
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	Ground	STOP#	Ground	STOP#	
39	LOCK#	+3.3V	LOCK#	+3.3V	
40	PERR#	SDONE	PERR#	SDONE	
41	+3.3V	SBO#	+3.3V	SBO#	
42	SERR#	Ground	SERR#	Ground	

Table 4-11: PCI Connector Pinout *(continued)*

Pin	5V System Environment		3.3V System Environment		Comments
	Side B	Side A	Side B	Side A	
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	Ground	AD[09]	M66EN	AD[09]	66 MHz / gnd
50	CONNECTOR KEY		Ground	Ground	5 volt key
51	CONNECTOR KEY		Ground	Ground	5 volt 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	+5V (I/O)	+5V (I/O)	+3.3V (I/O)	+3.3V (I/O)	
60	ACK64#	REQ64#	ACK64#	REQ64#	
61	+5V	+5V	+5V	+5V	
62	+5V	+5V	+5V	+5V	32-bit connector end
	CONNECTOR KEY		CONNECTOR KEY		64-bit spacer
	CONNECTOR KEY		CONNECTOR KEY		64-bit spacer
63	Reserved	Ground	Reserved	Ground	64-bit connector 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]#	+5V (I/O)	C/BE[4]#	+3.3V (I/O)	
67	Ground	PAR64	Ground	PAR64	
68	AD[63]	AD[62]	AD[63]	AD[62]	
69	AD[61]	Ground	AD[61]	Ground	
70	+5V (I/O)	AD[60]	+3.3V (I/O)	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]	+5V (I/O)	AD[53]	+3.3V (I/O)	
76	Ground	AD[52]	Ground	AD[52]	
77	AD[51]	AD[50]	AD[51]	AD[50]	
78	AD[49]	Ground	AD[49]	Ground	
79	+5V (I/O)	AD[48]	+3.3V (I/O)	AD[48]	
80	AD[47]	AD[46]	AD[47]	AD[46]	
81	AD[45]	Ground	AD[45]	Ground	
82	Ground	AD[44]	Ground	AD[44]	

Table 4-11: PCI Connector Pinout *(continued)*

Pin	5V System Environment		3.3V System Environment		Comments
	Side B	Side A	Side B	Side A	
83	AD[43]	AD[42]	AD[43]	AD[42]	
84	AD[41]	+5V (I/O)	AD[41]	+3.3V (I/O)	
85	Ground	AD[40]	Ground	AD[40]	
86	AD[39]	AD[38]	AD[39]	AD[38]	
87	AD[37]	Ground	AD[37]	Ground	
88	+5V (I/O)	AD[36]	+3.3V (I/O)	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 connector end

Pins labeled "+5V (I/O)" and "+3.3V (I/O)" are special power pins for defining and driving the PCI signaling rail on the Universal Board. On the motherboard, these pins are connected to the main +5V or +3.3V plane, respectively.

Special attention must be paid to the connection of **REQ64#** and **ACK64#** on the motherboard. The associated pull-up resistors are placed on the motherboard (not the expansion board) to ensure that there is never more than a single pull-up of the appropriate value connected to either of these pins. The 64-bit connector extension is not always present on the motherboard, so these pins are located on the 32-bit section of the connector. These signals must be bused together (with a single pull-up resistor on each signal) on all connectors that provide the 64-bit data path. For 32-bit connectors, these signals must remain "open." They must not be connected together, and each must have its own pull-up resistor so that 64-bit boards plugged into 32-bit slots will operate correctly. For specifics of **REQ64#** during reset, refer to Section 4.3.2..

4.4. Expansion Board Specification

4.4.1. Board 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 a board is physically present in the slot and providing information about the total power requirements of the board. Table 4-12 defines the required setting of the **PRSNT#** pins for expansion boards.

Table 4-12: Present Signal Definitions

PRSNT1#	PRSNT2#	Expansion Configuration
Open	Open	No expansion board present
Ground	Open	Expansion board present, 25W maximum
Open	Ground	Expansion board present, 15W maximum
Ground	Ground	Expansion board present, 7.5W maximum

In providing a power level indication, the expansion board must indicate total maximum power consumption for the board. The system must assume that the expansion board could draw this power from either the 5V or 3.3V power rail. Furthermore, if the expansion board is configurable (e.g., sockets for memory expansion, etc.), the pin strapping must indicate the total power consumed by a fully configured board, which may be more than that consumed in its shipping configuration.

Boards 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 B49 is a special purpose pin that has logical significance in 66 MHz capable add-in boards, and in such it must be connected and decoupled as described in Section 7.8.. For all other add-in boards, 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 board.

Table 4-13: PCI Board Pinout

Pin	5V Board		Universal Board		3.3V Board		Comments
	Side B	Side A	Side B	Side A	Side B	Side A	
1	-12V	TRST#	-12V	TRST#	-12V	TRST#	32-bit start
2	TCK	+12V	TCK	+12V	TCK	+12V	
3	Ground	TMS	Ground	TMS	Ground	TMS	
4	TDO	TDI	TDO	TDI	TDO	TDI	
5	+5V	+5V	+5V	+5V	+5V	+5V	
6	+5V	INTA#	+5V	INTA#	+5V	INTA#	
7	INTB#	INTC#	INTB#	INTC#	INTB#	INTC#	
8	INTD#	+5V	INTD#	+5V	INTD#	+5V	
9	PRSNT1#	Reserved	PRSNT1#	Reserved	PRSNT1#	Reserved	
10	Reserved	+5V	Reserved	$+V_{I/O}$	Reserved	+3.3V	
11	PRSNT2#	Reserved	PRSNT2#	Reserved	PRSNT2#	Reserved	
12	Ground	Ground	KEYWAY		KEYWAY		3.3V key
13	Ground	Ground	KEYWAY		KEYWAY		3.3V key
14	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	
15	Ground	RST#	Ground	RST#	Ground	RST#	
16	CLK	+5V	CLK	$+V_{I/O}$	CLK	+3.3V	
17	Ground	GNT#	Ground	GNT#	Ground	GNT#	
18	REQ#	Ground	REQ#	Ground	REQ#	Ground	
19	+5V	Reserved	$+V_{I/O}$	Reserved	+3.3V	Reserved	
20	AD[31]	AD[30]	AD[31]	AD[30]	AD[31]	AD[30]	
21	AD[29]	+3.3V	AD[29]	+3.3V	AD[29]	+3.3V	
22	Ground	AD[28]	Ground	AD[28]	Ground	AD[28]	
23	AD[27]	AD[26]	AD[27]	AD[26]	AD[27]	AD[26]	
24	AD[25]	Ground	AD[25]	Ground	AD[25]	Ground	
25	+3.3V	AD[24]	+3.3V	AD[24]	+3.3V	AD[24]	
26	C/BE[3]#	IDSEL	C/BE[3]#	IDSEL	C/BE[3]#	IDSEL	
27	AD[23]	+3.3V	AD[23]	+3.3V	AD[23]	+3.3V	
28	Ground	AD[22]	Ground	AD[22]	Ground	AD[22]	
29	AD[21]	AD[20]	AD[21]	AD[20]	AD[21]	AD[20]	
30	AD[19]	Ground	AD[19]	Ground	AD[19]	Ground	
31	+3.3V	AD[18]	+3.3V	AD[18]	+3.3V	AD[18]	
32	AD[17]	AD[16]	AD[17]	AD[16]	AD[17]	AD[16]	
33	C/BE[2]#	+3.3V	C/BE[2]#	+3.3V	C/BE[2]#	+3.3V	
34	Ground	FRAME#	Ground	FRAME#	Ground	FRAME#	
35	IRDY#	Ground	IRDY#	Ground	IRDY#	Ground	
36	+3.3V	TRDY#	+3.3V	TRDY#	+3.3V	TRDY#	
37	DEVSEL#	Ground	DEVSEL#	Ground	DEVSEL#	Ground	
38	Ground	STOP#	Ground	STOP#	Ground	STOP#	
39	LOCK#	+3.3V	LOCK#	+3.3V	LOCK#	+3.3V	
40	PERR#	SDONE	PERR#	SDONE	PERR#	SDONE	
41	+3.3V	SBO#	+3.3V	SBO#	+3.3V	SBO#	
42	SERR#	Ground	SERR#	Ground	SERR#	Ground	

Table 4-13: PCI Board Pinout *(continued)*

Pin	5V Board		Universal Board		3.3V Board		Comments
	Side B	Side A	Side B	Side A	Side B	Side A	
43	+3.3V	PAR	+3.3V	PAR	+3.3V	PAR	
44	C/BE[1]#	AD[15]	C/BE[1]#	AD[15]	C/BE[1]#	AD[15]	
45	AD[14]	+3.3V	AD[14]	+3.3V	AD[14]	+3.3V	
46	Ground	AD[13]	Ground	AD[13]	Ground	AD[13]	
47	AD[12]	AD[11]	AD[12]	AD[11]	AD[12]	AD[11]	
48	AD[10]	Ground	AD[10]	Ground	AD[10]	Ground	
49	Ground	AD[09]	M66EN	AD[09]	M66EN	AD[09]	66 MHz /gnd
50	KEYWAY		KEYWAY		Ground	Ground	5V key
51	KEYWAY		KEYWAY		Ground	Ground	5V key
52	AD[08]	C/BE[0]#	AD[08]	C/BE[0]#	AD[08]	C/BE[0]#	
53	AD[07]	+3.3V	AD[07]	+3.3V	AD[07]	+3.3V	
54	+3.3V	AD[06]	+3.3V	AD[06]	+3.3V	AD[06]	
55	AD[05]	AD[04]	AD[05]	AD[04]	AD[05]	AD[04]	
56	AD[03]	Ground	AD[03]	Ground	AD[03]	Ground	
57	Ground	AD[02]	Ground	AD[02]	Ground	AD[02]	
58	AD[01]	AD[00]	AD[01]	AD[00]	AD[01]	AD[00]	
59	+5V	+5V	$+V_{I/O}$	$+V_{I/O}$	+3.3V	+3.3V	
60	ACK64#	REQ64#	ACK64#	REQ64#	ACK64#	REQ64#	
61	+5V	+5V	+5V	+5V	+5V	+5V	
62	+5V	+5V	+5V	+5V	+5V	+5V	32-bit end
	KEYWAY		KEYWAY		KEYWAY		64-bit spacer
	KEYWAY		KEYWAY		KEYWAY		64-bit spacer
63	Reserved	Ground	Reserved	Ground	Reserved	Ground	64-bit start
64	Ground	C/BE[7]#	Ground	C/BE[7]#	Ground	C/BE[7]#	
65	C/BE[6]#	C/BE[5]#	C/BE[6]#	C/BE[5]#	C/BE[6]#	C/BE[5]#	
66	C/BE[4]#	+5V	C/BE[4]#	$+V_{I/O}$	C/BE[4]#	+3.3V	
67	Ground	PAR64	Ground	PAR64	Ground	PAR64	
68	AD[63]	AD[62]	AD[63]	AD[62]	AD[63]	AD[62]	
69	AD[61]	Ground	AD[61]	Ground	AD[61]	Ground	
70	+5V	AD[60]	$+V_{I/O}$	AD[60]	+3.3V	AD[60]	
71	AD[59]	AD[58]	AD[59]	AD[58]	AD[59]	AD[58]	
72	AD[57]	Ground	AD[57]	Ground	AD[57]	Ground	
73	Ground	AD[56]	Ground	AD[56]	Ground	AD[56]	
74	AD[55]	AD[54]	AD[55]	AD[54]	AD[55]	AD[54]	
75	AD[53]	+5V	AD[53]	$+V_{I/O}$	AD[53]	+3.3V	
76	Ground	AD[52]	Ground	AD[52]	Ground	AD[52]	
77	AD[51]	AD[50]	AD[51]	AD[50]	AD[51]	AD[50]	
78	AD[49]	Ground	AD[49]	Ground	AD[49]	Ground	
79	+5V	AD[48]	$+V_{I/O}$	AD[48]	+3.3V	AD[48]	
80	AD[47]	AD[46]	AD[47]	AD[46]	AD[47]	AD[46]	
81	AD[45]	Ground	AD[45]	Ground	AD[45]	Ground	
82	Ground	AD[44]	Ground	AD[44]	Ground	AD[44]	

Table 4-13: PCI Board Pinout *(continued)*

Pin	5V Board		Universal Board		3.3V Board		Comments
	Side B	Side A	Side B	Side A	Side B	Side A	
83	AD[43]	AD[42]	AD[43]	AD[42]	AD[43]	AD[42]	
84	AD[41]	+5V	AD[41]	$+V_{I/O}$	AD[41]	+3.3V	
85	Ground	AD[40]	Ground	AD[40]	Ground	AD[40]	
86	AD[39]	AD[38]	AD[39]	AD[38]	AD[39]	AD[38]	
87	AD[37]	Ground	AD[37]	Ground	AD[37]	Ground	
88	+5V	AD[36]	$+V_{I/O}$	AD[36]	+3.3V	AD[36]	
89	AD[35]	AD[34]	AD[35]	AD[34]	AD[35]	AD[34]	
90	AD[33]	Ground	AD[33]	Ground	AD[33]	Ground	
91	Ground	AD[32]	Ground	AD[32]	Ground	AD[32]	
92	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	
93	Reserved	Ground	Reserved	Ground	Reserved	Ground	
94	Ground	Reserved	Ground	Reserved	Ground	Reserved	64-bit end

Table 4-14: Pin Summary - 32 bit Board

Pin Type	5V Board	Universal Board	3.3V Board
Ground	22	18 (Note)	22 (Note)
+5 V	13	8	8
+3.3 V	12	12	17
I/O pwr	0	5	0
Reserv'd	6	6	6

NOTE:

If the **M66EN** pin is implemented, the number of ground pins for a Universal board is 17 and the number of ground pins for a 3.3V board is 21.

Table 4-15: Pin Summary - 64 bit Board (incremental pins)

Pin Type	5V Board	Universal Board	3.3V Board
Ground	16	16	16
+5 V	6	0	0
+3.3 V	0	0	6
I/O pwr	0	6	0
Reserv'd	5	5	5

Pins labeled "+V I/O" are special power pins for defining and driving the PCI signaling rail on the Universal Board. On this board, the PCI component's I/O buffers must be powered from these special power pins only[37] -- not from the other +5V or +3.3V power pins.

4.4.2. Power Requirements

4.4.2.1. Decoupling

Under typical conditions, the V_{CC} plane to ground plane capacitance will provide adequate decoupling for the V_{CC} connector pins. The maximum trace length from a connector pad to the V_{CC}/GND plane via shall be 0.25 inches (assumes a 20 mil trace width).

However, on the Universal board, it is likely that the I/O buffer power rail will not have adequate capacitance to the ground plane to provide the necessary decoupling. Pins labeled "+V I/O" should be decoupled to ground with an average of 0.047 μF per pin.

Additionally, all +3.3V pins (even if they are not actually delivering power), and any unused +5V and V I/O pins on the PCI edge connector provide an AC return path, and must have plated edge fingers and be coupled to the ground plane on the add-in board as described below to ensure they continue to function as efficient AC reference points:

1. The decoupling must average at least 0.01 μF (high-speed) per V_{cc} pin.
2. The trace length from pin pad to capacitor pad shall be no greater than 0.25 inches using a trace width of at least 0.02 inches.
3. There is no limit to the number of pins that can share the same capacitor provided that requirements 1 and 2 are met.

4.4.2.2. Power Consumption

The maximum power allowed for any PCI board is 25 watts, and represents the total power drawn from all of the four power rails provided at the connector. In the worst case, all 25 watts could be drawn from either the +5V or +3.3V rail.

It is anticipated that many systems will not provide a full 25 watt per connector power budget for each power rail, because most boards will typically draw much less than this amount. For this reason, PCI boards that consume more than 10 watts should power up in and reset to a power saving state that consumes 10 watts or less, if possible. While in this state, the board must provide full access to its PCI configuration space, and must

[37] While the primary goal of the PCI 5V to 3.3V transition strategy is to spare vendors the burden and expense of implementing 3.3V parts that are "5V tolerant," such parts are not excluded. If a PCI component of this type is used on the Universal Board, 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 must 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 extra "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).

perform required bootstrap functions, such as basic text mode on a video board. All other board functions can be suspended if necessary. This power saving state can be achieved in a variety of ways. For example:

- Clock rates on the board can be reduced, which reduces performance but does not limit functionality.
- Power planes to non-critical parts could be shut off with an FET, which could limit functional capability.

After the driver for the board has been initialized, it may place the board 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 board in order to allow the system to determine if it has a sufficient power budget for all boards in the current configuration. The driver must be able to accurately determine the maximum power requirements for its board as currently configured and from which rail(s) this power will be drawn.

Add-in boards must never source 3.3V power back to the system board, except in the case where an add-in board has been specifically designed to provide a given system's 3.3V. power. Boards capable of 3.3V PCI signaling may have multiple mechanisms that indirectly source power back to the system. For example, boards 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 3.3V supply (bleed resistor or otherwise) to dissipate any power "generated" on the add-in board. This requirement does not apply to noise generated on the 3.3V 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 option card 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 cards. This includes all signal groups (refer to Section 2.2.) except those listed as, "System Pins," "Interrupt Pins," and "JTAG Pins."
- The trace lengths of the additional signals used in the 64-bit extension are limited to 2 inches on all 64-bit cards.
- The trace length for the PCI **CLK** signal is 2.5 inches ± 0.1 inches for 32-bit and 64-bit cards and must be routed to only one load.

4.4.3.2. Routing

The power pins have been arranged on the connector to facilitate layouts on four layer boards. A "split power plane" may be used, 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 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 that 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 expansion card shall be controlled to be in the 60Ω -100Ω range. 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 expansion card. Violation of expansion board trace length or loading limits will compromise system signal integrity. It is specifically a violation of this specification for expansion boards to:

- Attach an expansion ROM directly (or via bus transceivers) on any PCI pins.
- Attach two or more PCI devices on an expansion board, 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.

Chapter 5
Mechanical Specification

5.1. Overview

The PCI expansion card is based on a raw card design (see Figures 5-1 to 5-6) that is easily implemented in existing cover designs from multiple manufacturers. The card design adapts to ISA, EISA, and MC systems. PCI expansion cards have two basic form factors: standard length and short length. 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 expansion card has been defined for both the 32-bit and 64-bit interfaces.

PCI cards and connectors are keyed to manage the 5V to 3.3V transition. The basic 32-bit connector contains 120 pins. The logical numbering of pins shows 124 pin identification numbers, but four pins are not present and are replaced by the keying location. In one orientation, the connector is keyed to accept 5V system signaling environment boards; turned 180 degrees the key is located to accept 3.3V system signaling environment boards. Universal add-in cards, cards built to work in both 5V and 3.3V system signaling environments, have two key slots so that they can plug into either connector. 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 cards and 64-bit cards are inter-operable within the system's signaling voltage classes defined by the keying in the 32-bit connector subset. A 32-bit card identifies itself for 32-bit transfers on the 64-bit connector. A 64-bit card in a 32-bit connector must configure for 32-bit transfers.

Maximum card power dissipation is encoded on the **PRSNT1#** and **PRSNT2#** pins of the expansion 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, "Electrical Specification."

The PCI expansion card includes a mounting bracket for card location and retention. The backplate is the interface between the card and the system that provides for cable escapement. The card has been designed to accommodate PCI brackets for both ISA/EISA and MC systems. See Figures 5-8 and 5-9 for the ISA/EISA assemblies and Figures 5-10 and 5-11 for the MC assemblies. Bracket kits for each card type must be furnished with the PCI card so that end users may configure the card for their systems.

The ISA/EISA kit contains a PCI bracket, four 4-40 screws, and a card extender. The assembled length of the PCI expansion board is that of an MC card. The extender fastens to the front edge of the PCI card to provide support via a standard ISA card guide. The MC bracket kit contains a riveted MC bracket-bracket brace assembly and two pan head 4-40 screws.

The component side of a PCI expansion card is the opposite of ISA/EISA and MC cards. The PCI card is a mirror image of ISA/EISA and MC cards. A goal of PCI is to enable its implementation in systems with a limited number of expansion card slots. In these systems, the PCI expansion board connector can coexist, within a single slot, with an ISA/EISA or MC expansion connector. These slots are referred to as *shared slots*. Shared slots allow the end user to install a PCI, ISA/EISA, or MC card. However, only one expansion board can be installed in a shared slot at a time. For example, shared slots in PCI systems with an ISA expansion bus can accommodate an ISA or a PCI expansion board; shared slots in PCI systems with an MC expansion bus can accommodate an MC or a PCI expansion board; etc.

5.2. Expansion Card Physical Dimensions and Tolerances

The maximum component height on the primary component side of the PCI expansion card is not to exceed 0.570 inches (14.48 mm). The maximum component height on the back side of the card is not to exceed 0.105 inches (2.67 mm). Datum A on the illustrations is used to locate the PCI card to the planar and to the frame interfaces; the back of the frame and the 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 5-16 for PCI expansion card physical dimensions.

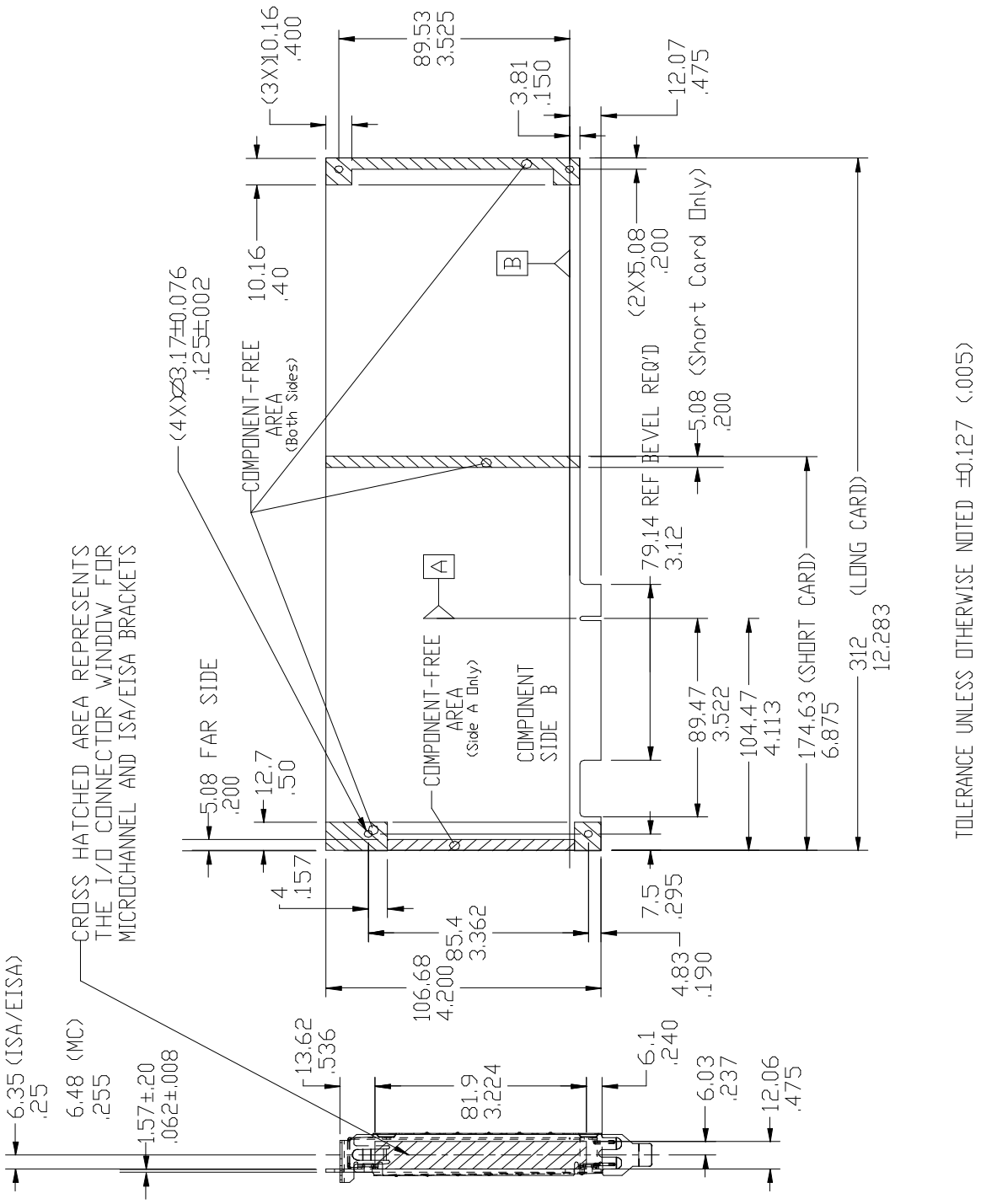


Figure 5-1: PCI Raw Card (5V)

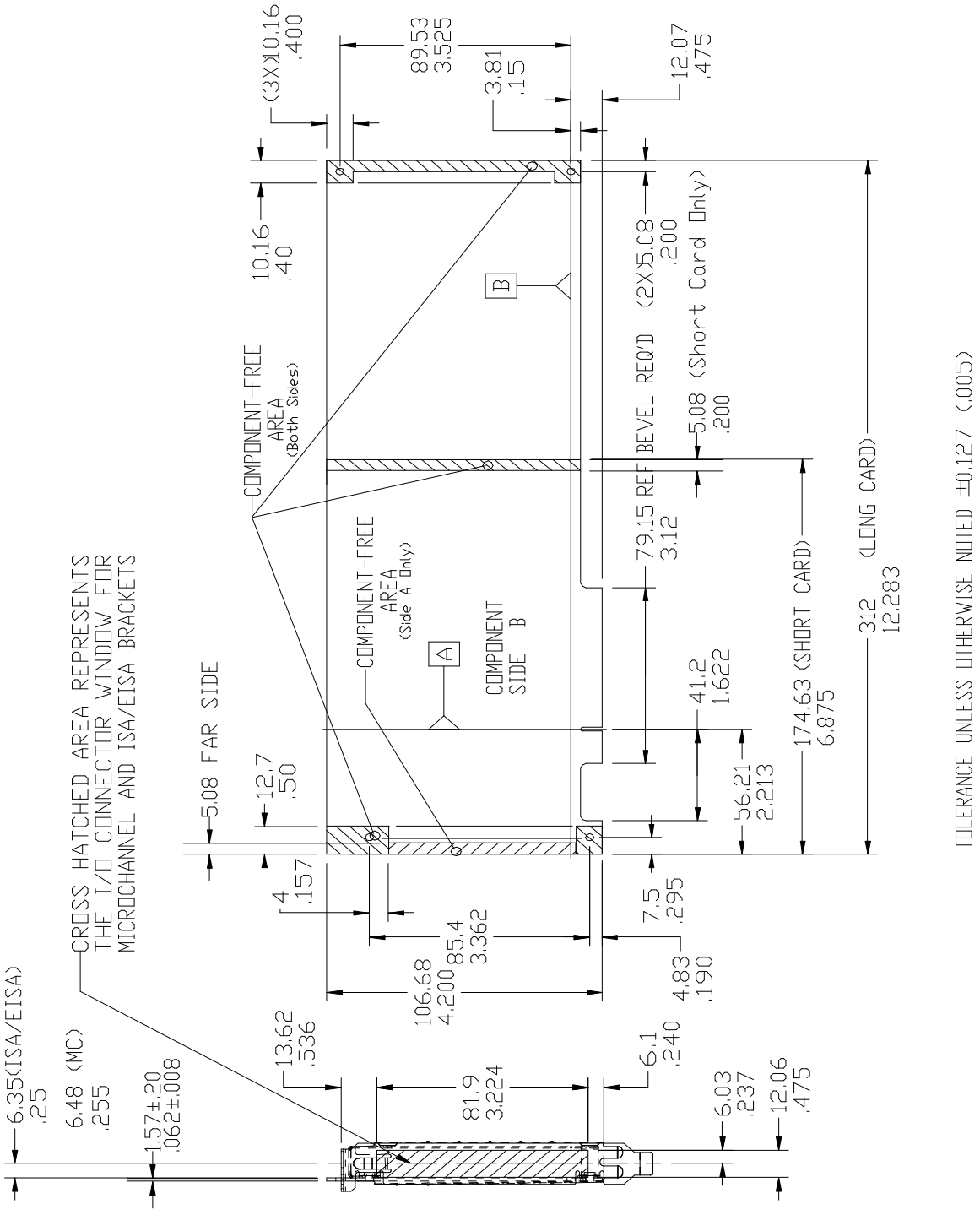
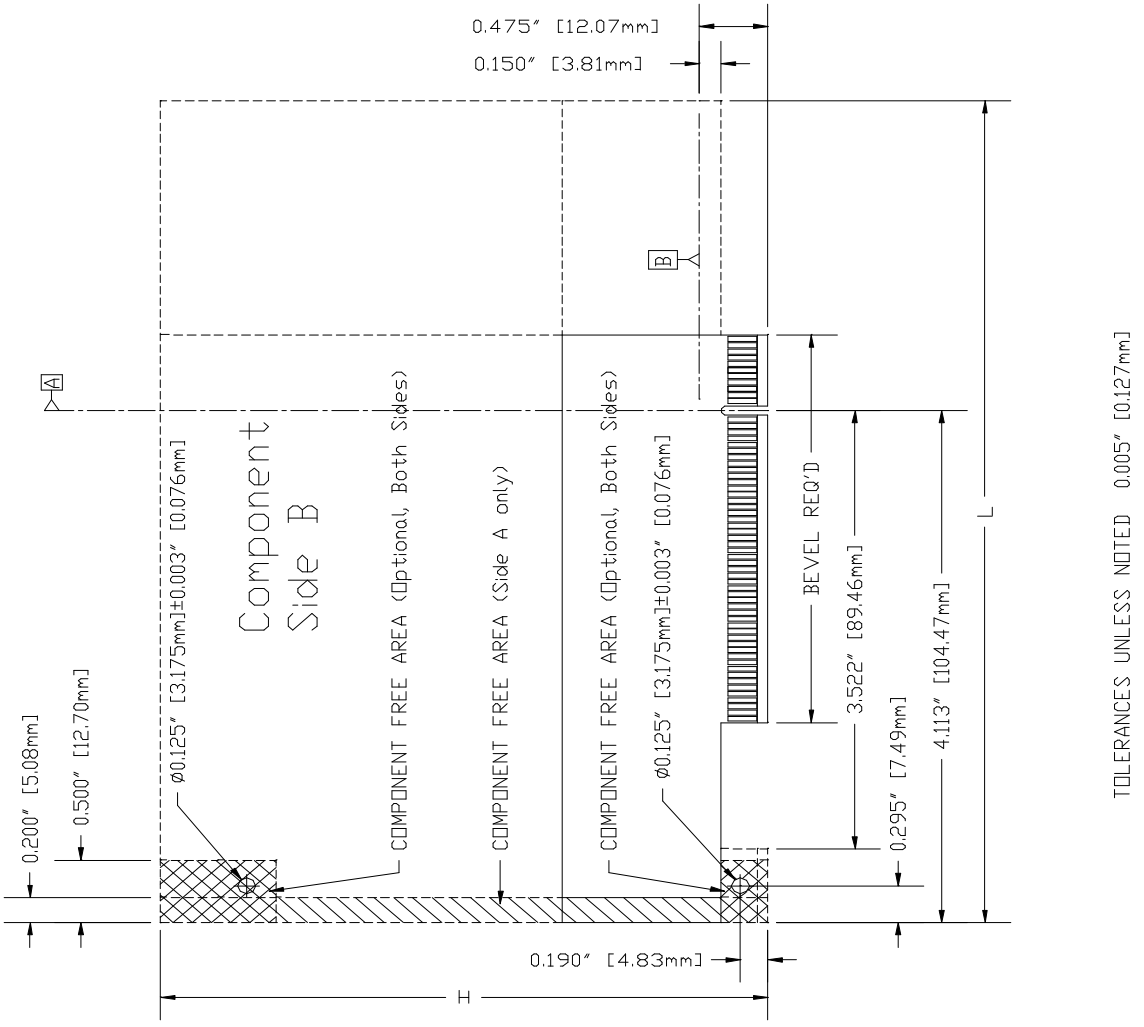
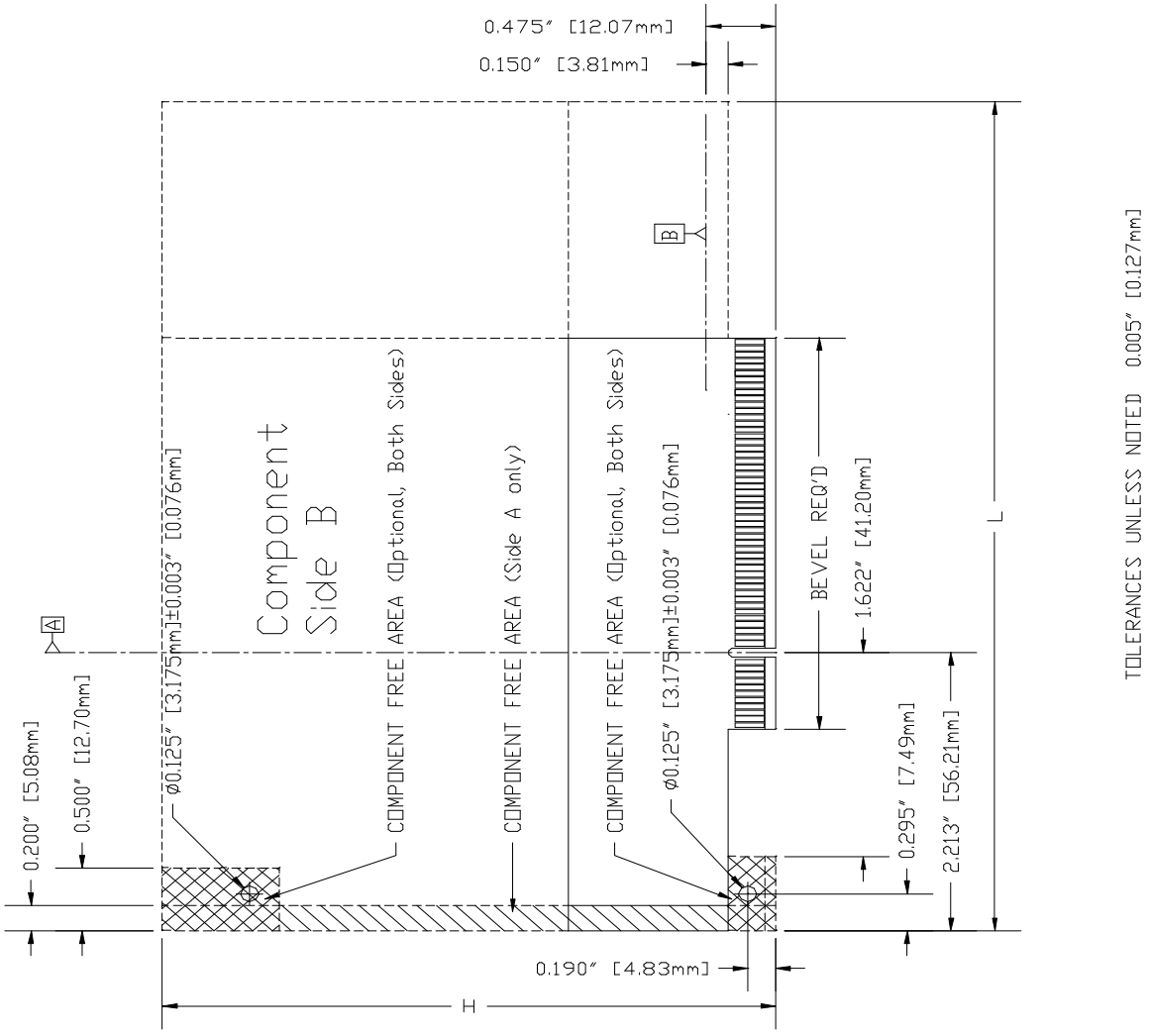


Figure 5-2: PCI Raw Card (3.3V and Universal)



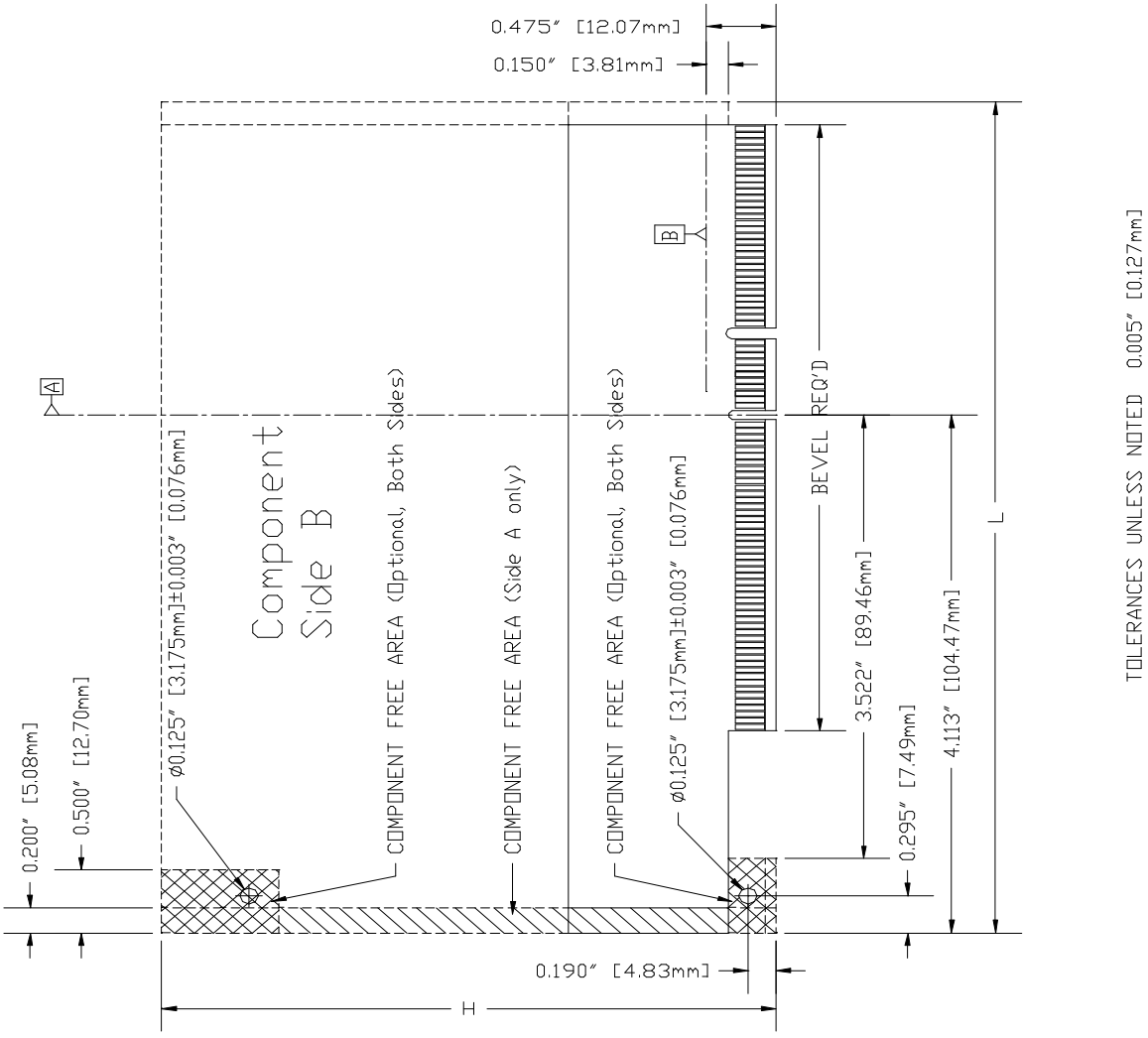
	Min.	Max.
L	4.721″ [119.91mm]	6.6″ [167.64mm]
H	1.42″ [36.07mm]	4.2″ [106.68mm]

Figure 5-3: PCI Raw Variable Height Short Card (5V, 32-bit)



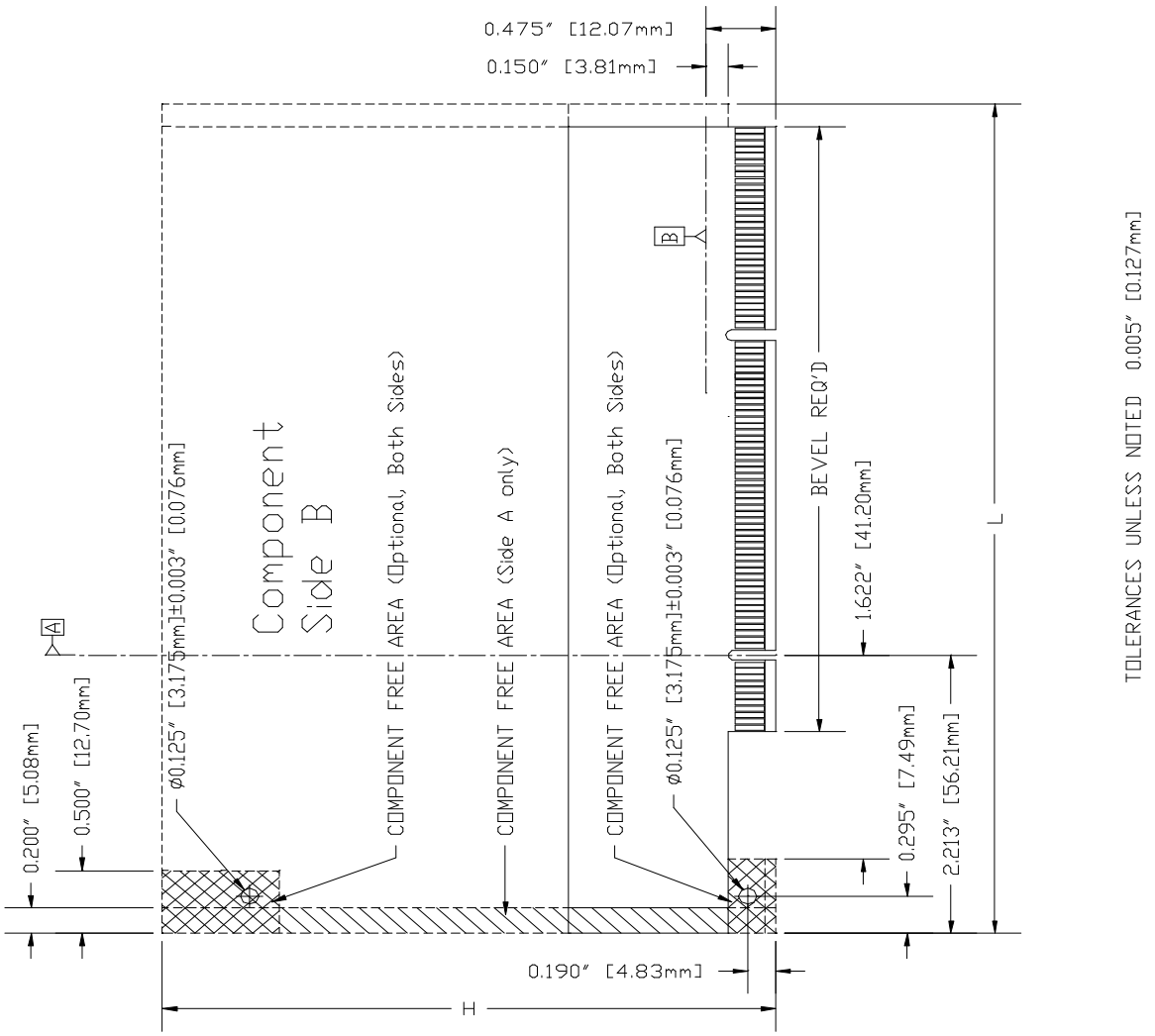
	Min.	Max.
L	4.721″ [119.91mm]	6.6″ [167.64mm]
H	1.42″ [36.07mm]	4.2″ [106.68mm]

Figure 5-4: PCI Raw Variable Height Short Card (3.3V, 32-bit)



	Min.	Max.
L	6.421″ [163.09mm]	6.6″ [167.64mm]
H	1.42″ [36.07mm]	4.2″ [106.68mm]

Figure 5-5: PCI Raw Variable Height Short Card (5V, 64-bit))



	Min.	Max.
L	6.421″ [163.09mm]	6.6″ [167.64mm]
H	1.42″ [36.07mm]	4.2″ [106.68mm]

Figure 5-6: PCI Raw Variable Height Short Card (3.3V, 64-bit)

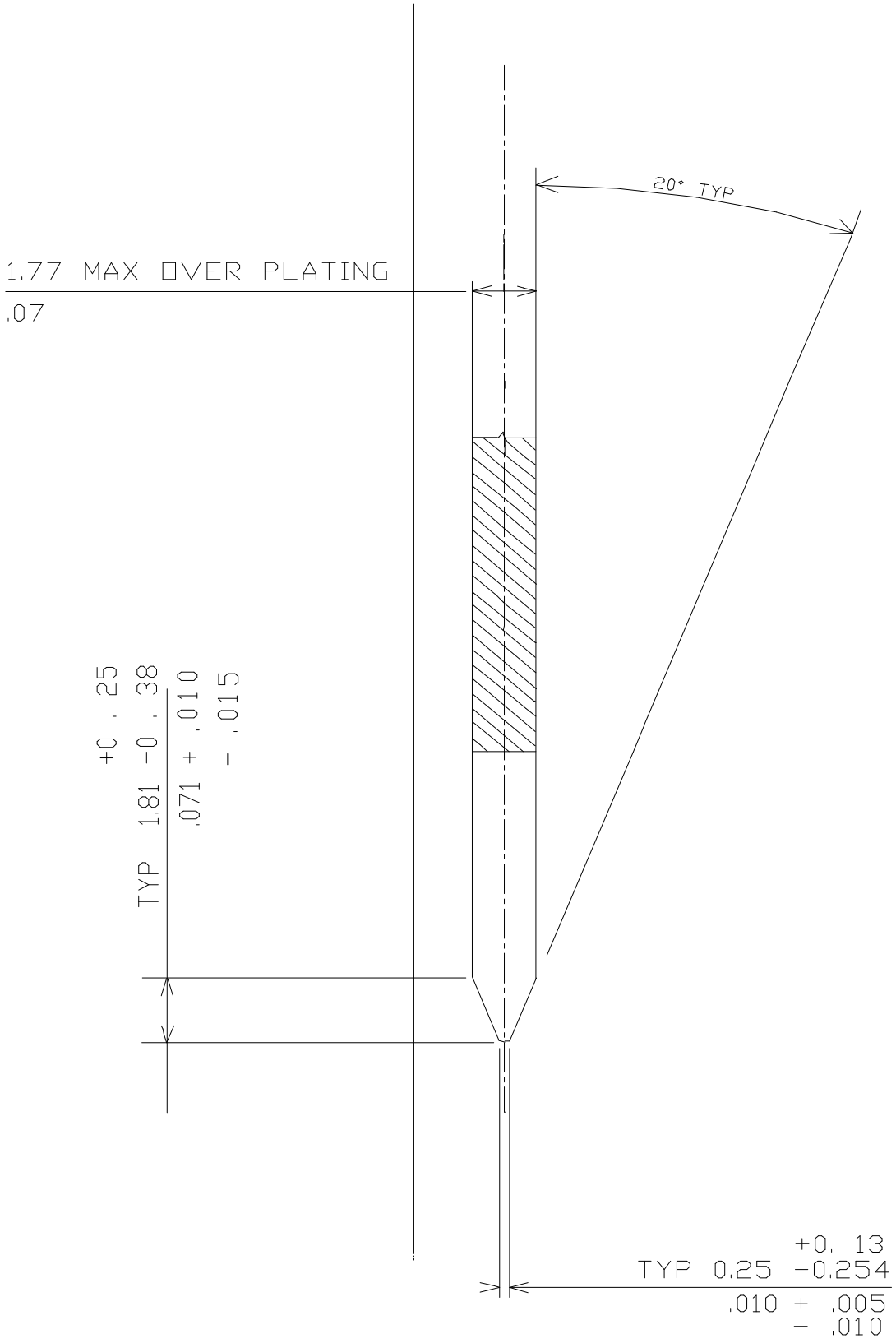


Figure 5-7: PCI Card Edge Connector Bevel

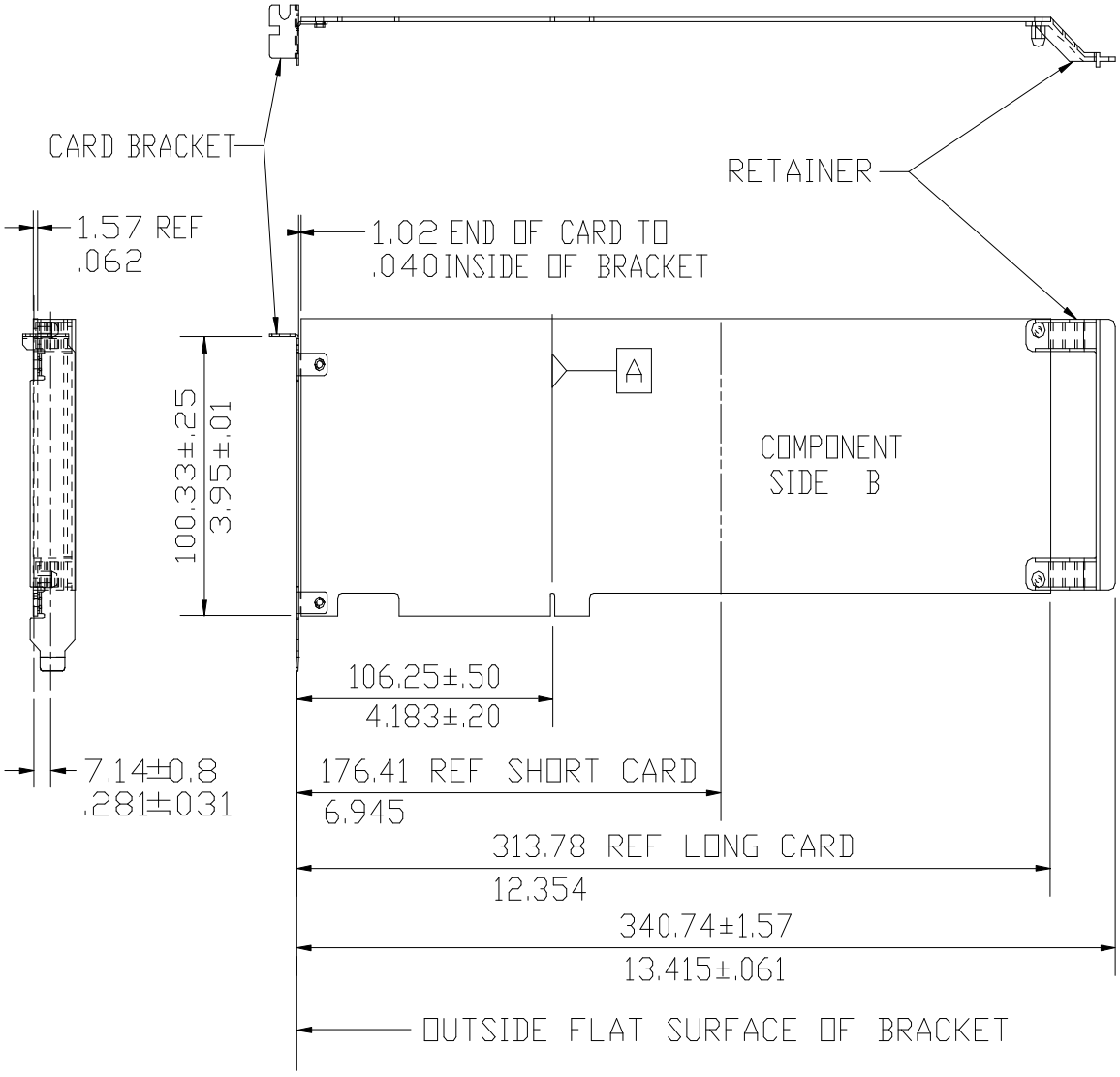


Figure 5-8: ISA Assembly (5V)

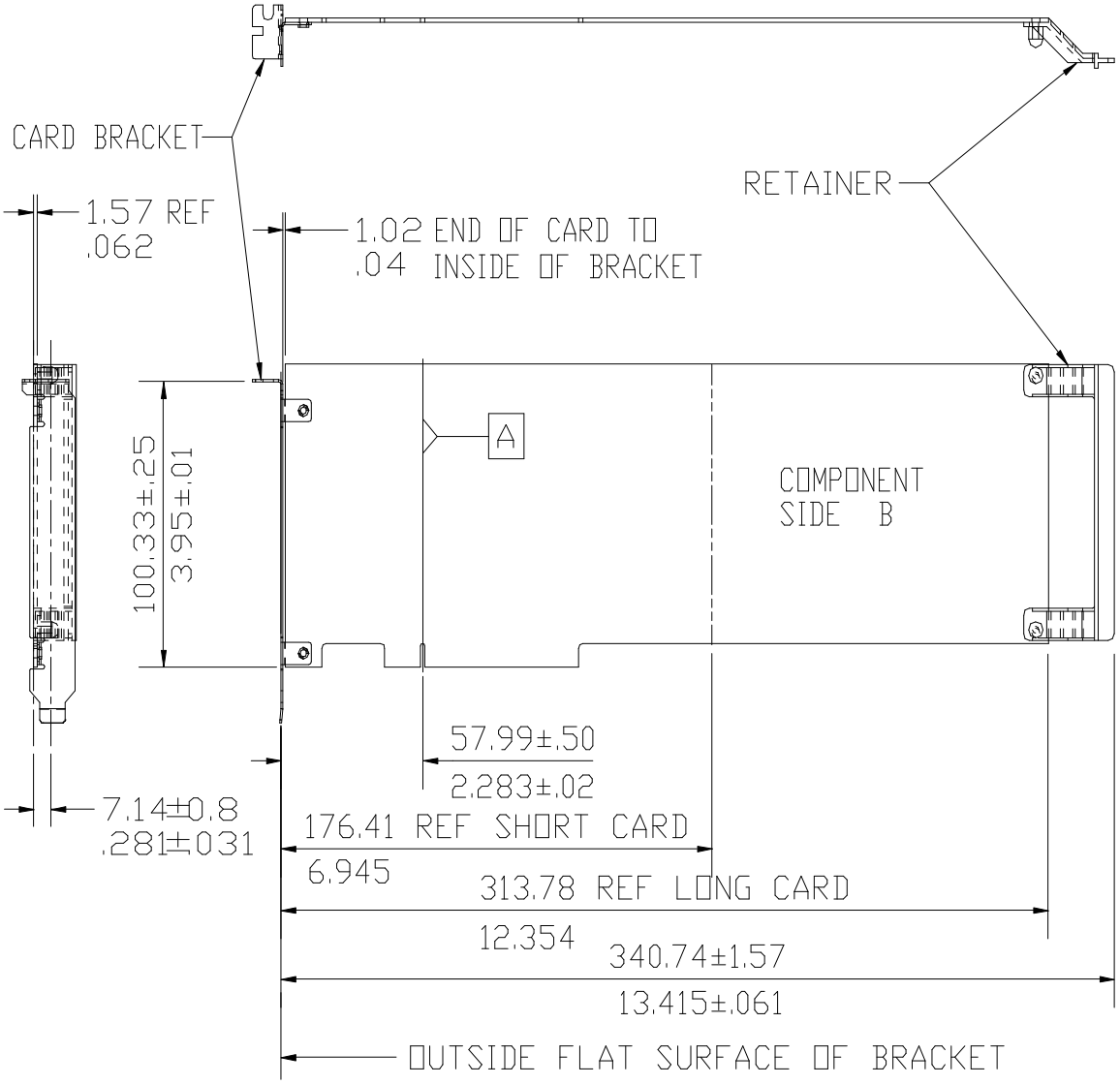


Figure 5-9: ISA Assembly (3.3V and Universal)

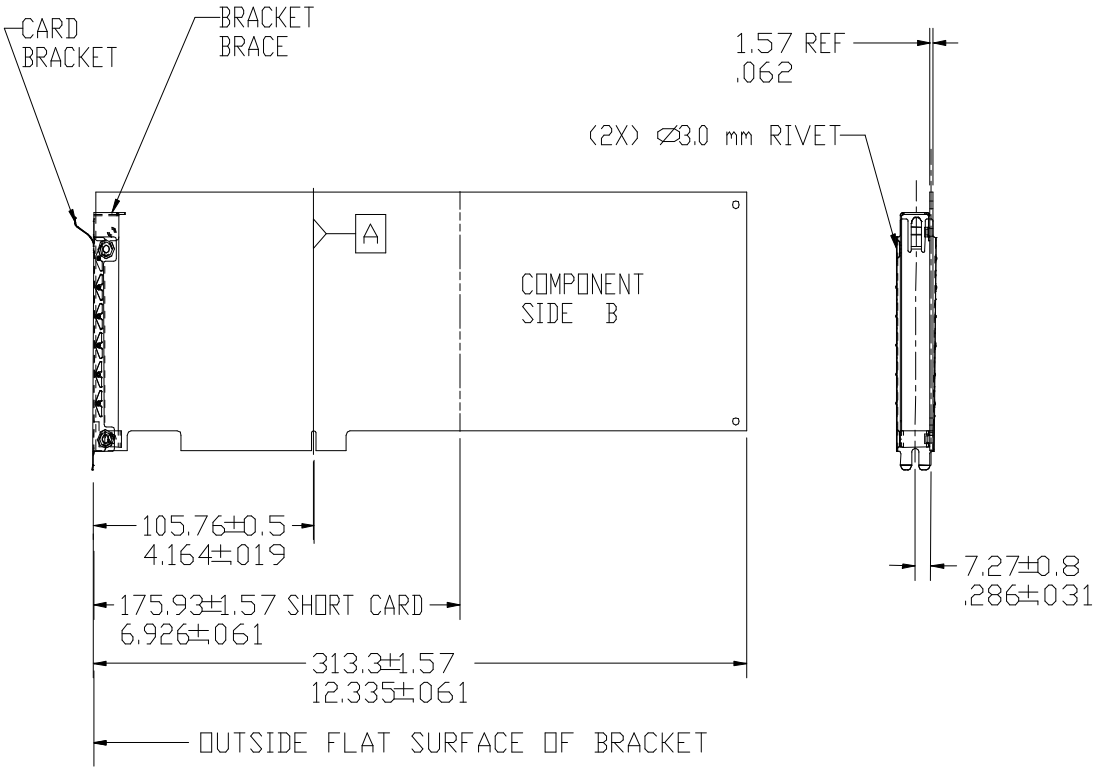


Figure 5-10: MC Assembly (5V)

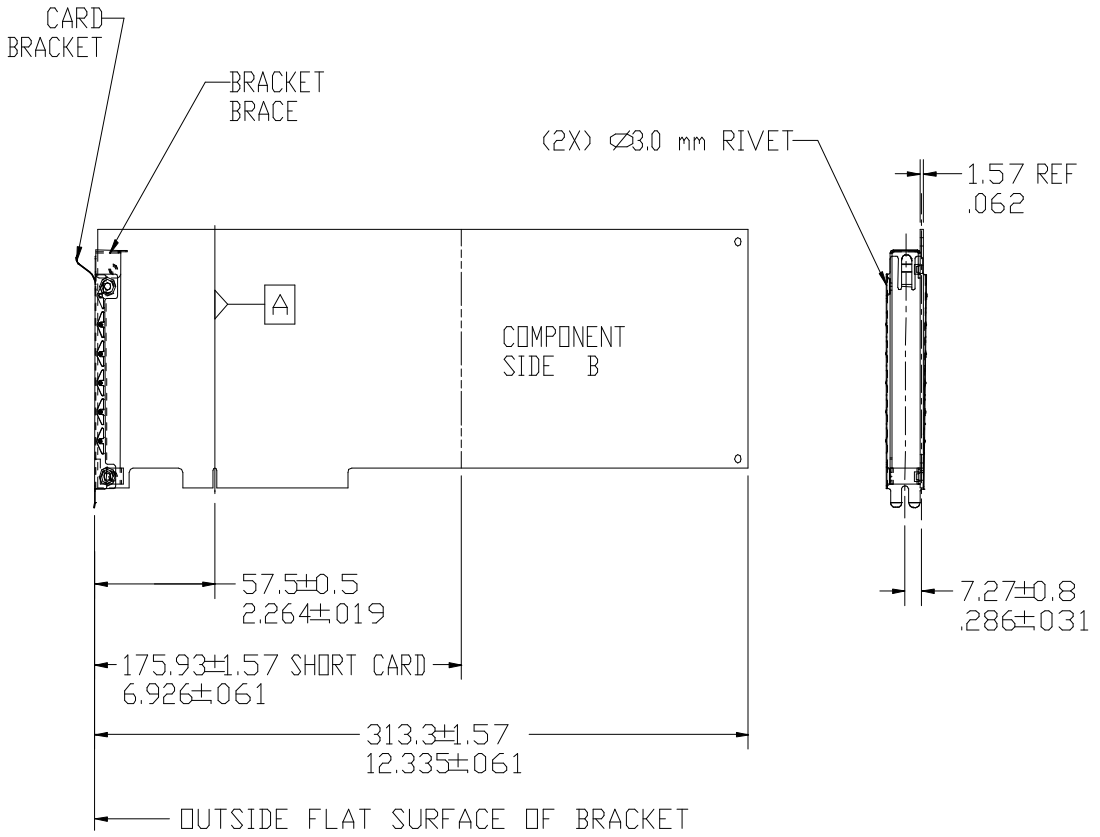


Figure 5-11: MC Assembly (3.3V)

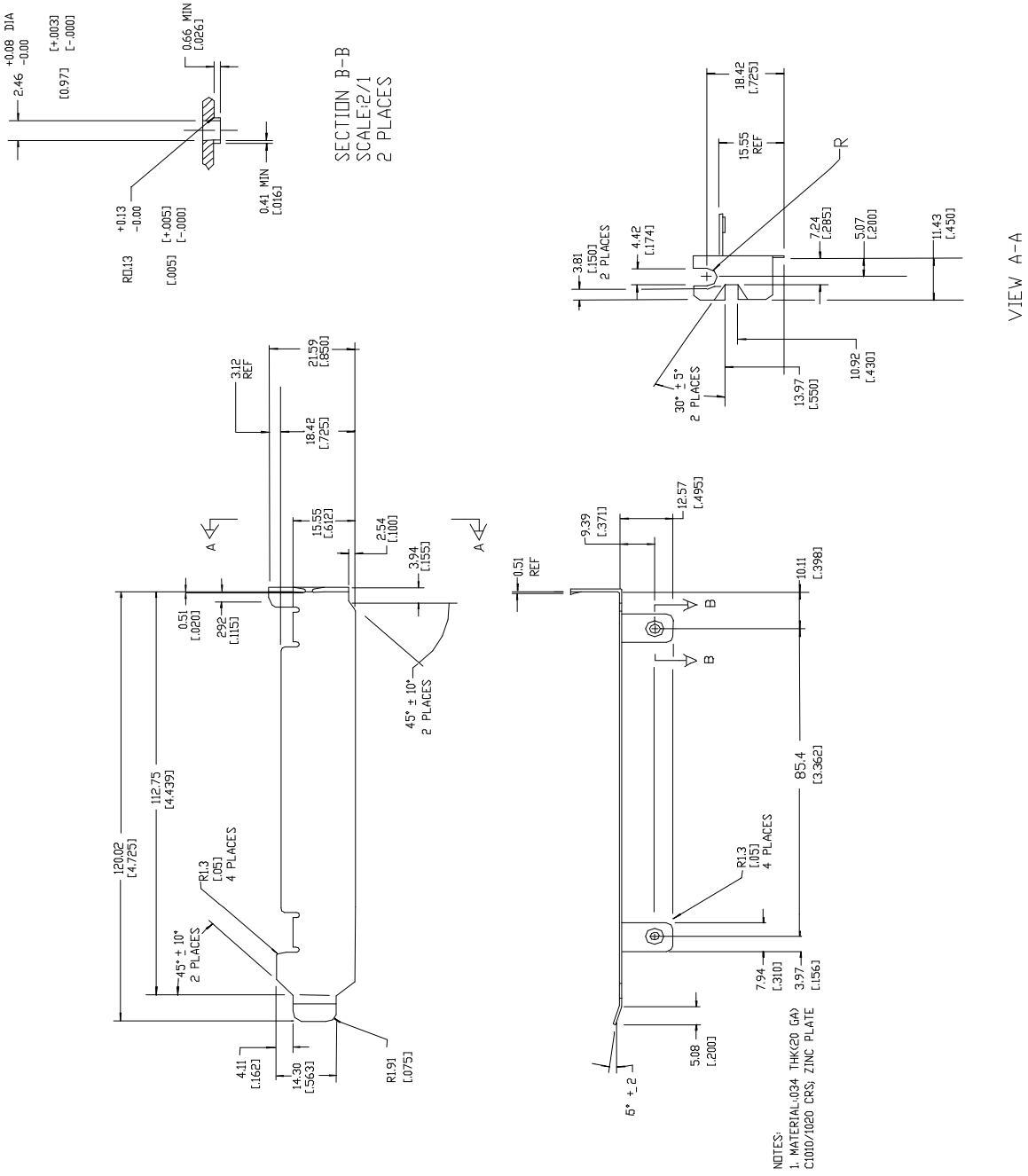


Figure 5-12: ISA Bracket

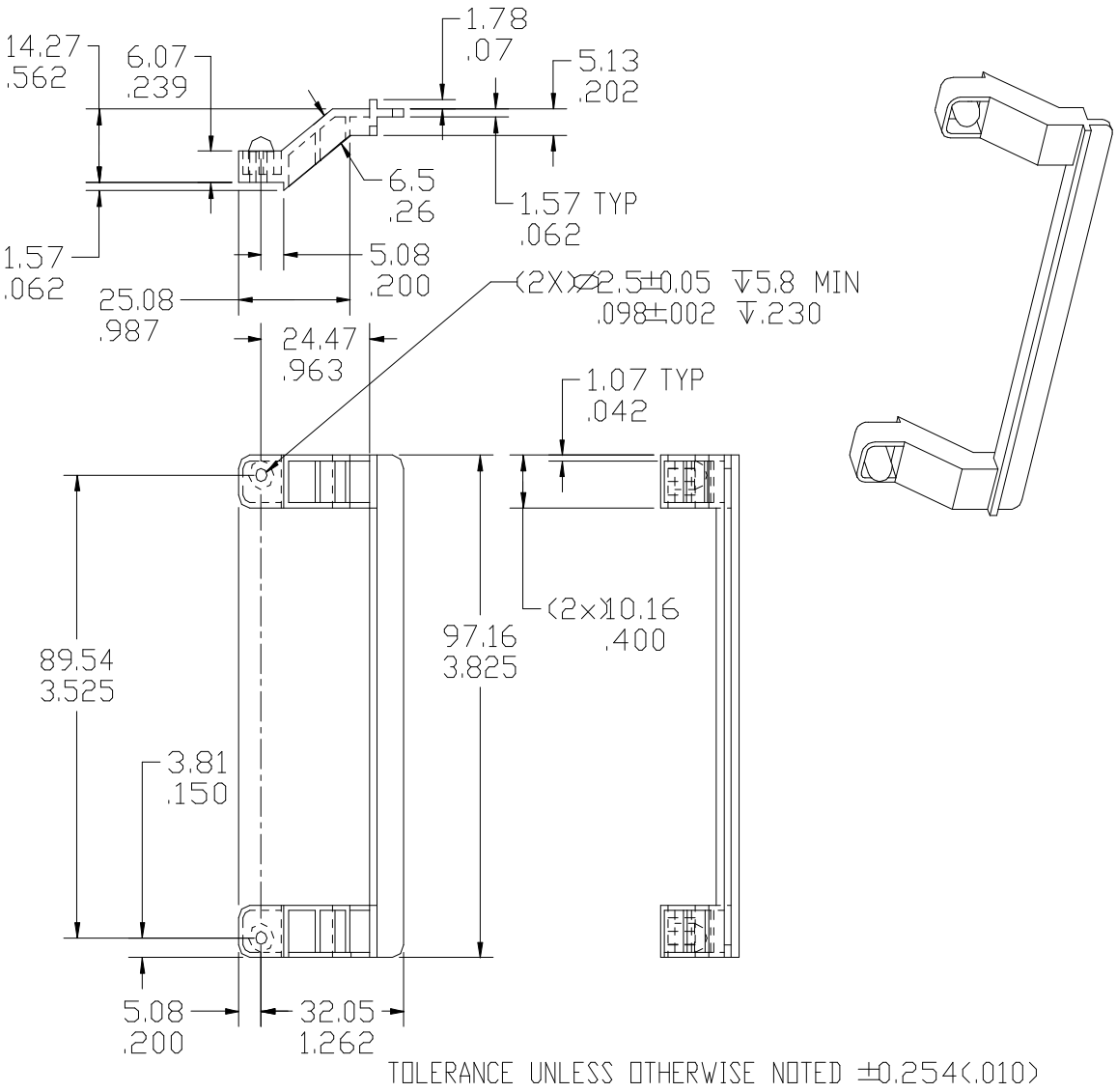


Figure 5-13: ISA Retainer

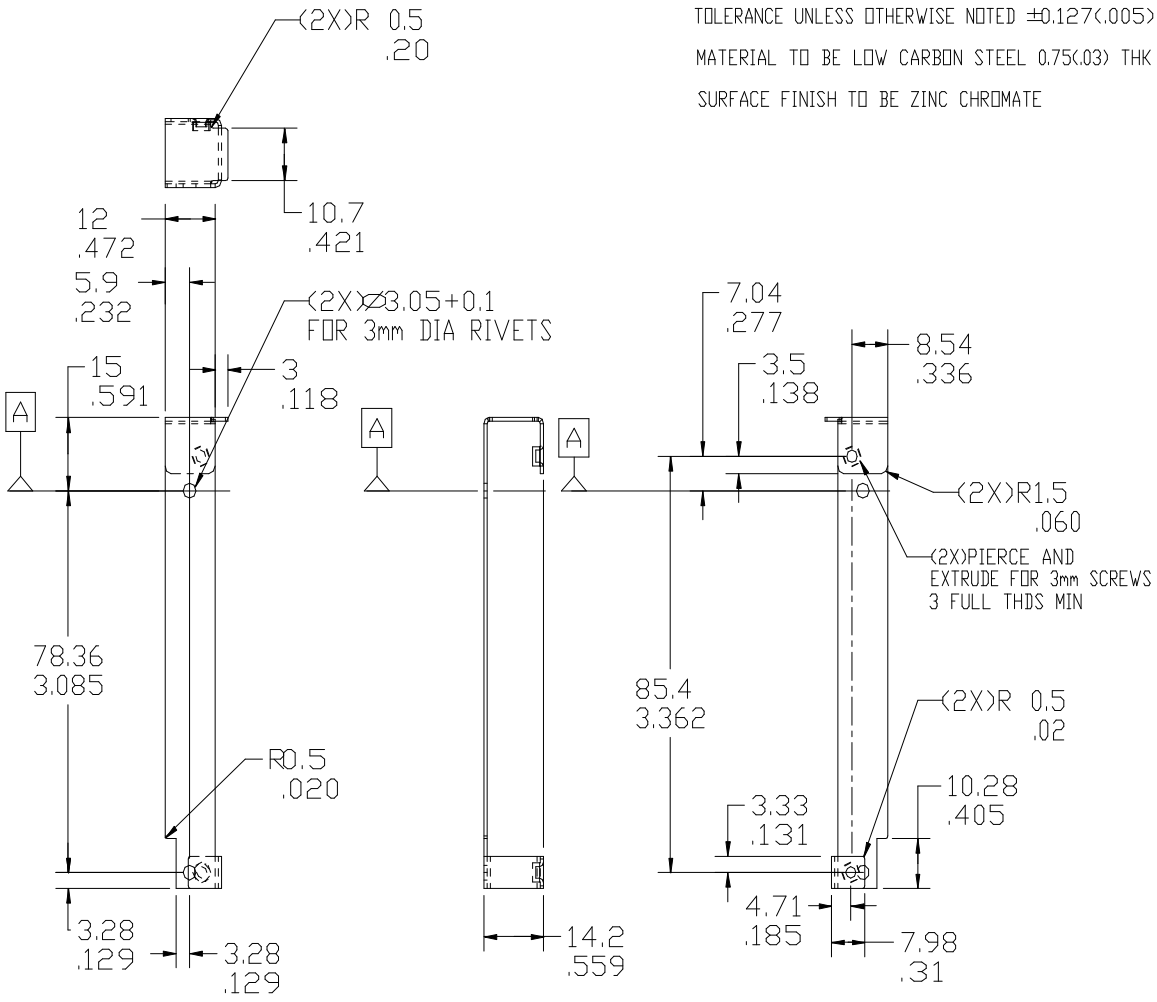


Figure 5-14: MC Bracket Brace

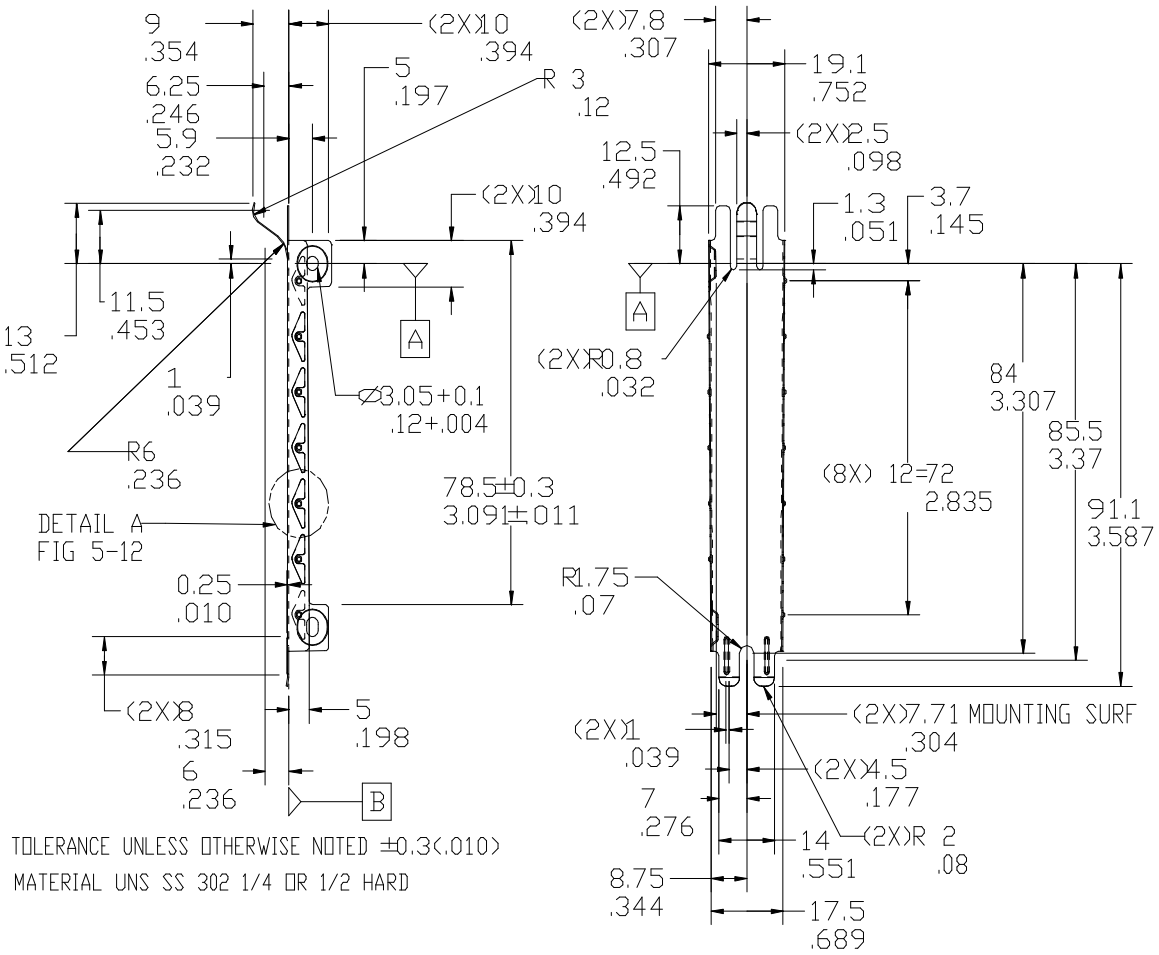


Figure 5-15: MC Bracket

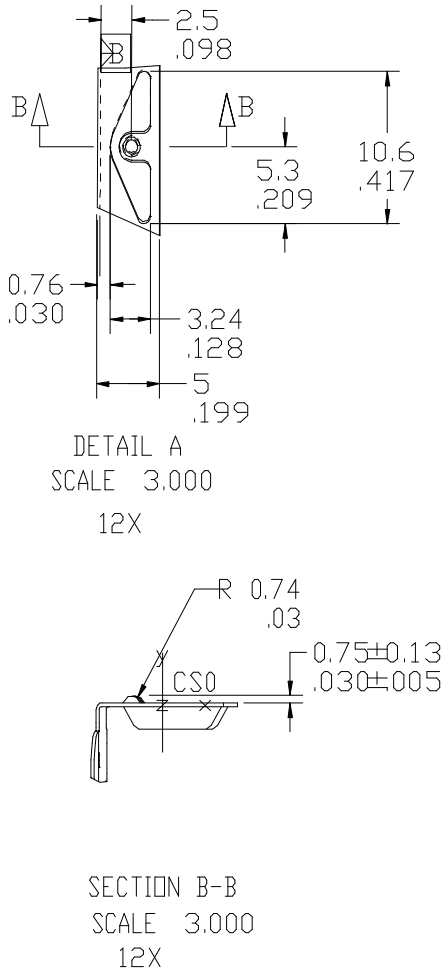


Figure 5-16: MC Bracket Details

5.2.1. Connector Physical Description

The connectors that support PCI expansion cards are derived from those on the MC bus. The MC connectors are well defined and have proven value and reliability. There are four connectors that can be used depending on the PCI implementation. The differences between connectors are 32 bit and 64 bit, and the 5V and 3.3V signaling environments. A key differentiates the signaling environment voltages. The same physical connector is used for the 32-bit signaling environments. In one orientation the key accepts 5V boards. Rotated 180 degrees, the connector accepts 3.3V signaling boards. The pin numbering of the connector changes for the different signaling environments to maintain the same relative position of signals on the connector (see Figures 5-18, 5-19, 5-21, and 5-23 for board layout details).

In the connector drawings, the recommended board layout details are given as nominal dimensions. Layout detail tolerancing should be consistent with the connector supplier's recommendations and good engineering practice.

See Figures 5-17 through 5-23 for connector dimensions and layout recommendations. See Figures 5-24 through 5-30 for card edge connector dimensions and tolerances. Tolerances for cards are given so that interchangeable cards can be manufactured.

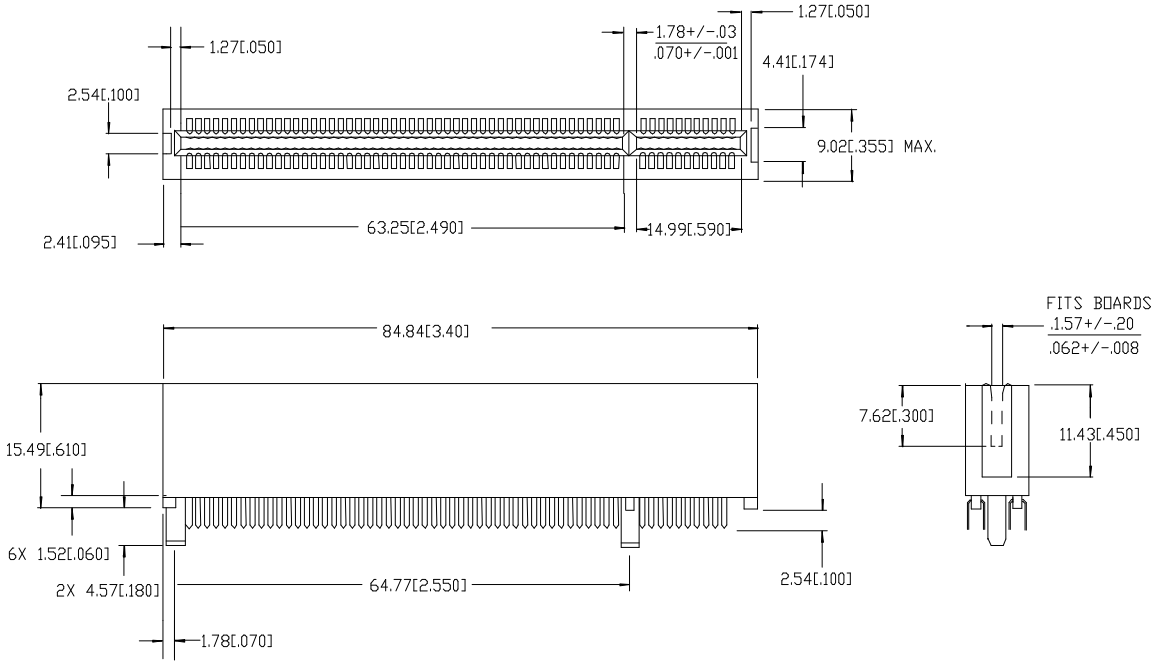


Figure 5-17: 32-bit Connector

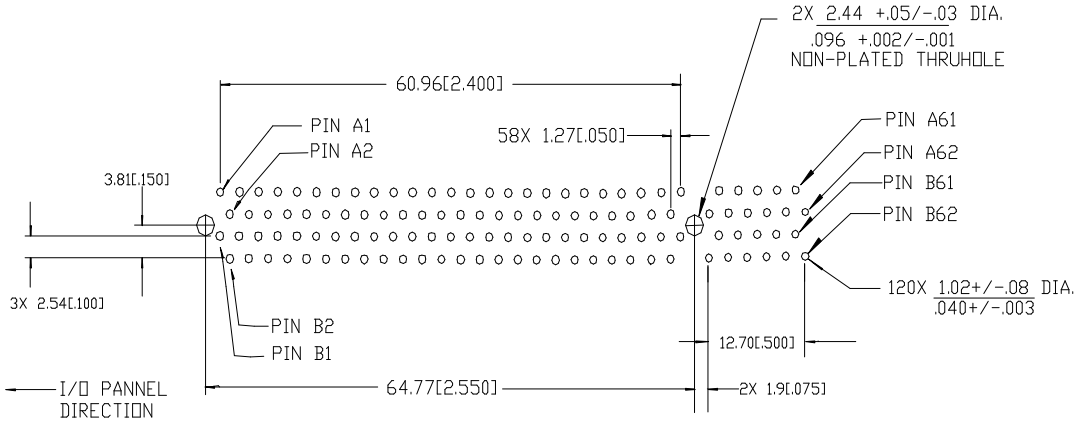


Figure 5-18: 5V/32-bit Connector Layout Recommendation

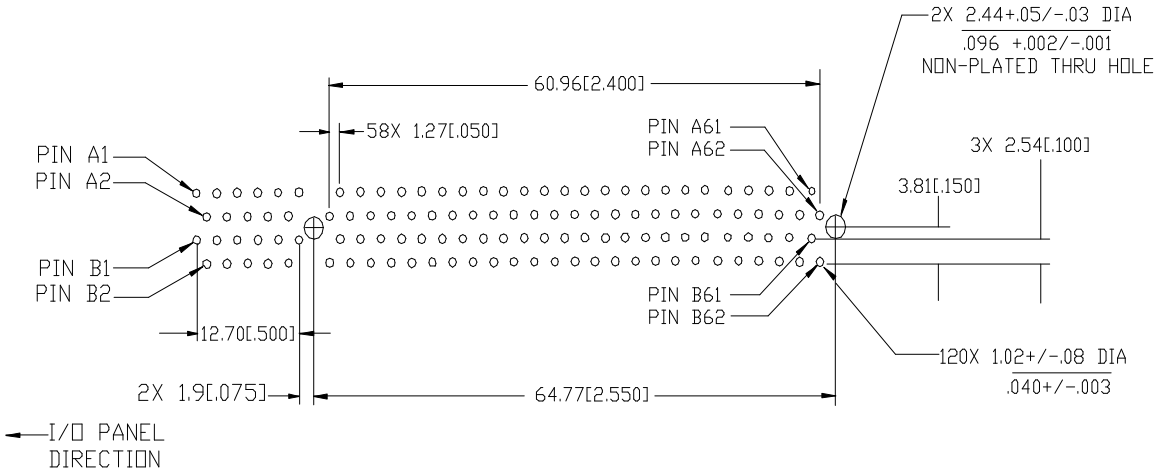


Figure 5-19: 3.3V/32-bit Connector Layout Recommendation

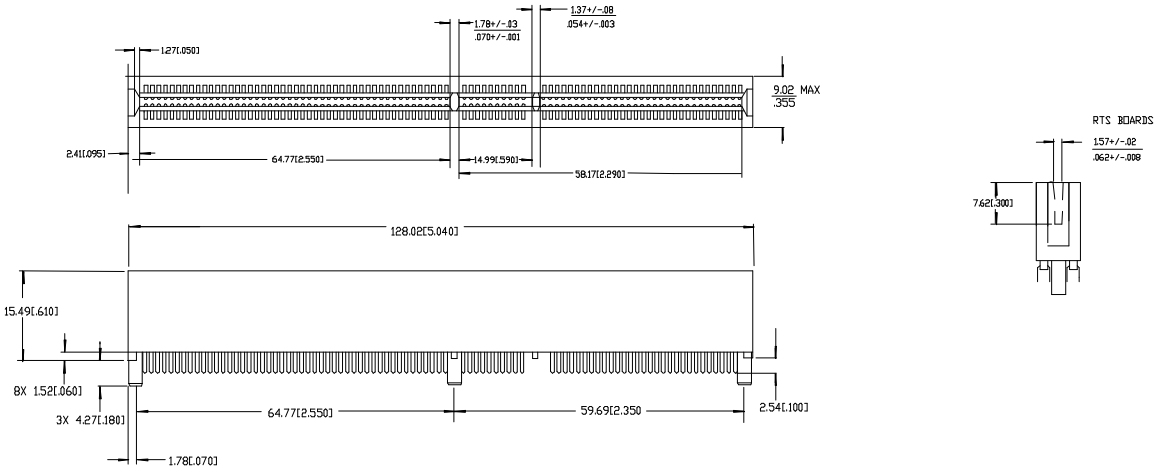


Figure 5-20: 5V/64-bit Connector

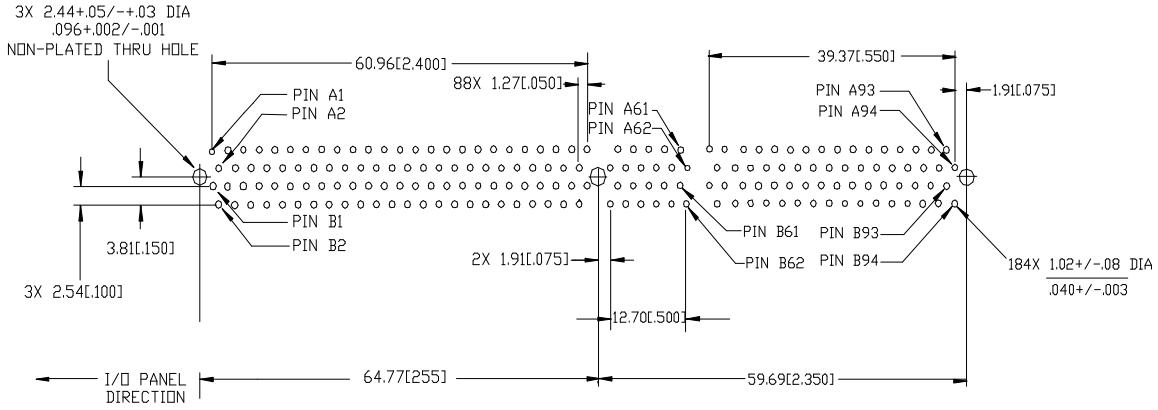


Figure 5-21: 5V/64-bit Connector Layout Recommendation

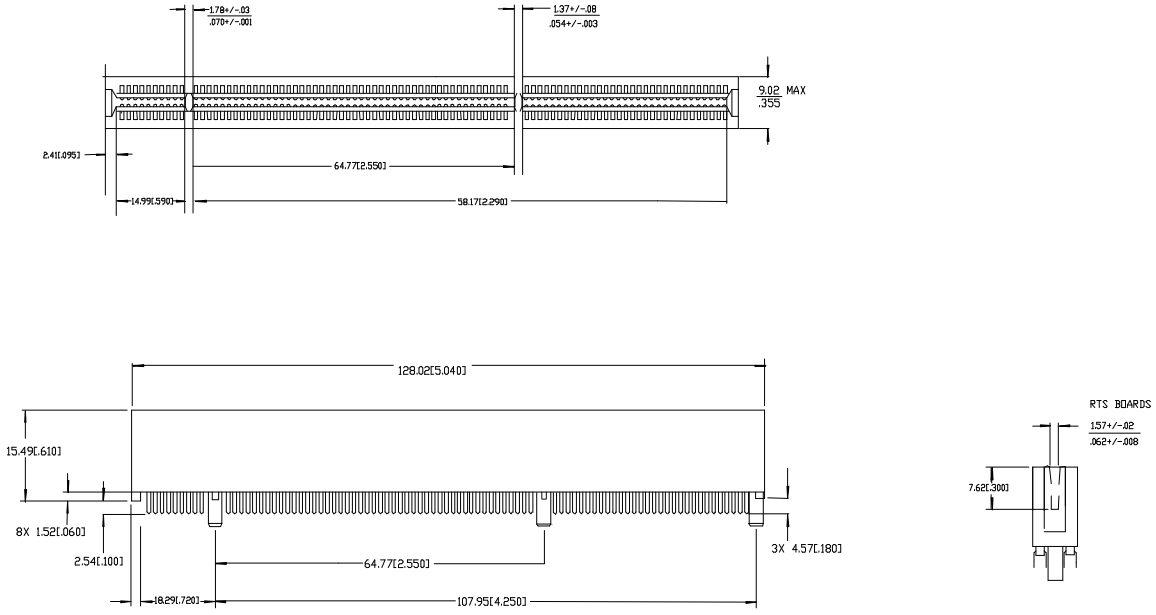


Figure 5-22: 3.3V/64-bit Connector

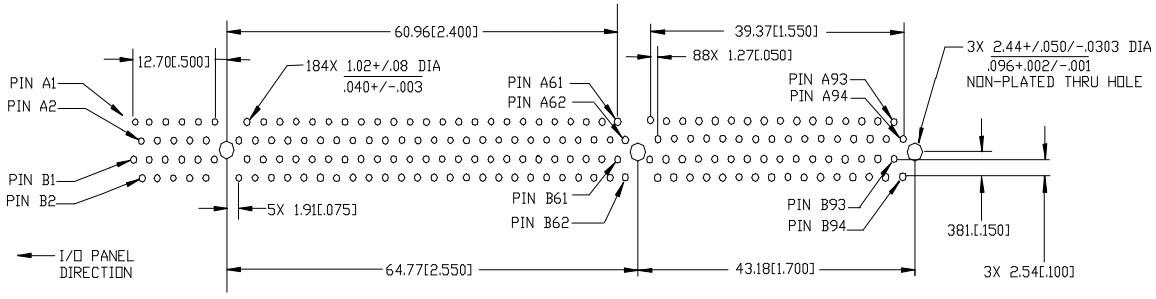


Figure 5-23: 3.3V/64-bit Connector Layout Recommendation

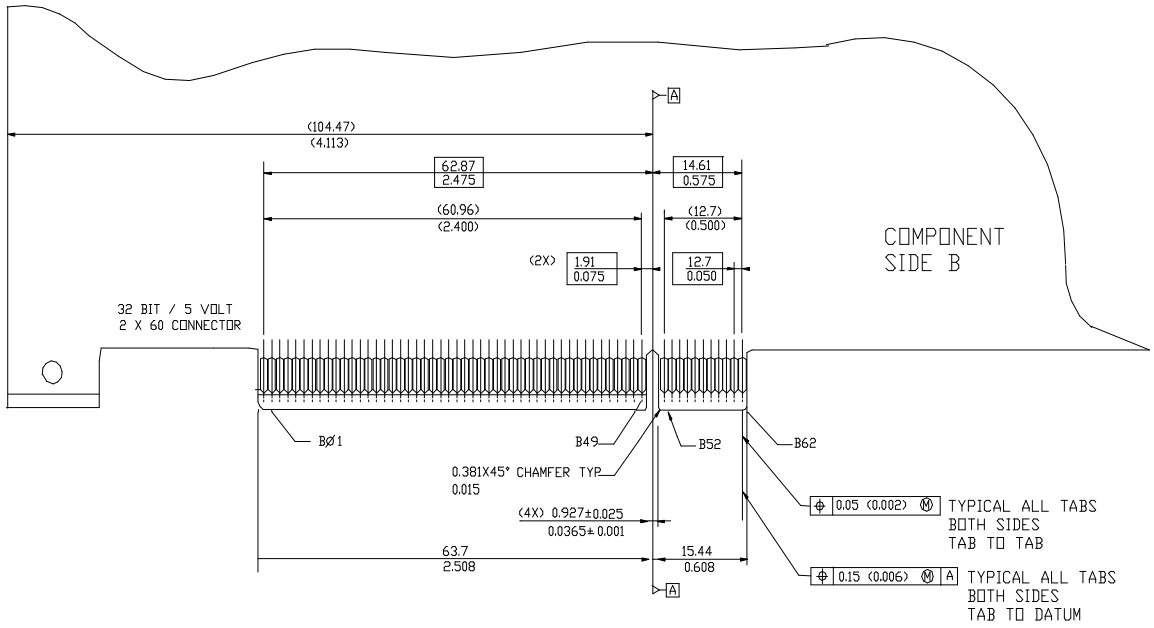


Figure 5-24: 5V/32-bit Card Edge Connector Dimensions and Tolerances

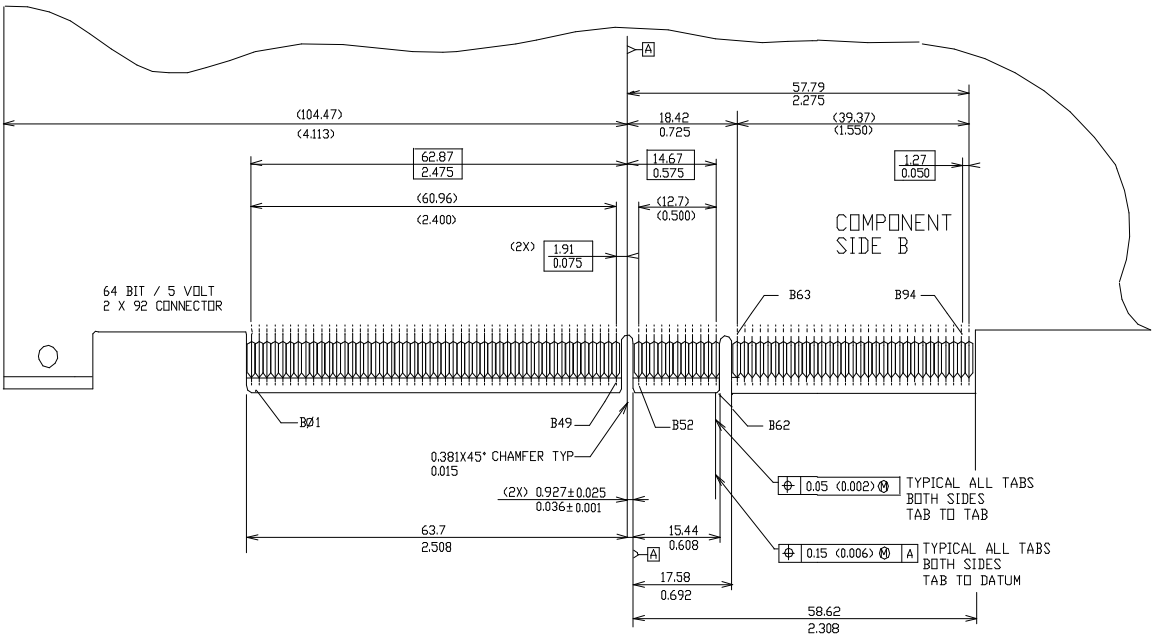


Figure 5-25: 5V/64-bit Card Edge Connector Dimensions and Tolerances

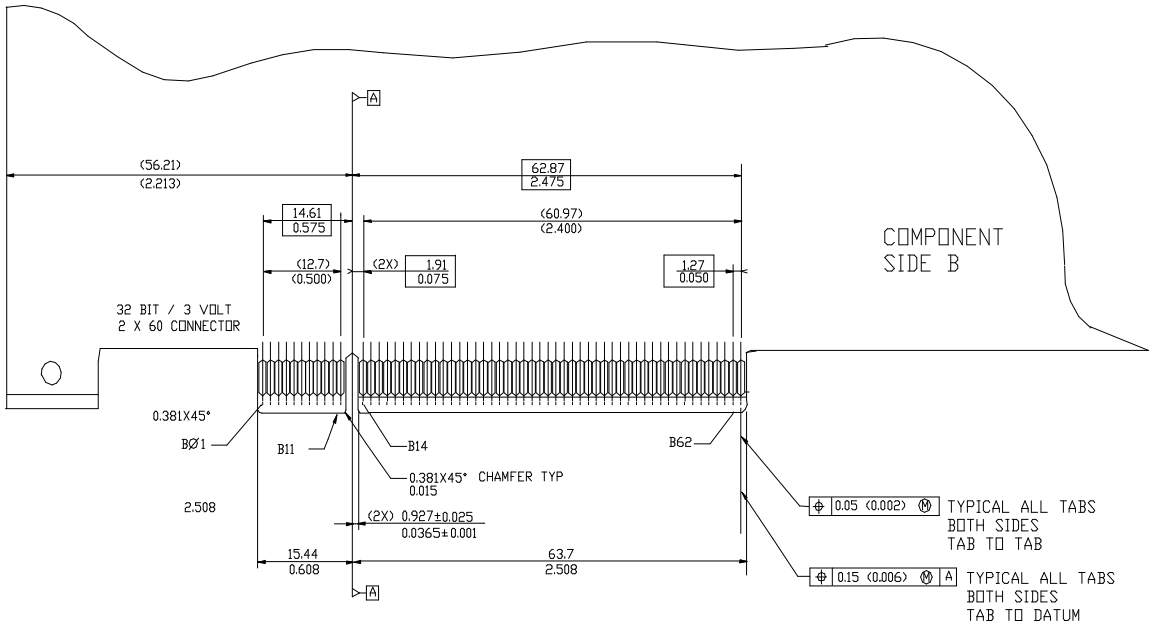


Figure 5-26: 3.3V/32-bit Card Edge Connector Dimensions and Tolerances

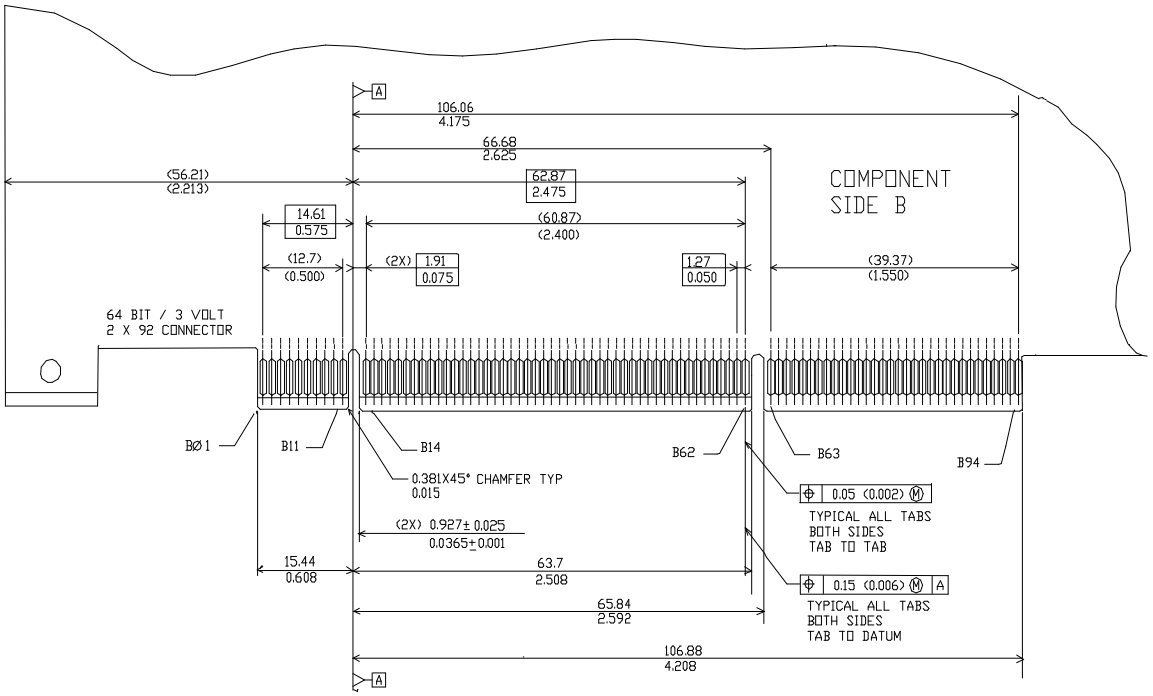


Figure 5-27: 3.3V/64-bit Card Edge Connector Dimensions and Tolerances

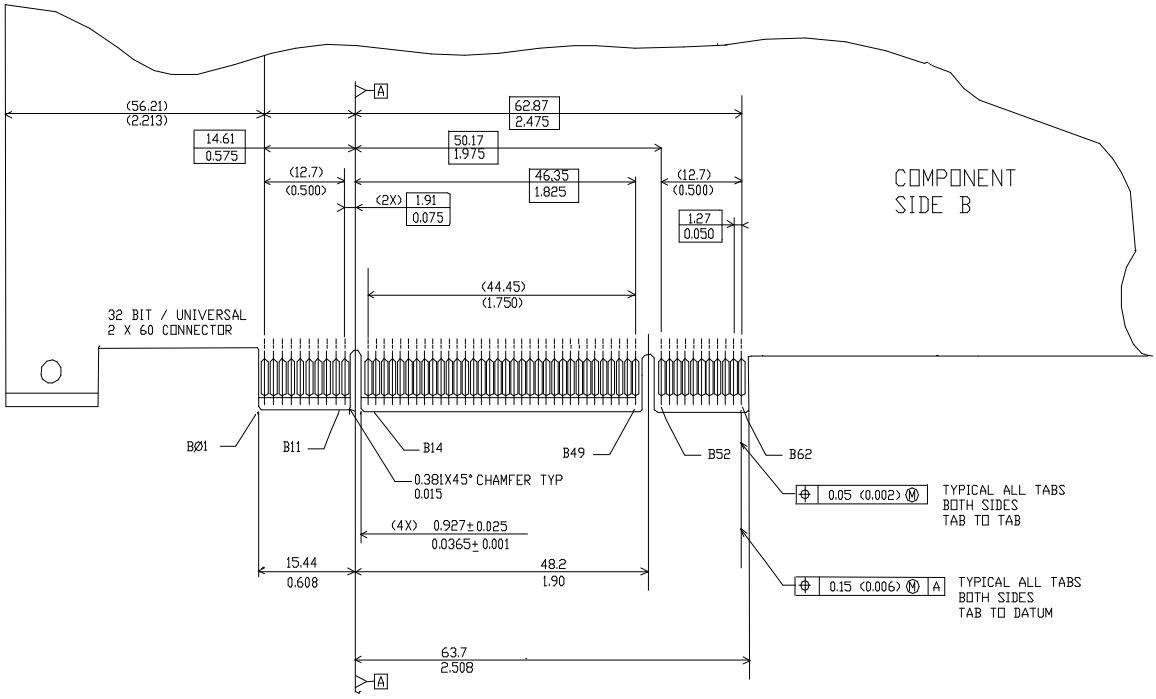


Figure 5-28: Universal 32-bit Card Edge Connector Dimensions and Tolerances

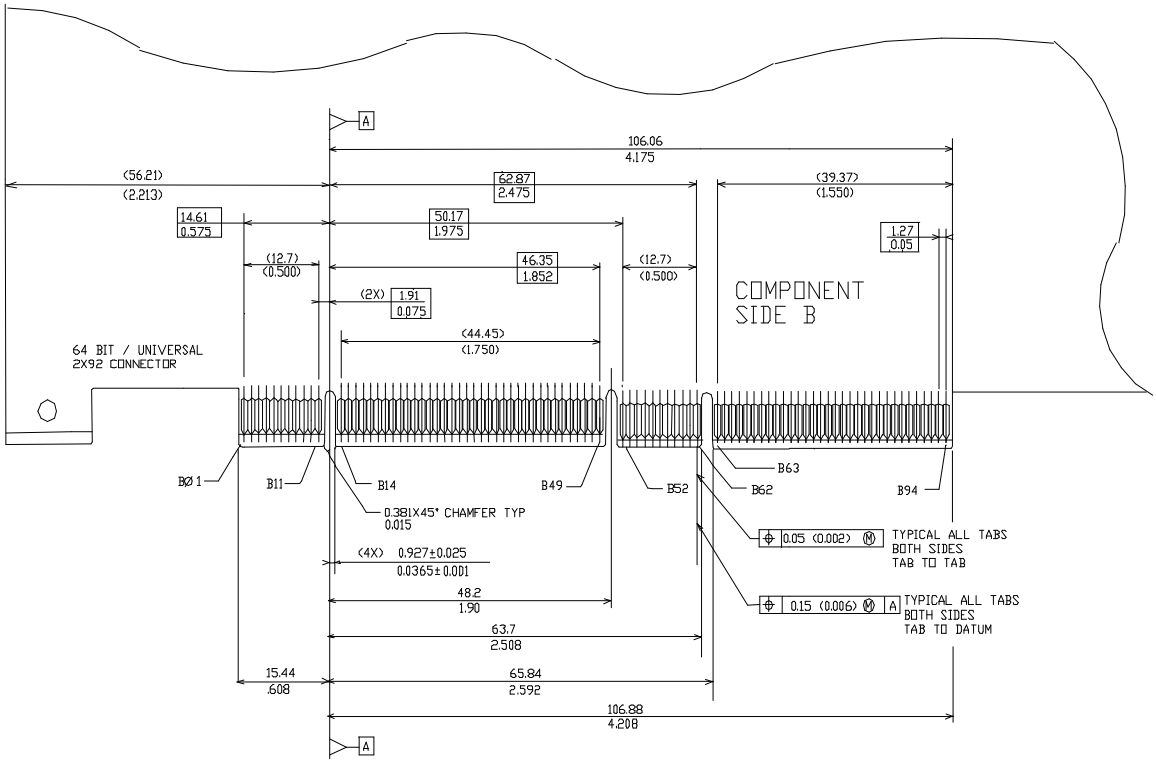


Figure 5-29: Universal 64-bit Card Edge Connector Dimensions and Tolerances

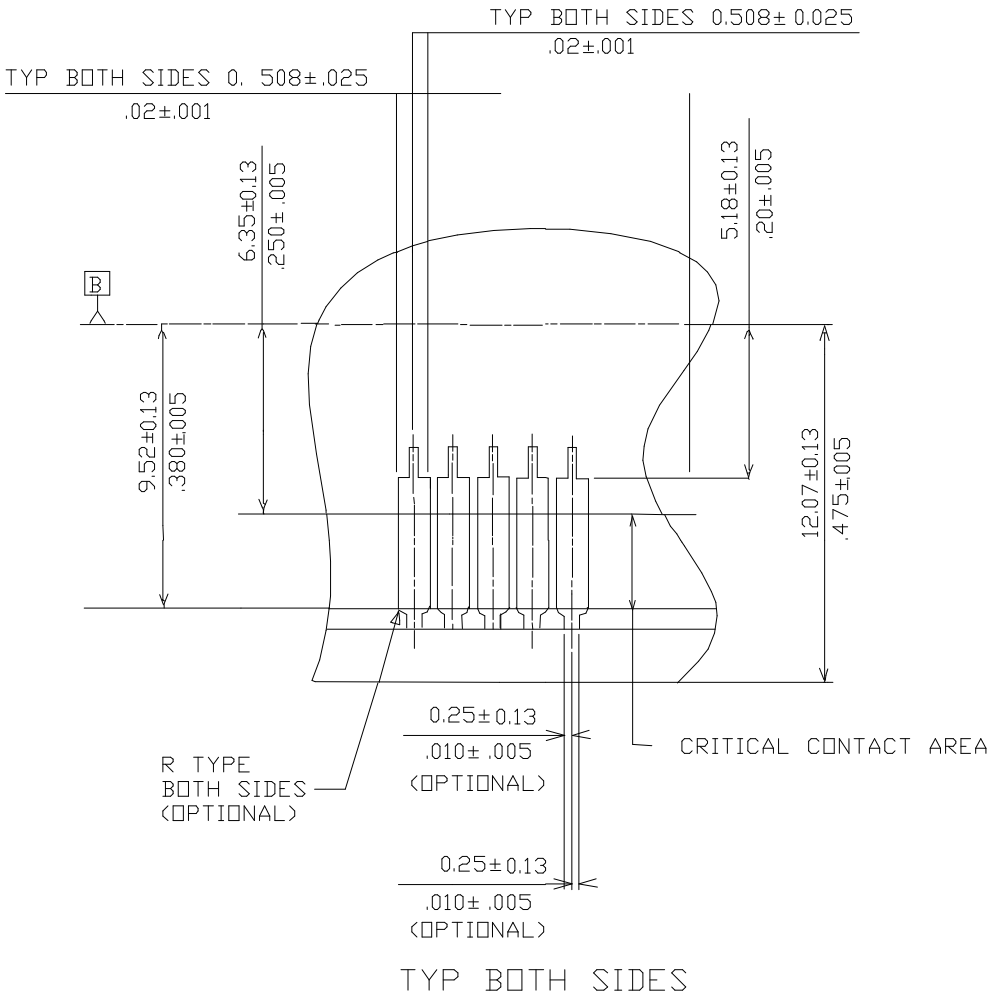


Figure 5-30: PCI Card Edge Connector Contacts

5.2.1.1. 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	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.

5.2.1.2. 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 card edge.
Mating Force	6 oz. (1.7N) 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 board specification.
Contact Normal Force	75 grams minimum.

Table 5-3: Connector Electrical Performance Requirements

Parameter	Specification
Contact Resistance	(low signal level) 30 milliohms max. initial, 10 milliohms 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	1A, 30 °C rise above ambient.
Voltage Rating	125V.
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 board and tested per Battelle method.

5.2.2. Planar Implementation

Two types of planar implementations are supported by the PCI expansion card design: expansion connectors mounted on the planar and expansion connectors mounted on a riser card. For illustrative purposes, only the planar mounted expansion connectors are detailed here. The basic principles may be applied to riser card designs. See Figures 5-31, 5-32, and 5-33 for planar details for ISA, EISA, and MC cards, respectively. The planar drawings show the relative locations of the PCI 5V and 3.3V connector datums to the ISA, EISA, and MC connector datums. Both 5V and 3.3V connectors are shown on the planar to concisely convey the dimensional information. Normally, a given system would incorporate either the 5V or the 3.3V PCI connector, but not both. Standard card spacing of 0.8 inches for ISA/EISA and 0.85 inches for MC allows for only one shared slot per system. If more PCI expansion slots are required, while using existing card spacing, additional slots must be dedicated to PCI. Viewing the planar from the back of the system, the shared slot is located such that dedicated ISA, EISA, or MC slots are located to the right and dedicated PCI slots are located to the left.

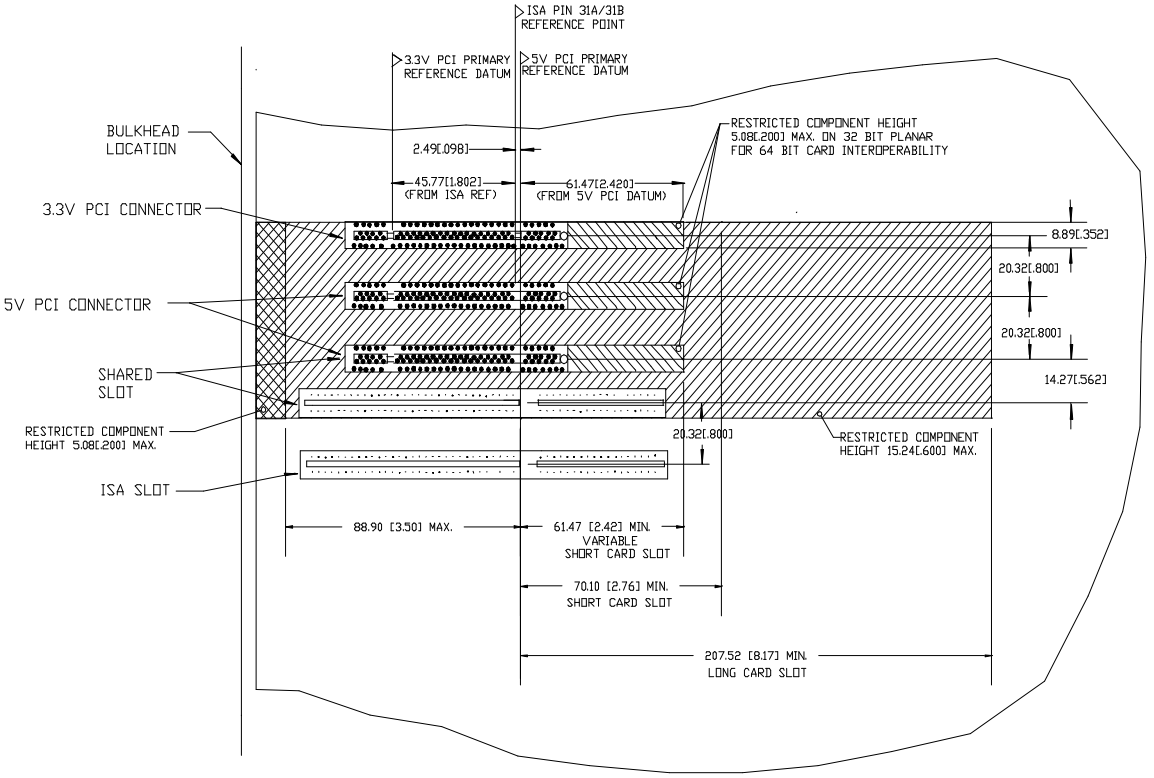


Figure 5-31: PCI Connector Location on Planar Relative to Datum on the ISA Connector

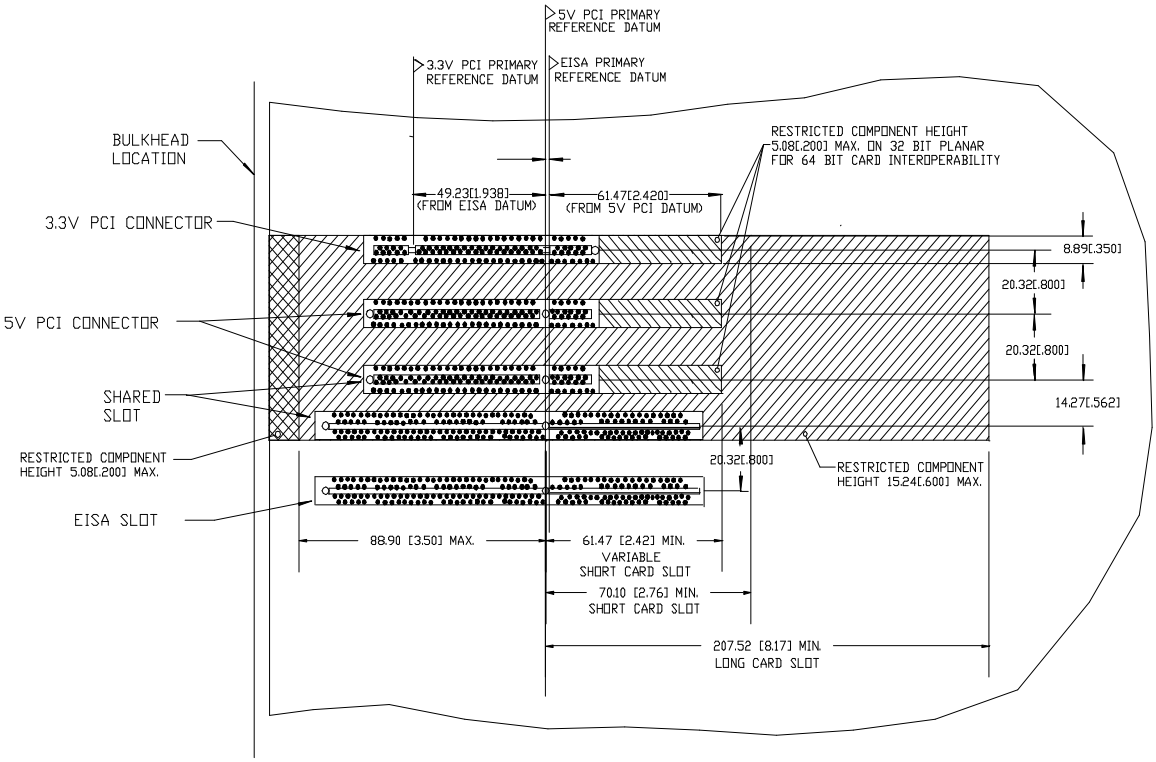


Figure 5-32: PCI Connector Location on Planar Relative to Datum on the EISA Connector

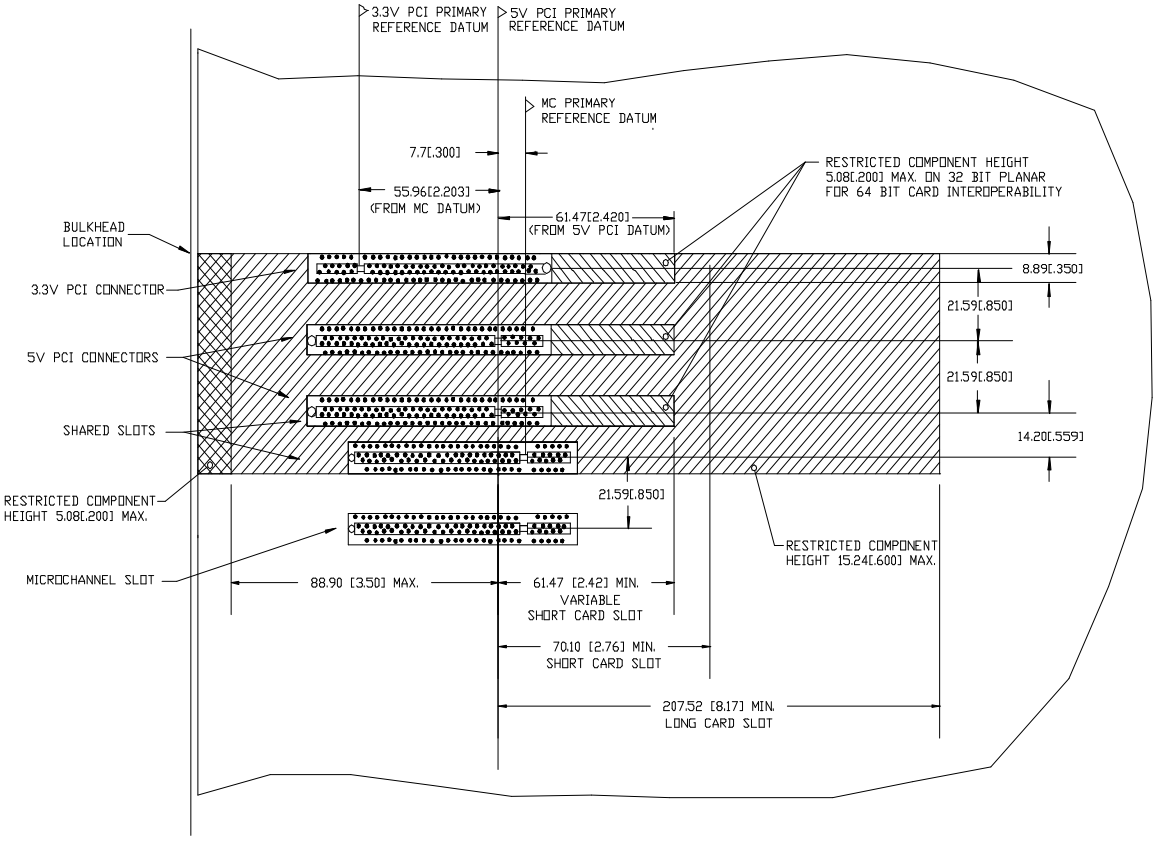


Figure 5-33: PCI Connector Location on Planar Relative to Datum on the MC Connector

Chapter 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
- Effective support of existing configuration mechanisms (e.g., EISA Configuration Utility)
- 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 must implement Configuration Space. Multifunction devices must provide a Configuration Space for each function implemented (refer to Section 6.4).

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.[38] 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 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 two Header Types are defined, 00h which has the layout shown in Figure 6-1 and 01h which is defined for PCI-to-PCI bridges and is documented in the *PCI to PCI Bridge Architecture Specification*.

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 0FFFFh 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.

[38] The device dependent region contains device specific information and is not described in this document.

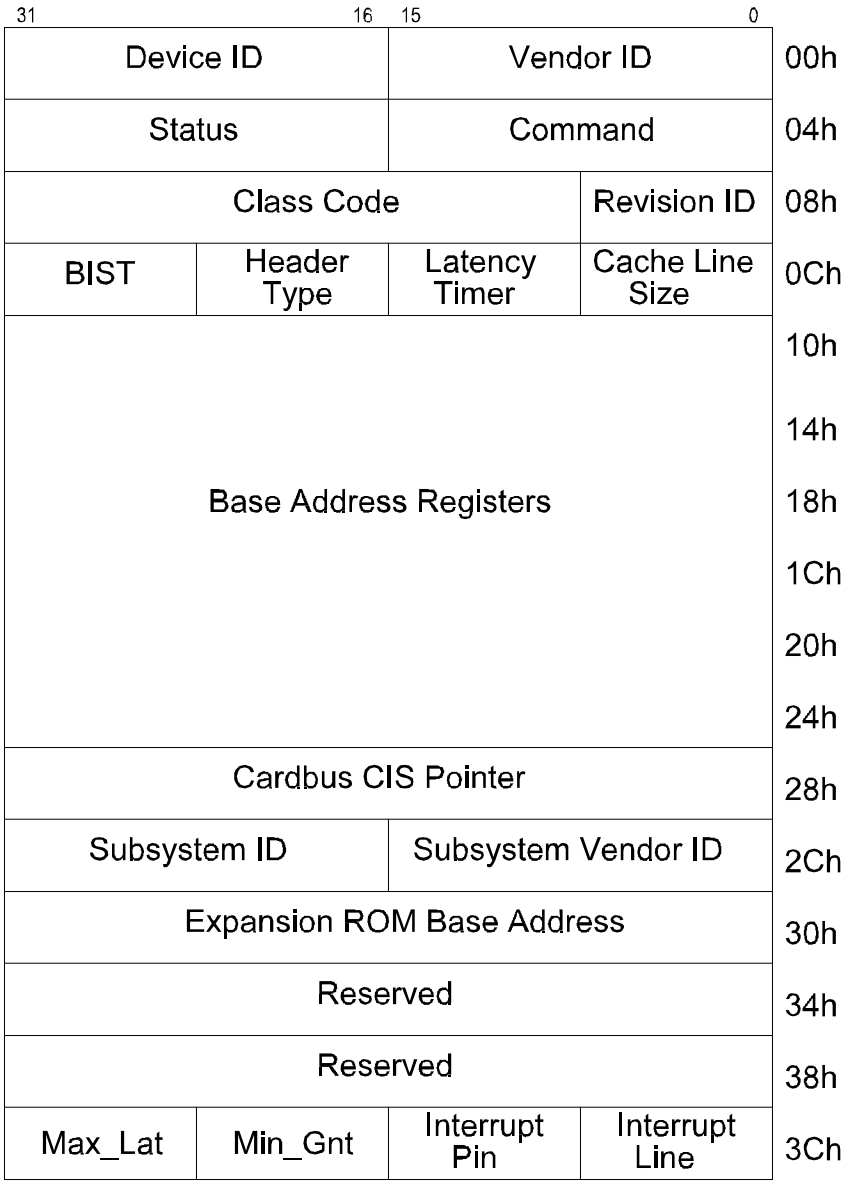


Figure 6-1: Type 00h Configuration Space Header

All PCI compliant devices must support the Vendor ID, Device ID, Command, Status, Revision ID, Class Code, and Header Type fields in the header. 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 back 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. 0FFFFh 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.* 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 specificly 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. Table 6-1 shows the defined values for base classes (i.e., the upper byte in the Class Code field).

Table 6-1: Class Code Register Encodings

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 - FEh	Reserved.
FFh	Device does not fit in any defined classes.

Encodings for 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 devices functionality. For instance, devices that do not implement an I/O Space probably 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 6-2 explains the meanings of the different bits in the Command register.

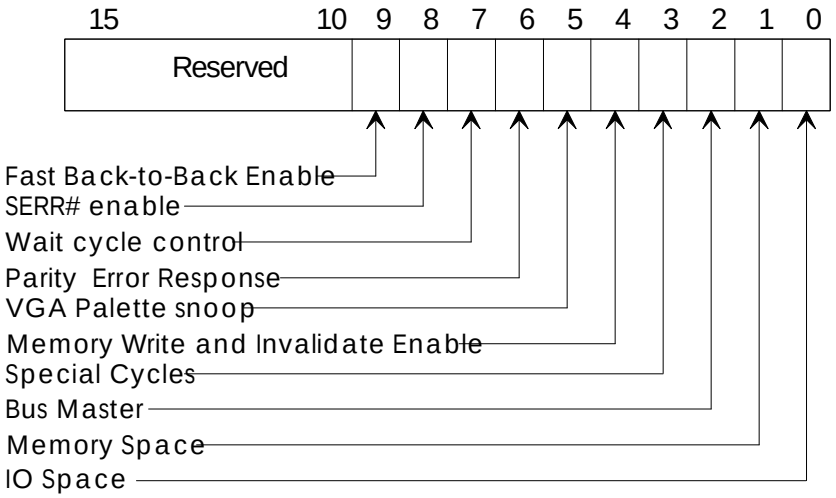


Figure 6-2: Command Register Layout

Table 6-2: 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 accesses like all other accesses. VGA compatible devices should implement this bit. See Section 3.10 for more details on VGA palette snooping.

Table 6-2: Command Register Bits *(continued)*

Bit Location	Description
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 must ignore any parity errors that it detects and continue 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	This bit is used to control whether or not a device does address/data stepping. Devices that never do stepping must hardwire this bit to 0. Devices that always do stepping must hardwire this bit to 1. Devices that can do either, must make this bit read/write and have it initialize to 1 after **RST#**.
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. This bit (and bit 6) must be on to report address parity errors.
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.
10-15	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 6-3 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.

Reads to this register behave normally. Writes are slightly different in that bits can be reset, but not set. A bit is reset whenever the register is written, and the 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.

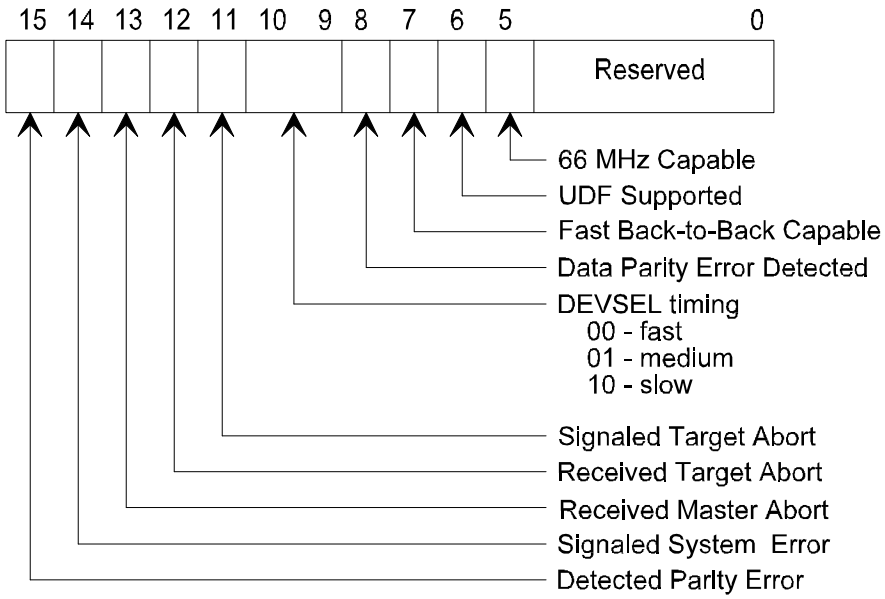


Figure 6-3: Status Register Layout

Table 6-3: Status Register Bits

Bit Location	Description
0-4	Reserved.
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 optional read-only bit indicates that this device supports User Definable Features. This bit is required to be set when a device function has device specific configuration selections that must be presented to the user. Functions that do not support user selectable configuration items would not implement this bit, and therefore return a 0 when read. Refer to Section 6.7. for a complete description of requirements for setting this bit.
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 or observed **PERR#** asserted; 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 3.7.1. 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.

Table 6-3: Status Register Bits (*continued*)

Bit Location	Description
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.
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 Functions

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 32-bit words. 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) must implement this register to know when a burst sequence wraps to the beginning of the cacheline.

Devices participating in the caching protocol (refer to Section 3.8) use this field to know when to Disconnect burst accesses at cacheline boundaries. These devices can ignore the PCI cache support lines (**SDONE** and **SBO#**) when this register is set to 0.

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.). 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 6-4 describes the bits in the register.

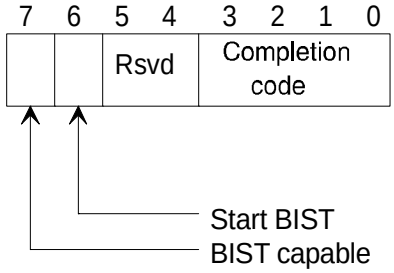


Figure 6-4: BIST Register Layout

Table 6-4: 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.[39]

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. 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 devices 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 of 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.

Subsystem Vendor ID* and *Subsystem ID

These registers are used to uniquely identify the add-in board or subsystem where the PCI device resides. They provide a mechanism for add-in card vendors to distinguish their cards from one another even though the cards may have the same PCI controller on them (and, therefore, the same Vendor ID and Device ID).

Implementation of these registers is optional and an all zero value indicates that the device does not support subsystem identification. Subsystem Vendor IDs can be obtained from the PCI SIG and are used to identify the vendor of the add-in board or subsystem. Values for Subsystem ID are vendor specific. Values in these registers are programmed during the manufacturing process or loaded from external logic (e.g., strapping options, serial ROMs, etc), prior to the system BIOS or any system software accessing the PCI Configuration Space. Devices loading these values from external logic 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 a Retry until the data is valid.

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.

---

[39] 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 255 are reserved.

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).

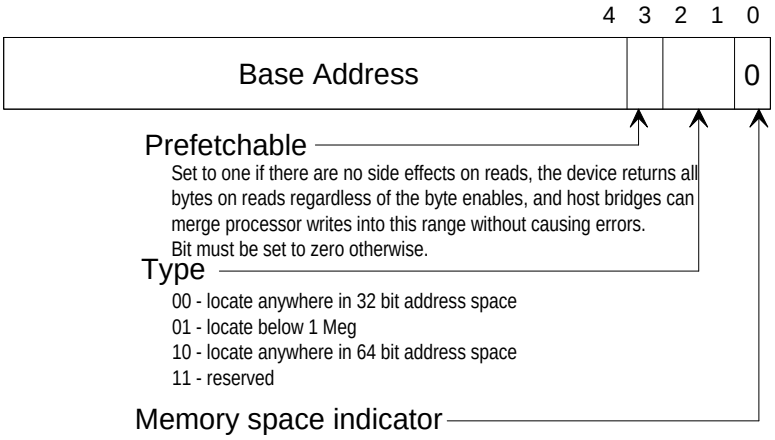


Figure 6-5: Base Address Register for Memory

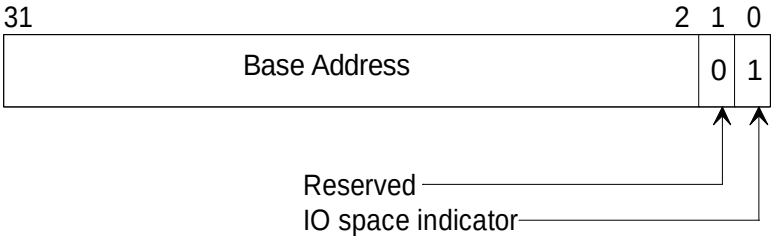


Figure 6-6: Base Address Register for I/O

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 6-5. 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[40] without causing errors. Bits 0-3 are read-only.

Table 6-5: 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	Base register is 32 bits wide but must be mapped below 1M in Memory Space.
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 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 required 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.

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 may not consume more than 256 bytes per I/O Base Address register.

A type 00h predefined header has six DWORD locations allocated for Base Address registers starting at offset 10h in Configuration Space. The first Base Address register is always located at offset 10h. The second register may be at offset 14h or 18h depending on the size of the first. The offsets of subsequent Base Address registers are determined by the size of previous Base Address registers.

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 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 should always allow control functions to be mapped into Memory Space.

[40] Any device that has a range that behaves like normal memory, but doesn't participate in PCI's caching protocol, 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.

6.2.5.2. Expansion ROM Base Address Register

Some PCI devices, especially those that are intended for use on add-in modules in PC architectures, require local EPROMs for expansion ROM (refer to Section 6.3. 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.

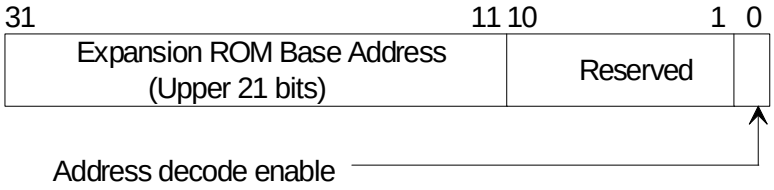


Figure 6-7: Expansion ROM Base Address Register Layout

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 on a device, it may share a decoder between the Expansion ROM Base Address register and other Base Address registers.[41] 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.

[41] 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.

6.2.5.3. Add-in Memory

A mechanism for handling add-in memory will be defined in a future revision of the specification. The mechanism will define a new Header Type value and configuration registers specific to add-in memory will be specified. These definitions will allow automatic detection, sizing, and configuration of the add-in memory devices.

6.3. PCI Expansion ROMs

The PCI specification provides a mechanism where devices can provide expansion ROM code that can be executed for device-specific initialization and, possibly, a system boot function (refer to Section 6.2.5.2.). The mechanism allows the ROM to contain several different images to accommodate different machine and processor architectures. This section specifies the required information and layout of code images in the expansion ROM. Note that PCI devices that support an expansion ROM must allow that ROM to be accessed with any combination of byte enables. This specifically means that DWORD accesses to the expansion ROM must be supported.

The information in the ROMs is laid out to be compatible with existing Intel x86 Expansion ROM headers for ISA, EISA, and MC adapters, but it will also support other machine architectures. The information available in the header has been extended so that more optimum use can be made of the function provided by the adapter and so that the minimum amount of Memory Space is used by the runtime portion of the expansion ROM code.

The PCI Expansion ROM header information supports the following functions:

- A length code is provided to identify the total contiguous address space needed by the PCI device ROM image at initialization.
- An indicator identifies the type of executable or interpretive code that exists in the ROM address space in each ROM image.
- A revision level for the code and data on the ROM is provided.
- The Vendor ID and Device ID of the supported PCI device are included in the ROM.

One major difference in the usage model between PCI expansion ROMs and standard ISA, EISA, and MC ROMs is that the ROM code is never executed in place. It is always copied from the ROM device to RAM and executed from RAM. This enables dynamic sizing of the code (for initialization and runtime) and provides speed improvements when executing runtime code.

6.3.1. PCI Expansion ROM Contents

PCI device expansion ROMs may contain code (executable or interpretive) for multiple processor architectures. This may be implemented in a single physical ROM which can contain as many code images as desired for different system and processor architectures (see Figure 6-8). Each image must start on a 512-byte boundary and must contain the PCI expansion ROM header. The starting point of each image depends on the size of previous images. The last image in a ROM has a special encoding in the header to identify it as the last image.

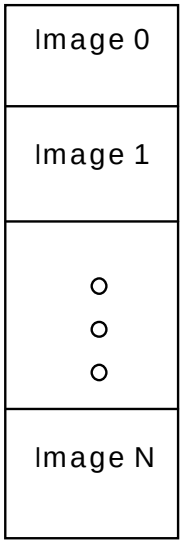


Figure 6-8: PCI Expansion ROM Structure

6.3.1.1. PCI Expansion ROM Header Format

The information required in each ROM image is split into two different areas. One area, the ROM header, is required to be located at the beginning of the ROM image. The second area, the PCI Data Structure, must be located in the first 64 KB of the image. The format for the PCI Expansion ROM header is given below. The offset is a hexadecimal number from the beginning of the image and the length of each field is given in bytes.

Extensions to the PCI Expansion ROM Header and/or the PCI Data Structure may be defined by specific system architectures. Extensions for PC-AT compatible systems are described in Section 6.3.3.

Offset	Length	Value	Description
0h	1	55h	ROM Signature, byte 1
1h	1	AAh	ROM Signature, byte 2
2h-17h	16h	xx	Reserved (processor architecture unique data)
18h-19h	2	xx	Pointer to PCI Data Structure

ROM Signature

The ROM Signature is a two-byte field containing a 55h in the first byte and AAh in the second byte. This signature must be the first two bytes of the ROM address space for each image of the ROM.

Pointer to PCI Data Structure

The Pointer to the PCI Data Structure is a two-byte pointer in little endian format that points to the PCI Data Structure. The reference point for this pointer is the beginning of the ROM image.

6.3.1.2. PCI Data Structure Format

The PCI Data Structure must be located within the first 64 KB of the ROM image and must be DWORD aligned. The PCI Data Structure contains the following information:

Offset	Length	Description
0	4	Signature, the string "PCIR"
4	2	Vendor Identification
6	2	Device Identification
8	2	Pointer to Vital Product Data
A	2	PCI Data Structure Length
C	1	PCI Data Structure Revision
D	3	Class Code
10	2	Image Length
12	2	Revision Level of Code/Data
14	1	Code Type
15	1	Indicator
16	2	Reserved

Signature — These four bytes provide a unique signature for the PCI Data Structure. The string "PCIR" is the signature with "P" being at offset 0, "C" at offset 1, etc.

Vendor Identification — The Vendor Identification field is a 16-bit field with the same definition as the Vendor Identification field in the Configuration Space for this device.

Device Identification — The Device Identification field is a 16-bit field with the same definition as the Device Identification field in the Configuration Space for this device.

Pointer to Vital Product Data — The Pointer to Vital Product Data (VPD) is a 16-bit field that is the offset from the start of the ROM image and points to the VPD. This field is in little-endian format. The VPD must be within the first 64 KB of the ROM image. A value of 0 indicates that no Vital Product Data is in the ROM image. Section 6.4 describes the format and information contained in Vital Product Data.

PCI Data Structure Length — The PCI Data Structure Length is a 16-bit field that defines the length of the data structure from the start of the data structure (the first byte of the Signature field). This field is in little-endian format and is in units of bytes.

PCI Data Structure Revision — The PCI Data Structure Revision field is an eight-bit field that identifies the data structure revision level. This revision level is 0.

Class Code — The Class Code field is a 24-bit field with the same fields and definition as the class code field in the Configuration Space for this device.

Image Length — The Image Length field is a two-byte field that represents the length of the image. This field is in little-endian format, and the value is in units of 512 bytes.

Revision Level — The Revision Level field is a two-byte field that contains the revision level of the code in the ROM image.

Code Type — The Code Type field is a one-byte field that identifies the type of code contained in this section of the ROM. The code may be executable binary for a specific processor and system architecture or interpretive code. The following code types are assigned:

Type	Description
0	Intel x86, PC-AT compatible
1	Open Firmware standard for PCI[42]
2-FF	Reserved

Indicator — Bit 7 in this field tells whether or not this is the last image in the ROM. A value of 1 indicates "last image;" a value of 0 indicates that another image follows. Bits 0-6 are reserved.

6.3.2. Power-on Self Test (POST) Code

For the most part, system POST code treats add-in PCI devices identically to those that are soldered on to the motherboard. The one exception is the handling of expansion ROMs. POST code detects the presence of an option ROM in two steps. First the code determines if the device has implemented an Expansion ROM Base Address register in Configuration Space. If the register is implemented, the POST must map and enable the ROM in an unused portion of the address space, and check the first two bytes for the AA55h signature. If that signature is found, there is a ROM present; otherwise, no ROM is attached to the device.

If a ROM is attached, POST must search the ROM for an image that has the proper code type and whose Vendor ID and Device ID fields match the corresponding fields in the device.

After finding the proper image, POST copies the appropriate amount of data into RAM. Then the device's initialization code is executed. Determining the appropriate amount of data to copy and how to execute the device's initialization code will depend on the code type for the field.

6.3.3. PC-compatible Expansion ROMs

This section describes further requirements on ROM images and the handling of ROM images that are used in PC-compatible systems. This applies to any image that specifies Intel x86, PC-AT compatible in the Code Type field of the PCI Data Structure, and any platform that is PC-compatible.

---

[42] Open Firmware is a processor architecture and system architecture independent standard for dealing with device specific option ROM code. Documentation for Open Firmware is available in the *IEEE 1275-1994 Standard for Boot (Initialization, Configuration) Firmware Core Requirements and Practices.* A related document, *PCI Bus Binding to IEEE 1275-1994*, specifies the application of Open Firmware to the PCI local bus, including PCI-specific requirements and practices. This document may be obtained using anonymous FTP to the machine *playground.sun.com* with the filename /pub/p1275/bindings/postscript/PCI.ps.

6.3.3.1. ROM Header Extensions

The standard header for PCI Expansion ROM images is expanded slightly for PC-compatibility. Two fields are added, one at offset 02h provides the initialization size for the image. Offset 03h is the entry point for the expansion ROM INIT function.

Offset	Length	Value	Description
0h	1	55h	ROM Signature byte 1
1h	1	AAh	ROM Signature byte 2
2h	1	xx	Initialization Size - size of the code in units of 512 bytes.
3h	3	xx	Entry point for INIT function. POST does a FAR CALL to this location.
6h-17h	12h	xx	Reserved (application unique data)
18h-19h	2	xx	Pointer to PCI Data Structure

6.3.3.1.1. POST Code Extensions

POST code in these systems copies the number of bytes specified by the Initialization Size field into RAM, and then calls the INIT function whose entry point is at offset 03h. POST code is required to leave the RAM area where the expansion ROM code was copied to as writable until after the INIT function has returned. This allows the INIT code to store some static data in the RAM area, and to adjust the runtime size of the code so that it consumes less space while the system is running.

The PC-compatible specific set of steps for the system POST code when handling each expansion ROM are:

1. Map and enable the expansion ROM to an unoccupied area of the memory address space.
2. Find the proper image in the ROM and copy it from ROM into the compatibility area of RAM (typically 0C0000h to 0E0000h) using the number of bytes specified by Initialization Size.
3. Disable the Expansion ROM Base Address register.
4. Leave the RAM area writable and call the INIT function.
5. Use the byte at offset 02h (which may have been modified) to determine how much memory is used at runtime.

Before system boot, the POST code must make the RAM area containing expansion ROM code read-only.

POST code must handle VGA devices with expansion ROMs in a special way. The VGA device's expansion BIOS must be copied to 0C0000h. VGA devices can be identified by examining the Class Code field in the device's Configuration Space.

6.3.3.1.2. INIT Function Extensions

PC-compatible expansion ROMs contain an INIT function that is responsible for initializing the I/O device and preparing for runtime operation. INIT functions in PCI expansion ROMs are allowed some extended capabilities because the RAM area where the code is located is left writable while the INIT function executes.

The INIT function can store static parameters inside its RAM area during the INIT function. This data can then be used by the runtime BIOS or device drivers. This area of RAM will not be writable during runtime.

The INIT function can also adjust the amount of RAM that it consumes during runtime. This is done by modifying the size byte at offset 02h in the image. This helps conserve the limited memory resource in the expansion ROM area (0C0000h - 0DFFFFh).

For example, a device expansion ROM may require 24 KB for its initialization and runtime code, but only 8 KB for the runtime code. The image in the ROM will show a size of 24 KB, so that the POST code copies the whole thing into RAM. Then when the INIT function is running, it can adjust the size byte down to 8 KB. When the INIT function returns, the POST code sees that the runtime size is 8 KB and can copy the next expansion BIOS to the optimum location.

The INIT function is responsible for guaranteeing that the checksum across the size of the image is correct. If the INIT function modifies the RAM area in any way, then a new checksum must be calculated and stored in the image.

If the INIT function wants to completely remove itself from the expansion ROM area, it does so by writing a zero to the Initialization Size field (the byte at offset 02h). In this case, no checksum has to be generated (since there is no length to checksum across).

On entry, the INIT function is passed three parameters: the bus number, device number, and function number of the device that supplied the expansion ROM. These parameters can be used to access the device being initialized. They are passed in x86 registers, [AH] contains the bus number, the upper five bits of [AL] contain the device number, and the lower three bits of [AL] contain the function number.

Prior to calling the INIT function, the POST code will allocate resources to the device (via the Base Address and Interrupt Line registers) and will complete any User Definable Features handling (refer to Section 6.7).

6.3.3.1.3. Image Structure

A PC-compatible image has three lengths associated with it, a runtime length, an initialization length, and an image length. The image length is the total length of the image and it must be greater than or equal to the initialization length.

The initialization length specifies the amount of the image that contains both the initialization and runtime code. This is the amount of data that POST code will copy into RAM before executing the initialization routine. Initialization length must be greater than or equal to runtime length. The initialization data that is copied into RAM must checksum to 0 (using the standard algorithm).

The runtime length specifies the amount of the image that contains the runtime code. This is the amount of data the POST code will leave in RAM while the system is operating. Again, this amount of the image must checksum to 0.

The PCI Data structure must be contained within the runtime portion of the image (if there is any) otherwise it must be contained within the initialization portion. Figure 6-9 shows the typical layout of an image in the expansion ROM.

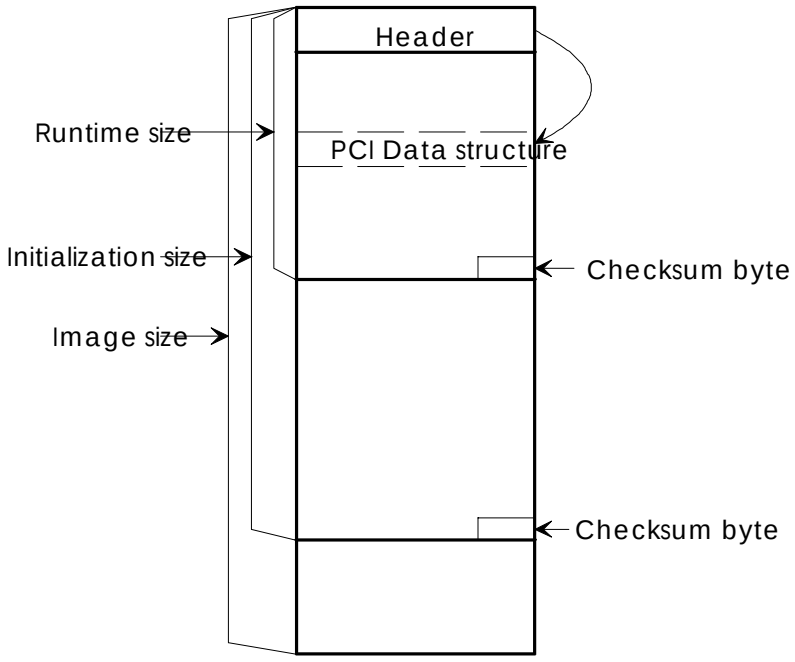


Figure 6-9: Typical Image Layout

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 PCI adapters is optional depending on the manufacturer. However, if VPD is implemented, the recommended fields should be included. Conditionally recommended fields and additional fields may also be included depending on the particular device. 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, adapter manufacturers are encouraged to provide VPD due to its inherent benefits for the adapter, system manufacturers, and for Plug and Play.

6.4.1. Importance of Vital Product Data

The availability of configuration information and other VPD beyond what is currently available in the PCI Configuration Space will help ensure a smooth transition to a true plug and play environment. The availability of VPD on electronic devices allows a system manufacturer to implement creative product delivery processes for systems and feature upgrades by providing an ability to ship accurate and complete hardware orders to

their customers. Availability of VPD on electronic devices allows the component manufacturer to obtain, and monitor information related to the field performance of their product(s) with respect to failure rates and operational compatibility with other devices in the machine. VPD can also be very effective for systems manufacturers in providing on-line technical support by identifying systems configuration information.

6.4.2. VPD Location

Vital Product Data for PCI devices is located in the device's expansion ROM. The VPD, when provided, must be located in the first 64 KB of a ROM image. A value of 0 in the pointer to VPD indicates that no VPD is in the ROM image (refer to Section 6.3.1.2 . If no expansion ROM is present on a PCI device other than VPD, there will only be one expansion ROM image (image 0), which will contain the VPD for that device. If multiple expansion ROM images are present and VPD is also provided, each image will contain VPD for the device. The VPD that describes the hardware may be a duplicate copy in each image but the VPD information that pertains to software may be different for each expansion ROM image.

6.4.3. VPD Data Structure Description

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 Tables 6-6 and 6-7.

Table 6-6: Small Resource Data Type Tag Bit Definitions

Offset	Field		
Byte 0	Tag Bit[7]	Tag Bits[6:3]	Tag Bits [2:0]
	Type = 0	Small item name	Length = n bytes
Bytes 1 to n	Actual information		

Table 6-7: Large Resource Data Type Tag Bit Definitions

Offset	Field Name
Byte 0	Value = 1xxxxxxxB (Type = 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)
Bytes 3 to n	Actual data items

Vital Product Data in PCI expansion ROM uses several of the predefined[43] tag item names and one new one defined specifically for PCI Vital Product Data. The existing item names that are used are: Compatible Device ID (0x3), Vendor Defined (0xE), and End Tag (0xF) for Small Resource Data Types; and Identifier String (0x2), and Vendor

[43] Already defined in the *Plug and Play ISA Specification, Version 1.0a* .

Defined (0x4) for Large Resource Data Types. The new large resource item name for Vital Product Data is VPD with a value of 0x10.

One or more VPD tags can be used to wrap the keywords described below. Other PnP tags such as the Identifier String (0x02) may be included to provide additional product information. The last tag must be the End Tag (0x0F) which provides a checksum across all of the Vital Product Data. The checksum is correct if the sum of all bytes in the Vital Product Data is zero. A small example of the resource data type tags used in a typical VPD is shown below:

TAG	Identifier String
TAG	VPD containing one or more VPD keywords
...	...
TAG	VPD containing one or more VPD keywords
TAG	End Tag

6.4.4. VPD Format

Information fields within a VPD resource type consist of a three-byte header followed by some amount of data (see Figure 6-10). The three-byte header contains a two-byte keyword and a one-byte length. A keyword is 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 6-10: VPD Format

VPD keywords identified below are listed in three categories: Recommended Fields, Conditionally Recommended Fields, and Additional 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.

6.4.4.1. Recommended Fields

Support of VPD is optional, but if implemented, the following fields are recommended to be included. Note that OEM preference, or manufacturing process, may cause some fields to be omitted (e.g., Serial Number).

PN	*Part Number of Assembly*	The characters are alphanumeric and represent the Part Number for this device.
FN	*FRU Part Number*	The characters are alphanumeric and represent the FRU Part Number (Field Replaceable Unit) for this device.
EC	*EC Level of Assembly*	The characters are alphanumeric and represent the Engineering Change for this board. This field should not be confused with the Revision ID in the Configuration Space Header shown in Figure 6-1 which is a vendor defined extension to the Device ID.
MN	*Manufacture ID*	This keyword may be optionally provided as an extension to the 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.
SN	*Serial Number*	The characters are alphanumeric and represent the unique board Serial Number or Inventory Identification Number.

6.4.4.1. Conditionally Recommended Fields

The conditionally recommended fields are needed only when the device or subassembly supports or includes the related functionality.

LI	*Load ID*	The Load Identification is a part of the name of the base "down load" which may be required by an adapter to load it with the software necessary to make it a functional device.
RL	*ROM Level*	This descriptor is used to identify the revision level of any non-alterable ROM code on the adapter.
RM	*Alterable ROM Level*	This descriptor is used to identify the part number of any alterable ROM code on the adapter.

NA	*Network Address*	This field is needed by those adapters that require a unique network address for a local area network. Adapters such as Token Ring, Baseband, or Ethernet use this field. Data in the field may be encoded in binary on the device but is externalized in ASCII or a hexadecimal representation of a binary value in ASCII.
DD	*Device Driver Level*	This field represents the minimum device driver level required.
DG	*Diagnostic Level*	This field represents the minimum diagnostic level required.
LL	*Loadable Microcode Level*	This field represents the minimum loadable microcode level required. If this field is not present, level zero is implied. Loadable microcode is associated with a given Card ID rather than Part Number/EC level. Therefore, as changes are made to a particular adapter, a corresponding microcode level may be required for correct operation. This field is required if loadable microcode is required for functional operation of the adapter. Its presence notifies the initialization code of this additional requirement.
VI	*Vendor ID/Device ID*	The Vendor ID and Device ID as they appear in the Configuration Space header. Only one *VI* keyword may appear per VPD Tag. Data in this field is binary encoded.
FU	*Function Number*	This field identifies which function in a multifunction device the VPD data applies to. Only one *FU* keyword may appear per VPD Tag. Data in this field is binary encoded.
SI	*Subsystem Vendor ID/Subsystem ID*	The Subsystem Vendor ID and Subsystem ID as they should appear in the type 00h Configuration Space header. Data in this field is binary encoded.

6.4.4.2. Additional Fields

Z0-ZZ	*User/Product Specific*	These fields are available for device specific data for which no unique keyword has been defined.

6.4.5. VPD Example

VPD keywords are wrapped by one or more large resource VPD tags. Each VPD item is identified by a two-character ASCII code. The two-character VPD keyword is followed by a one-byte data length field. The binary data length value (in bytes) identifies the length of the keyword data only. Table 6-8 is an example of a VPD. Hexadecimal digits are packed two per byte.

Table 6-8: VPD Example

Offset	Item	Value
0	Large Resource Type ID String Tag (0x02)	0x82
1	Length	0x0021
3	Data	"ABC Super-Fast Widget Controller"
36	Large Resource Type VPD Tag (0x10)	0x90
37	Length	0x0033
39	VPD Keyword	"PN"
41	Length	0x08
42	Data	"6181682A"
50	VPD Keyword	"EC"
52	Length	0x0A
53	Data	"4950262536"
63	VPD Keyword	"SN"
65	Length	0x08
66	Data	"00000194"
74	VPD Keyword	"FN"
76	Length	0x06
77	Data	"135722"
83	VPD Keyword	"MN"
85	Length	0x04
86	Data	"1037"
90	Large Resource Type VPD Tag (0x10)	0x90
91	Length	0x000A
93	VPD Keyword	"DG"
95	Length	0x02
96	Data	"01"
98	VPD Keyword	"DD"
100	Length	0x02
101	Data	"01"
103	Small Resource Type End Tag (0xF)	0x79
104	Data	Checksum

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 sharable. 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. User Definable Configuration Items

This section describes the mechanism to support the configuring of PCI adapters that have User Definable Features (UDFs) using system configuration mechanisms (such as the EISA Configuration Utility). UDFs are defined to be device configuration items that are dependent on the environment into which the device is installed and whose settings cannot be automatically determined by hardware or system software. For example, the token ring speed setting for token ring network devices will be dependent on the specific token ring network into which the device is installed. Therefore, the default value of these configuration items may prevent successful system boot given the environment in which it is installed and the user may be required to insure a proper configuration. UDFs do not apply to devices that have a common compatible default configuration, such as VGA compatible graphics adapters, since a successful system boot can be achieved using the device's default configuration.

6.7.1. Overview

Device UDFs are described in a text based "file" that is supplied with an adapter. This file will be referred to as a PCI Configuration File, or PCF. The PCF will specify to the system configuration mechanism the device specific user definable features (UDFs). Adapters that do not support device specific UDFs are not required to supply a PCF.

Adapter vendors are required to supply a separate PCF for each adapter function that supports device specific UDFs. The PCF can be supplied with an adapter on a 1.44 MB diskette formatted with the PC/MS-DOS File Allocation Table (FAT) format. The filename for the file containing the PCF must be XXXXYYYY.PCF, where XXXX is the two-byte Vendor ID as specified in the device's Configuration Space header (represented as hexadecimal digits), and YYYY is the two-byte Device ID as specified in the device's Configuration Space header (represented as hexadecimal digits). The file must be in the root directory on the diskette.

A function on an adapter is required to indicate that it has user definable features via the UDF_Supported bit. This read-only UDF_Supported bit resides in the Status Register and will be set when a device function has device specific configuration selections that must be presented to the user. Functions that do not support user selectable configuration items would not implement this bit, and, therefore, return a 0 when read. Refer to Section 6.2.3. for a description of the Status register.

For devices where the UDF_Supported bit is set, system startup and/or configuration software will recognize the function on the adapter as one that supports user definable features. Systems are not required to be capable of interpreting a PCF. For such systems, the user will need to rely on a vendor supplied device specific configuration utility if the user requires the ability to alter user definable features of that device.

Systems that choose to support interpreting PCFs are also responsible for supplying non-volatile storage (NVS) to hold the device specific configuration selected by the user. In this scenario, system POST software will, at system boot time, copy the appropriate values for each PCI adapter from the non-volatile storage to the appropriate Configuration Space registers for each function. The mechanism for interpreting a PCF, presenting the information to the user, and storing the selections in the NVS is system specific. Note that when sizing NVS for a given system, the number of adapters supported, the number of functions per adapter, and the number of bytes of configuration information per function must be analyzed. In addition, the system will need to store

enough overhead information such that POST knows what address of which device/function each configuration byte will be written to, masked appropriately as specified in the PCF. It is recommended that system non-volatile storage be sized such that an average of 32 bytes of configuration data (potentially non-contiguous) will be written to each adapter device function. In addition, vendors should design adapters such that they do not require more than 32 bytes of configuration information per function as a result of PCF specified configuration options.

6.7.2. PCF Definition

6.7.2.1. Notational Convention

The PCF contains ISO Standard 8859-1 character set text, commonly referred to as Code Page 850. The text includes keywords that aid the system configuration mechanism's interpretation of the PCF information, as well as provides generic text representing device specific information. All text is case insensitive, unless otherwise noted. White space, including spaces, tabs, carriage returns, and linefeeds, is ignored outside of quoted strings. All PCF selections must be for device specific configuration options and be targeted for the device specific portion (192 bytes) of the function's Configuration Space. The PCF cannot be used for requesting allocation of system level resources such as interrupt assignments, or memory, I/O or expansion ROM address allocations. The PCF cannot request writes to the PCI Configuration Space Header (addresses less than 40h). This must be enforced by the system configuration mechanism.

System configuration software will use the PCF to present to the user the device specific configuration options. User selections will be stored in system non-volatile storage, presumably as values to be written to device specific Configuration Space addresses. POST software will use the information stored in non-volatile memory to write appropriate configuration settings into each device's Configuration Space. The device's logic can use the information as loaded in Configuration Space, or require its expansion ROM logic or device driver software to copy the device specific Configuration Space values into appropriate I/O or memory based device registers at system initialization. In addition, the device can choose to alias the device specific Configuration Space registers into appropriate I/O or memory based device registers if needed. Any configuration information required to be accessible after device initialization should not be accessible exclusively via Configuration Space.

6.7.2.1.1. Values and Addresses

A value or address can be given in hexadecimal, decimal, or binary format. The radix, or base identifier, is specified by attaching one of the following characters to the end of the value or address:

H or h - Hexadecimal

D or d - Decimal

B or b - Binary

The radix character must be placed immediately after the value, with no space in between. If no radix is specified, decimal is assumed.

Example: 1FOOh

Hexadecimal numbers beginning with a letter must have a leading zero.

Example: 0C000h

6.7.2.1.2. Text

Text fields contain information that is to be presented to the user. These fields are free form and are enclosed in quotation marks. These text fields can be tailored to a specific international market by using the Code Page 850 character set to support international languages (see the LANG statement description below). Text field maximum lengths are given for each instance. Text fields can contain embedded tabs, denoted by \t, and embedded linefeeds, denoted by \n. Quotation marks and backslashes can also be placed in the text using \" and \\ respectively.

Embedded tabs are expanded to the next tab stop. Tab length is eight characters (tab stops are located at 9, 17, 25, etc.).

6.7.2.1.3. Internal Comments

Comments can be embedded in the PCF for annotational purposes. Comments are not presented to the user. Comments can be placed on separate lines, or can follow other PCF statements. Comments begin with a semi-colon (;) and continue through the end of the line.

6.7.2.1.4. Symbols Used in Syntax Description

This description of the PCF syntax uses the following special symbols:

[]	The item or statement is optional.
x|y	Either x or y is allowed.

6.7.2.2. PCI Configuration File Outline

A PCF is structured as follows:

Device Identification Block

Function Statement Block(s)

[Device Identification Block

Function Statement Block(s)]

The Device Identification Block identifies the device by name, manufacturer, and ID. The PCF must begin with this block.

The Function Statement Blocks define the user presentable configuration items associated with the device.

The Device Identification Block and Function Statement Block set can optionally be repeated within the PCF file to support multiple languages.

6.7.2.2.1. Device Identification Block

The Device Identification Block within the PCF is defined as follows:

```
BOARD
      ID="XXXXYYYY"
      NAME="text"
      MFR="text"
      SLOT=PCI
      [VERSION=value]
      [LANG=XXX]
```

The BOARD statement appears at the beginning of each PCF. This statement, along with the other required statements, must appear before the optional statements contained in brackets []. The statements should occur in the order shown.

ID is a required statement containing the Vendor and Device IDs, XXXXYYYY, where XXXX is the two-byte Vendor ID as specified in the device's Configuration Space header (represented as hexadecimal digits), and YYYY is the two-byte Device ID as specified in the device's configuration header (represented as hexadecimal digits). The ID must contain eight characters and must be placed within quotation marks.

NAME is a required statement that identifies the device. Vendor and product name should be included. A maximum length of 90 characters is allowed. The first 55 characters are considered significant (i.e., only the first 55 characters will be shown if truncation or horizontal scrolling is required).

MFR is a required statement that specifies the board manufacturer. A maximum length of 35 characters is allowed.

SLOT=PCI is a required statement that identifies the device as PCI. This is included to assist configuration utilities that must also parse EISA or ISA CFG files.

VERSION is an optional statement that specifies the PCF standard that this PCF was implemented to. The syntax described by this section represents version 0. This statement allows future revisions of the PCF syntax and format to be recognized and processed accordingly by configuration utilities. Version 0 will be assumed when the VERSION statement is not found in the Device Identification Block.

LANG is an optional statement that specifies the language used within the quote enclosed text found within the given Device Identification Block/Function Statement Block set. When no LANG statement is included, then the default language is English. XXX can have the following values which are defined in the IS0-639 standard[44]:

CS	Czech
DA	Danish
NL	Dutch
EN	English (default)
FI	Finnish
FR	French
DE	German
HU	Hungarian
IT	Italian
NO	Norwegian
PL	Polish
PT	Portuguese
SK	Slovak
ES	Spanish
SV	Swedish

6.7.2.2.2. Function Statement Block

Function Statement Blocks define specific configuration choices to be presented to the user. A Function Statement Block is defined as follows:

```
FUNCTION="text"
      [HELP="text"]
      [SHOW=[YES|NO|EXP]}
      Choice Statement Block
      .
      .
      .
      [Choice Statement Block]
```

The FUNCTION statement names a function of the device for which configuration alternatives will be presented to the user. A maximum of 100 characters is allowed for the function name.

HELP is an optional text field containing additional information that will be displayed to the user if the user requests help while configuring the function. This text field can contain a maximum of 600 characters.

SHOW is an optional statement used to specify whether this function is displayed or not. YES is the default value and indicates that the function will be displayed. NO indicates that the function will never be displayed. EXP indicates that the function will be displayed only when the system is in expanded mode. This feature allows the configuration utility to generate INIT statements that are not presented to the user (SHOW=NO). In addition, more advanced features can be hidden from the casual user (SHOW=EXP).

Each Choice Statement Block names a configuration alternative for the function and lists the register addresses, sizes, and values needed to initialize that alternative. Each

[44] The ISO-639 standard can be obtained from ANSI Customer Service, 13th Floor, 11 West 42nd Street, New York City, New York 10036.

Function Statement Block must contain at least one Choice Statement Block. The first choice listed for a given function will be the default choice used for automatic configuration.

6.7.2.2.2.1. Choice Statement Block

```
CHOICE = "text"
      [HELP="text"]
      INIT Statement
      .
      .
      .
      [INIT Statement]
```

CHOICE statements are used to indicate configuration alternatives for the function. Each FUNCTION must have at least one CHOICE statement, and can have as many as necessary. A maximum of 90 characters is allowed for the choice name.

HELP is an optional text field containing additional information that will be displayed to the user if the user requests help with the CHOICE. This text field can contain a maximum of 600 characters.

A Choice Statement Block can contain one or more INIT statements. INIT statements give the register addresses and values needed to initialize the configuration alternative named by the CHOICE statement.

6.7.2.2.2.1.1. INIT Statements

INIT=PCI(*address*) [BYTE|WORD|DWORD] *value*

INIT statements provide the register addresses and values needed to initialize the device's vendor specific registers.

The PCI keyword is used to indicate that this is a PCI INIT statement. This is included to assist configuration utilities that must also parse EISA or ISA CFG files.

Address is the register's offset in the PCI Configuration Space. This address value must be within the 192 bytes of device specific Configuration Space (offsets 64-255).

An optional BYTE, WORD, or DWORD qualifier can be used to indicate the size of the register. The default is BYTE.

Value gives the value to be output to the register. Bit positions marked with an "r" indicate that the value in that position is to be preserved. The "r" can only be used as a bit position in a binary value, or as a hex digit (four bit positions) in a hex value. The length of the value must be the same as the data width of the port: 8, 16, or 32 bits.

Examples:

```
INIT = PCI(58h) 11110000b
INIT = PCI(5Ah) 0000rr11b
INIT = PCI(0A6h) WORD R8CDh
INIT = PCI(48h) WORD RR0000001111RR11b
```

6.7.3. Sample PCF

```
BOARD
ID="56781234"          ; Vendor is 5678h, Device is 1234h
                       ; Filename would be "56781234.PCF"
NAME=           "Super Cool Widget PCI Device"
MFR=            "ABC Company"
SLOT=            PCI
VERSION=         0

FUNCTION="Type of Widget Communications"
HELP="This choice lets you select which type of
communication you want this device to use."
  CHOICE="Serial"
   INIT=PCI(45h)         rrr000rrb  ;Default size is BYTE
   INIT=PCI(8Ch)         DWORD       0ABCDRRRRh
  CHOICE="Parallel"
    INIT=PCI(45h)        rrr010rrb
    INIT=PCI(8Ch)        DWORD       1234abcdh
  CHOICE="Cellular"
    INIT=PCI(45h)        rrr100rrb
    INIT=PCI(8Ch)        DWORD       5678abcdh

FUNCTION="Communication Speed"
  CHOICE="4 Mbit/Sec"       INIT=PCI(56h) WORD R12Rh
  CHOICE="16 Mbit/Sec"      INIT=PCI(56h) WORD R4CRh
  CHOICE="64 Gbit/Sec"      INIT=PCI(56h) WORD R00Rh

FUNCTION="Enable Super Hyper Turbo Mode"
HELP="Enable Super Hyper Turbo Mode only if the 64 Gbit
Speed has been selected."
  CHOICE="No"     INIT=PCI(49h)      rrrrr0rrb
  CHOICE="Yes"    INIT=PCI(49h)      rrrrr1rrb

FUNCTION="Widget Host ID"
  CHOICE="7"      INIT=PCI (9Ah)     rrrrr000b
  CHOICE="6"      INIT=PCI (9Ah)     rrrrr001b
  CHOICE="5"      INIT=PCI (9Ah)     rrrrr010b
  CHOICE="4"      INIT=PCI (9Ah)     rrrrr011b
```

Chapter 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 purpose of 66 MHz PCI is to provide connectivity to very high bandwidth devices in applications such as HDTV, 3D graphics, and advanced video. 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. If set, this bit signifies that the device is capable of operating in 66 MHz mode.

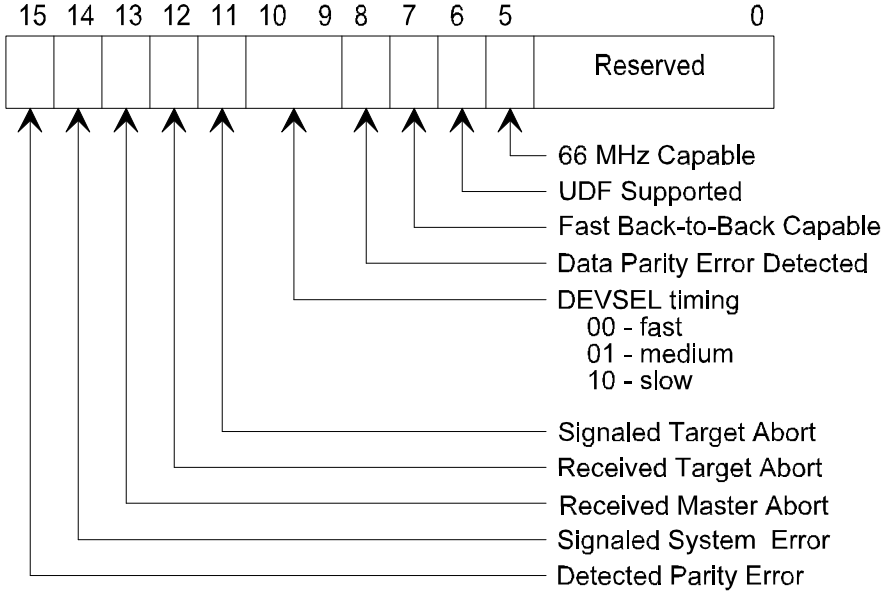


Figure 7-1: Status Register Layout

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.[45]

[45] Configuration software may identify all agent capabilities when it probes the agents by checking the 66MHZ_CAPABLE bit in all Status registers. 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.

7.5. Protocol

7.5.1. 66MHZ_ENABLE (M66EN) Pin Definition

An existing ground pin on the 33 MHz PCI connector (pin 49, Side B) has been designated **M66EN**. A 66 MHz PCI planar 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 planar-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 may use **M66EN** as an input. If a 66 MHz PCI agent can run without any knowledge of the speed, it may leave **M66EN** disconnected.

Note that pin 49, Side B is already bused as a ground in 33 MHz PCI systems. Refer to the *PCI Compliance Checklist* for more information.

Table 7-1: Bus and Agent Combinations

Bus 66MHZ_CAPABLE[46]	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[47]
1	0	33 MHz PCI agent located on a 66 MHz PCI bus[47]
1	1	66 MHz PCI agent located on a 66 MHz PCI bus

7.5.2. Latency

The 66 MHz PCI bus is intended for low latency devices. It is required that the first data phase of a read transaction not exceed 16 clocks. For further information, system designers should refer to the *PCI Multimedia Design Guide*.

[46] The bus 66MHZ_CAPABLE status bit is located in a bridge.

[47] This condition may cause the configuration software to generate a warning to the user stating that the card is installed in an inappropriate socket and should be relocated.

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 boards, 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 with PCI, 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 7-2.

$$T_{cyc} \geq T_{val} + T_{prop} + T_{skew} + T_{su}$$

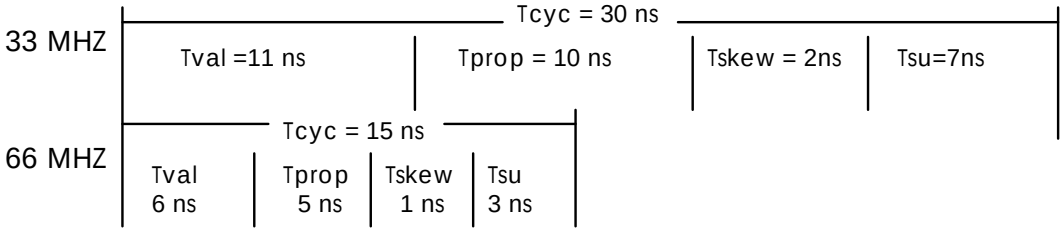


Figure 7-2: 33 MHz PCI vs. 66 MHz PCI Timing

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 planar segment must use the PCI 3.3V keyed connector. Therefore, 66 MHz PCI planar segments accept either 3.3V or universal add-in boards; 5V add-in boards 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
I_{OH} (min)	Output high Minimum current	$V_{out} = 0.3V_{cc}$	$-12V_{cc}$	-	mA	1
I_{OH} (max)	Output high Maximum current	$V_{out} = 0.7V_{cc}$	-	$-32V_{cc}$	mA	
I_{OL} (min)	Output low Minimum current	$V_{out} = 0.6V_{cc}$	$16V_{cc}$	-	mA	1
I_{OL} (max)	Output low Maximum current	$V_{out} = 0.18V_{cc}$	-	$38V_{cc}$	mA	
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
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	
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

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#, 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 7-8) is optional. The designer may elect to meet this parameter with an unloaded output per revision 2.0 of the PCI specification. However, adherence to both maximum and minimum parameters is now required (the maximum is no longer simply a guideline). The V/I curves define the minimum and maximum output buffer drive strength. These curves should be interpreted as traditional DC curves with one exception; from a quiescent or steady state condition, 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 time specification), the more time allowed for logic delay inside the part.

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 boards, compliance with the clock specification is measured at the add-in board component, not at the connector slot. Figure 7-3 shows the clock waveform and required measurement points for 3.3V signaling environments. Table 7-3 summarizes the clock specifications.

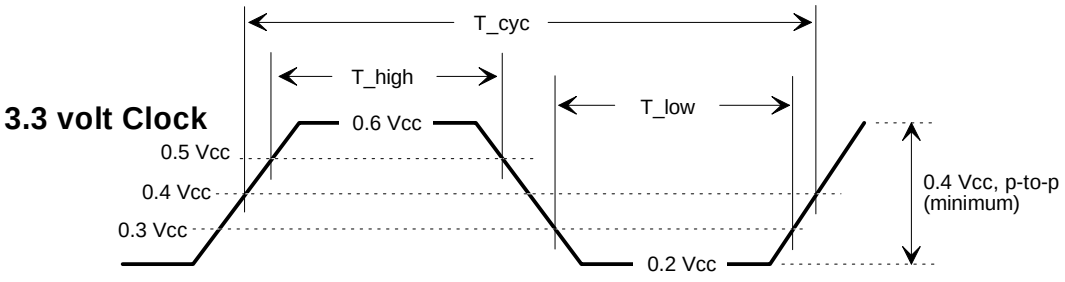


Figure 7-3: 3.3V Clock Waveform

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

NOTES:

1. In general, all 66 MHz PCI components must work with any clock frequency up to 66 MHz. Device operational parameters at frequencies 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 planar only. For clock frequencies between 33 MHz and 66 MHz, the clock frequency may not change except in conjunction with a PCI reset.
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 7-3. Clock slew rate is measured by the slew rate circuit shown in Figure 7-8.
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.

7.6.4.2. Timing Parameters

Table 7-4: 66 MHz and 33 MHz Timing Parameters

Symbol	Parameter	66 MHz		33 MHz[7]		Units	Notes
		Min	Max	Min	Max		
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 Set up Time to **CLK** - bused signals	3		7		ns	3, 4
$t_{su}(ptp)$	Input Set up 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
$t_{rst-clk}$	Reset Active Time after **CLK** stable	100		100		µs	5
$t_{rst-off}$	Reset Active to output float delay		40		40	ns	5, 6
t_{rrsu}	**REQ64#** to **RST#** setup time	$10T_{cyc}$		$10T_{cyc}$		ns	
t_{rrh}	**RST#** to **REQ64#** hold time	0	50	0	50	ns	

NOTES:

1. See the timing measurement conditions in Figure 7-4. 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 7-8. Maximum times are measured with the load circuit shown in Figures 7-6 and 7-7.
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 7-5.
5. **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.

7.6.4.3. Measurement and Test Conditions

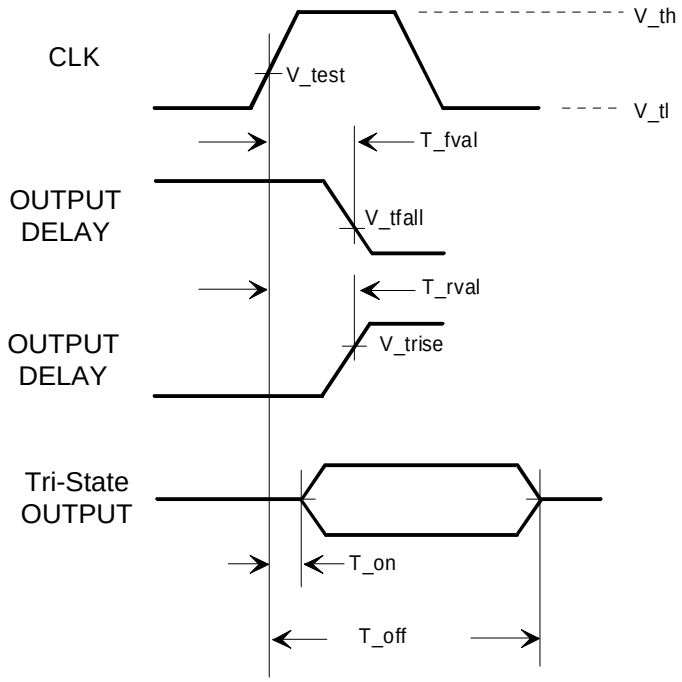


Figure 7-4: Output Timing Measurement Conditions

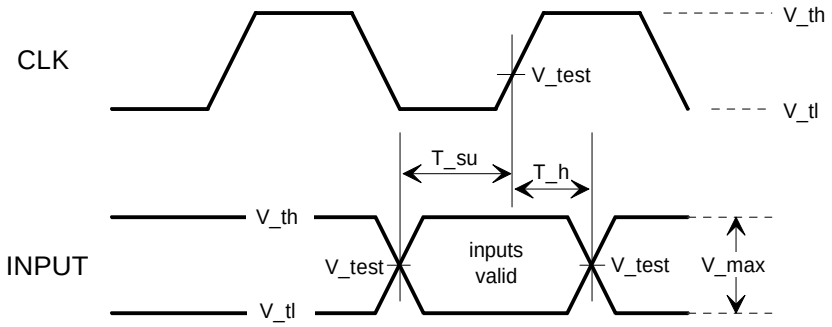


Figure 7-5: Input Timing Measurement Conditions

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 25Ω 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 7-8. Input signal slew rate will be measured between $0.3V_{cc}$ and $0.6V_{cc}$.

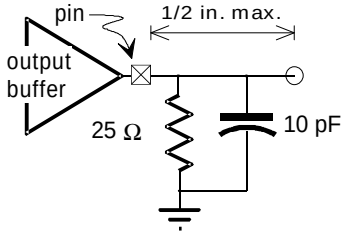


Figure 7-6: T_{val}(max) Rising Edge

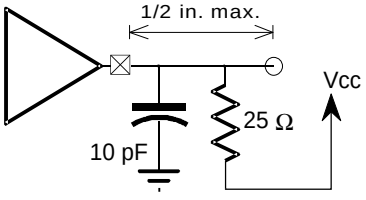


Figure 7-7: T_{val}(max) Falling Edge

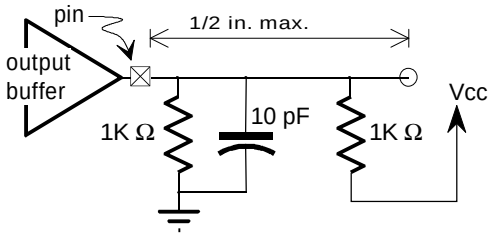


Figure 7-8: T_{val} (min) and Slew Rate

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 may modify the suggested pinout shown in Figure 4-10 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 planar and add-in board 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 7-9 illustrates the recommended method for routing the 66 MHz PCI clock signal.

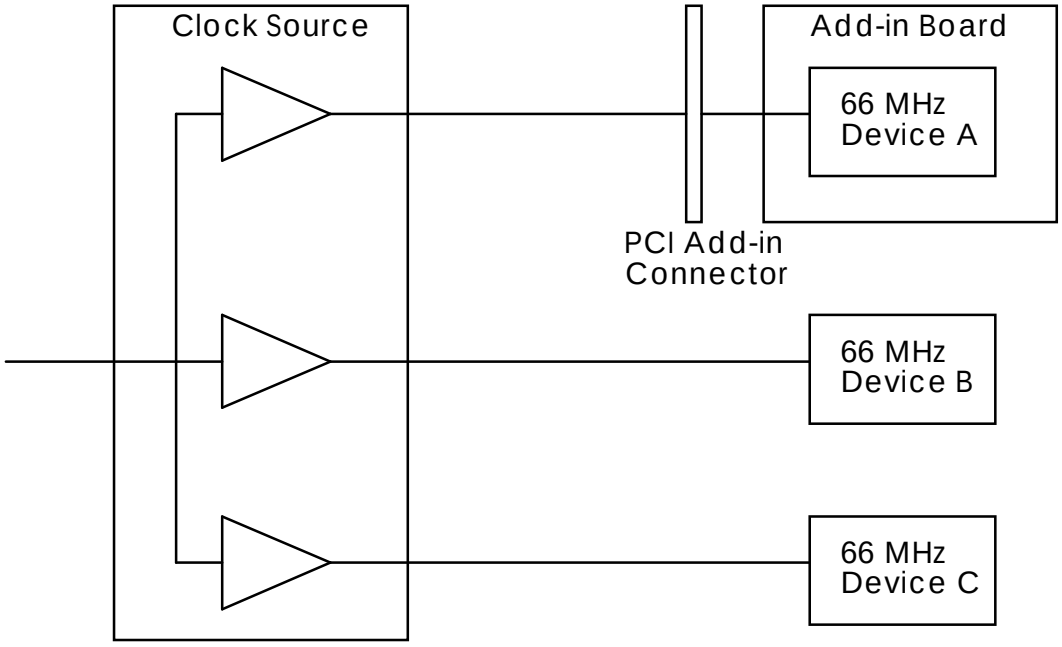


Figure 7-9: Recommended Clock Routing

7.7. System (Planar) 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 7-10. The maximum skew is measured between any two components,[48] not between connectors. To correctly evaluate clock skew, the system designer must take into account clock distribution on the add-in board as specified in Section 4.4.

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

[48] The system designer may need to 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.

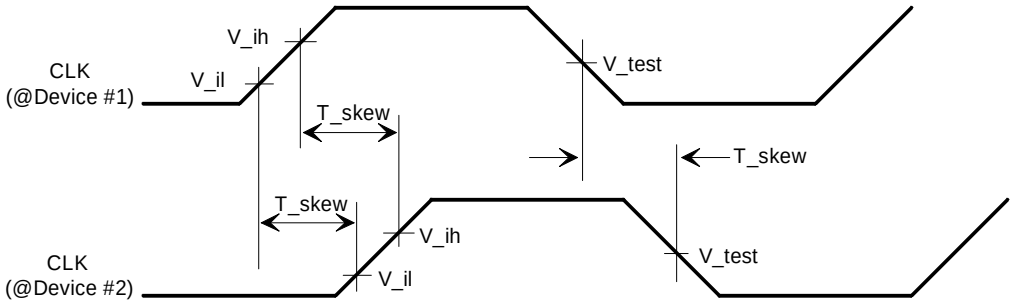


Figure 7-10: Clock Skew Diagram

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 planar, 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

When computing a total 66 MHz PCI load model, designers must pay careful attention to maximum trace length and loading of add-in boards. Refer to Section 4.4.3. Also, the maximum pin capacitance of 10 pF must be assumed for add-in boards, whereas the actual pin capacitance may be used for planar devices.

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.

T_{prop} is measured as shown in Figure 7-11. It begins at the time the output buffer would have crossed the threshold point (V_{trise} or V_{tfall}) had it been driving the specified T_{val}(max) load. It ends when the slowest input crosses V_{ih} (high going) or V_{il} (low going) and never rings back across that level again. Care must be taken in evaluating this exact timing point. Note that input buffer timing is tested with a certain amount of overdrive (past V_{ih} and V_{il}). This may be needed to guarantee input buffer timings. For example, the input may not be valid (and consequently T_{prop} time is still running) unless it goes up to V_{th} and does not ring back across V_{ih}.

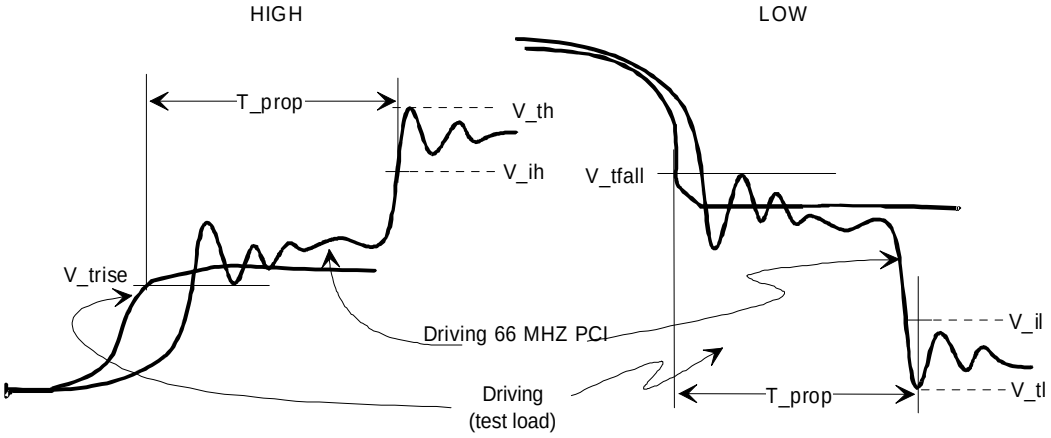


Figure 7-11: Measurement of T_{prop}

(refer to Table 7-5 for parameter values)

The relevant timing budget can be expressed by the equation:

$$T_{cyc} \geq T_{val} + T_{prop} + T_{su} + T_{skew}$$

The following table compares PCI timing budgets at various speeds.

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	

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.

7.7.6. Physical Requirements

7.7.6.1. Routing and Layout of Four Layer Boards

Refer to Section 4.3.6.1.

7.7.6.2. Planar 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

With the exception of the **M66EN** pin (side B, pin 49), the standard PCI connector pinout is used. Refer to Section 4.3.7 for planar 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[49] 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 may be connected to the **M66EN** input pin of components located on the same logical bus segment of the system planar. This signal is static, there is no stub length restriction.

To complete an AC return path, a 0.01 μF capacitor shall be located, within 0.25" of the **M66EN** pin, to each such add-in connector and shall decouple the **M66EN** signal to ground. Any attached component or installed add-in board that is not 66 MHz capable, must pull the **M66EN** net to the V_{il} input level. The remaining components, add-in boards, and the logical bus segment clock resource are, thereby, signaled to operate in 33 MHz mode.

7.8. Add-in Board Specifications

Refer to Section 4.4.

With the exception of the **M66EN** pin (side B, pin 49), the standard PCI edge-connector pinout is used. Refer to Section 4.4.1 for the add-in board edge-connector. 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 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 board shall operate in the 33 MHz mode.

[49] As a general rule, there will be only one such connector, but more than one may be possible in certain cases.

Appendix A
Special Cycle Messages

Special Cycle message encodings are defined in this appendix. The current list of defined encodings is small, but it is expected to grow. Reserved encodings should not be used.

Message Encodings

AD[15::0]	Message Type
0000h	SHUTDOWN
0001h	HALT
0002h	x86 architecture-specific
0003h 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.

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.

Future Encodings

PCI SIG member companies that require special encodings outside the range of currently defined encodings should send a written request to the PCI SIG Steering Committee. The Steering Committee will allocate and define special cycle encodings based upon information provided by the requester specifying usage needs and future product or application direction.

Appendix 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 do 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.).

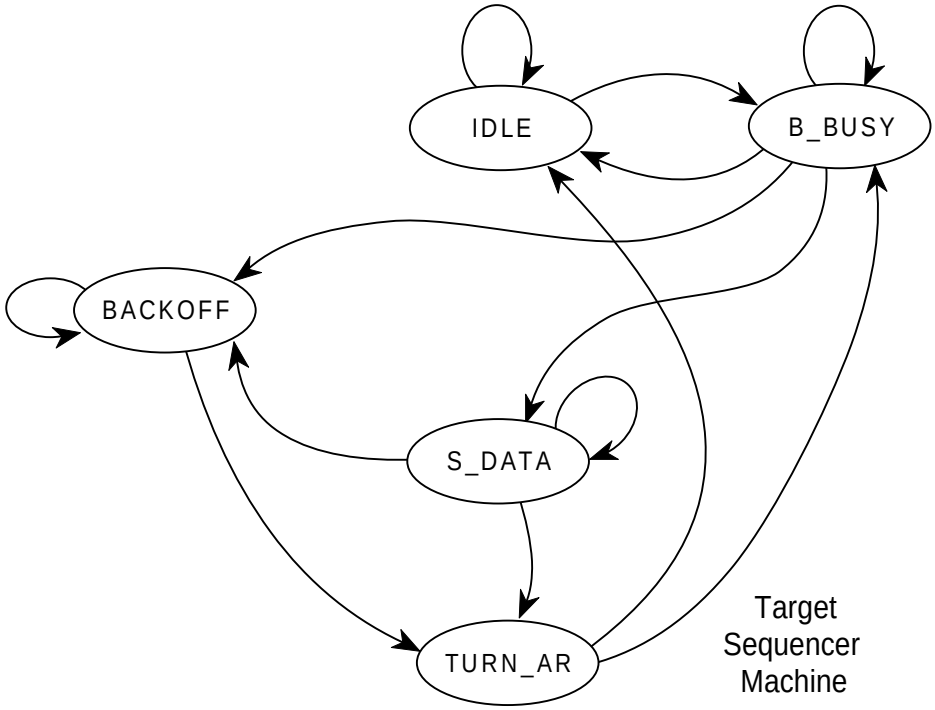
The bus interface consists of two parts. The first is the bus sequencer that performs the actual bus operation. The second part is the backend or hardware application. In a master, the backend generates the transaction and provides the address, data, command, Byte Enables, and the length of the transfer. 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.

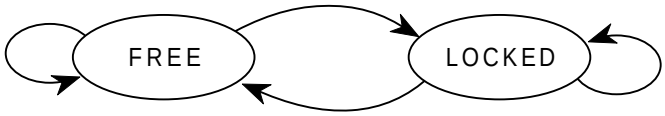
The state machine equations assume a logical operation where "*" is an AND function and has precedence over "+" which is an OR function. Parentheses have precedence over both. The "!" character is used to indicate the NOT of the variable. In the state machine equations, the PCI **SIGNAL**s represent 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.





Target LOCK Machine

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) * (FREE + LOCKED * L_lock#)
goto BACKOFF	*if (!**FRAME#** + !**IRDY#**) * Hit * (Term * !Ready + LOCKED * !L_lock#)

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#**

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:[50]

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.

[50] 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 * Tar_dly))
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 stop.
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_target = 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.

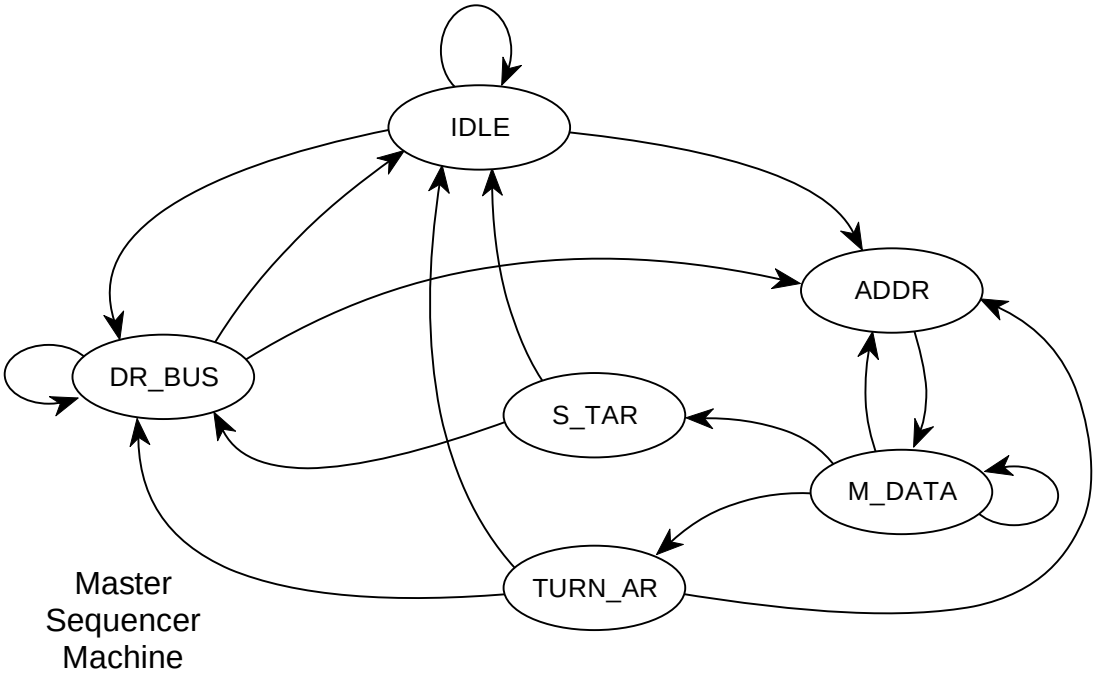
B_BUSY is a state where the agent waits for the current transaction to complete and the bus to return to the Idle state. B_BUSY is useful for devices that do slow address decode or perform subtractive decode. If the target does neither of these two options, the path to S_DATA and BACKOFF may be removed. 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.

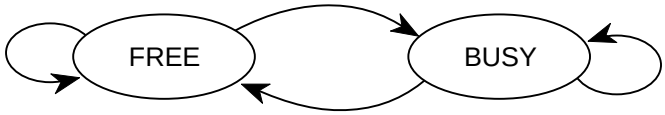
BACKOFF is where the target goes after it asserts **STOP#** and waits for the master to deassert **FRAME#**.

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.



Master Sequencer Machine



Master LOCK Machine

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#**

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.

Appendix C
Operating Rules

This appendix is not a complete list of rules of the specification or 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.

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#**, **SBO#**, **SDONE**, **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 the **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 unto 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 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 or each completed data phase and remain valid every clock during the entire data phase regardless if wait states are inserted and 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 sampled asserted when the access is a configuration command. **IDSEL** is non-deterministic at any other time.

6. **RST#, INTA#**, **INTB#**, **INTC#**, and **INTD#** are not qualified or synchronous.

Master Signals

7. A transaction starts when **FRAME#** is sampled 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.

9. When **FRAME#** and **IRDY#** are deasserted, the transaction has ended.

10. 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. The master 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.

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.

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.

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)
    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.

Arbitration

21. When **FRAME#** and **IRDY#** are deasserted and **GNT#** is asserted, the agent may start an access.
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[51] 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.

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#**.
26. The target is required to complete a subsequent data phase within 8 clocks from the completion of the previous data phase.
27. A master is required to assert its **IRDY#** within 8 clocks for any given data phase (initial and subsequent).

Exclusive Access

28. Ownership of **LOCK#** maybe obtained when **LOCK#** is deasserted, the bus is Idle and the agent's **GNT#** is asserted.
29. **LOCK#** must be released when the master is terminated with Retry before data is transferred.
30. **LOCK#** is owned and only driven by a single agent and may be retained when the bus is released.

[51] 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.

31. A target that supports **LOCK#** on PCI must adhere to the following rules:
    a. The target of an access locks itself when **LOCK#** is deasserted during the address phase.
    b. Once lock is established, the target remains locked until it samples both **FRAME#** and **LOCK#** deasserted together or it signals Target-Abort.
    c. Guarantee exclusivity to the owner of **LOCK#** (once lock is established) of a minimum of 16 bytes (aligned) of the resource.[52] This includes accesses that do not originate on PCI for multiport devices.
32. A master that uses **LOCK#** on PCI must adhere to the following rules:
    a. A master can access only a single resource during a lock operation.
    b. A lock cannot straddle a device boundary.
    c. Sixteen bytes (aligned) is the maximum resource size a master can count on as being exclusive during a lock operation. An access to any part of the 16 bytes locks the entire 16 bytes.
    d. First transaction of lock operation must be a read transaction.
    e. **LOCK#** must be asserted the clock following the address phase and kept asserted to maintain control.
    f. **LOCK#** must be released if Retry is signaled before a data phase has completed and the lock has not been established.[53]
    g. **LOCK#** must be released whenever an access is terminated by Target-Abort or Master-Abort.
    h. **LOCK#** must be deasserted for a minimum of one Idle cycle between consecutive lock operations.

Device Selection

33. A target must do a full decode before driving/asserting **DEVSEL#**, or any other target response signal.
34. A target must assert **DEVSEL#** (claim the transaction) before it is allowed to issue any other target response.
35. 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.
36. A PCI device is a target of a type 0 configuration command (read or write) only if its **IDSEL** is asserted and **AD[1::0]** are "00" during the address phase of the command.

---

[52] The maximum is the complete resource.

[53] Once lock has been established, the master retains ownership of **LOCK#** when terminated with Retry or Disconnect.

Parity

37. 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.
38. 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#**).

Appendix D
Class Codes

This appendix describes the current Class Code encodings. This list may be enhanced at any time. Companies wishing to define a new encoding should contact the PCI SIG. All unspecified values are reserved for SIG assignment.

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.

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 below).
	02h	00h	Floppy disk controller.
	03h	00h	IPI bus controller.
	04h	00h	RAID controller.
	80h	00h	Other mass storage controller.

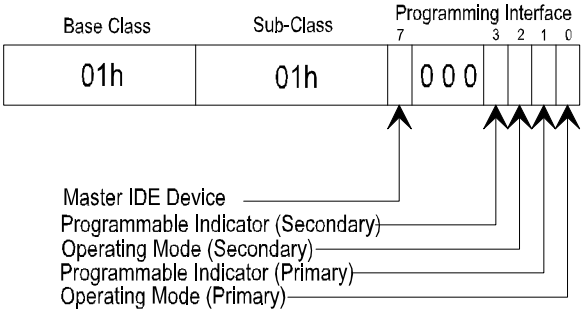


Figure D-1: Programming Interface Byte Layout for IDE Controller Class Code

The SIG document *PCI IDE Controller Specification* completely describes the layout and meaning of bits 0 thru 3 in the Programming Interface byte. The document *Bus Master Programming Interface for IDE ATA Controllers* describes the meaning of bit 7 in the Programming Interface byte. This document can be obtained via FAX by calling (408)741-1600 and requesting document 8038.

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.
	80h	00h	Other network controller.

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.

Base Class	Sub-Class	Interface	Meaning
03h	00h	00000000b	VGA-compatible controller. Memory addresses 0A0000h thru 0BFFFFh. I/O addresses 3B0h to 3BBh and 3C0h to 3DFh and all aliases of these addresses.
		00000001b	8514-compatible controller -- 2E8h and its aliases, 2EAh-2EFh.
	01h	00h	XGA controller.
	80h	00h	Other display controller.

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.
	80h	00h	Other multimedia device.

Base Class 05h

This base class is defined for all types of memory controllers (refer to Section 6.2.5.3). 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.

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. There are no register-level programming interfaces 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.
	05h	00h	PCMCIA bridge.
	06h	00h	NuBus bridge.
	07h	00h	CardBus bridge.
	80h	00h	Other bridge device.

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.
	01h	00h	Parallel port.
		01h	Bidirectional parallel port.
		02h	ECP 1.X compliant parallel port.
	80h	00h	Other communications device.

Base Class 08h

This base class is defined for all types of generic system peripherals. Several sub-class values are defined, each 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.
	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.
	80h	00h	Other system peripheral.

Base Class 09h

This base class is defined for all types of input devices. Several sub-class values are defined. No specific register-level programming interfaces are defined.

Base Class	Sub-Class	Interface	Meaning
09h	00h	00h	Keyboard controller.
	01h	00h	Digitizer (pen).
	02h	00h	Mouse controller.
	80h	00h	Other input controller.

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.

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.
	40h	00h	Co-processor.

Base Class 0Ch

This base class is defined for all types of serial bus controllers. Several sub-class values are defined. There are no specific register-level programming interfaces defined.

Base Class	Sub-Class	Interface	Meaning
0Ch	00	00h	FireWire (IEEE 1394).
	01h	00h	ACCESS.bus.
	02h	00h	SSA.
	03h	00h	Universal Serial Bus (USB).
	04h	00h	Fibre Channel.

Appendix 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 which 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.

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.

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 causes 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.

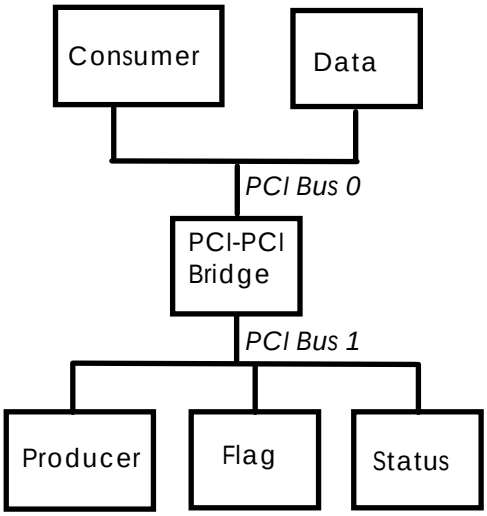


Figure E-1: Example Producer - Consumer Model

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 flag can be read by the Consumer. 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 flag is read by the Consumer. 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.

Summary of PCI Ordering Rules

Following is a summary of the general PCI ordering rules presented in Section 3.2.5. These rules apply to all PCI transactions, whether they are handled by a target or bridge as Delayed Transactions or not.

General Rules

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 device can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-posted transaction as a master. Otherwise a deadlock may occur.

Following is a summary of the PCI ordering rules specific to Delayed Transactions, presented in Section 3.3.3.3.

Delayed Transaction Rules

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. Until that time, it is only a request and may be discarded at anytime.
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 while completing a request using Delayed Transaction termination.

7. Delayed Requests and Delayed Completions have no ordering requirements with respect to themselves or each other. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything accept another Posted Memory Write.
8. A single master may have any number of outstanding request 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.

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 is 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 present 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. Where 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.

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 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 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 a 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. Until that time, the DRR is only a request and may be discarded at anytime 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 a I/O Write or Configuration Write commands. 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 anytime 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 are 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. The four Yes boxes are found in row 1 and prevent deadlocks from occurring when Delayed Transactions are used with devices designed to an earlier version of this specification. For example, a Posted Memory Write (PMW) transaction is being blocked by a Delayed Read Completion (DRC) or a Delayed Write Completion (DWC). When blocking occurs, the PMW is required to pass the DRC or the DWC. 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[6]	Yes[7]	Yes[8]
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/No	Yes/No	Yes/No	Yes/No
DWC (Row 5)	Yes/No	Yes/No	Yes/No	Yes/No	Yes/No

Col 2, Row 1 (Rule 1) -- A subsequent PMW cannot pass a previously accepted PMW. Posted Memory write transactions must complete in the order they are received. If the subsequent write is to the flag that covers the data, stale data may be used by the Consumer if the writes are allowed to pass each other.

Col 2, Row 2 (Rule 2) -- A read transaction must push posted write data to maintain ordering. For example, a memory write to a location and 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.

Col 2, Row 3 (Rule 3) -- A Delayed Write Request may be the flag that covers the data previously written (PMW) and therefore cannot pass data that it potentially covers.

Col 2, Row 4 (Rule 4) -- A read transaction must pull write data back to the originating bus of the read transaction. 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.

Col 2, Row 5 -- There are no ordering requirements for this case. 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.

Cols 3 and 4 -- The third and fourth columns are requests. Since requests have no ordering relationship with any other request that has been presented on the bus, there are no ordering requirements between them. Therefore, these two columns are don't cares, except the first row. The designer can (for performance or cost reasons) allow or disallow other transactions to pass requests that have been enqueued.

Col 3 and 4, Row 1 (Rules 5 and 6) -- These boxes are required to be Yes to prevent deadlocks. The deadlock can occur when bridges that support Delayed Transactions are used with bridges that do not support Delayed Transactions.

Referring to Figure E-2, the deadlock occurs 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 long memory write transactions 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.

Col 5 and 6 -- The fifth and sixth columns are Delayed Transactions that have completed on the destination bus and are moving toward the originating bus. Except for the first row all, other rows are either requests or completions of requests and, therefore, have no ordering associated with them. Because of this the system can allow other accesses to pass them if an advantage in cost or performance can be had.

Col 5 and 6, Row 1 (Rules 7 and 8) -- These boxes are required to be Yes to prevent deadlocks. This deadlock can also 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.

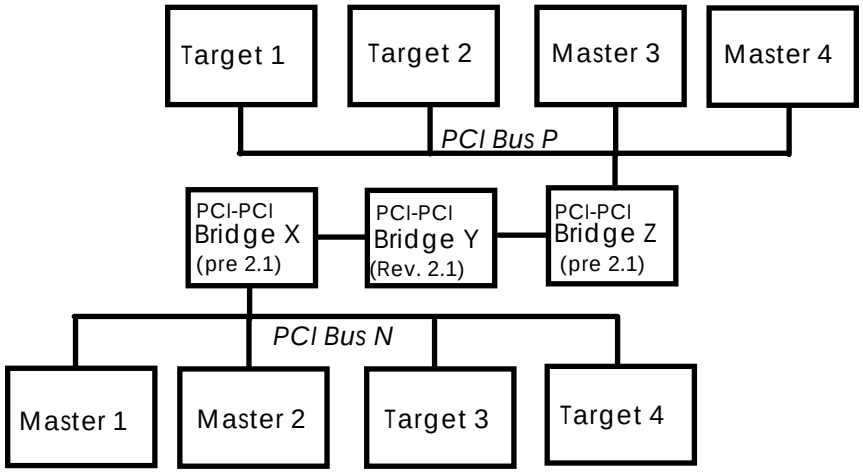


Figure E-2: Example System with PCI-to-PCI Bridges

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. When a locked transaction is initiated on the primary bus, where 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 proceeding 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.

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). However, it is recommended that the bridge not discard the transaction until 2^{15} PCI clocks (about 983 μs) after it acquired the data or status. The shorter number is useful in 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 does not have a device driver.

Glossary

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.
add-in board	A circuit board that plugs into a motherboard and provides added functionality.
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 expansion board 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.

central resources	Bus support functions supplied by the host system, typically in a PCI compliant bridge or standard chipset.
clean snoop	A snoop that does not result in a cache providing modified data.
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 cycle	Bus cycles 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.
Delayed Transaction	The process of a target latching a request and completing it after the master was terminated with Retry.
device dependent region	The last 48 DWORDS of the PCI configuration space. The contents of this region are not described in this document.
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 board	*See* add-in board.
green machine	A system designed for minimum power consumption.
header region	The first 16 DWORDS of the PCI 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).
ISA	Industry Standard Architecture expansion bus built into the IBM PC AT computer.
keepers	Another name for pullup resistors that are only used to sustain a signal state.

latency	*See* arbitration latency, master data latency, target initial latency, 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.
livelock	A condition in which two or more operations require completion of another operation before they can complete.
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.
motherboard	A circuit board containing the basic functions (e.g., CPU, memory, I/O, and expansion connectors) of a computer.
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 (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.
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.
shared slot	An arrangement on a PCI motherboard that allows a PCI connector to share the system bus slot nearest the PCI bus layout with an ISA, EISA, or MC bus connector. In an MC system, for example, the shared slot can accommodate either an MC expansion board or a PCI expansion board.
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 agents. *See also* positive decoding.
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.
transfer state	Any bus clock, during a data phase, in which data is transferred.

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.

Index

A

B

C

D

E

F

G

H

I

J

L

M

P

R

S

T

U

V

W