

PCI Bus Demystified

Doug Abbott



CD-ROM Included

Contains a full, searchable version of the book!

Demystifying Technology Series

By Engineers, For Engineers

PCI Bus Demystified

by Doug Abbott

A VOLUME IN THE
DEMYSTIFYING TECHNOLOGY™
SERIES



www.LLH-Publishing.com



Copyright © 2000 by LLH Technology Publishing

All rights reserved. No part of this book may be reproduced, in any form or means whatsoever, without written permission of the publisher. While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of information contained herein.

Printed in the United States of America.

ISBN 1-878707-78-7 (LLH eBook)

LLH Technology Publishing and HighText Publications are trademarks of Lewis Lewis & Helms LLC, 3578 Old Rail Road, Eagle Rock, VA, 24085

To Susan:

**My best friend, my soul mate. Thanks for sharing
life's journey with me.**

To Brian:

**Budding DJ, future pilot and all around neat kid.
Thanks for keeping me young at heart.**

Contents

Introduction	1
Intended Audience	2
The Rest of This Book	3
 Chapter 1: Introducing the Peripheral Component	
Interconnect (PCI) Bus	5
So What is a Computer Bus?	6
Bus Taxonomy	7
What's Wrong with ISA and Attempts to Fix It	9
The VESA Local Bus	10
Introducing PCI	11
Features	11
The PCI Special Interest Group	12
PCI Signals	13
Signal Groups	13
Signal Types	18
Sideband Signals	19
Definitions	20
Summary	21
 Chapter 2: Arbitration	22
The Arbitration Process	22
An Example of Fairness	25
Bus Parking	26
Latency	27
Summary	31

Chapter 3: Bus Protocol	32
PCI Bus Commands	32
Basic Read/Write Transactions	34
Transaction Termination — Master	45
Transaction Termination — Target	45
Error Detection and Reporting	51
Summary	54
 Chapter 4: Optional and Advanced Features	 56
Interrupt Handling	56
The Interrupt Acknowledge Command	59
“Special” Cycle	60
64-bit Extensions	62
Summary	66
 Chapter 5: Electrical and Mechanical Issues	 67
A “Green” Architecture	67
Signaling Environments — 3.3V and 5V	70
5 Volt Signaling Environment	72
3.3 Volt Signaling Environment	77
Timing Specifications	81
66 MHz PCI	85
Mechanical Details	88
Summary	90
 Chapter 6: Plug and Play Configuration	 92
Background	92
Configuration Address Space	93
Configuration Header — Type 0	95
Base Address Registers (BAR)	103

Expansion ROM	107
Capabilities List	110
Vital Product Data	111
Summary	115
 Chapter 7: PCI BIOS	 116
Operating Modes	116
Is the BIOS There?	117
BIOS Services	118
Generate Special Cycle	120
Summary	124
 Chapter 8: PCI Bridging	 125
Bridge Types	125
Configuration Address Types.....	128
Configuration Header — Type 1	129
Bus Hierarchy and Bus Number Registers	130
Address Filtering — the Base and Limit Registers	132
Prefetching and Posting to Improve Performance	135
Interrupt Handling Across a Bridge	136
Bridge Support for VGA — Palette “Snooping”	140
Resource Locking.....	142
Summary	146
 Chapter 9: CompactPCI	 148
Why CompactPCI?	148
Mechanical Implementation	150
Electrical Implementation	155
CompactPCI Bridging	162
Summary	165

Chapter 10: Hot Plug and Hot Swap	166
PCI Hot Plug	166
Hot Plug Primitives	170
CompactPCI Hot Swap	174
Resources for Full Hot Swap	180
Summary	185
 Appendix A: Class Codes	 187
 Appendix B: Connector Pin Assignments	 191
 Index	 195

Introduction

Today's computer systems, with their emphasis on high resolution graphics, full motion video, high bandwidth networking, and so on, go far beyond the capabilities of the architecture that ushered in the age of the personal computer in 1982. Modern PC systems demand high performance interconnects that also allow devices to be changed or upgraded with a minimum of effort by the end user.

In response to this need, PCI (*p*eripheral component *i*nterconnect) has emerged as the dominant mechanism for interconnecting the elements of modern, high performance computer systems. It is a well thought out standard with a number of forward looking features that should keep it relevant well into the next century. Originally conceived as a mechanism for interconnecting peripheral components on a motherboard, PCI has evolved into at least a half dozen different physical implementations directed at specific market segments yet all using the same basic bus protocol. In the form known as Compact PCI, it is having a major impact in the rapidly growing telecommunications market. PC-104 Plus offers a building-block approach to small, deeply embedded systems such as medical instruments and information kiosks.

PCI offers a number of significant performance and architectural advantages over previous busses:

- *Speed*: The basic PCI protocol can transfer up to 132 Mbytes per second, well over an order of magnitude faster than ISA. Even so, the demand for bandwidth is

insatiable. Extensions to the basic protocol yield bandwidths as high as 512 Mbytes per second and development currently under way will push it to a gigabyte.

- *Configurability:* PCI offers the ability to configure a system automatically, relieving the user of the task of system configuration. It could be argued that PCI's success owes much to the very fact that users need not be aware of it.
- *Multiple Masters:* Prior to PCI, most busses supported only one "master," the processor. High bandwidth devices could have direct access to memory through a mechanism called DMA (*direct memory access*) but devices, in general, could not talk to each other. In PCI, any device has the potential to take control of the bus and initiate transactions with any other device.
- *Reliability:* "Hot Plug" and "Hot Swap," defined respectively for PCI and Compact PCI, offer the ability to replace modules without disrupting a system's operation. This substantially reduces MTTR (*mean time to repair*) to yield the necessary degree of up-time required of mission-critical systems such as the telephone network.

Intended Audience

This book is intended as a thorough introduction to the PCI bus. It is not a replacement for the specification nor does it go into that level of detail. Think of it as a "companion" to the specification.

If you have a basic understanding of computer architecture and can read timing diagrams, this book is for you. Some knowledge of the Intel x86 processor family is useful but not essential.

The Rest of This Book

Chapter 1: Begins with a brief introduction to and history of computer busses and then introduces the PCI bus, its features and benefits, and describes the signals that make up PCI.

Chapter 2: Describes the arbitration process by which multiple masters share access to the bus. This also includes a discussion of bus latency.

Chapter 3: Explains the bus protocol including basic data transfer transactions, transaction termination and error detection and reporting.

Chapter 4: Covers the advanced and optional features of PCI including interrupt handling, the “Special” cycle and extensions to 64 bits.

Chapter 5: Describes the electrical and mechanical features of PCI with emphasis on its “green” specifications. This also covers 66 MHz PCI.

Chapter 6: Explores the extensive topic of Plug-and-Play configuration. This is the feature that truly distinguishes PCI from all of the bus architectures that have preceded it.

Chapter 7: Describes the PCI BIOS, a platform-independent API for accessing PCI’s configuration space.

Chapter 8: Explores the concept of PCI bridging as a way to build larger systems. This also describes an alternative interrupt mechanism using ordinary PCI transactions.

Chapter 9: Introduces Compact PCI, the industrial strength version of the PCI bus.

Chapter 10: Wraps things up with a look at Hot Plug and Hot Swap, two approaches to the problem of maintaining mission-critical systems by allowing modules to be swapped while the system is running.

CHAPTER 1

Introducing the Peripheral Component Interconnect (PCI) Bus

The notion of a computer “bus” evolved in the early 1960s along with the minicomputer. At that time, the minicomputer was a radical departure in computer architecture. Previously, most computers had been one-of-a-kind, custom built machines with relatively few peripherals—a paper tape reader and punch, a teletype, a line printer and, if you were lucky, a disk. The peripheral interface logic was tightly coupled to the processor logic.

The integrated circuit shrank the CPU from a refrigerator-sized cabinet down to one or two printed circuit boards. The interface electronics to peripheral devices shrank accordingly. Now computers could be cranked out on an assembly line, but only if they could be assembled efficiently. The engineers of the day quickly recognized the obvious solution—design all the boards to a common electrical and protocol interface specification. Assembling the computer is now just a matter of plugging boards into a backplane consisting of connectors and a large number of parallel wires.

The computer bus also solved a marketing problem. After all, there’s no point in mass producing computers unless you can sell them. A single company possesses limited expertise and resources

to address only a small segment of the potential applications for computers. The major minicomputer vendors solved this problem by making their bus specifications public to encourage third party vendors to build compatible equipment addressing different market segments.

So What is a Computer Bus?

Fundamentally, a computer bus consists of a set of parallel “wires” attached to several connectors into which peripheral boards may be plugged, as shown in Figure 1-1. Typically the processor is connected at one end of these wires. Memory may also be attached via the bus.

The wires are split into several functional groups such as:

- *Address*: Specifies the peripheral and register within the peripheral that is being accessed.
- *Data*: The information being transferred to or from the peripheral.
- *Control*: Signals that effect the data transfer operation. It is the control signals and how they are manipulated that embody the bus *protocol*.

Beyond basic data transfer, busses typically incorporate advanced features such as:

- Interrupts
- DMA
- Power distribution

Additional control lines manage these features.

The classic concept of a bus is a set of boards plugged into a passive backplane as shown in Figure 1-1. But there are also many bus implementations based on cables interconnecting stand-alone

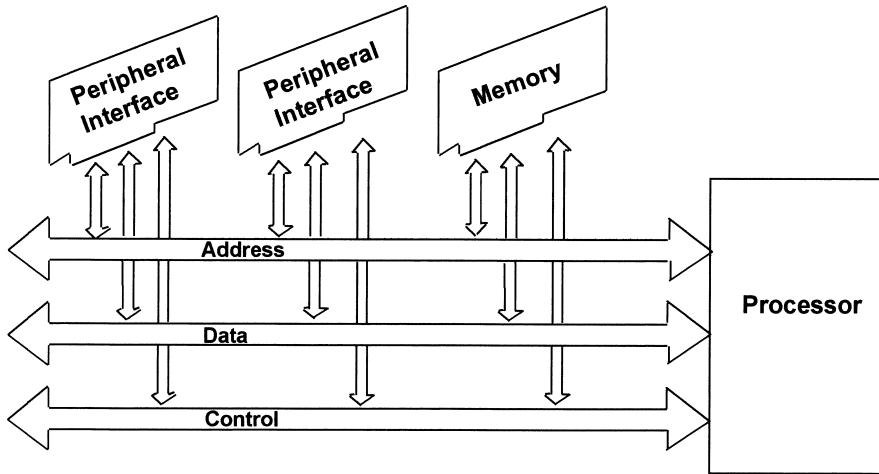


Figure 1-1: Functional diagram of a computer bus.

boxes. The GPIB (general purpose interface bus) is a classic example. Contemporary examples of cable busses include USB (universal serial bus) and IEEE-1394 (trademarked by Apple Computer under the name Firewire™). Nor is the backplane restricted to being passive as illustrated by the typical PC motherboard implementation.

Bus Taxonomy

Computer busses can be characterized along a number of dimensions. Architecturally, busses can be characterized along two binary dimensions: synchronous vs. asynchronous and multiplexed vs. non-multiplexed. In a synchronous bus, all operations occur on a specified edge of a master clock signal. In asynchronous busses operations occur on specified edges of control signals without regard to a master clock. Early busses tended to be asynchronous. Contemporary busses are generally synchronous.

A bus can be either *multiplexed* or *non-multiplexed*. In a multiplexed bus data and address share the same signal lines. Control

signals identify when the common lines contain address information and when they contain data. A non-multiplexed bus has separate wires for address and data.

The basic advantage of a multiplexed bus is fewer wires which in turn means fewer pins on each connector, fewer high-power driver circuits and so on. The disadvantage is that it requires two *phases* to carry out a single data transfer—first the address must be sent, then the data transferred. Contemporary busses are about evenly split between multiplexed and non-multiplexed.

Table 1-1 lists some of the quantifiable dimensions of bus design. Busses can be characterized in terms of the number of bits of address and data. Contemporary busses are typically either 32 or 64 bits wide for both address and data. Not surprisingly, multiplexed busses tend to have the same number of address and data bits.

Table 1-1: Bus parameters

Address width	8, 16, 32, 64
Data width	1, 8, 16, 32, 64
Transfer rate	1 MHz up to several hundred MHz
Maximum length	Several centimeters to several meters
Number of devices	A few up to many

A key element of any bus protocol is performance. How fast can it transfer data? Early busses were limited to a few megahertz, which closely matched processor performance of the era. The problem in contemporary systems is that the processor is often many times faster than the bus and so the bus becomes a performance bottleneck.

Bus length is related to transfer speed. Early busses with transfer rates of one or two megahertz allowed maximum lengths of several


meters. But with higher transfer rates comes shorter lengths so that propagation delay doesn't adversely impact performance.

The maximum number of devices that can be connected to a bus is likewise restricted by high performance considerations. Early busses could tolerate high-power, relatively slow driver circuits and could thus support a large number of attached devices. High performance busses such as PCI limit driver power and so are severely restricted in terms of number of devices.

What's Wrong with ISA and Attempts to Fix It

PCI evolved, at least in part, as a response to the shortcomings of the then venerable ISA (*industry standard architecture*) bus. ISA in turn was an evolutionary enhancement of the bus defined by IBM for its first personal computer. It was well matched to the processor performance and peripheral requirements of early PCs.

ISA began to run out of steam about 1992 when Windows had become firmly established as the dominant computing paradigm. To be truly effective, graphical computing requires much more than the 8 MB/sec that ISA is capable of. ISA's 16-bit data path is a bottleneck for contemporary 32-bit processors. Also, falling DRAM prices coupled with the extensive memory requirements of graphical computing soon rendered ISA's 16 Mbyte address space inadequate.

 Another problem concerned how computing systems were configured. ISA peripherals rely primarily on jumpers and DIP switches to resolve conflicts involving I/O addresses, interrupt and DMA channel allocation. Successful configuration of such a system requires a fairly detailed understanding of the devices and how they interact. This level of expertise is expected of hobbyists and geeks but is completely unacceptable in a mass-market consumer product.

The VESA Local Bus

The VESA Local Bus, promoted by the Video Electronics Standards Association, was one of the first attempts to overcome the limitations of ISA. The VL Bus strategy is to attach the video controller, and possibly other high-bandwidth devices, directly to the processor's local bus, either directly or through a buffer. The direct connection supports only one device, the buffered approach supports up to three devices. See Figure 1-2 for more detail.

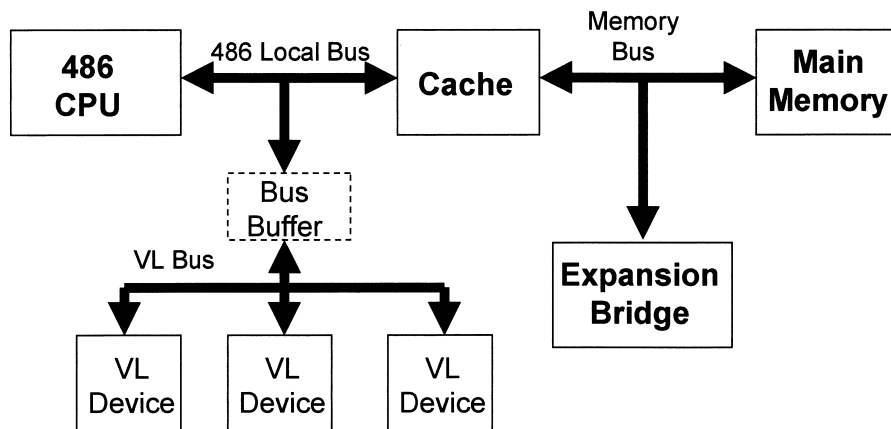



Figure 1-2: Functional diagram of the VL Bus.

 The VL Bus solved the bandwidth problem (in the short term anyway). On a 33 MHz, 32-bit processor bus, the VL Bus could achieve 132 Mbytes/sec. VESA also made an attempt to address the configuration issue by mandating that all VL Bus devices must support automatic configuration. Unfortunately, they didn't bother to define a configuration protocol so every device manufacturer invented their own.

VESA also did not specify with any precision the electrical characteristics of VL devices. They were just expected to be

compatible with the 486 bus. But the principal drawback of the VL Bus is that it's processor-specific. As soon as the Pentium came out, it was no longer relevant.

Introducing PCI

Intel developed the original PCI specification in an attempt to bring some coherence to an otherwise chaotic marketplace. Intel chose not to support the VL Bus because it failed to take a sufficiently long-term view of the emerging issues and trends in the development of PC architecture.

Revision 1 of the PCI specification appeared in June of 1992. Revision 2.0 came out in April 1993 to be followed by Rev. 2.1 in the first quarter of 1995 and finally the current revision, 2.2, which was released in February 1999.

Features

PCI implements a set of forward-looking features that should keep it relevant well into the next century:

- The maximum theoretical transfer rate of the base configuration is 132 Mbytes/sec. Currently defined extensions can boost this by a factor of four to 528 Mbytes/sec.
- Any device on the bus can be a bus master and initiate transactions. One consequence of this is that there is no need for the traditional notion of DMA.
- The transfer protocol is optimized around transferring blocks of data. A single transfer is just a block transfer with a length of one.
- Although PCI is officially processor-independent, it inevitably reflects its origins with Intel and its primary

application in the PC architecture. Among other things it uses little-endian byte ordering.

- PCI implements Plug and Play configurability. Every device in a system is automatically configured each time the system is turned on. The configuration protocol supports up to 256 devices in a system.
- The electrical specifications emphasize low power use including support for both 3.3 and 5 volt signaling environments. PCI is a “green” architecture.

The PCI Special Interest Group

PCI is embodied in a set of specifications maintained by the PCI Special Interest Group, an unincorporated association of several hundred member companies worldwide representing all aspects of the microcomputer industry including:

- Chip vendors
- OEM motherboard suppliers
- BIOS and operating system vendors
- Add-in card suppliers
- Tool suppliers

The specifications currently include:

- PCI Local Bus Spec., Rev. 2.2
- Mobile Design Guide, Rev. 1.1
- Power Management Interface Spec., Rev. 1.1
- PCI to PCI Bridge Architecture Spec., Rev. 1.1
- PCI Hot-Plug Spec., Rev. 1.0
- Small PCI Spec., Rev. 1.5a
- PCI BIOS Spec., Rev. 2.1

Copies of the specifications may be ordered from:

PCI Special Interest Group
2575 N.E. Kathryn #17
Hillsboro, OR 97124
(800) 433-5177
(503) 693-6232 (International)
(503) 693-8344 (FAX) www.pcisig.com

All of the specifications are available in PDF format on a single CD-ROM. (This address, URL, and phone numbers may have changed since publication of this book.)

PCI Signals

Figure 1-3 shows the signals defined in PCI. A PCI interface requires a minimum of 47 pins for a *target-only* device and 49 pins for a *master*. This is sufficient for a 32-bit data path running at up to 33 MHz and is mandatory for all devices claiming PCI compatibility. An additional 51 pins define optional features such as 64-bit transfers, interrupts and a JTAG interface.

A note about notation: A # sign at the end of a signal name, such as FRAME#, indicates that the signal is active or asserted in the low voltage state. Signal names without a # are asserted in the high voltage state. The notation [n::m], where n and m are integers such that n is greater than m, represents an “array” of signals with n – m + 1 members. Thus AD[31::0] represents the 32-bit data bus consisting of signals AD[0] to AD[31] with AD[0] being the least significant bit.

Signal Groups

For purposes of definition, the PCI signals can be classified in several functional groups.

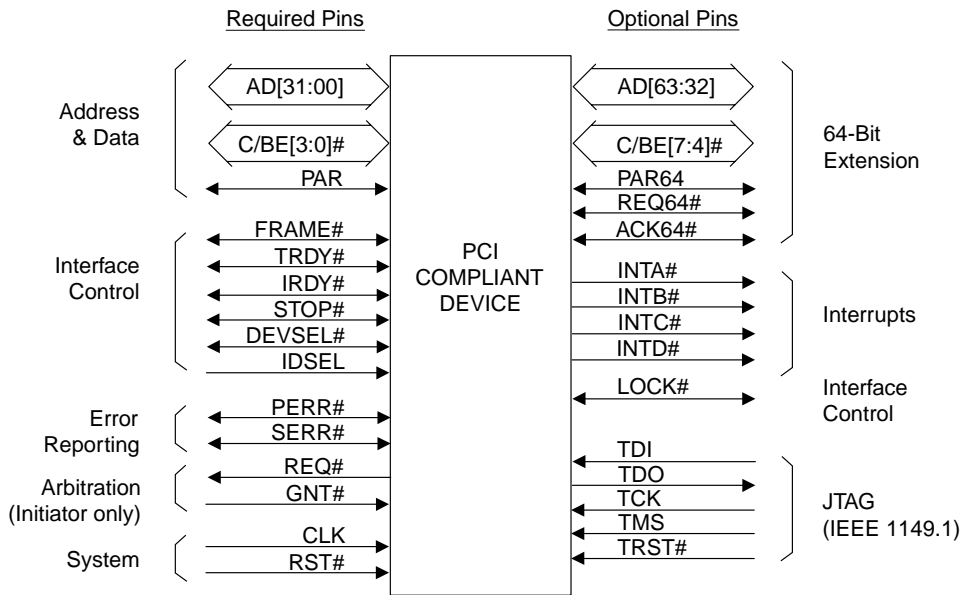


Figure 1-3: Signals defined in the PCI standard.

System

CLK Provides timing for all PCI transactions and is an input to every PCI device. All other PCI signals except RST# and INTA# through INTD# are sampled on the rising edge of CLK. (in)

RST# Brings PCI-specific registers, sequencers, and signals to a consistent state. Whenever RST# is asserted, all PCI output signals must be driven to their benign state. In general, this means they must be tri-stated. (in)

Address and Data

AD[31:0] Address and data are multiplexed on the same set of pins. A PCI transaction consists of an *address phase* followed by one or more *data phases*. (t/s)

C/BE[3::0] Bus command and byte enables are multiplexed on the same pins. During the address phase of a transaction, C/BE[3::0] define a *bus command*. During each data phase, C/BE[3::0] are used as *byte enables* to determine which byte lanes carry valid data. C/BE[0] applies to byte 0 (lsb) and C/BE[3] applies to byte 3 (msb). (t/s)

PAR Even *Parity* across AD[31::0] and C/BE[3::0]. All PCI agents are required to generate parity. (t/s)

Interface Control

FRAME# Driven by the current master to indicate the beginning and duration of a transaction. Data transfer continues while FRAME# is asserted. When FRAME# is de-asserted, the transaction is in its final data phase or has completed. (s/t/s)

IRDY# Initiator Ready indicates that the bus master is able to complete the current data phase. During a write, IRDY# indicates that valid data is present on AD[31::0]. During a read it indicates that the master is prepared to accept data. (s/t/s)

TRDY# Target Ready indicates that the selected target device is able to complete the current data phase. During a read, TRDY# indicates that valid data is present on AD[31::0]. During a write, it indicates that the target is prepared to accept data. A data phase completes on any clock cycle during which both IRDY# and TRDY# are asserted. (s/t/s)

STOP# Indicates that the selected target requests the master to terminate the current transaction. (s/t/s)

LOCK# Indicates an *atomic* operation that may require multiple transactions to complete. (s/t/s)

IDSEL *Initialization Device Select* is a chip select used during configuration transactions. (in)

DEVSEL# *Device Select* indicates that a device has decoded its address as the target of the current transaction. (s/t/s)

Arbitration

REQ# *Request* indicates to the central arbiter that an agent desires to use the bus. Every potential bus master has its own point-to-point REQ# signal. (t/s)



GNT# *Grant* indicates to an agent that is asserting its REQ# signal that access to the bus has been granted. Every potential bus master has its own point-to-point GNT# signal. (t/s)

Error Reporting

PERR# For reporting data *Parity Errors* during all PCI transactions except a Special Cycle. (s/t/s)

SERR# *System Error* is for reporting address parity errors, data parity errors on Special Cycle commands, and any other potentially catastrophic system error. (o/d)

Interrupt (optional)

INTA# through INTD# are used by a device to request attention from its device driver. A single-function device may only use INTA#. Multi-function devices may use any combination of INTx# signals. (o/d)

64-bit Bus Extension (optional)

AD[63::32] Upper 32 address and data bits. (t/s)

C/BE[7::4] Upper byte enable signals. Generally not valid during address phase. (t/s)

REQ64# *Request 64-bit Transfer* indicates that the current bus master desires to execute a 64-bit transfer. (s/t/s)

ACK64# *Acknowledge 64-bit Transfer* indicates that the selected target is willing to execute 64-bit transfers. 64-bit transfers can only occur when both REQ64# and ACK64# are asserted. (s/t/s)

PAR64 *Even Parity* over AD[63::32] and C/BE[7::4]. (t/s)

JTAG/Boundary Scan (optional)

The PCI specification reserves a set of pins for implementing a *Test Access Port (TAP)* conforming to IEEE Standard 1149.1, *Test Access Port and Boundary Scan Architecture*. This provides a reliable, well-defined mechanism for testing a device or board.

Additional Signals

These signals are not part of the basic PCI protocol but implement additional features that are useful in certain operating environments.

PRSNT[1:2]# These are defined for add-in boards but not for motherboard devices. The *Present* signals indicate to the motherboard that a board is physically present and, if it is, its total power requirements. All boards are required to ground one or both Present signals as follows: (in)

PRSNT1#	PRSNT2#	State
Open	Open	No expansion board present
Ground	Open	Present, 25 W maximum
Open	Ground	Present, 15 W maximum
Ground	Ground	Present, 7.5 W maximum

Add-in boards are required to implement the Present signals but they are optional for motherboards.

CLKRUN# *Clock Running* is an optional input to a device to determine the state of CLK. It is output by a device that wishes to control the state of the clock. Assertion means the clock is running at its normal speed. De-assertion is a request to slow down or stop the clock. This is intended as a power saving mechanism in mobile environments and is described in the *PCI Mobile Design Guide*. The standard PCI connector does not have a pin for CLKRUN#. (in, o/d, s/t/s)

M66EN 66MHz_Enable indicates to a device that the bus segment is running at 66 MHz. (in)

PME# *Power Management Event* is an optional signal that allows a device to request a change in the device or system power state. The operation of this signal is described in the *PCI Bus Power Management Interface Specification*. (o/d)

3.3Vaux *Auxiliary 3.3 volt Power* allows an add-in card to generate power management events even when main power to the card is turned off. The operation of this signal is described in the *PCI Bus Power Management Interface Specification*. (in)

Signal Types

Each of the signals listed above included a somewhat cryptic set of initials in parentheses. These designate the *signal type*. The signal types are:

in: Input only

- CLK, RST#, IDSEL, TCK, TDI, TMS, TRST#, PRSNT[1:2]#,¹ CLKRUN#, M66EN, 3.3Vaux

¹ Although the specification calls these input only signals, this author believes they are really outputs because the information is being communicated from the add-in card to the motherboard.

out: Standard *totem-pole* active output only

- TDO

t/s: Bidirectional tri-state input/output

- AD[63:0], C/BE[7:0], PAR, PAR64, REQ#, GNT#, CLKRUN#

s/t/s: Sustained tri-state. Driven by one *owner* at a time. Note that all of the *s/t/s* signals are assertion low. The owner must drive the signal high, that is to the unasserted state, for one clock before tri-stating. Another agent must not drive an *s/t/s* signal sooner than one clock after the previous owner has tri-stated it. *s/t/s* signals require a pull-up to sustain the signal in the unasserted state until another agent drives it. The pull-up is provided by the central resource.

- FRAME#, TRDY#, IRDY#, STOP#, LOCK#, PERR#, REQ64#, ACK64#

o/d: Open drain, wire-OR allows multiple devices to assert the signal simultaneously. A pull-up is required to sustain the signal in the unasserted state when no device is driving it. The pull-up is provided by the central resource.

- SERR#, INTA# - INTD#, CLKRUN#, PME#

Sideband Signals

The specification acknowledges that there may be a need for application-specific signals that fall outside the scope of the PCI specifications. These are called *sideband signals* and are loosely defined as “... any signal not part of the PCI specifications that connects two or more PCI compliant agents and has meaning only to those agents.”

Such signals are *allowed* provided they don't interfere with the

PCI protocol. No pins are provided on the add-in card connector to support sideband signals so they are restricted to so-called “planar devices” on the motherboard.

Definitions

There are a number of terms that will crop up again and again throughout this book. Some of them have already been used without being defined.

Agent: An entity or device that operates on a computer bus.

Master: An agent capable of initiating bus transactions.

Transaction: In the context of PCI, a transaction consists of an address phase and one or more data phases. This is also called a *burst transfer*.

Initiator: A master that has arbitrated for and won access to the bus. The initiator is the agent that “initiates” bus transactions.

Target: An agent that recognizes its address during the address phase. The target responds to the transaction initiated by the initiator.

Central Resource: An element of the host system that provides bus support functions such as CLK and RST# generation, bus arbitration and pull-up resistors. The central resource is usually a part of the host processor’s chipset.

DWORD: A 32-bit block of data. A basic PCI bus can transfer data in DWORDs.

Latency: The number of clocks between specific state transitions during a bus transaction. Latency measures the time an agent requires to respond to an action initiated by another agent and is thus an indicator of overall performance.

Summary

This chapter has described the main features of PCI, identified the relevant specifications and the group responsible for maintaining those specifications. Some basic terms have been defined and the PCI signals have been described.

CHAPTER 2

Arbitration

Since the PCI Bus accommodates multiple masters—any of which could request the use of the bus at any time—there must be a mechanism that allocates use of bus resources in a reasonable way and resolves conflicts among multiple masters wishing to use the bus simultaneously. Fundamentally, this is called *bus arbitration*.

The Arbitration Process

Before a bus master can execute a PCI transaction, it must request, and be granted, use of the bus. For this purpose, each bus master has a pair of REQ# and GNT# signals connecting it directly to a central arbiter as shown in Figure 2-1. When a master wishes to use the bus, it asserts its REQ# signal. Sometime later the arbiter will assert the corresponding GNT# indicating that this master is next in line to use the bus.

Only one GNT# signal can be asserted at any instant in time. The master agent who sees his GNT# asserted may initiate a bus transaction when it detects that the bus is idle. The bus idle state is defined as both FRAME# and IRDY# de-asserted.

Figure 2-2 is a timing diagram illustrating how arbitration works when two masters request use of the bus simultaneously.

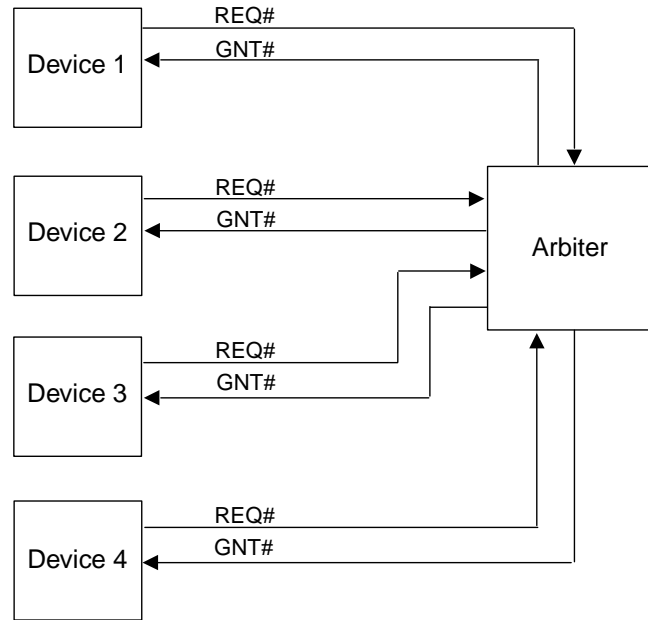


Figure 2-1: Arbitration process under PCI.

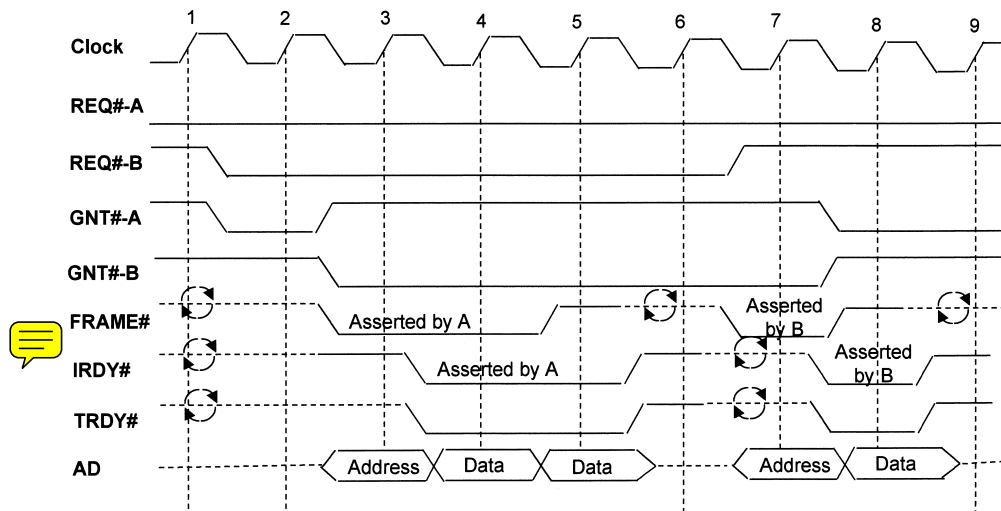


Figure 2-2: Timing diagram for arbitration process involving two masters.

Clock

- 1 The arbiter detects that device A has asserted its REQ#. No one else is asserting a REQ# at the moment so the arbiter asserts GNT#-A. In the meantime device B asserts its REQ#.
- 2 Device A detects its GNT# asserted, the bus is idle and so it asserts FRAME# to begin its transaction. Device A keeps its REQ# asserted indicating that it wishes to execute another transaction after this one is complete. Upon detecting REQ#-B asserted, the arbiter deasserts GNT#-A and asserts GNT#-B.
- 3 Device B detects its GNT# asserted but can't do anything yet because a transaction is in process. Nothing more of interest happens until clock ...
- 6 Device B detects that the bus is idle because both FRAME# and IRDY# are deasserted. In response, it asserts FRAME# to start its transaction. It also deasserts its REQ# because it does not need a subsequent transaction.
- 7 The arbiter detects REQ#-B deasserted. In response it deasserts GNT#-B and asserts GNT#-A since REQ#-A is still asserted.



Arbitration is “hidden,” meaning that arbitration for the next transaction occurs at the same time as, or in parallel with, the current transaction. So the arbitration process doesn't take any time. The specification does not stipulate the nature of the arbitration algorithm or how it is to be implemented other than to say that arbitration must be “fair.” This is not to say that there cannot be a relative priority scheme among masters but rather that every master gets a chance at the bus. Note in Figure 2-2 that even though Device A wants to execute another transaction, he must wait until Device B has executed his transaction.



An Example of Fairness

Figure 2-3 offers an example of what the specification means by fairness. This is taken directly from the specification. In this example, a bus master can be assigned to either of two arbitration *levels*. Agents assigned to Level 1 have a greater need for use of the bus than those assigned to Level 2. Agents at Level 2 have equal access to the bus with respect to other second level agents. Furthermore, Level 2 agents, *as a group*, have equal access to the bus as Level 1 agents.

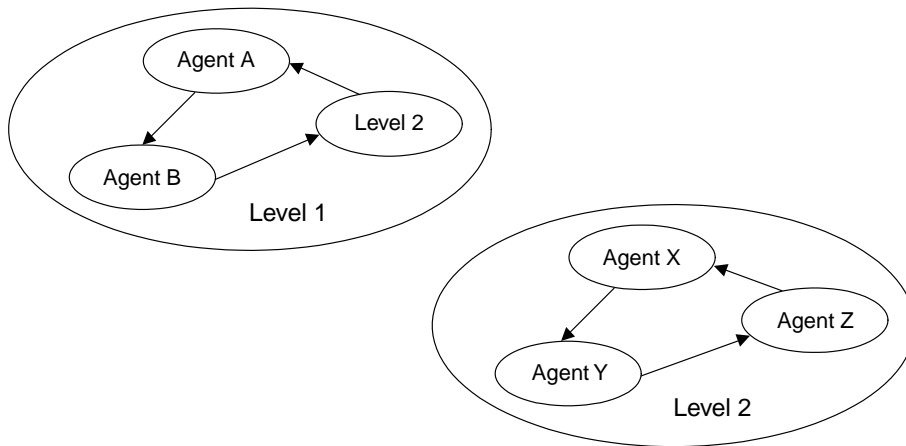


Figure 2-3: Example of fairness in arbitration.

Consider the case that all agents in the figure above have their REQ# signals asserted and continue to assert them. If Agent A is the next Level 1 agent to receive the bus and Agent X is next for Level 2, then the order of bus access would be:

A, B, Level 2 (X)
 A, B, Level 2 (Y)
 A, B, Level 2 (Z)
 and so forth.

If only Agents B and Y had their REQ# signals asserted, the order would be:

B, Level 2 (Y)

B, Level 2 (Y)

Typically, high performance agents like video, ATM or FDDI would be assigned to Level 1 while devices like a LAN or SCSI disk would go on Level 2. This allows the system designer to tune the system for maximum throughput and minimal latency without the possibility of starvation.

It is often the case that when a standard offers an example or suggestion of how some feature may be implemented, it becomes a de facto standard as most vendors choose that particular implementation. So it is with arbitration algorithms. Many chipset and bridge vendors have implemented the priority scheme described by this example.

Bus Parking

A master device is only allowed to assert its REQ# when it actually needs the bus to execute a transaction. In other words, it is not allowed to continuously assert REQ# in order to monopolize the bus. This violates the low-latency spirit of the PCI spec. On the other hand, the specification does allow the notion of “bus parking.”

The arbiter may be designed to “park” the bus on a default master when the bus is idle. This is accomplished by asserting GNT# to the default master when the bus is idle. The agent on which the bus is parked can initiate a transaction without first asserting REQ#. This saves one clock. While the choice of a default master is up to the system designer, the specification recommends parking on the last master that acquired the bus.

Latency

When a bus master asserts REQ#, a finite amount of time expires until the first data element is actually transferred. This is referred to as *bus access latency* and consists of several components as shown in Figure 2-4:

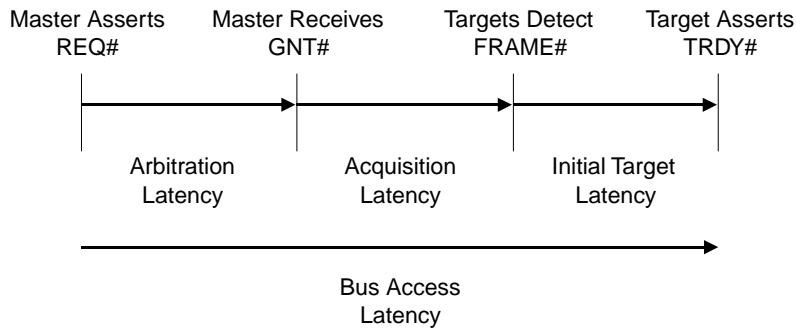


Figure 2-4: Components of bus access latency.

Arbitration Latency. The time from when the master asserts REQ# until it receives GNT#. This is a function of the arbitration algorithm and the number of other masters requesting use of the bus that may be ahead of this one in the arbitration queue.

Acquisition Latency. The time from when the master receives GNT# until the targets recognize that FRAME# is asserted. If the bus is idle, this is only one or two clock cycles. Otherwise it is a function of the *Latency Timer* in the master currently using the bus.

Initial Target Latency. The time from when the selected target detects FRAME# asserted until it asserts TRDY#. Target latency for the first data transfer is often longer than the latency on subsequent transfers because the device may need extra time to prepare a block of data—a disk may have to wait for the sector to come around for example. The specification limits initial target latency to 16 clocks and subsequent latency to 8 clocks.

Latency Timer

The PCI specification goes to great lengths to give designers and integrators facilities for balancing and fine tuning systems for optimal performance. One of these facilities is the *Latency Timer* that is required in every master device that is capable of burst lengths greater than two.

The purpose of the Latency Timer is to prevent a master from hogging the bus if other masters require access. The value programmed into the Latency Timer (or hardwired) represents the minimum number of clock cycles a master gets when it initiates a transaction.

When a master asserts FRAME#, the Latency Timer is loaded with the hardwired or configuration-programmed value. Each clock cycle thereafter decrements the counter. If the counter reaches 0 before the transaction completes and the master's GNT# is *not* asserted, that means another master needs to use the bus and so the current master must terminate its transaction. The current master will most likely immediately request the bus so it can finish its transaction. But of course it won't get the bus until all other masters currently requesting the bus have finished.

Bandwidth vs. Latency

In PCI there is a tradeoff between the desire for low latency and the complementary desire for high bandwidth (throughput). High throughput is achieved by allowing devices to use long burst transfers. Conversely, low latency results from reducing the maximum burst length.

A master is required to assert its IRDY# within eight clocks for any given data phase. The selected target is required to assert TRDY#

within 16 clocks from the assertion of FRAME# for the first data phase (32 clocks if the access hits a modified cache line). For subsequent data phases the target must assert TRDY# or STOP# within 8 clocks.

If we ignore the effects of the Latency Timer, it is a straightforward exercise to develop equations for worst case latencies.

If a modified cache line is hit:

$$\text{Latency}_{\max} = 32 + 8*(n - 1) + 1 \text{ (clocks)}$$

Otherwise:

$$\text{Latency}_{\max} = 16 + 8*(n - 1) + 1 \text{ (clocks)}$$

where n is the total number of data transfers. The extra clock is the idle cycle introduced between most transactions.

Nevertheless, it is more useful to consider transactions that exhibit typical behavior. PCI bus masters typically don't insert wait states because they only request transactions when they are prepared to transfer data. Likewise, once a target begins transferring data it can usually sustain the full data rate of one transfer per clock cycle. Targets typically have an initial access latency of less than 16 (or 32) clock cycles. Again ignoring the effects of the Latency Timer, typical latency can be expressed as:

$$\text{Latency}_{\text{typ}} = 8 + (n - 1) + 1 \text{ (clocks)}$$

The Latency Timer effectively controls the tradeoff between high throughput and low latency.

Table 2-1 illustrates this tradeoff between latency and throughput for different burst lengths based on the typical latency equation just developed.

Table 2-1: Bandwidth vs. latency.

Data Phases	Bytes Transferred	Total Clocks	Bandwidth (Mb/sec)	Latency (us)
8	32	16	60	0.48
16	64	24	80	0.72
32	128	40	96	1.20
64	256	72	107	2.16

Total Clocks: total number of clocks required to complete the transaction. Same as $\text{Latency}_{\text{typ}}$.

Latency Time: The Latency Timer is set to expire on the next to the last data transfer.

Bandwidth: calculated bandwidth in MB/sec

$$\text{Bandwidth} = \text{bytes transferred} / (\text{total clocks} * 30\text{ns})$$

Latency: latency in microseconds resulting from the transaction

$$\text{Latency} = \text{total clocks} * 0.030 \text{ us}$$

Notice that the amount of data transferred per transaction doubles from row to row but the latency doesn't quite double. From first row to last row the amount of data transferred increases by a factor of 8 while latency increases by about 4.5. This reflects the fact that there is some overhead in every PCI transactions and so the longer the transaction, the more *efficient* the bus is.

Note by the way that it's not uncommon to find devices that routinely violate the latency rules, particularly among older devices derived from ISA designs. How should an agent respond to excessive latency, or indeed any protocol violations? The specification states "A device is not encouraged actively to check for protocol errors."

In effect, the protocol rules define “good behavior” that well-behaved devices are expected to observe. Devices that aren’t so well behaved are tolerated.

Summary

PCI incorporates a hidden arbitration mechanism that regulates access to the bus by multiple masters. The arbitration algorithm is not specified but is required to be “fair.” The arbiter may include a mechanism to “park” the bus on a specific master when the bus is idle.

Bus access latency is the time from when a master requests use of the bus until the first item of data is transferred. There is a tradeoff between low latency and high bandwidth that can be regulated through the Latency Timer.

CHAPTER 3

Bus Protocol

The essence of any bus is the set of rules by which data moves between devices. This set of rules is called a *protocol*. This chapter describes the basic protocol that controls the transfer of data between devices on a PCI bus.

PCI Bus Commands

The PCI bus command for a transaction is conveyed on the C/BE# lines during the address phase. Note that when C/BE# is carrying command data it is assertion high (high level = logic 1) whereas when it carries byte enable data it is assertion low.

The PCI bus defines three distinct address spaces with corresponding read and write commands as shown in Table 3-1. The principal distinction between memory and I/O spaces is that memory is generally considered to be “prefetchable” and thus reads from memory space have no “side effects.” Configuration address space is used only at bootup time to configure the community of PCI cards in a system.

There are some additional read/write commands that apply to prefetchable memory space only. The purpose of Memory Read Line is to tell the target that the master intends to read most of, if not the full current cache line. The target may gain some performance

Table 3-1

C/BE#3	C/BE#2	C/BE#1	C/BE#0	Command Type
0	0	0	0	Interrupt Acknowledge
0	0	0	1	Special Cycle
0	0	1	0	I/O Read
0	0	1	1	I/O Write
0	1	0	0	Reserved
0	1	0	1	Reserved
0	1	1	0	Memory Read
0	1	1	1	Memory Write
1	0	0	0	Reserved
1	0	0	1	Reserved
1	0	1	0	Configuration Read
1	0	1	1	Configuration Write
1	1	0	0	Memory Read Multiple
1	1	0	1	Dual-Address Cycle
1	1	1	0	Memory Read Line
1	1	1	1	Memory Write and Invalidate

advantage by knowing that it is expected to supply up to an entire cache line. When a master issues the **Memory Read Multiple command**, it is saying that it intends to read more than one cache line before disconnecting. This tells the target that it is worthwhile to prefetch the next cache line.

Memory Write and Invalidate is semantically identical to Memory Write with the addition that the master commits to write a full cache line in a single PCI transaction. This is useful when a transaction hits a “dirty” line in a writeback cache. Because the current master

is updating the entire line, the cache can simply invalidate the line without bothering to write it back.

The *Interrupt Acknowledge* command is a read implicitly addressed to the system interrupt controller. The contents of the AD bus during the address phase are irrelevant and the C/BE# indicate the size of the returned vector during the corresponding data phase.

The *Special Cycle* command provides a message broadcast mechanism as an alternative to separate physical signals for sideband communication. The *Dual Address Cycle* (DAC) command is a way to transfer a 64-bit address on a 32-bit backplane.

Basic Read/Write Transactions

Figure 3-1 shows the timing of a typical read transaction—one that transfers data from the Target to the Initiator. Let's follow it cycle-by-cycle.

Clock

- 1 The bus is idle and most signals are tri-stated. The master for the upcoming transaction has received its GNT# and detected that the bus is idle so it drives FRAME# high initially.
- 2 Address Phase: The master drives FRAME# low and places a target address on the AD bus and a bus command on the C/BE# bus. All targets latch the address and command on the rising edge of clock 2.
- 3 The master asserts the appropriate lines of the C/BE# (byte enable) bus and also asserts IRDY# to indicate that it is ready to accept read data from the target. The target that recognizes its address on the AD bus asserts DEVSEL# to acknowledge its selection.

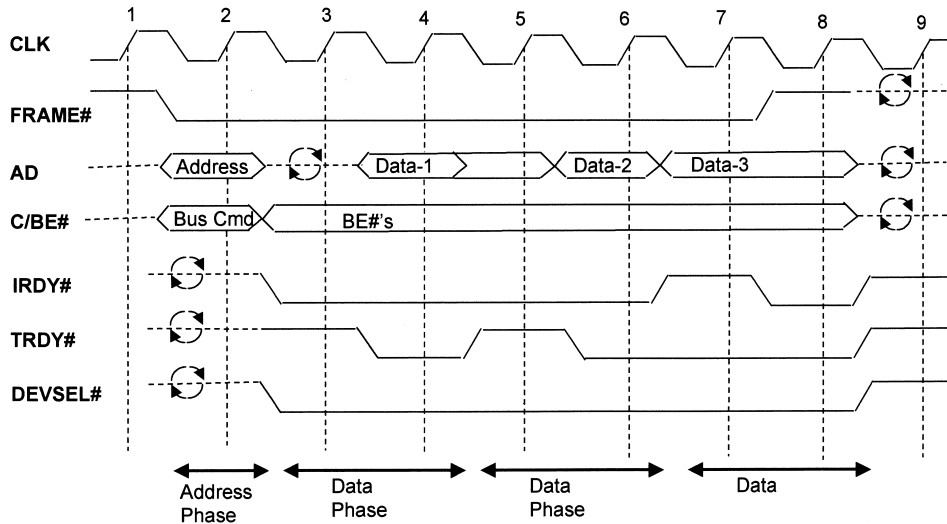


Figure 3-1: Timing diagram for a typical read transaction.

This is also a *turnaround cycle*: In a read transaction, the master drives the AD lines during the address phase and the target drives it during the data phases. Whenever more than one device can drive a PCI bus line, the specification requires a *one-clock-cycle turnaround*, during which neither device is driving the line, to avoid possible contention that could result in noise spikes and unnecessary power consumption. Turnaround cycles are identified in the timing diagrams by the two circular arrows chasing each other.

- 4 The target places data on the AD bus and asserts TRDY#. The master latches the data on the rising edge of clock 4. Data transfer takes place on any clock cycle during which both IRDY# And TRDY# are asserted.

- 5 The target deasserts TRDY# indicating that the next data element is not ready to transfer. Nevertheless, the target is required to continue driving the AD bus to prevent it from floating. This is a *wait cycle*.
- 6 The target has placed the next data item on the AD bus and asserted TRDY#. Both IRDY# and TRDY# are asserted so the master latches the data bus.
- 7 The master has deasserted IRDY# indicating that it is not ready for the next data element. This is another wait cycle.
- 8 The master has reasserted IRDY# and deasserted FRAME# to indicate that this is the last data transfer. In response the target deasserts AD, TRDY# and DEVSEL#. The master deasserts C/BE# and IRDY#. This is a *master-initiated termination*. The target may also terminate a transaction as we'll see later.

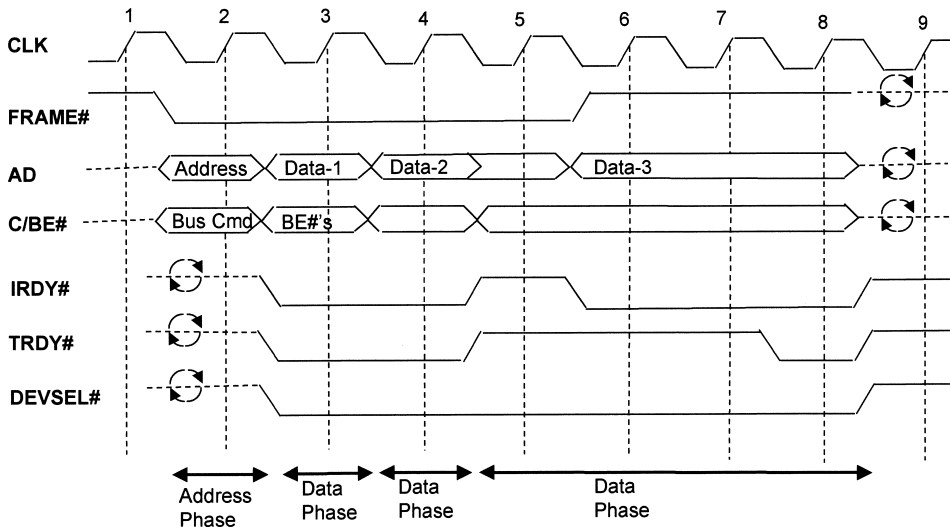


Figure 3-2: Timing diagram for a typical write transaction.

Figure 3-2 shows the details of a typical write transaction where data moves from the master to the target. The primary difference between the write transaction and the read transaction detailed in Figure 3-1 is that write does not require a turnaround cycle between the address and first data phase because the same agent is driving the AD bus for both phases. Thus the master can drive data onto the AD bus during clock 3.

Byte Enable Usage

During the data phases of a transaction, the C/BE# signals indicate which *byte lanes* convey meaningful data. The master may change byte enables between data phases but they must be valid on the clock that starts each data phase and remain valid for the entire data phase. The master is free to use any contiguous or non-contiguous combination of byte enables, including none, i.e. no byte enables asserted.

Independent of the byte enables, the agent driving the AD bus is required to drive all 32 lines to stable values. This is to assure valid parity generation and checking and to prevent the AD lines from floating.

Use of AD[1:0] During Address Phase

Since C/BE# conveys information about which of four bytes are to be transferred during each data phase, AD[1:0] can be used for something else during the address phase of a memory transaction. Specifically, AD[1:0] indicate how the target should advance the address during a multi-data phase burst as shown in Table 3-2. Linear addressing is the normal case wherein the target advances the address by 4 (32-bit transfer) or 8 (64-bit transfer) for each data phase.

Table 3-2

AD1	AD0	Address Sequence
0	0	<i>Linear (sequential) addressing.</i> Target increments address by 4 after each data phase.
0	1	<i>Reserved.</i> Target disconnects after first data phase.
1	0	<i>Cache line wrap.</i> New in Rev. 2.1. If initial address was not beginning of cache line, wrap around until cache line filled.
1	1	<i>Reserved.</i> Target disconnects after first data phase.

Cache line wrap mode only applies if a burst begins in the middle of a cache line. When the end of the cache line is reached, the address wraps around to the beginning of the cache line until the entire line has been transferred. If the burst continues beyond this point, the next transfer is to/from the same location in the next cache line where the transfer began.

Here's an example: Consider a cache line size of 16 bytes (4 DWORDs) and a transfer that begins at location 8. The first transfer is to location 8, the second to location C hex which is the end of the cache line. The third data phase is to address 0 and the fourth to address 4. If the burst continues, the next data phase will be to location 18 hex.


Targets are not required to support cache line wrap. If a target does not support this feature it should terminate the transaction after the first data phase.

Addresses for transfers to I/O space are fully qualified to the byte level. That is, AD[1:0] convey valid address information inferring the

least significant valid byte. This in turn implies which C/BE# signals are valid. Thus for example if AD[1:0] = 00, at a minimum C/BE#[0] must be 0 to transfer the low-order byte but up to four bytes could be transferred. Conversely if AD[1:0] = 11, only the high-order byte can be transferred so C/BE#[3] is 0 and C/BE#[2:0] must be 1. See Table 3-3.

Table 3-3

AD1:0 implies which BE# lines are valid



AD1	AD0	C/BE#3	C/BE#2	C/BE#1	C/BE#0
0	0	X	X	X	0
0	1	X	X	0	1
1	0	X	0	1	1
1	1	0	1	1	1

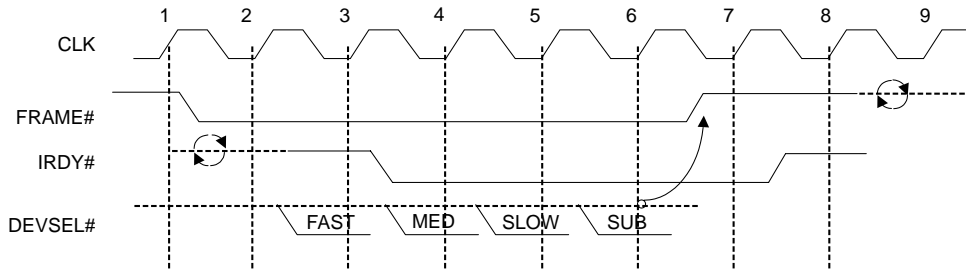
0: line must be asserted

1: line must not be asserted

X: line may be asserted

DEVSEL# Timing

The selected target is required to “claim” the transaction by asserting DEVSEL# within **three clock cycles** of the assertion of FRAME# by the current master as shown in Figure 3-3. This leads to three categories of target devices based on their response time to FRAME#. A *fast* target responds in one clock cycle, a *medium* target in two cycles and a *slow* target in three cycles. A target’s DEVSEL# timing is encoded in the Configuration Space Status Register. The target must assert DEVSEL# before it can assert **TRDY#** (or AD on a read transaction).



“SUB” = Subtractive Decoder

Figure 3-3: DEVSEL# timing.

If no agent claims the transaction within three clocks, a *subtractive-decode* agent may claim it on the fourth clock. A PCI bus segment can have at most one subtractive decode agent which is typically a bridge to another PCI segment or an expansion bus such as ISA, EISA, etc. The strategy is that if no agent claims the transaction on this bus segment, then its probably intended for some agent on the expansion bus segment on the other side of the bridge. So the bridge claims the transaction by asserting DEVSEL# and forwards it to the expansion bus.

The problem with subtractive decoding is that every transaction on the expansion bus incurs an additional latency of four clock cycles. As an alternative, the bridge could—and in most cases does—implement *positive decoding* whereby it is programmed at configuration time with one or more address ranges to which it will respond. Then it can claim transactions like any other target.

Finally, if all targets on a segment are either fast or medium, as indicated by their status registers, a subtractive decoding bridge could be programmed to tighten up its DEVSEL# response by one or two clock cycles.

If DEVSEL# is not asserted after 4 clocks following FRAME# assertion, the initiator terminates the transaction with a **Master-Abort**. This means the initiator tried to access an address that doesn't exist in the system.

Address/Data Stepping

Turning on 32 drivers simultaneously can lead to large current spikes on the power supply and crosstalk on the bus. One solution is to stagger the driver turn on as shown in Figure 3-4. In this example, the 32-bit AD bus is divided into four groups that are turned on in successive clock cycles.

For address stepping the master asserts FRAME# only when all four driver groups are on. Data can likewise be stepped. The example here is a write cycle so the master asserts IRDY# only when all four driver groups have switched to the current data item.

Although Figure 3-4 shows stepping synchronized to the PCI clock, this is not required.

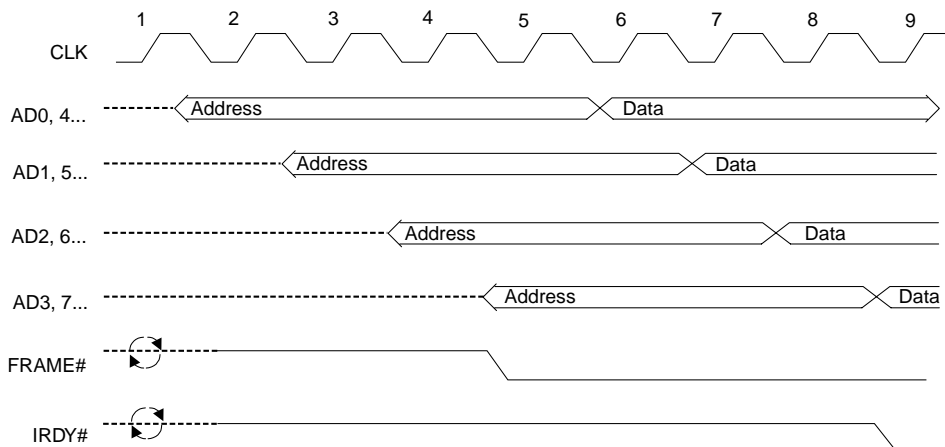


Figure 3-4: Address/data stepping.

Address/Data stepping only applies to qualified signals—those whose value is only considered valid when one or more control signals are asserted. The qualified signals consist of AD, PAR and PAR64, and IDSEL. AD is qualified by FRAME# during the address phase and IRDY# or TRDY# during a data phase. PAR and PAR64 are valid one clock cycle after the corresponding address or data phase. IDSEL is qualified by FRAME# and a configuration command.

There are a couple of problems with address/data stepping. First, it reduces performance by using additional clock cycles. Second, during a stepped address phase, another higher priority master may request the bus causing the arbiter to remove GNT# from the agent in the process of stepping. Since the stepping agent hasn't asserted FRAME# the bus is technically idle. In this case the stepping agent must tri-state its AD drivers and recontend for the bus.

A device indicates its ability to do address/data stepping through a bit in its configuration command register.

IRDY#/TRDY# Latency

The specification characterizes PCI as a “low latency, high throughput I/O bus.” In keeping with that objective, the specification imposes limits on the number of wait states initiators and targets can add to a transaction.

Specifically, an initiator must assert IRDY# within 8 clock cycles of the assertion of FRAME# on the first data phase and within 8 clock cycles of the deassertion of IRDY# on subsequent data phases. As a general rule, master latency should be fairly short because the agent shouldn't request the bus until it is either ready to supply data for a write transaction or accept data for a read transaction.

Similarly, a target is required to assert TRDY# within 16 clocks of the assertion of FRAME# for the first data phase and within 8 clocks

of the completion of the previous data phase. This acknowledges the case where a target may need additional time to get a buffer ready when it is first selected but should be able to deliver subsequent data items with relatively short latency.

Fast Back-to-back Transactions

Normally, an idle turnaround cycle must be inserted between transactions to avoid contention on the bus. However, there are some circumstances under which the turnaround cycle can be eliminated thus improving overall performance. The primary requirement is that there be no contention on any PCI bus line.

Depending on circumstances, either the master or the target can guarantee lack of contention.

If a master keeps its REQ# line asserted after it asserts FRAME#, it is asking to execute another transaction. As long as its GNT# remains asserted (i.e. no other agents are requesting the bus), the next transaction will be executed by the same master. There is no contention on any lines driven by the master as long as the first transaction was a write.

Furthermore, the second transaction must address the same target so that the same agent is driving DEVSEL# and TRDY#. This implies that the master has knowledge of target address boundaries in order to know that it is addressing the same one.

Figure 3-5 illustrates fast back-to-back timing for a master. The master keeps REQ# asserted through the first transaction to request a second transaction. In clock 3 the master drives write data followed immediately in clock 4 by the address phase of the next transaction. This example shows the second transaction as being a write. If it were a read, a turnaround cycle would need to be inserted after the second address phase.

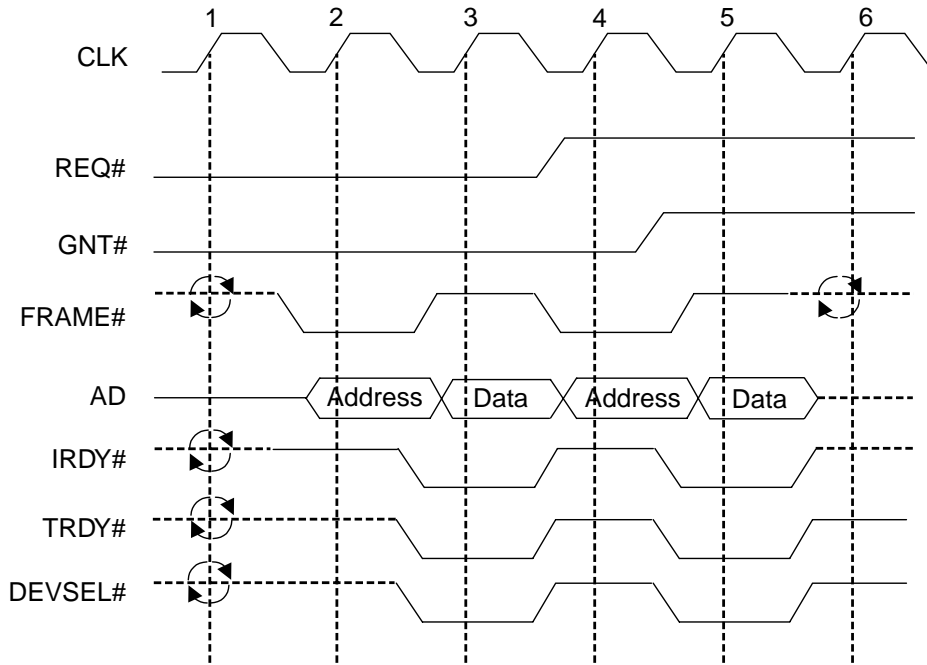


Figure 3-5: Fast back-to-back timing for a master.

The entire community of targets on a bus segment can guarantee a lack of bus contention if:

- All targets have medium or slow address decoders *AND*
- All targets can detect the start of a new transaction without the transition through the idle state

Because fast back-to-back timing includes no idle cycle (both FRAME# and IRDY# deasserted), targets must detect a new transaction as the falling edge of FRAME#. Such targets have the FAST BACK-TO-BACK CAPABLE bit set in their configuration status registers. If all targets are fast back-to-back capable and all targets are either medium or slow, then the target of the second half of a fast back-to-back transaction can be different because the delay in DEVSEL# guarantees a lack of contention.



Transaction Termination—Master

A transaction is “normally” terminated by the master when it has read or written as much data as it needs to. The master terminates a normal transaction by deasserting FRAME# during the last data phase. There are two circumstances under which a master may be forced to terminate a transaction prematurely

Master Preempted

If another agent requests use of the bus and the current master's latency timer expires, it must terminate the current transaction and complete it later.



Master Abort

If a master initiates a transaction and does not sense DEVSEL# asserted within four clocks, this means that no target claimed the transaction. This type of termination is called a master abort and usually represents a serious error condition.

Transaction Termination—Target

There are also several reasons why the target may need to terminate a transaction prematurely. For example, its internal buffers may be full and it is momentarily unable to accept more data. It may be unable to meet the maximum latency requirements of 16 clocks for first word latency or 8 clocks for subsequent word latency. Or it may simply be busy doing something else.

The target uses the STOP# signal together with other bus control signals to indicate its need to terminate a transaction. There are three types of target-initiated terminations:

Retry: Termination occurs before any data is transferred. The target is either busy or unable to meet the initial latency requirements

and is simply asking the master to try this transaction again later. The target signals retry by asserting **STOP#** and not asserting **TRDY#** on the initial data phase.

Disconnect: Once one or more data phases are completed, the target may terminate the transaction because it is unable to meet the subsequent latency requirement of eight clocks. This may occur because a burst crosses a resource boundary or a resource conflict occurs. The target signals a disconnect by asserting **STOP#** with **TRDY#** either asserted or not.

Target-Abort: This indicates that the target has detected a fatal error condition and will never be able to complete the requested transaction. Data may have been transferred before the Target-Abort is signaled. The target signals Target-Abort by asserting **STOP#** at the same time as deasserting **DEVSEL#**.

Retry — The Delayed Transaction

Figure 3-6 shows the case of a target retry. The target claims the transaction by asserting **DEVSEL#** but, at the same time, signals that it is not prepared to participate in the transaction at this time by asserting **STOP#** instead of **TRDY#**. The master deasserts **FRAME#** to terminate the transaction with no data transferred. In the case of a retry the master is obligated to retry the exact same transaction at some time in the future.

A common use for the target retry is the **delayed transaction**. A target that knows it can't meet the initial latency requirement can "memorize" the transaction by latching the address, command and byte enables and, if a write the write data. The latched transaction is called a **Delayed Request**. The target immediately issues a retry to the master and begins executing the transaction internally. This allows the bus to be used by other masters while the target is busy.

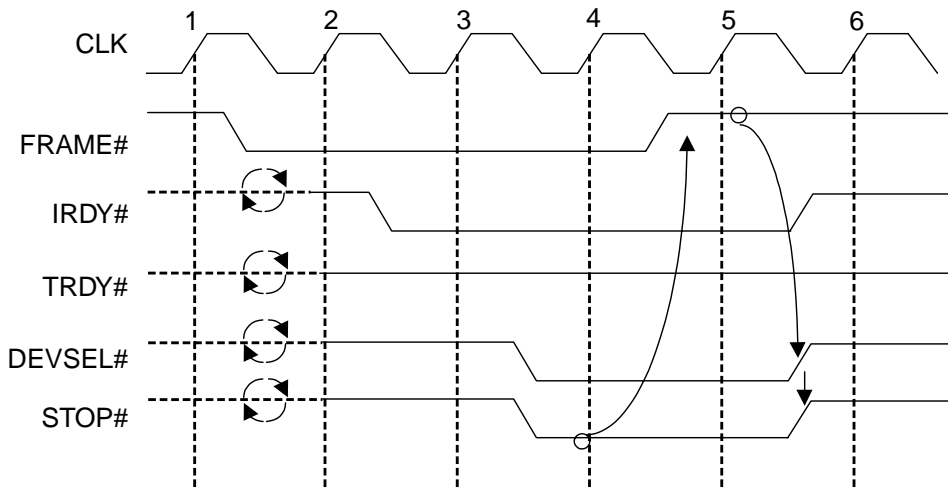


Figure 3-6: Target retry.

Later when the master retries the exact same transaction and the target has completed the transaction, the target replies appropriately. The result of completing a Delayed Request produces a *Delayed Completion* consisting of completion status and data if the request was a read. Bus bridges, particularly bridges to slower expansion buses like ISA, make extensive use of the delayed transaction.

Note that in order for the target to recognize a transaction as the retry of a previous transaction, the master must duplicate the transaction exactly. Specifically, the address, command and byte enables and, if a write the write data, must be the same as when the transaction was originally issued. Otherwise it looks like a new transaction to the target. ←

Typical targets can handle only one delayed transaction at a time.

While a target is busy executing a delayed transaction it must retry all other transaction requests without memorizing them until the current transaction completes.

Note that there is a possibility that another master may execute exactly the same transaction after the target has internally completed a delayed transaction but before the original initiator retries. The target can't distinguish between two masters issuing the same transaction so it replies to the second master with the Delayed Completion information. When the first master retries, it looks like a new transaction to the target and the process starts over.

What happens if a master never retries the transaction? Targets capable of executing delayed transactions must implement a **Discard Timer**. A target must discard a Delayed Completion if the master has not retried the transaction after 2^{32} clocks.

Disconnect

The target may terminate a transaction with a Disconnect if it is unable to meet the maximum latency requirements. There are two possibilities—either the target is prepared to execute one last data

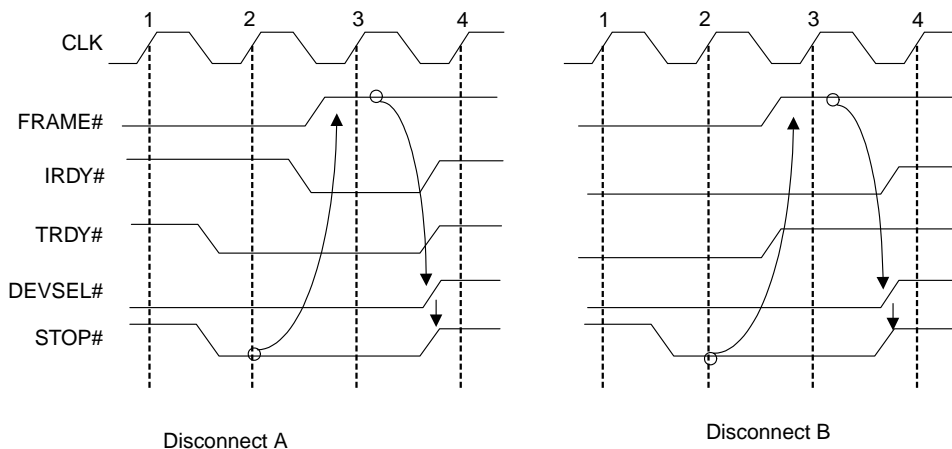


Figure 3-7: Target disconnect — with data.

phase or it is not. If TRDY# is asserted when STOP# is asserted, the target indicates that it is prepared to execute one last data phase. This is called a “Disconnect with data”. There are two cases as shown in Figure 3-7: Disconnect-A and Disconnect-B. The only difference between the two is the state of IRDY# when STOP# is asserted. In the case of Disconnect-A, IRDY# is not asserted when STOP# is asserted. The master is thus notified that the next transfer will be the last. It deasserts FRAME# on the same clock that it asserts IRDY#.

In Disconnect-B, the final transfer occurs in the same clock when STOP# is sampled asserted. The master deasserts FRAME# but the rules require that IRDY# remain asserted for one more clock. To prevent another data transfer, the target must deassert TRDY#. In both cases the target must not deassert DEVSEL# or STOP# until it detects FRAME# deasserted. The target may resume the transaction later at the point where it left off.

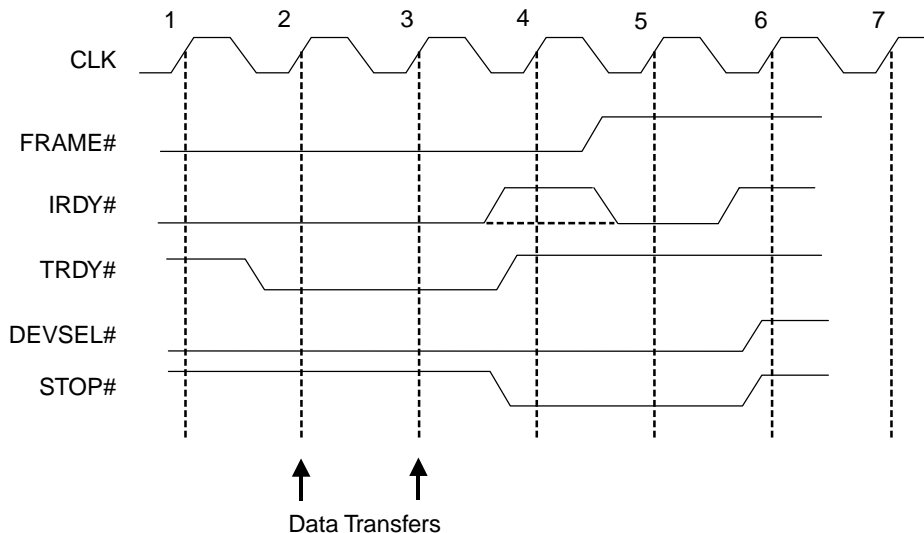


Figure 3-8: Target disconnect — without data.

If the target asserts **STOP#** when **TRDY#** is not asserted, it is telling the initiator that it is not prepared to execute another data phase. This is called a “**Disconnect without data**”. The initiator responds by deasserting **FRAME#**. There are two possibilities: either **IRDY#** is asserted when **STOP#** is detected or it is not. In the latter case, the initiator must **assert IRDY#** in the clock cycle where it deasserts **FRAME#**. This is illustrated in Figure 3-8. Note that the Disconnect without data looks exactly like a Retry except that one or more data phases have completed.

Target Abort

As shown in Figure 3-9, Target Abort is distinguished from the previous cases because **DEVSEL#** is not asserted at the time that **STOP#** is asserted. Also, unlike the previous cases where the master is invited (or required) to retry or resume the transaction, Target Abort

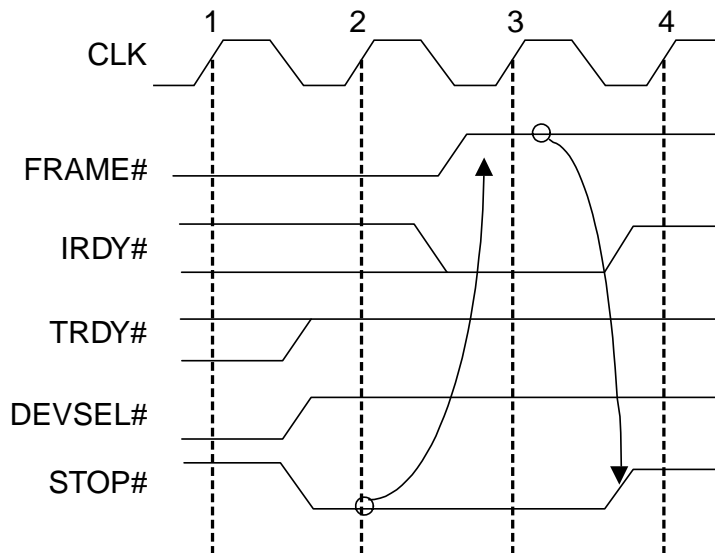


Figure 3-9: Target Abort.

specifically says do not retry this transaction. Target Abort typically means that the target has experienced some fatal error condition. The master should probably raise an exception back to its host. One or more data phases may have completed before the target signaled Target Abort.

Error Detection and Reporting

Parity Generation & Detection — PAR and PERR#

All bus agents are required to generate even parity over the AD and C/BE# busses. The result of the parity calculation appears on the PAR line. Even parity means that the PAR line is set so that the number of bus lines in the logical 1 state, including PAR, is even. All 32 AD lines are always included in the parity calculation even if they are not being used in the current transaction. This is another reason why the driving agent must always drive all 32 AD lines.

With two minor exceptions, all agents are required to have the ability to check parity. The two exceptions are:

- Devices (i.e. silicon) designed exclusively for use on a motherboard.
- “Devices that never deal with, contain or access any data that represents permanent or residual system or application state, e.g. human interface and video/audio devices”.

The agent driving the AD bus during any clock phase computes even parity and places the result on the PAR line one clock cycle later. The receiving agent checks the parity and, upon detecting an error, may assert PERR#. So on a read transaction, PAR is driven by the target and PERR# is driven by the initiator. The target then senses PERR# and may take action if appropriate. On a write transaction, the opposite occurs. ←

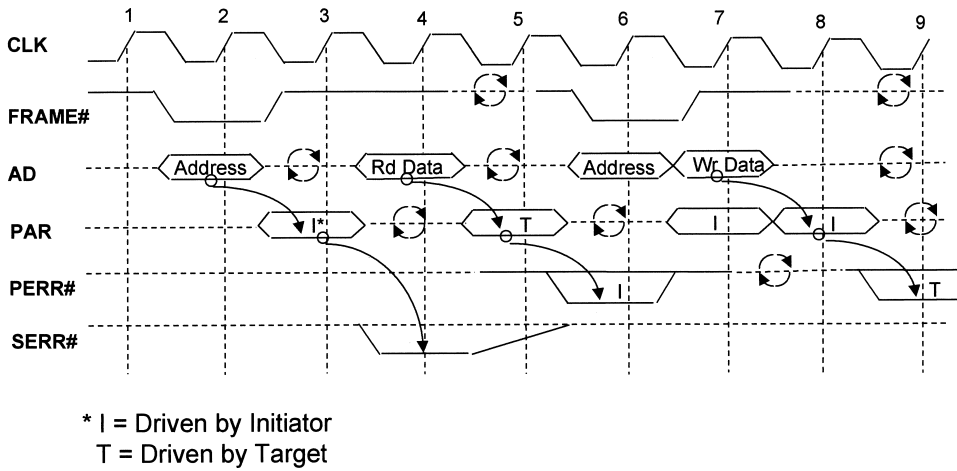


Figure 3-10: Timing diagram for parity generation and detection.

Figure 3-10 illustrates the timing of parity generation and detection. The key point to note is that one clock cycle is required to generate parity and another is required to check it. Looking at it in more detail:

Clock

- 2 Address phase. The selected master places the target address and command on the bus. All targets latch this information.
- 3 Turnaround cycle for read transaction. The master places computed parity for the address phase on PAR.
- 4 If any agent has detected a parity error in the address phase it asserts SERR# here. This is the first read data phase and also a turnaround cycle for PAR.
- 5 Target places computed parity on PAR. Otherwise this is an idle cycle.

- 6 Master reports any parity error here by asserting PERR#. This also happens to be the address phase for the next transaction.

Clocks 7 to 9 illustrate the same process for write transactions. Note that no turnaround is required on either AD or PAR.

Note that because SERR# is open-drain it may require more than one clock cycle to return to the non-asserted state.

Upon detection of a parity error, the agent that is checking parity must set the DETECTED PARITY ERROR bit in its Configuration Status Register. If the PARITY ERROR RESPONSE bit in its Configuration Command Register is a 1, then it asserts PERR#. Any error recovery strategies are the responsibility of the host attached to the agent that detects the error.

Although bus agents are required to generate parity, there is no requirement that they act on a detected parity error. The ability to detect parity errors and take action is controlled by bits in the device's Configuration Control Register.

System Errors — SERR#

PERR# only reports parity errors during data phases. That is, it is intended to signal an error condition between a specific master/target pair. Parity is also generated and checked during the address phase. But if there is an error on the address bus, which target should check and report that error? The answer is they all should. Any target which detects a parity error during the address phase asserts SERR# and sets the SIGNALLED SYSTEM ERROR bit in its Status Register if the SERR# ENABLE bit in its Command Register is set. SERR# is an open-drain signal so it is permissible for more than one agent to assert it simultaneously.

An agent that “thinks” it has been selected in the presence of an address parity error can respond in one of three ways:

- Claim the transaction and proceed as if everything were OK
- Claim the transaction and terminate with Target-Abort
- Don’t claim the transaction and let the master terminate with Master-Abort

SERR# is also used to signal parity errors on Special Cycles because, like the address phase, a Special Cycle is not directed at a specific target. It may also be used to signal other catastrophic error conditions.

The assertion of SERR# should be considered a fatal condition. The specification suggests that SERR# would most likely be handled as a non-maskable interrupt.

Summary

The PCI specification defines a precise set of rules, called a protocol, for how data is transferred across the bus. Every bus transaction consists of an address phase and one or more data phases. Both the initiator and the target of a transaction can regulate the flow of data by controlling their respective “ready” signals, IRDY# and TRDY#.

A transaction may be terminated by either the initiator or the target. One reason the target may terminate a transaction is because it is temporarily busy or unable to meet the initial latency requirements. In this case it tells the initiator to “Retry” the transaction later.

All PCI agents are required to generate even parity on the AD and C/BE lines. With two exceptions, all agents are required to have

the ability to check parity whether or not they choose to take any action in response to a detected parity error. Parity errors during data phases are reported on the PERR# line. The SERR# line is used to report parity errors during address phases and Special Cycle transactions. It can also be used to report other system errors. SERR# is considered to be a fatal condition.

CHAPTER 4

Optional and Advanced Features

The previous chapter described the basic data transfer protocol, the process of moving data from one place on the bus to another. PCI incorporates a number of optional and advanced features that substantially extend its capabilities.

Interrupt Handling

The PCI specification considers interrupt support “optional.” There are four interrupt lines, INTA# to INTD#, defined on the PCI connector. However, a *single-function* device can only use INTA#. *Multi-function* devices can use any combination of the four interrupt signals. A single-function device is a component or add-in board that embodies exactly one logical device or function. A multi-function device may incorporate anywhere from two to eight logical functions. Each function has its own PCI configuration space. In all cases, the interrupt connection is encoded in the read-only Interrupt Pin register of the function’s configuration space. Each function may only be connected to a single interrupt line.

PCI interrupts are defined as level-sensitive, assertion-low and asynchronous with respect to the PCI clock. A device requests

attention from its device driver by asserting (driving low) its INTx# signal. The interrupt signal remains asserted until the device driver clears the condition that caused the interrupt. The device then deasserts its INTx#.

Note that the INTx# signals are not necessarily bussed. They are open-drain, so they could be and in fact often are. The specification allows complete freedom in the matter of how interrupt sources are connected to the interrupt controller. But as in many similar situations, the specification suggests an implementation that has become a de facto standard.

Consider an interrupt controller with four IRQs available for PCI usage. We'll call them IRQW, IRQX, IRQY and IRQZ. Now consider that we have four PCI slots (numbered 0 to 3) on our motherboard, each of which has four interrupt pins—INTA, INTB, INTC and INTD. We connect the PCI interrupt pins to the interrupt controller inputs as shown in Table 4-1 and illustrated graphically in Figure 4-1.

The result of this configuration is that the INTA from each of the slots is connected to a different interrupt input. Since most devices are single function and thus can only use INTA, each device gets a separate interrupt input. The concept can be extended beyond four slots or devices. The pattern simply repeats itself and the INTA pins from two slots share the same interrupt input.

The shared nature of PCI interrupts introduces a complexity to device drivers that is typically not present in drivers for ISA devices. Since interrupts can be shared, the Interrupt Service Routines (ISRs) for devices sharing an interrupt must cooperate in servicing interrupt requests. In an environment such as ISA where interrupts are unique, an ISR can generally assume that when it is invoked, its device caused the interrupt. In a shared environment that's not the case.

Table 4-1

Interrupt Controller Inputs	PCI Slot 0	PCI Slot 1	PCI Slot 2	PCI Slot 3
IRQW	INTA	INTB	INTC	INTD
IRQX	INTB	INTC	INTD	INTA
IRQY	INTC	INTD	INTA	INTB
IRQZ	INTD	INTA	INTB	INTC

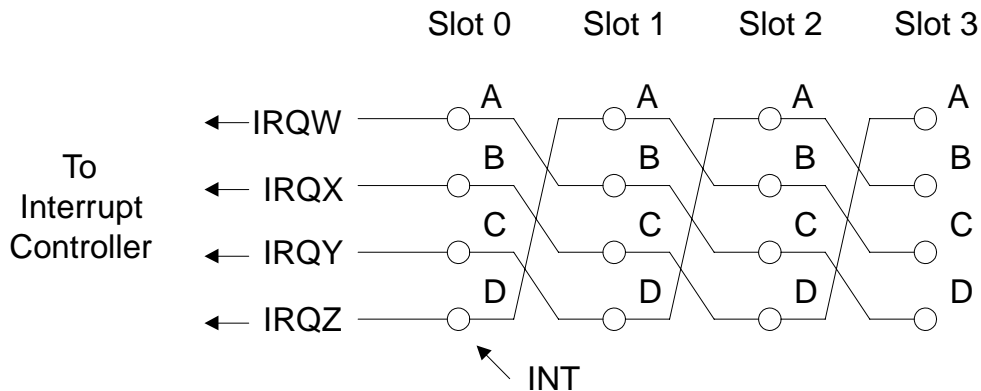


Figure 4-1: Connections between the PCI interrupt pins and the interrupt controller inputs.

When an interrupt occurs, the ISRs for *all* devices sharing the interrupt line must be invoked and they must each test their respective device(s) to find the one that asserted the interrupt signal.

The mechanism for invoking multiple ISRs is called “chaining.”

The Interrupt Acknowledge Command

Figure 4-2 illustrates the Interrupt Acknowledge command, which is generated by the agent whose interrupt *input* is asserted. In a typical single processor system this would be the main processor. Only one agent in the system responds to the Interrupt Acknowledge — typically the APIC.

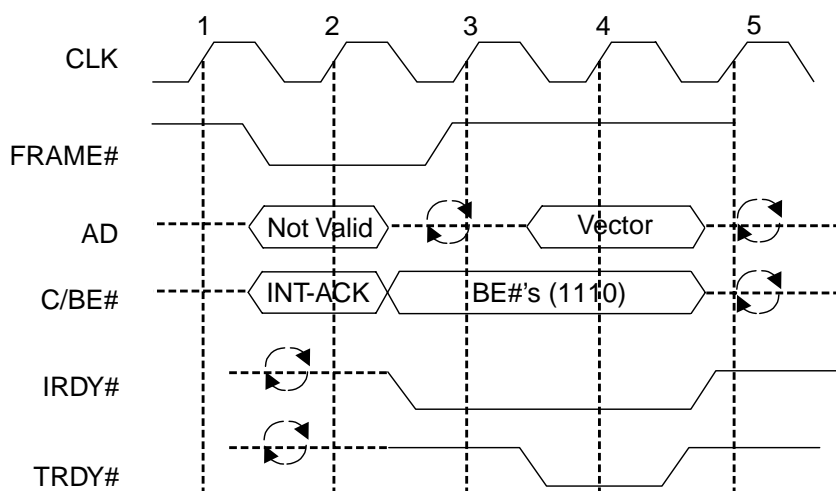


Figure 4-2: Interrupt Acknowledge command.

The AD bus is invalid during the address phase because the target of the transaction, the APIC, recognizes it is being selected by virtue of the Interrupt Acknowledge command. But again, the AD bus must be driven to generate valid parity and prevent the receiver inputs from floating.

The Interrupt Acknowledge cycle proceeds like any other PCI cycle. The initiator asserts IRDY#. The interrupt controller asserts DEVSEL# to claim the transaction and TRDY# when it is ready to

supply the interrupt vector. The C/BE# bus indicates which bytes of the interrupt vector are valid. Because PCI is processor independent, we don't necessarily know the nature or size of an interrupt vector. That's a function of the host processor architecture. The example shows a typical x86 system where the interrupt vector is a single byte.

“Special” Cycle

The Special Cycle provides a mechanism to broadcast information simultaneously to multiple targets. The specification suggests that it is a useful way to convey **sideband information** to one or more devices without the need for additional wires on the backplane. One use for this facility is to broadcast processor status such as Halt and Shutdown.

By definition, a Special Cycle is not directed at a specific target but rather to any and all targets that have an interest in the message being broadcast. This has several consequences:

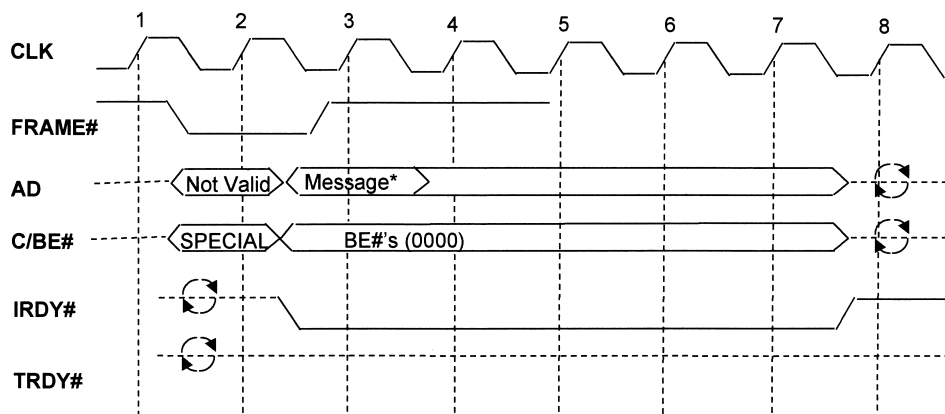
- The AD bus is not valid during the address phase. Of course it must still be driven in order to generate parity correctly.
- Targets do not assert DEVSEL# or TRDY#.
- Since DEVSEL# is not asserted, the only way for the transaction to terminate is with a Master Abort.

During the data phase AD[15:0] conveys a predefined message type. AD[31:16] may optionally carry message-dependent data. Table 4-2 shows the currently defined messages.

Figure 4-3 shows the timing of a Special Cycle.

Table 4-2

Message Code (AD[15:0])	Message Type
0x0000	<i>Shutdown.</i> Processor is entering a shut-down mode, probably due to an unrecoverable software problem.
0x0001	<i>Halt.</i> Processor has executed a halt instruction.
0x0002	<i>X86-specific message.</i> AD[31:16] contains an Intel-specific message.
0x0003 to 0xffff	<i>Reserved.</i> Assigned by PCI SIG steering committee.



*Message must be latched on first cycle of IRDY

Figure 4-3: Timing diagram of a Special Cycle.

Clock

- 2 Master asserts FRAME#. AD is not valid, C/BE# = Special Cycle.
- 3 Master places message on AD, asserts IRDY# and deasserts FRAME#. All targets must latch the message on the first clock in which IRDY# is asserted.
- 4–7 Master waits to time out with a Master Abort.

A Special Cycle is always a full DWORD transfer so all four C/BE# lines are asserted during the data phase.

Because of the Master Abort requirement, a Special Cycle is a minimum of six clock cycles (more if the master delays the assertion of IRDY#).

Multiple data phases are permitted but at present there are no messages that would require more than one data phase. The requirement that data be latched on the first clock following the assertion of IRDY# implies that IRDY# must be deasserted for at least one clock before executing a second, or subsequent, data phase.

64-bit Extensions

The PCI specification defines an optional extension to 64 bits for memory targets in a way that allows 64-bit agents to seamlessly inter-operate with 32-bit agents. 64-bit transfers only occur if both the initiator and target support 64 bits. Otherwise, transfers default to 32 bits. The “negotiation” to transfer 64 bits occurs on a per-transaction basis and is facilitated by two optional signals; REQ64# and ACK64#.

64-bit Bus

Figure 4-4 shows a 64-bit transaction. A 64-bit master asserts REQ64# at the same time as FRAME# in clock 2. In this case the selected target also supports 64-bit transfers so it asserts ACK64# together with DEVSEL# in clock 3. This example shows a read transaction. The target places the low-order 32 bits on AD[31:0] and the high-order on AD[63:32]. The master places byte enable information for AD[63:32] on C/BE#[7:4]. Parity for AD[63:32] and C/BE#[7:4] is computed and checked on PAR64.

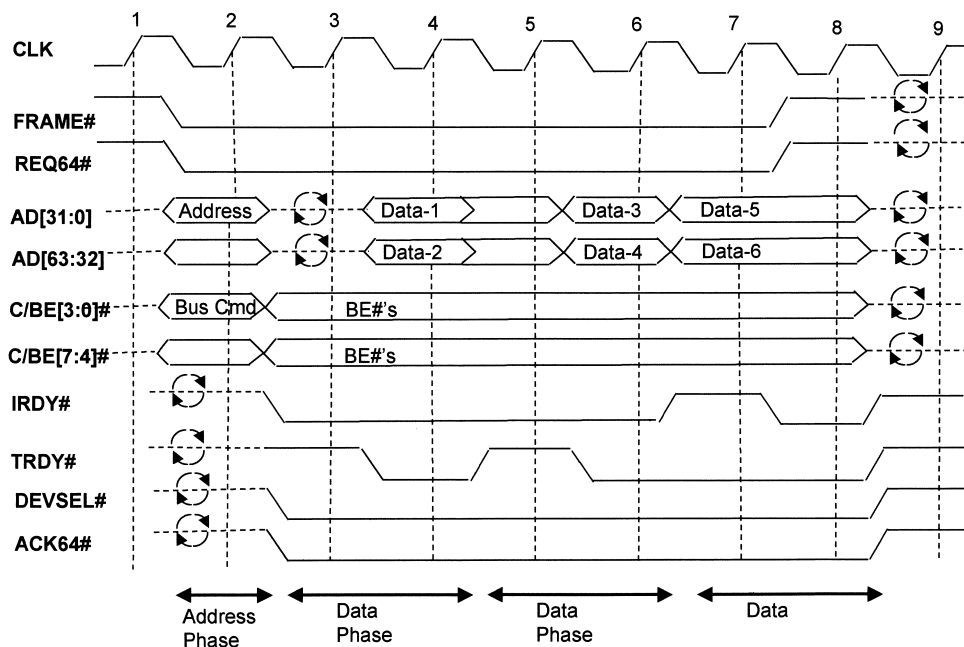


Figure 4-4: Timing diagram of a 64-bit transaction.

Figure 4-5 shows what happens when a 64-bit master executes a write transaction to a 32-bit target. In clock 2 the master asserts REQ64# as before. In clock 3 the master places up to eight bytes of data on AD[63:0] and corresponding byte enables on C/BE#[7:0]. At the same time the master detects DEVSEL# asserted with ACK64# not asserted indicating that the target only supports 32 bits. In clock 4 the master moves the upper four bytes (Data-2) down to AD[31:0].

A 64-bit target communicating with a 32-bit master knows that it must revert to 32 bits because it detects REQ64# unasserted.

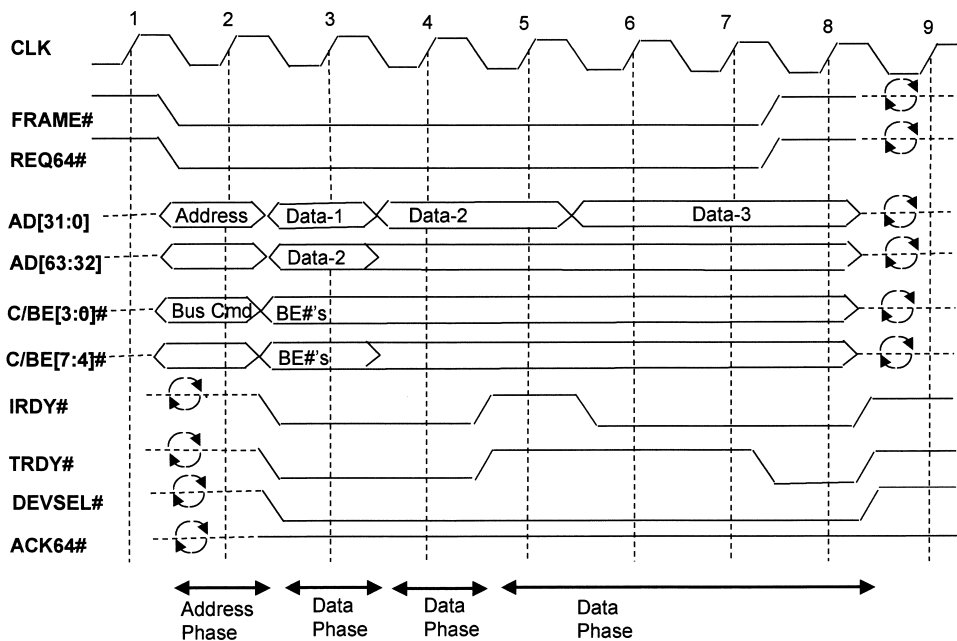


Figure 4-5: Execution of a write transaction from a 64-bit master to a 32-bit target.

64-bit Addressing — The Dual Address Cycle

There is another optional mechanism that permits 32-bit agents to address memory locations above 4 GBytes. This is accomplished by adding a second address phase to a transaction in the form of a Dual Address Cycle command (DAC). Note that even if a target supports DAC, standard single address commands (SAC) must be used for locations below 4 GBytes. 64-bit addressing is only supported in the memory space.

Figure 4-6 illustrates the Dual Address Command. In clock 2, the master issues the DAC command on C/BE#[3:0] and puts the low-order address on AD[31:0]. A 64-bit master puts the high-order address on AD[63:32] and the transaction command (in this case Mem Read) on C/BE#[7:4]. In clock 3 the master places the high-order address on AD[31:0] and the normal transaction command on C/BE#[3:0].

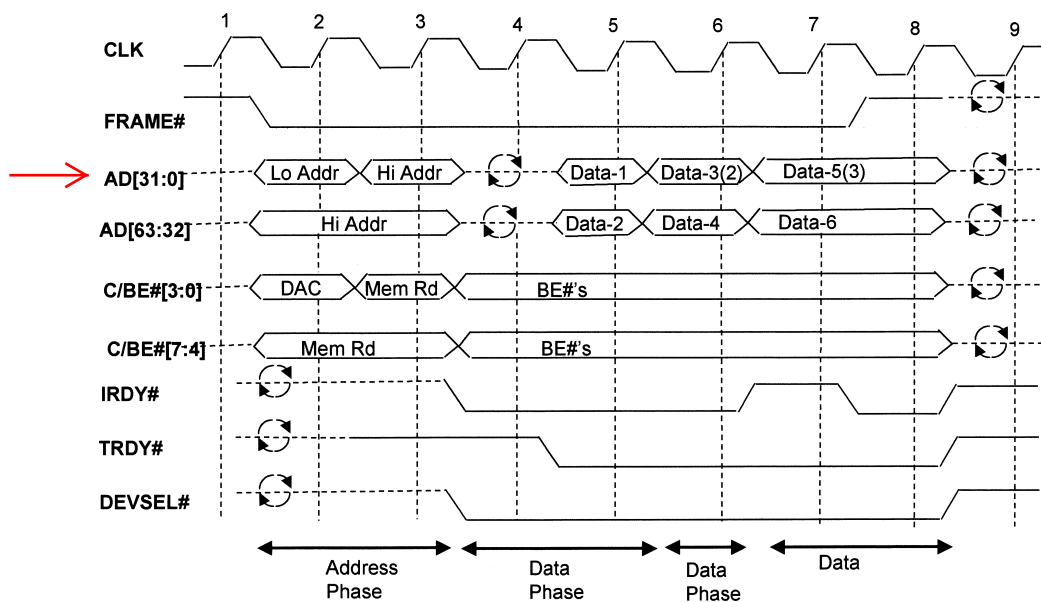


Figure 4-6: Dual Address Command used in 64-bit addressing.

A 64-bit target can decode the entire address and transaction command during the first address phase. However, the master must still execute the DAC because it won't know until the target is selected that the target is 64-bit capable. But by decoding the address and command in the first address phase, a medium or slow DEVSEL target saves one clock cycle.

The DAC command is always exactly one clock cycle. Consequently, address stepping is not permitted for the DAC command.

Summary

The topics covered in this chapter, interrupts, the Special Cycle command and 64-bit extensions, are all optional features of PCI.

The specification provides for four interrupt signals from each PCI device. A single function device may only use one of the interrupt signals, INTA#. Multi function devices may use any combination of the four. The routing of the four signals among the devices in a system is at the discretion of the designer. Interrupts are defined as assertion low, level-sensitive and asynchronous to the clock.

The Special Cycle command is a broadcast mechanism that may, in certain cases, substitute for sideband signals. Special Cycles are not directed at a specific target and so no target responds. The Special Cycle is always terminated by a Master Abort.

The PCI bus may be extended to 64 bits in a way that allows 32-bit agents to interoperate with 64-bit agents. The Dual Address command (DAC) provides a way for 32-bit agents to access a 64-bit memory address space.

CHAPTER 5

Electrical and Mechanical Issues

This chapter summarizes the electrical signaling environment of PCI and mechanical issues related to add-in cards. The objective is to highlight the electrical features of PCI without getting bogged down in details that are primarily of interest to integrated circuit designers. To dig deeper, refer to the current revision of the PCI specification.

A “Green” Architecture

Many aspects of PCI’s electrical specification are explicitly intended to reduce power consumption. Not only is this environmentally correct, it is essential for mobile and portable devices. PCI is based on CMOS, which means that steady state DC currents are minimal and in fact most DC drive current goes to pull-up resistors. The bus protocol assures that bus receivers are not allowed to float such that they might oscillate and consume unnecessary power. Finally, the most interesting aspect of low power consumption is that PCI is based on “reflected wave” switching rather than the more traditional “incident wave” switching.

Incident Wave Switching — the Old Way

Traditional bus architectures have stressed the need for proper termination of all bus lines to prevent unwanted reflections. Every

signal on a backplane bus is really a transmission line with a characteristic impedance of about 120 ohms. If the ends are not terminated, a pulse travelling down the line will be reflected back from the end possibly causing unwanted interference.

The solution is to terminate both ends of the bus in the characteristic impedance. Figure 5-1 shows a typical termination arrangement. The “Thevenin equivalent” impedance of the 180/330 ohm divider is 120 ohms while the divider maintains an open-circuit voltage of 3.4 volts.

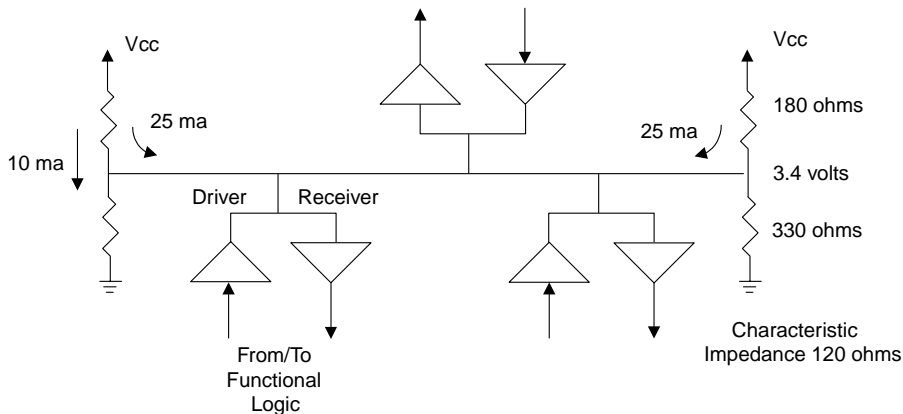


Figure 5-1: “Traditional bus” — incident wave switching.

This “incident wave” approach is fundamentally incompatible with the objective of low power consumption. Each of the divider networks in this example consumes 10 ma, or 20 ma per signal line. For the 46 bussed signals of the PCI, that’s almost an amp. At 5 volts, that’s about 5 watts just for the termination resistors!

Each driver must be capable of sinking 50 ma when it drives the line to the low voltage state. Such high power drivers require a lot of silicon real estate and dissipate substantial power themselves.

The current surges resulting from many drivers switching on or off at once can cause large noise spikes on the power lines, not to mention crosstalk between bus signals.

Reflected Wave Switching — the New Way

Not surprisingly then, PCI takes a radically different approach to bus termination. It eliminates the termination networks altogether and actually *takes advantage* of the reflected wave front. As shown in Figure 5-2, a PCI bus driver is designed to drive the line about “half way”, and *only* half way. As the wave front propagates to the end of the line, it is insufficient to switch the receivers that it passes. When the wave front reaches the end of the bus, it is reflected back *doubled in magnitude*. So the receivers switch as the wave front passes them the second time going in the other direction.

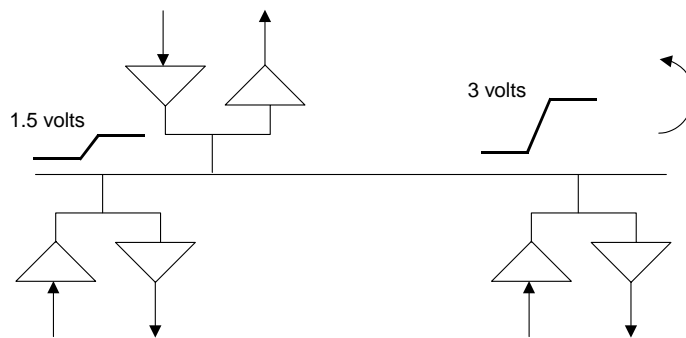


Figure 5-2: Reflected wave switching.

Reflected wave switching requires twice the propagation time of incident wave switching. It also requires much more careful attention to trace length and layout. The specification limits propagation time to 10 ns at 33 MHz and, as we’ll see shortly, sets very specific limits on trace length.

Preventing Receiver Inputs from Floating

If a tri-state bus line is not driven, i.e. it is tri-stated, and it is not terminated with a pull-up resistor, it is said to be “floating”. The voltage level of a floating bus line tends to settle around the switching point of the bus receivers. This may cause the receiver to oscillate and consume more power than it should. There are basically two approaches to preventing a bus line from floating:

1. Always drive the line, or
2. Pull it up to the signaling voltage (3.3V or 5V) through a resistor

The PCI spec requires that AD[31::0], PAR and C/BE[3::0] be driven to stable states when the bus is idle. If the bus is parked, the agent on which it is parked should drive AD and C/BE. If the bus is not parked then the central resource should drive AD and C/BE. AD[63::32], PAR64 and C/BE[7:4] require pull up resistors because otherwise they would float when a 32-bit agent is driving the bus.

The control signals all require pull ups since they can't be driven while the bus is idle. This includes FRAME#, DEVSEL#, IRDY#, TRDY#, STOP#, SERR#, PERR#, LOCK#, REQ64#, ACK64# and the INTx# signals. Typical resistor values are 2.7 kilohm in the 5V signaling environment and 8.2 kilohm in the 3V signaling environment.

Signaling Environments—3.3V and 5V

At the present time most computer busses use 5 volt TTL-compatible signaling levels. There is, however, a trend toward 3.3 volt logic, particularly in portable and mobile environments where power consumption must be minimized. Unfortunately, these two logic families don't mix well together, so PCI has developed

separate electrical specifications for each signaling environment. When we speak of a “signaling environment,” we are referring to the *signal level* on the PCI pins and not to the voltage that powers the board.

The motherboard (including connectors) defines the signaling environment for the bus, whether it be 5V or 3.3V. A 5V expansion board is designed to work only in a 5V signaling environment. Similarly, a 3.3V board works only in a 3.3V signaling environment. To prevent boards from being installed incorrectly, the connector has different keying for the two signaling environments (see Figure 5-3).

There is also a provision for a “universal board”, one that can operate in either signaling environment. A universal board has notches for both signaling keys. There are three pins on the connector labeled Vio. A universal board powers its PCI transceivers from the Vio pins. The motherboard connects the Vio pins to the power rail corresponding to system’s signaling environment.

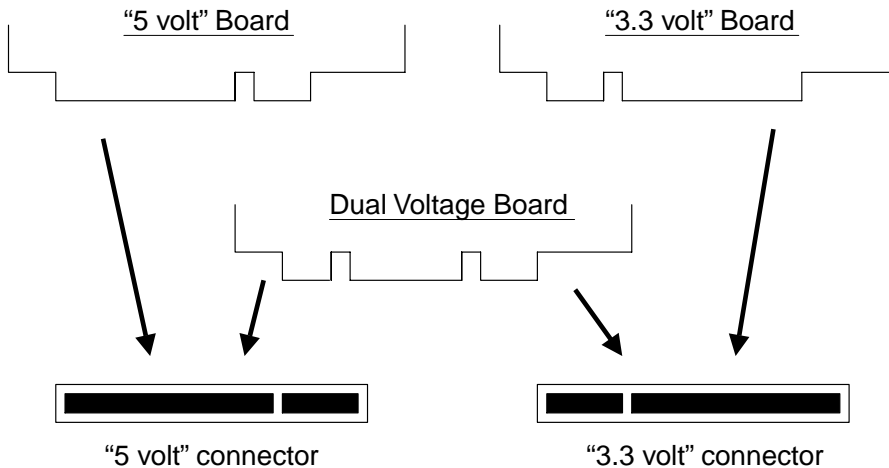


Figure 5-3: 3.3V vs. 5V keying.

PCI defines four power rails: +5V, +3.3V, +12V and –12V. Systems implementing the 3.3V signaling environment are required to provide all four supplies. Systems with 5V signaling are not required to provide 3.3V but it is strongly “encouraged.”

Many of the figures, tables and their accompanying notes in the remainder of this chapter have been taken from Rev. 2.2 of the PCI Specification. As always, refer to the specification for more details.

5 Volt Signaling Environment

The 5 volt specifications are given in terms of absolute voltages based on standard TTL levels.

DC Specifications

Table 5-1 summarizes the 5 volt DC specifications.

Notes for Table 5-1

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 5 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].

Table 5-1: 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	$I_{out} = -2 \text{ ma}$	2.4		V	
V_{ol}	Output Low Voltage	$I_{out} = 3 \text{ ma}, 6 \text{ ma}$		0.55	V	2
C_{in}	Input Pin Capacitance			10	pF	3
C_{clk}	CLK Pin Capacitance		5	12	pF	
C_{IDSEL}	IDSEL Pin Capacitance			8	pF	4
L_{pin}	Pin Inductance			20	nH	5
I_{off}	PME# Input Leakage	$V_o \leq 5.25 \text{ V}$ V_{cc} off or floating		1	μA	6

5. This is a recommendation, not an absolute requirement. The actual value should be provided with the component data sheet.
6. This input leakage is the maximum allowable leakage in the PME# open drain driver when power is removed from V_{cc} of the component. This assumes that no event has occurred to cause the device to attempt to assert PME#.

AC Specifications

For the reflected wave switching mechanism to work properly, the output driver must source or sink enough instantaneous current to develop the initial half amplitude voltage step on a bus wire loaded with PCI components. But it must not source or sink *too much* current such that it drives the line too far possibly resulting in undesirable reflections. Table 5-2 summarizes the AC specifications for the 5 volt signaling environment while Figure 5-4 shows the V/I curves that characterize a PCI driver. These numbers are based on a maximum of ten AC loads where each expansion board connector is considered one AC load. Typical configurations are six motherboard loads plus two expansion connectors or two motherboard loads and four expansion connectors.

Notes for Table 5-2

1. Refer to the V/I curves in Figure 5-4. Switching current characteristics for REQ# and GNT# are permitted to be one half of that specified here; i.e., half size output drivers may be used on these signals. This specification does not apply to CLK and RST# which are system outputs. “Switching Current High” specifications are not relevant to SERR#, PME#, INTA#, INTB#, INTC#, and INTD# which are open drain outputs.
2. 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 5-4. 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.

Table 5-2: AC specifications for 5V signaling.

Symbol	Parameter	Condition	Min	Max	Units	Notes
$I_{oh(AC)}$	Switching Current High	$0 < V_{out} \leq 1.4$	-44		mA	1
		$1.4 < V_{out} < 2.4$	$\frac{-44 + (V_{out} - 1.4)}{0.024}$		mA	1,2
		$3.1 < V_{out} < V_{cc}$		Eq. A		1,3
	(Test Point)	$V_{out} = 3.1$		-142	mA	3
$I_{ol(AC)}$	Switching Current Low	$V_{out} \geq 2.2$	95		mA	1
		$2.2 > V_{out} > 0.55$	$\frac{V_{out}}{0.023}$		mA	1
		$0.71 > V_{out} > 0$		Eq. B		1,3
	(Test Point)	$V_{out} = 0.71$		206	mA	3
I_{cl}	Low Clamp Current	$-5 < V_{in} \leq -1$	$\frac{-25 + (V_{in} + 1)}{0.015}$		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

- 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 (Figure 5-5) 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.

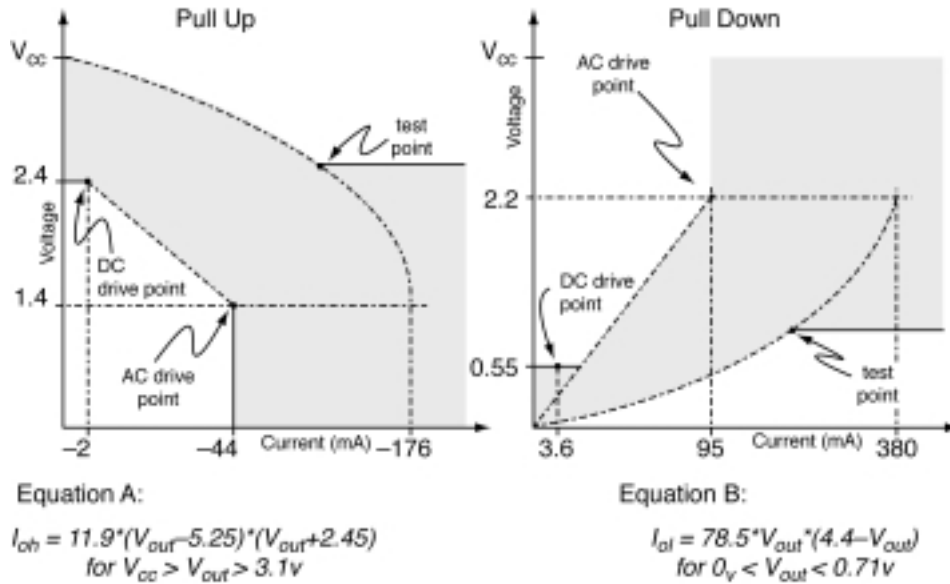


Figure 5-4: Characteristic V/I curves for a PCI driver in the 5V signaling environment.

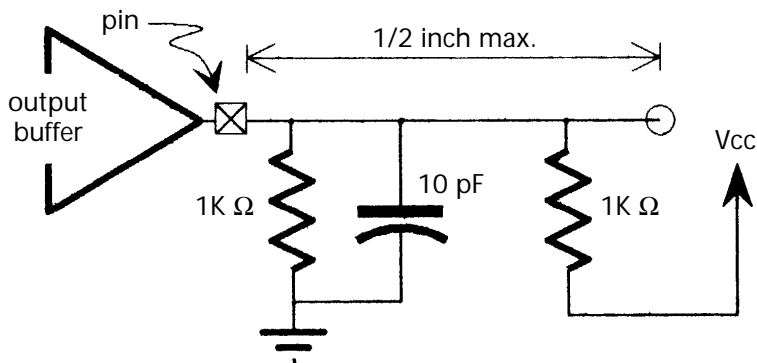


Figure 5-5: Specified load for output rise and fall slew rate measurements.

3.3 Volt Signaling Environment

The 3.3 volt environment is based on V_{cc} -relative switching voltages and is optimized for CMOS. The intent is that components connect directly together, whether on the motherboard or an expansion board, without any external buffers or other “glue.”

DC Specifications

Table 5-3 summarizes the DC specifications for the 3.3 volt environment.

Table 5-3: DC specifications for 3.3V signaling.

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

Notes for Table 5-3

1. This specification should be guaranteed by design. It is the minimum voltage to which pull-up resistors are calculated to pull a floated network. Applications sensitive to static power utilization must assure that the input buffer is conducting minimum current at this input voltage.
2. Input leakage currents include hi-Z output leakage for all bi-directional buffers with tri-state outputs.
3. Absolute maximum pin capacitance for a PCI input is 10 pF (except for CLK) 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.
6. This input leakage is the maximum allowable leakage in the PME# open drain driver when power is removed from V_{cc} of the component. This assumes that no event has occurred to cause the device to attempt to assert PME#.

AC Specifications

Table 5-4 summarizes the AC specifications for the 3.3 volt signaling environment while Figure 5-6 illustrates the corresponding V/I curves.

Notes for Table 5-4

1. Refer to the V/I curves in Figure 5-6. 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 specifi-

Table 5-4: AC specifications for 3.3V signaling.

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

cation does not apply to CLK and RST# which are system outputs.

“Switching Current High” specifications are not relevant to SERR#, PME#, INTA#, INTB#, INTC#, and INTD# which are open drain outputs.

- 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 5-6. 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 (Figure 5-5) 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.

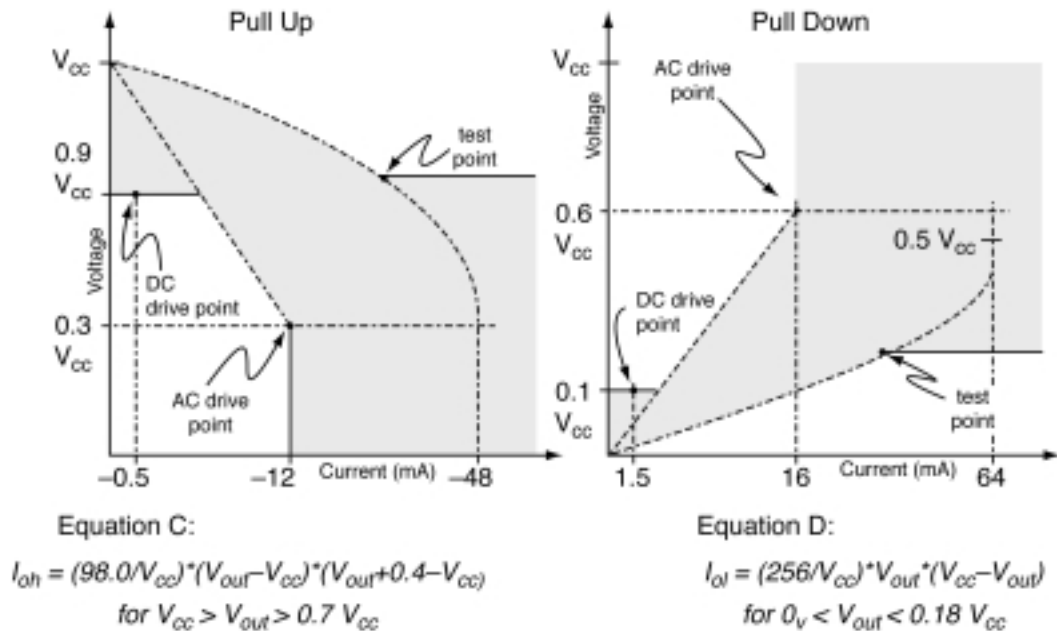


Figure 5-6: Characteristic V/I curves for a PCI driver in the 3.3 V signaling environment.

Timing Specifications

Clock

Figure 5-7 shows the clock waveform and the required measurement points. Table 5-5 summarizes the specifications. For expansion boards, clock measurements are made at the expansion board PCI component and not at the connector. Note again the distinction between the 5V and 3.3V signaling environments.

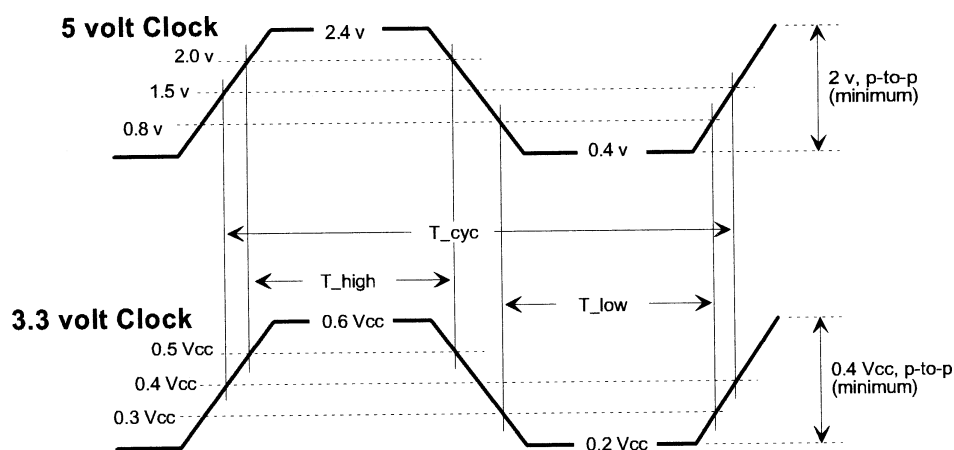


Figure 5-7: Clock waveform and required measurement points.

Table 5-5: 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 for Table 5-5

1. In general, all PCI components must work with any clock frequency between nominal DC and 33 MHz. Device operational parameters at frequencies under 16 MHz may be guaranteed by design rather than by testing. The clock frequency may be changed at any time during the operation of the system so long as the clock edges remain “clean” (monotonic) and the minimum cycle and high and low times are not violated. For example, the use of spread spectrum techniques to reduce EMI emissions is included in this requirement. 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 5-7.
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.

Timing Parameters

Table 5-6 lists the timing parameters for both the 5V and 3.3V signaling environments.

Notes for Table 5-6

1. See the output timing measurement conditions in Figure 5-8.
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 must be correlated to these specifications. Note that faster buffers may exhibit some ring back when attached to a 50 pF lump load which should be of no consequence as long as the output buffers are in full compliance with slew rate and V/I curve specifications.

Table 5-6: Timing parameters.

Symbol	Parameter	Min	Max	Units	Notes
t_{val}	CLK to Signal Valid Delay — bussed signals	2	11	ns	1,2,3
$T_{val}(ptp)$	CLK to Signal Valid Delay — point to point	2	12	ns	1,2,3
t_{on}	Float to Active Delay	2		ns	1,7
t_{off}	Active to Float Delay		28	ns	1,7
t_{su}	Input Setup Time to CLK — bussed signals	7		ns	3,4,8
$t_{su}(ptp)$	Input Setup Time to CLK — point to point	10, 12		ns	3,4
t_h	Input Hold Time from CLK	0		ns	4
T_{rst}	Reset active time after power stable	1		ms	5
$T_{rst-clk}$	Reset active time after CLK stable	100		μ s	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	
T_{rhfa}	RST# high to first configuration access	2^{25}		clocks	
T_{rhff}	RST# high to first FRAME# assertion	5		clocks	

For parts compliant to the 3.3V signaling environment:

Minimum times are evaluated with same load used for slew rate measurement (Figure 5-5); maximum times are evaluated with the load circuits shown in Figure 5-9.

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 input timing measurement conditions in Figure 5-8.
5. CLK is stable when it meets the requirements in the previous section. RST# is asserted and deasserted asynchronously with respect to CLK.

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.
8. Setup time applies only when the device is not driving the pin. Devices cannot drive and receive signals at the same time.

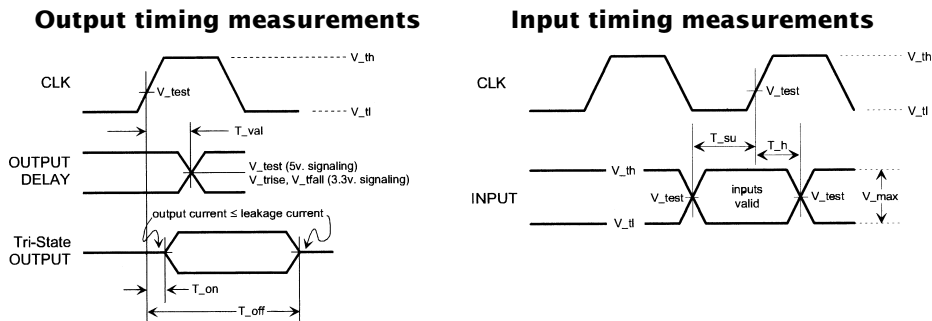


Figure 5-8: Input and output timing measurement conditions.

Table 5-7: Measurement condition parameters.

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

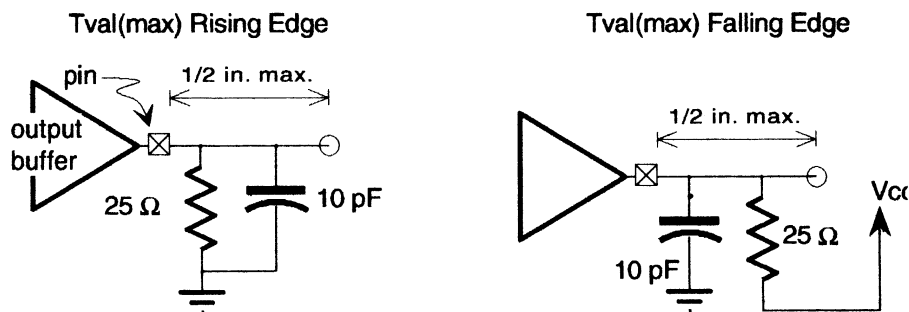


Figure 5-9: Load circuits for 3.3V slew measurements.

66 MHz PCI

66 MHz operation is defined in a way that allows 33 MHz cards to coexist with 66 MHz cards in much the same way that 32-bit cards coexist with 64-bit cards. 66 MHz is supported only in a 3.3 volt signaling environment. A read-only bit in the Status Register of an add-in card, 66MHZ_CAPABLE, identifies it as capable of 66 MHz operation.

The M66EN pin was formerly defined as ground. It is pulled up on a 66 MHz capable motherboard. 33 MHz cards will connect this pin to the ground plane thus pulling it low to signify that the system is limited to 33 MHz. So only if all cards are 66 MHz capable will the system run at 66 MHz.

M66EN is an input to the clock generation circuit. If M66EN is low, the clock reverts to 33 MHz.

Clock Specification

Table 5-8 shows the clock specifications for 66 MHz operation. Not surprisingly, the numbers are roughly half the same values for 33 MHz operation as shown in Table 5-5.

Table 5-8: Clock specifications for 66 MHz operation.

Symbol	Parameter	Min	Max	Units	Notes
T_{cyc}	CLK Cycle Time	15	30	ns	2,4
T_{high}	CLK High Time	6		ns	
T_{low}	CLK Low Time	6		ns	
—	CLK Slew Rate	1.5	4	V/ns	3

Notes for Table 5-8

1. Refer to Figure 5-7 for details of clock waveform.
2. In general, all 66 MHz PCI components must work with any clock frequency up to 66 MHz. CLK requirements vary depending upon whether the clock frequency is above 33 MHz.
 - a. Device operational parameters at frequencies at or under 33 MHz will conform to the specifications in Table 5-5. 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 motherboard only.
 - b. For clock frequencies between 33 MHz and 66 MHz, the clock frequency may not change except while $RST\#$ is asserted or when spread spectrum clocking (SSC) is used to reduce EMI emissions.
3. 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 5-7.
4. The minimum clock period must not be violated for any single clock cycle; i.e., accounting for all system jitter.

Timing Parameters

Table 5-9 shows those timing parameters that change from 33 MHz to 66 MHz.

Table 5-9: Timing parameters for 66 MHz operation.

Symbol	Parameter	Min	Max	Units	Notes
t_{val}	CLK to Signal Valid Delay — bussed signals	2	6	ns	1,2,3,5
$T_{val}(ptp)$	CLK to Signal Valid Delay — point to point	2	6	ns	1,2,3,5
t_{on}	Float to Active Delay	2		ns	1,5,7
t_{off}	Active to Float Delay		14	ns	1,7
t_{su}	Input Setup Time to CLK — bussed signals	3		ns	3,4,7
$t_{su}(ptp)$	Input Setup Time to CLK — point to point	5		ns	3,4

Notes for Table 5-9

1. See the output timing measurement conditions in Figure 5-8.
2. Minimum times are evaluated with same load used for slew rate measurement (Figure 5-5); maximum times are evaluated with the load circuits shown in Figure 5-9.
3. REQ# and GNT# are point-to-point signals and have different output valid delay and input setup times than do bussed signals. GNT# and REQ# have a setup time of 5 ns. All other signals are bused.
4. See the input timing measurement conditions in Figure 5-8.
5. When M66EN is asserted, the minimum specification for T_{val} , $T_{val}(ptp)$, 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.

6. 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. Setup time applies only when the device is not driving the pin.
Devices cannot drive and receive signals at the same time.

Mechanical Details

Connector

PCI expansion cards utilize a connector derived from the connector used by IBM's Microchannel (see Figure 5-10). The basic 32-bit bus uses a 124-pin connector where 4 pins are used for a keyway that distinguishes 5 volt signaling from 3.3 volt signaling. The same physical connector is used for both signaling environments. In one orientation, the key accommodates 5V cards. Rotated 180 degrees, it accommodates 3.3V cards.

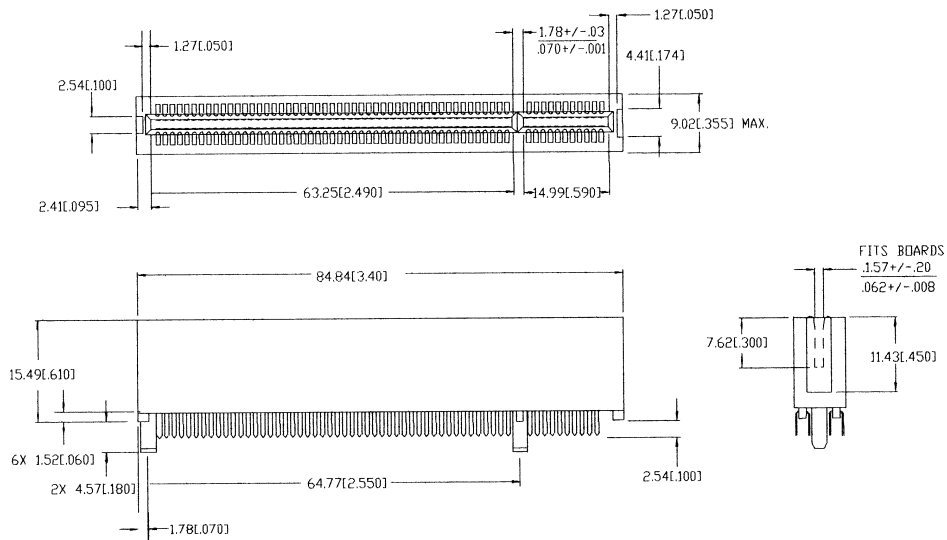


Figure 5-10: 32-bit PCI expansion card connector.

The 64-bit extension, built into the same connector molding, extends the total number of pins to 184 as shown in Figure 5-11. Note that the 64-bit connector requires two different implementations to accommodate signaling environment keying.

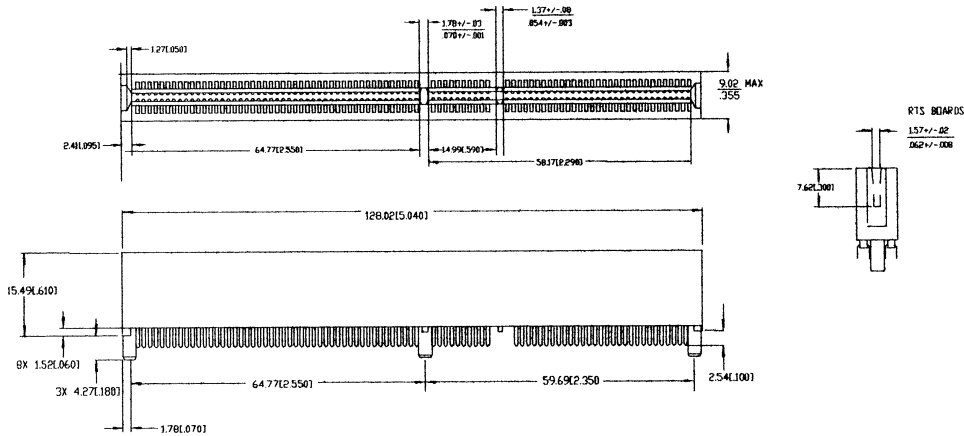


Figure 5-11: 64-bit PCI expansion card connector.

Card

The basic PCI expansion card is designed to fit in standard PC chassis available from any number of vendors. The card looks essentially like an ISA or EISA card except that the components are on the opposite side. This allows the implementation of *shared slots* where a single chassis slot could accommodate either an ISA card or a PCI card.

Because of the tight timing requirements imposed by operation up to 66 MHz, the specification places limits on the trace length of PCI signals on expansion boards. The 32-bit interface signals are limited to 1.5" from the top edge of the connector to the PCI interface device. The 64-bit extension signals are limited to 2". The CLK signal *must* be $2.5" \pm 0.1"$.

The specification also strongly recommends that the pinout of the interface chip connecting to the PCI align exactly with the PCI connector pinout as shown in Figure 5-12. This contributes to shorter, more consistent stub lengths.

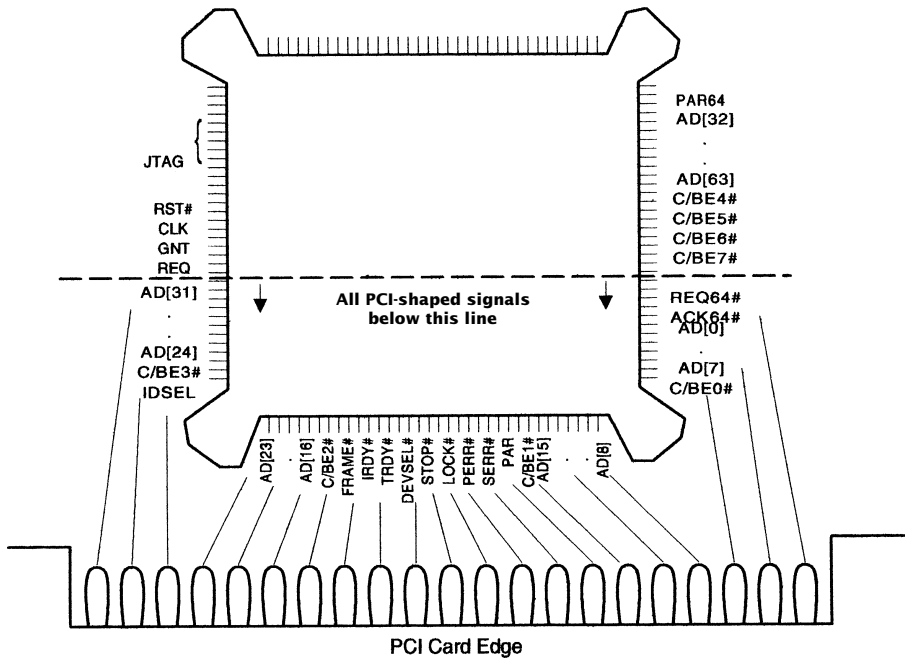


Figure 5-12: Suggested pinout for PQFP PCI component.

Summary

PCI's electrical characteristics are explicitly designed for low power consumption. The bus does away with power-consuming termination resistors and instead takes advantage of the wavefront reflected from an unterminated bus line to minimize the drive requirements of interface silicon. Because the specification is based on CMOS, DC current requirements are almost nil and drivers must be characterized in terms of a V/I curve during switching.

PCI supports two *signaling environments*, 5 volts and 3.3 volts. Again, the motivation is lower power consumption. Keying in the expansion card connector prevents a card from being plugged into the wrong signaling environment. There is provision for a *universal* card that can work in either environment.

Like the 64-bit extension, the 66 MHz extension is implemented in a way that allows 33 MHz cards to coexist with 66 MHz cards. The CLK for a bus segment operates at 66 MHz only if all cards are 66 MHz capable.

CHAPTER 6

Plug and Play Configuration

A key feature of PCI that distinguishes it from earlier busses such as ISA is the ability to dynamically configure a system to avoid resource conflicts. This is known as Plug and Play configurability or, if you're less optimistic, "Plug and Pray."

Background

In the "old days," configuration issues were generally handled by jumpers on each add-in card. The jumpers would select operating characteristics such as memory or I/O address space, interrupt vectors and perhaps a DMA channel. Configuring such a card correctly requires a fairly detailed knowledge of the system and its hardware.

Configure such a card wrong and it will likely conflict with something else. This often leads to bizarre system behavior that is difficult to diagnose.

In the PC world, various device types such as serial controllers, video adapters and so on have a limited range of defined configurations. Software drivers for these devices expect that the card will be configured to one of the **default settings**. Information about the device's settings is typically conveyed by the command line that starts the driver.

In the world of Plug and Play, an add-in card tells the system what it needs—how much memory or I/O space, does the device require an interrupt and so on. Configuration software scans the system at boot up time to determine total resource requirements and then assigns resources like memory and I/O space and interrupts to individual cards in a way that avoids resource conflicts.

The device driver can make no assumptions about a device's configuration. Instead, **it must interrogate the device** to determine what resources have been allocated to it.

Configuration Address Space

PCI defines a third address space in addition to memory and I/O. This is called **configuration space** and every logical function gets **256** bytes in this space. A function is selected for configuration space access by asserting the corresponding device's IDSEL signal together with executing a Config Read or Config Write bus command.

Configuration Transactions

PCI-based systems require a mechanism that allows software to generate transactions to Configuration space. This mechanism will generally be located in the Host-to-PCI bridge. The specification defines an appropriate mechanism for x86 processors. Other processors may, and probably will, use a similar approach.

The x86 configuration mechanism uses two DWORD read/write registers in I/O space. These are:

CONFIG_ADDRESS	0x3f8
CONFIG_DATA	0x3fc

The layout of CONFIG_ADDRESS is shown in Figure 6-1. Bit 31 is an enable that determines when access to CONFIG_DATA is to be

interpreted as a configuration transaction on the PCI bus. When bit 31 is 1, reads and writes to CONFIG_DATA are translated to PCI configuration read and write cycles at the address specified by the contents of CONFIG_ADDRESS. When bit 31 is 0, reads and writes to CONFIG_DATA are simply passed through as PCI I/O reads and writes. Bits 30 to 24 are reserved, read-only, and must return 0 when read. Bits 23 to 16 identify a specific bus segment in the system. Bits 15 to 11 select a device on that segment. Bits 10 to 8 select a function within the device (if the device supports multiple functions). Bits 7 to 2 select a DWORD configuration register within the function. Finally, bits 1 and 0 are reserved, read-only, and must return 0 when read.

CONFIG_ADDRESS can only be accessed as a DWORD. Byte or word accesses to CONFIG_ADDRESS are passed through to the PCI bus.

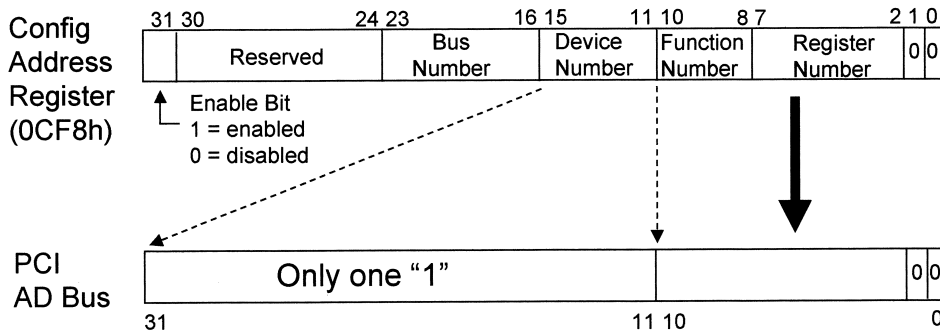


Figure 6-1: x86 configuration address.

Driving IDSEL

A device is selected as the target of a configuration transaction by asserting its IDSEL pin. The specification does not define the nature of the mapping between the Device Number field and the individual

IDSEL signals. In the defined x86 configuration mechanism, the host bridge decodes the Device Number field to drive one of the lines in the range AD[31:11]. Every device then has its IDSEL pin connected to exactly one of AD[31:11] as shown in Figure 6-2.

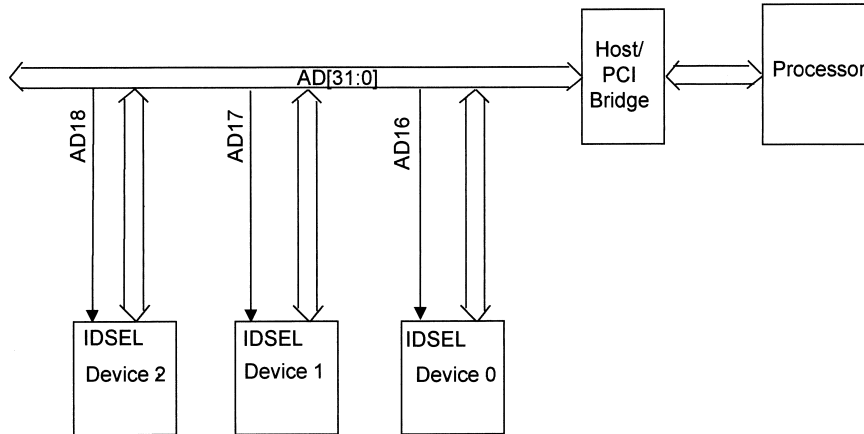


Figure 6-2: Asserting IDSEL.

Configuration Header—Type 0

Of the 256 bytes of configuration space allocated to every function, the first 64 bytes are defined by the specification and are called the Configuration Header. The remaining 192 bytes are available for device-specific configuration functions. Figure 6-3 shows the layout of the Configuration Header.

Header Type

Currently, three different header types are defined as indicated by the value in byte 0xE (14 decimal). The Type 0 header is for most devices. The Type 1 header describes a bridge device and the Type 2 header describes a PC Card device. In all cases, the first three DWORDS and the Header Type byte of the fourth DWORD are the same.

The most significant bit of the Header Type is set to 1 if the device is a multi-function device.

Identification Registers

Several fields in the header are read-only and serve to identify the device along with various operational characteristics.

- *Vendor ID*: Identifies the vendor of the device. More specifically, it identifies the vendor of the PCI silicon.

31		16 15		0	
Device ID		Vendor ID			00h
Status		Command			04h
Class Code			Revision ID		08h
BIST	Header Type	Latency Timer	Cache Line Size		0ch
Base Address Registers					10h
					14h
					18h
					1Ch
					20h
					24h
Cardbus CIS Pointer					28h
Subsystem ID		Subsystem Vendor ID			2ch
Expansion Bus ROM Base					30h
Reserved			Cap Pntr		34h
Reserved					38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line		3Ch

Figure 6-3: Type 0 configuration header.

Vendor ID codes are assigned by the PCI SIG.

- *Device ID*: Identifies the device. This value is assigned by the vendor.
- *Revision ID*: Assigned by the device vendor to identify the revision level of the device.

Two additional registers allow makers of PCI plug in adapters to identify their devices.

- *Subsystem Vendor ID*: Identifies the vendor of a functional PCI device.
- *Subsystem Device ID*: Assigned by the vendor to identify a functional PCI device. Can also be used to identify individual functions in a multi-function device.

The Class Code is a 24-bit read-only register that identifies the basic function of the device. It is divided into three sections:

- *Base Class*: Defines the basic functional category.
- *Sub-class*: Identifies a device type or implementation within the Base Class. For example, a mass storage controller can be SCSI, IDE, floppy, etc. A network controller can be Ethernet, token ring and so on.
- *Programming Interface*: Defines specific register-level implementations. For most classes this is simply 0, but it is used for IDE controllers and other traditional PC peripherals.

Command Register

The read/writable Command Register provides coarse control over a device's ability to generate and respond to PCI cycles.

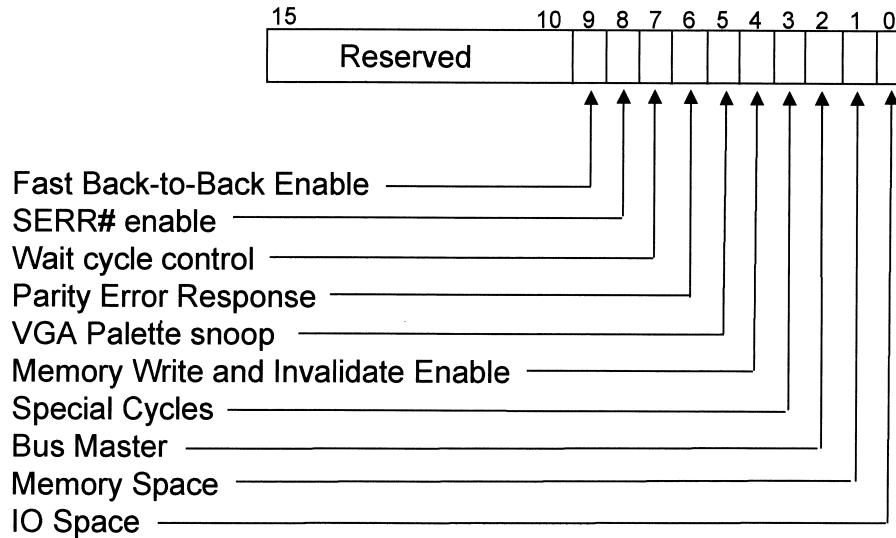


Figure 6-4: Configuration Command Register.

Bit

- 0 When 1, allows the device to respond to PCI I/O space accesses.
- 1 When 1, allows the device to respond to PCI memory space accesses
- 2 When 1, enables the device to act as a bus master
- 3 When 1, allows a device to monitor Special Cycle operations.
- 4 When 1, a master is allowed to use the Memory Write and Invalidate command if so capable. When 0, the master must use Memory Write instead.
- 5 Controls how VGA devices handle access to VGA palette registers.
- 6 When 1, the device responds to a detected parity error by asserting PERR#. If 0, the device ignores parity errors although it is still required to generate parity.
- 7 Controls whether a device does address/data stepping. A device not capable of stepping hardwires this bit to 0. A device that always steps hardwires it to 1. A device that can do either must implement this bit as writable.

- 8 When 1, allows the device to assert SERR#.
- 9 When 1, allows a master to execute fast back-to-back transactions to different targets. This bit will only be set if all targets are fast back-to-back capable.

Note that writing all zeros to this register effectively disconnects the device from the PCI bus for all accesses except configuration cycles.

Status Register

The Status Register contains two types of information—Read only bits that convey additional information about a device’s capabilities and read/write bits that track bus related events.

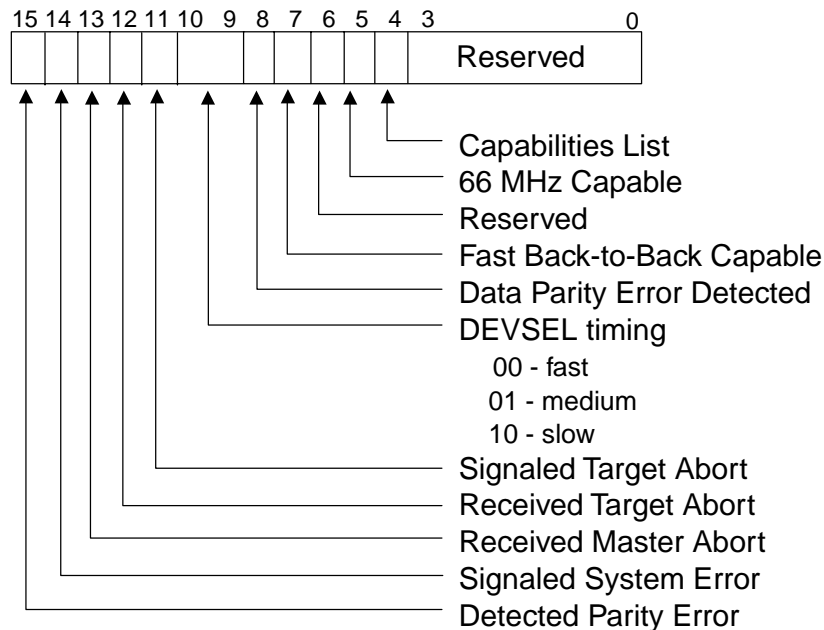


Figure 6-5: Configuration Status Register.

The writable bits operate differently than normal. A bit is set to 1 by the occurrence of an event. Writing a 1 to a bit from the PCI bus *clears* it. This simplifies programming. After reading the register and determining that error bits are set, you simply write the same value back to clear them.

Bit

- 4 RO. 1 = Extended capabilities pointer exists.
- 5 RO. 1 = device is capable of 66 MHz operation.
- 6 RO. 1 = device supports “user definable features”.
- 7 RO. 1 = target device supports fast back-to-back transactions to different targets.
- 8 RW. Only implemented by masters. Set if
 - The agent asserted PERR# itself or observed PERR# asserted
 - The agent was the bus master for the operation in which the error occurred AND
 - Its Parity Error Response bit is set
- 9–10 RO. DEVSEL# timing
 - 00 = Fast
 - 01 = medium
 - 10 = slow
 - 11 = reserved
- 11 RW. Set by a target when it terminates a transaction with Target Abort
- 12 RW. Set by a master when its transaction is terminated by Target Abort
- 13 RW. Set by a master when it terminates a transaction with Master Abort
- 14 RW. Set by a device that asserts SERR#
- 15 RW. Set by a device whenever it detects a parity error, even if parity error handling is disabled.

Built-in Self-Test Register (BIST)

This optional mechanism provides a standardized way of implementing self-test on plug-in cards. Devices that don't support BIST must return a value of 0 when this register is read.

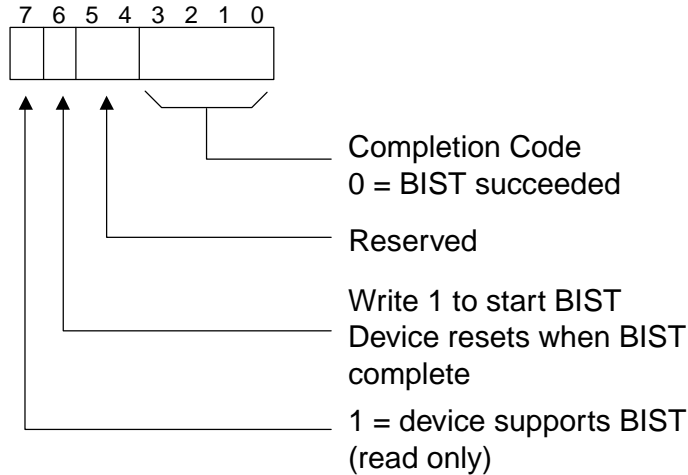


Figure 6-6: Built-in self-test (BIST) register.

Bit

- 7 RO. 1 = device supports BIST
- 6 RW. Write 1 to invoke BIST. Device resets bit when BIST is complete.
- 5-4 Reserved. Read as 0.
- 3-0 Completion code. 0 = device has passed test. Non-zero value indicates failure. Failure codes are device specific.

Latency Timer

The Latency Timer is required and must be a read/writable register for any master capable of bursting more than two data phases. The value written here is the minimum number of clock cycles that the

master can retain ownership of the bus. Typically, the lower three bits are hardwired to 0 and only the upper 5 bits are writable. This yields a maximum of 255 clock cycles with a granularity of eight clock cycles. The Latency Timer may be read-only if the master never bursts more than two data phases.

Cache-Line Size

Configuration software writes the system cache line size in DWORD increments to this register. It is required for any master that implements the **Memory Write and Invalidate command** and for any target that implements **cache-line wrap addressing**. Masters that implement the advanced read commands should take advantage of this register to optimize their use of the read commands.

Cardbus CIS Pointer

Optional. Implemented by devices that share silicon between cardbus and PCI devices. It points to the *Card Information Structure* for the Cardbus implementation. Details of the CIS can be found in revision 3.0 of the PC Card specification.

Capabilities Pointer

If Status Register bit 4 = 1, this read-only byte is a pointer to the first entry of the *Capabilities List*. It is a byte offset into the device-specific configuration space.

Max_Lat (Maximum Latency)

The specification says that this optional register specifies “how often the device needs to gain access to the PCI bus”. A better interpretation might be how *quickly* the master needs access to the bus. Values of Max_Lat are in increments of 250 ns which happens to be about eight clocks at 33 MHz.

The intention is that configuration software can use this value to assign the master to an arbitration priority level. Devices with lower values, implying a need for low latency, would be assigned to the higher priority levels.

Min_Gnt (Minimum Grant)

This register indicates how long the master would like to retain bus ownership when it initiates a transaction. Values of Min_Gnt are in increments of 250 ns or eight clocks at 33 MHz.

Configuration software uses this value to set the device's Latency Timer.

Base Address Registers (BAR)

The Base Address Registers provide the mechanism that allows configuration software to determine the memory and I/O resources that a device requires. Once the system topology is determined, configuration software maps all devices into a set of reasonable, non-conflicting address ranges and writes the corresponding starting addresses into the Base Address Registers. The Type 0 configuration header supports up to six Base Address Registers, allowing a device to have up to six independent address ranges.

There are two formats for the Base Register as shown in Figure 6-7. Read-only bit 0 determines whether the Base Address Register represents memory or I/O space.

For memory space, read-only bits 1 and 2 indicate how the memory space must be mapped and the size of the Base Address Register. Memory can be mapped into either 32-bit or 64-bit address space implying respectively a 32-bit register or a 64-bit register. A 64-bit register occupies two adjacent BAR locations in the Configuration Header. Prior to revision 2.2 the combination 01

in bits 2 and 1 identified memory space that must be located below the one megabyte real mode boundary. Although this is no longer supported, “System software should recognize this encoding and handle appropriately.” Bit 3 identifies prefetchable memory.

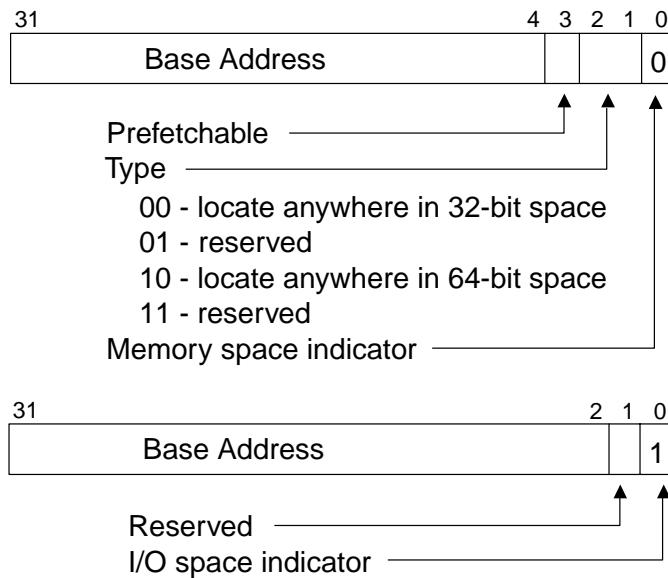


Figure 6-7: Base Address Register.

For I/O space, bit 1 is hardwired to 0 and the remaining bits are used to map the device. An I/O Base Address Register is always 32 bits.

Determining Block Size

How does configuration software determine the size of the memory or I/O space represented by each BAR? A Base Address Register only implements as many bits as are necessary to decode the block size that it represents. Thus, for example, a BAR that represents 1 Megabyte of memory space would only need to implement the upper 12 bits of the

32 bit address. The lower 20 bits decode an address within the 1 Megabyte range. When you read a BAR, the undecoded bits read back as 0.

So the procedure for determining block size is to:

Step	1 MB Example
1. Write all 1's to the register	0xFFFFFFFF
2. Read it back	0xFF000008
3. Mask off the lower four read-only bits	0xFFF00000
4. Take the 1's complement	0x00FFFFFF
5. Add 1. This is the block size.	0x00100000

The same procedure applies to I/O space and 64-bit memory space.

This strategy has two interesting consequences. Block sizes are always powers of 2 and the base address is always “naturally aligned.” This means, for example, that a 2 MB address space can't have a starting address of 3 MB.

Note that the minimum block size inferred by the Memory BAR format is 16 bytes. Likewise the minimum I/O block size is four bytes. In the interest of minimizing the number of bits in a BAR, devices are allowed to consume more space than they actually use. The specification suggests that decoding down to 4 KB of memory space is appropriate for devices that need less than that. A device that decodes more space than it uses need not respond to the unused space. **Devices that map into I/O space must not use more than 256 bytes per Base Address Register.**

Use Memory Space if Possible

Although PCI fully supports “I/O” space, the specification recommends that device registers be mapped into memory space if at all

possible. There are several reasons for this. In the PC architecture I/O space is limited and highly fragmented making it potentially difficult to allocate I/O space. Secondly, I/O space is assumed to have read side effects and is ~~is not prefetchable~~. This precludes certain optimizations that PCI-to-PCI bridges are allowed to perform. Finally, some processor architectures simply don't support the notion of I/O address space.

In practice, some devices use two Base Address Registers to represent the same set of device registers. One of these BARs maps into memory space, the other into I/O space. Configuration software will allocate space to both registers if possible. Later when the device's driver is invoked, it will decide, based on its environment and other considerations, which space to use.

What is “Prefetchable”?

Fundamentally, prefetchable memory space has no read “side effects.” This in turn means that the act of reading a memory location does not in any way change the contents. No matter how many times you read it, you get the same result. Conventional memory is prefetchable. A FIFO is not. Each time you read a FIFO you get the next data element.

The primary objective in defining prefetchable memory is to allow PCI bridges to prefetch read data. In many cases prefetching can substantially reduce read latency. Consider a master agent executing a read to a location on the other side of a bridge. If the bridge recognizes that the location is prefetchable, it can go ahead and read subsequent locations (prefetch) on the assumption that the master intends to read further. If, on the other hand, the master chooses not to read further, no harm is done because the prefetch has not altered the contents of the prefetched registers.

A further requirement on PCI prefetchable memory is that it must return all four bytes on a read independent of the BE# signals.

Back in the days when processor cycles were at a premium, clever hardware designers would build I/O registers with read side effects as a way to simplify device programming. For example, the act of reading a status register could clear the interrupt flag if it were set. This would eliminate the need to write a zero back to that bit.

Today, trying to save a couple of instructions by using a non-prefetchable register might actually slow the system down by precluding other optimization strategies. Good design practice emphasizes avoiding read side effects unless there is no alternative.

Expansion ROM

The Expansion ROM Base Address register operates similarly to the Base Address registers just described. Since the expansion ROM is assumed to exist in memory space, bit 0 is used as a ROM enable. Bits 1 to 10 are reserved and bits 11 to 31 set the base address. The ROM's block size is determined in the same way as for other address ranges with a granularity of 2k. The Expansion ROM Base Address register is limited to 32 bits. See Figure 6-8.

The expansion ROM itself is organized as one or more "images" with a specific format based on existing ROM headers for ISA, EISA and Microchannel adapters. One major difference between PCI expansion ROMs and previous implementations is that ROM code is never executed in place. It must first be copied to RAM. There are two reasons for this: RAM is generally faster than ROM and the initialization code can be discarded after it is executed.

Just because a device implements an Expansion ROM Base Address register doesn't necessarily mean a ROM is present.

Configuration software must test for the presence of a ROM by testing for the ROM signature in the first two bytes of the header. See Figure 6-9.

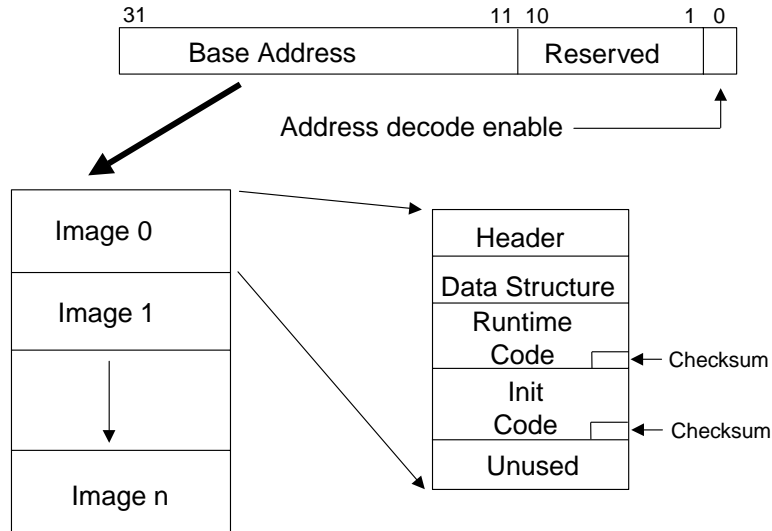


Figure 6-8: Expansion ROM Base Address Register.

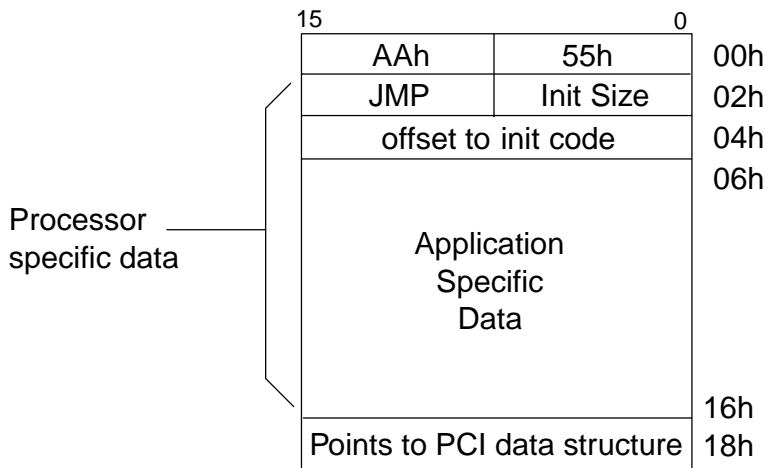


Figure 6-9: ROM image header.

The next 24 bytes (16h) of the header are processor specific. For x86 implementations, byte 2 is the length in 512 byte chunks of the initialization code and the next three bytes are a short jump to the init code. The POST code executes a far call to this location. The remainder of the processor-specific field is available to the application for various identifying information.

Finally, the last two bytes of the header are a pointer to a PCI data structure. The reference point for this pointer is the beginning of the ROM image.

Figure 6-10 shows the PCI Data Structure that provides additional information about the ROM image. The first four bytes are the text string “PCIR”, a signature that verifies the existence of the data structure. The vendor ID, device ID and class code fields must match the corresponding fields in the device’s configuration header for the image to be considered valid. Think of this as a “sanity check” to be sure the right ROM is installed.

31			0
“PCIR”			00h
Device ID		Vendor ID	04h
PCI Struct Len		Reserved	08h
Class Code		Struct Rev	0ch
Code Rev		Image Length	
Reserved		Indi- cator	Code Type

Figure 6-10: PCI data structure.

PCI Struct Len: The length of the PCI data structure itself, currently 24 (18h) bytes.

Struct Rev: Revision level of the data structure. This is 0 for Rev. 2.2 of the specification.

Image Length: Entire length of this image in 512 byte increments.

Code Rev: Revision level of the contents of this image. Assigned by vendor.

Code Type: Identifies the type of executable code in the image, either native machine language for a particular processor or interpretive code conforming to the Open Firmware standard (IEEE 1275-1994). 0 = Intel x86 code, 1 = interpretive code, 2 = Hewlett-Packard PA RISC and the values from 3 through FFh are reserved.

Capabilities List

Figure 6-11 shows the *Capabilities List*, a new mechanism in Rev. 2.2 that supports new and optional PCI capabilities in the form of an open-ended linked list. If bit 4 of the Status Register is 1, then the byte at offset 34h in the header contains the offset to the first element of a linked list of capabilities. The Capabilities List resides in the device-specific portion of a function's configuration space.

Each capability consists of an 8-bit ID code assigned by the PCI SIG, an 8-bit offset to the next element in the list and some number of additional bytes that may be either read-only or read/writable. The offset field of the last capability in the list is set to 0.

The following capabilities are currently defined:

0. Reserved
1. PCI Power Management Interface, documented in the *PCI Power Management Interface Specification*.

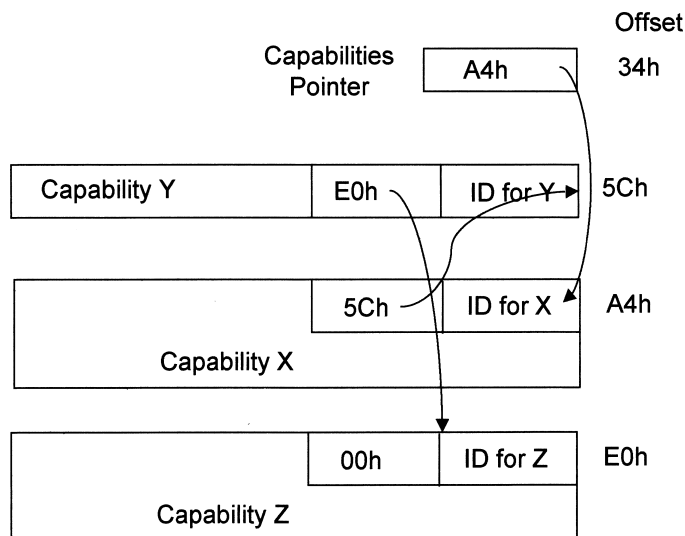


Figure 6-11: Capabilities List.

2. AGP. Identifies a graphics controller using the features of the Accelerated Graphics Port.
3. VPD. Provides support for Vital Product Data (see next section).
4. Slot Identification. Identifies a bridge that provides external expansion capabilities.
5. Message Signaled Interrupts.
6. Compact PCI Hot Swap CSR.

Vital Product Data

Vital Product Data (VPD) is additional information that uniquely identifies items such as hardware, software and microcode elements of the system. Among other things, it can provide the system with information on FRUs (Field Replaceable Unit) such as part number, serial number, Engineering Change level and so on. VPD also provides a mechanism for storing information about performance and failure data.

Prior to Rev. 2.2, VPD resided in the ROM space accessed by the Expansion ROM BAR. VPD now resides in an unspecified storage device such as serial EEPROM on a PCI device. The storage device is then read and written through the VPD capability shown in Figure 6-12. To read an element of VPD, you write its address into the VPD Address field setting the flag bit, “F”, to 0. When the device has read the specified four bytes from storage and placed them into the VPD Data field it sets F to 1. To write a VPD field, you first write the data to the VPD Data field, then write the address to the VPD Address field setting F to 1. After the device has written the data to storage it sets F to 0.

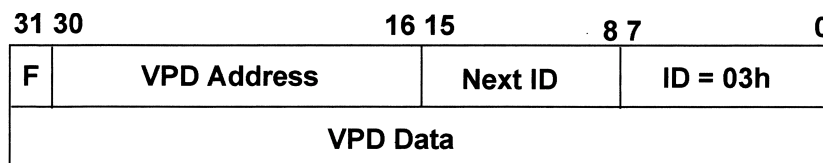


Figure 6-12: VPD capability.

VPD is organized as lists of information fields as shown in Figure 6-13. The information field has a 3-byte header followed by some amount of data as indicated by the length entry in the header. There are two categories of VPD keywords: read-only fields and read/write fields. The defined keywords are all ASCII and it is expected that the data will be ASCII as well. Here is an example of the “expansion board serial number” VPD.

```
Keyword:    SN
Length:     8
Data:       "01734672"
```

Keyword		Length	Data
Byte 0	Byte 1	Byte 2	Bytes 3 through n

Figure 6-13: VPD information field.

The information fields are contained within tagged data structures consisting of large and small resource descriptors as shown in Figure 6-14. The format is described in *Plug and Play ISA Specification, Version 1.0a*. Specifically, VPD uses four tag types as follows:

Tag Type	Resource Type	Description
Identifier String Tag (0x2)	Large	First item in the VPD list. Contains the name of the board in ASCII.
VPD-R Tag (0x10)	Large	List of read-only VPD fields.
VPD-W Tag (0x11)	Large	List of read/write VPD fields.
End Tag (0xf)	Small	Identifies end of VPD data. The End Tag has a zero data length.

Vital Product Data consists of one each of the above resource descriptors in the order shown.

The read-only fields include:

- PN *Board Part Number*. An extension of the Device ID (Subsystem ID) in the Configuration Header.
- EC *EC Level*. Identifies the Engineering Change Level of the board.

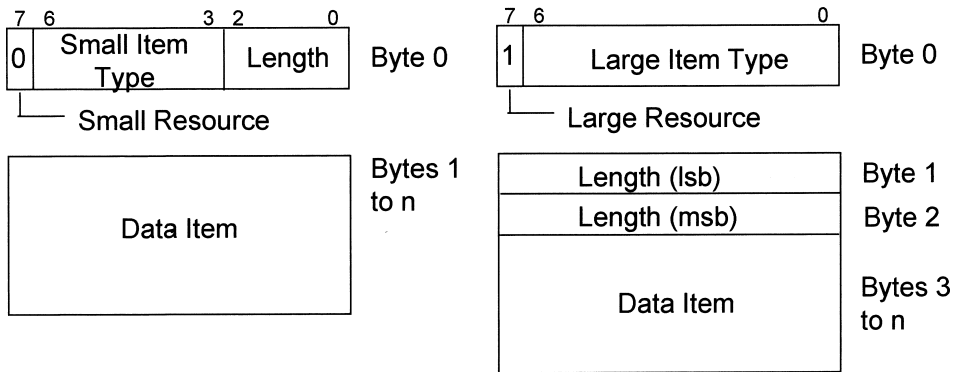


Figure 6-14: Resource data tags.

- MN *Manufacturer ID*. An extension of the Vendor ID (Subsystem Vendor ID) in the Configuration Header.
- SN *Serial Number*. Identifies a board's unique serial number.
- Vx *Vendor Specific*. Permits a vendor to create his own fields. The second character (x) may be 0 to Z
- CP *Extended Capability*. Allows a new capability to be identified in the VPD area. The data field is four bytes of *binary* pointing to the control/status registers for the capability.
 - Byte 0: Capability ID
 - Byte 1: Index of Base Address Register that contains the capabilities CSR
 - Bytes 2 and 3: Offset from BAR to CSR
- RV *Checksum and Reserved*. First data byte is a checksum from the Identifier String Tag up to and including this byte. Sum of all bytes must add up to zero. The remainder is reserved space as needed to fill up the read-only space. This field is required.

The read/write fields include:

Vx *Vendor Specific*. Permits a vendor to create his own fields. The second character (x) may be 0 to Z.

Yx *System Specific*. The second character of the keyword can be 0 to 9 or B to Z.¹

YA *Asset Tag Identifier*. Contains an asset identifier provided by the system owner. Primarily of interest to the bean counters.

RW *Remaining Read/Write Area*. Fills up the unused portion of the read/write space.²

Summary

PCI supports Plug and Play configuration that allows a system to be automatically configured at boot time. Each PCI function has 256 bytes of Configuration Space of which the first 64 bytes constitute a pre-defined header that provides all of the functionality required to configure the function.

Configuration Space also includes support for an expansion ROM that can provide device initialization and BIOS extensions. The Capabilities List provides an open-ended way to identify new and optional PCI features. Vital Product Data is an optional feature that offers additional information about a specific PCI device.

¹ It's not clear from the specification who gets to assign these keywords.

² The specification goes on to say "One or more of the Vx, Yx and RW items are required." I take this to mean that unless one of these items is present, there's no point in having a read/write section. The read/write section doesn't include a checksum.

CHAPTER 7

PCI BIOS

It is entirely possible for device drivers to access the Configuration Space directly using the mechanism described in the last chapter. However, any software that does so is platform-dependent and may not run on some platforms. This violates the spirit of PCI, which is intended to be platform-independent. To solve this problem, the PCI BIOS defines a platform-independent API to access configuration features.

Operating Modes

x86 processors can operate in any of four modes:

- *Real Mode*. The original 8088, 1 Mbyte address space
- *16-bit Protected Mode*. The 80286, 16 Mbyte address space
- *32-bit Protected Mode*. The 80386 and above, 4 Gbyte address space, protected segments
- *Flat Protected Mode*. Same as 32-bit Protected Mode except everything is in one “flat” 4 Gbyte address space.

The PCI BIOS functions must be accessible from any of these operating modes. Real mode and 16-bit protected mode use the conventional INT mechanism that all traditional BIOS functions use.

32-bit and flat protected modes require a far call to an entry point obtained from the BIOS32 Service Directory.

The PCI BIOS functions use x86 CPU registers to pass arguments and return status.

Is the BIOS There?

The PCI BIOS is based on the *Standard BIOS 32-bit Service Directory Proposal* put forward by Phoenix Technologies Ltd. Before we can use the PCI BIOS, we have to determine if it's present. In real or 16-bit protected mode we can simply invoke INT 1Ah with the appropriate function code and see what comes back. In 32-bit protected mode we have to get the entry point from the BIOS32 Service Directory and so the first step is to determine if it exists.

The BIOS32 Service Directory is identified by the data structure shown in Figure 7-1. The strategy is to scan the address range from 0xE0000 to 0xFFFFF looking for the signature “_32_”. If the signature is found, the Service Directory can be accessed by calling the specified entry point.

31				0	
“_32_”					00h
Entry point, 32-bit physical addr.					04h
		Check-sum	Length	Rev Level	08h
Reserved					0ch

Figure 7-1: BIOS32 Service Directory.

Having found the BIOS32 Service Directory, we can now inquire if the PCI BIOS is present. We call the Service Directory entry point passing in a 4-byte service identifier string. If the service is present, the Service Directory returns the base address, length and entry point of the code image for the service.

ENTRY:

EAX	Service identifier. 4-character string "\$PCI" (049435024h)
EBX	Function code in BL. 0 is the only function currently defined. Other bytes 0

EXIT:

AL	Return code 0 = service present 80h = service not present 81h = bad function code
EBX	Base address of service
ECX	Length of service
EDX	Entry point

BIOS Services

The functions making up the PCI BIOS fall into a few categories:

Identifying PCI Resources

PCI BIOS Present

Find PCI Device

Find PCI Class Code

Accessing PCI Configuration Space

Read/Write byte/word/dword

PCI Support Functions

Generate Special Cycle

Get IRQ Routing Options

Set PCI IRQ

PCI BIOS Present

ENTRY

AX B101h

EXIT

CF 1 = no BIOS present
 0 = BIOS present IFF EDX set properly

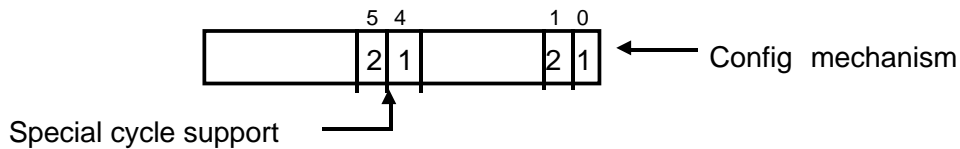
EDX "PCI "

CL Number of last PCI bus in system

BX Interface version: BH - major, BL - minor

AH Present status: 0 = BIOS present IFF EDX
 set properly

AL Hardware mechanism



This is the way to determine if the PCI BIOS is present in real mode. Even though we already know the PCI BIOS is present in 32-bit protected mode, this function returns some additional necessary information. AL returns information about which configuration and special cycle mechanisms are supported. CL returns the number of the last PCI bus segment in the system. Segments are numbered sequentially from 0 to the value returned in CL.

Find PCI Device/Class

This pair of functions allows us to locate PCI devices either by class code or specific vendor and device ID. The first time either of these functions is called, SI is set to 0. Then before each subsequent call, SI is incremented. The function is called repeatedly until it

returns `DEVICE_NOT_FOUND`. The returned values are the location in Configuration Space of the specified device.

ENTRY	
AX	B1 [02h 03h]
CX	Device ID (find device)
ECX	Class code (find class)
DX	Vendor ID (find device only)
SI	Index (0 .. n)
EXIT	
CF	1 = error, 0 success
BH	Bus number
BL	Device number (upper 5 bits)
	Function number (lower 3 bits)
AH	Return code
	SUCCESSFUL
	DEVICE_NOT_FOUND
	BAD_VENDOR_ID (find device only)

Generate Special Cycle

This function generates a Special Cycle on the specified bus. Note however that Configuration Mechanism 2 can only generate special cycles on Bus 0 and will return `FUNCTION_NOT_SUPPORTED` if you specify a non-zero bus number.

ENTRY	
AX	B106h
BH	Bus number
EDX	Special cycle data
EXIT	
CF	1 = error, 0 success
AH	Return code
	SUCCESSFUL
	FUNCTION_NOT_SUPPORTED

Read Configuration Register (Byte, Word, Dword)

This set of functions allows you to read Configuration Space by specifying the bus, device, function and register numbers. The service will return the value BAD_REGISTER_NUMBER if the register number is not properly aligned for the data size being requested.

ENTRY

AX	B1 [08h 09h 0Ah]
BH	Bus number
BL	Device number (upper 5 bits)
	Function number (lower 3 bits)
DI	Register number

EXIT

CF	1 = error, 0 success
CL, CX, ECX	Returned data
AH	Return code
	SUCCESSFUL
	BAD_REGISTER_NUMBER

Write Configuration Register (Byte, Word, Dword)

This set of functions allows you to write Configuration Space by specifying the bus, device, function and register numbers and the data to write. The service will return the value BAD_REGISTER_NUMBER if the register number is not properly aligned for the data size being requested.

ENTRY

AX	B1 [0Bh 0Ch 0Dh]
BH	Bus number
BL	Device number (upper 5 bits)
	Function number (lower 3 bits)
CL, CX, ECX	Data to write
DI	Register number

EXIT

CF	1 = error, 0 success
AH	Return code
	SUCCESSFUL
	BAD_REGISTER_NUMBER

Get Interrupt Routing Options

This function is used to determine what options are available for routing INTx# lines to IRQs. The argument passed to this function is a pointer to a data structure.

ENTRY	
AX	B10Eh
BX	0000h
DS	Segment or selector for BIOS data. Must resolve to 0F0000h
ES	Segment or selector of data structure
DI, EDI	Offset to data structure
EXIT	
CF	1 = error, 0 success
AH	Return code
	SUCCESSFUL
	FUNCTION_NOT_SUPPORTED
	BUFFER_TOO_SMALL
BX	Bitmap of IRQs exclusively dedicated to PCI devices

The structure pointed to by ES:DI(EDI) contains two fields: a far pointer to a buffer to contain the returned interrupt routing information and the length of that buffer represented in two bytes. A far pointer is four bytes in real and 16-bit protected modes and six bytes in 32-bit protected mode. The Get PCI Interrupt Routing function will return an error if the buffer size is insufficient to store an Interrupt Routing Table Entry for each device that requires an interrupt.

The buffer returned by the Get PCI Interrupt Routing Options function contains an *Interrupt Routing Table Entry* for each PCI device that requires interrupt support. See Figure 7-2. After identifying the bus number and device number, an Interrupt Routing Table Entry supplies two values for each of the four PCI bus interrupt lines. The *IRQ bit map* values show which of the processor IRQs the interrupt pin may be connected to. Bit 0 corresponds to IRQ 0 and so on.

Offset	Size	Description
0	byte	PCI bus number
1	byte	PCI device number
2	byte	Link value for INTA#
3	word	IRQ bit map for INTA#
5	byte	Link value for INTB#
6	word	IRQ bit map for INTB#
8	byte	Link value for INTC#
9	word	IRQ bit map for INTC#
11	byte	Link value for INTD#
12	word	IRQ bit map for INTD#
14	byte	Slot number
15	byte	Reserved

Figure 7-2: Interrupt routing table entry.

The *link value* fields show which interrupt pins are wire-ORed together. Interrupt pins that are wired together have the same link value. The value is arbitrary except that the value zero means that the interrupt pin is not connected to the interrupt controller.

Slot number indicates whether this table entry is for a motherboard device or an add-in slot. A value of 0 indicates a motherboard device, a non-zero value is a slot. This provides a way to correlate PCI device numbers with physical slots. Assignment of slot numbers is implementation dependent. The spec does recommend however that slots should be “clearly labeled.”

Upon successful return, the buffer length field is updated to reflect the actual length of the Interrupt Routing Table.

Set PCI Interrupt

Finally, having determined what possible routings exist, we can establish a binding between an interrupt pin on a specific connector and an IRQ at the processor. This function is intended to be used by

a system-wide configuration utility or a Plug and Play operating system rather than by device drivers.

ENTRY

AX	B10Fh
BH	Bus Number
BL	Device (high 5 bits), Function (low 3 bits)
CH	IRQ. Valid values: 0..0Fh
CL	Int Pin. Valid values: 0Ah..0Dh
DS	Segment or selector for BIOS data. Must resolve to 0F0000h

EXIT

CF	1 = error, 0 success
AH	Return code
	SUCCESSFUL
	SET_FAILED
	FUNCTION_NOT_SUPPORTED

Summary

The PCI BIOS provides a platform-independent means to access Configuration Space. The BIOS is accessible from all operating modes of the x86 processors. PCI BIOS services allow you to find specific devices or device classes, read and write Configuration Space and set interrupt options.

CHAPTER 8

PCI Bridging

The notion of bridging plays a significant role in PCI architecture primarily due to electrical limitations that impose a severe limit on the number of devices residing on a single PCI bus segment. In some cases it is also desirable to functionally isolate portions of the system so they can operate in parallel.

Bridge Types

In this chapter we're primarily concerned with the PCI-to-PCI (P2P) bridge, that is, a bridge that connects two PCI bus segments. The P2P bridge is defined in *PCI-to-PCI Bridge Architecture Specification*, Rev. 1.1, December 1998. But before delving into the details of the P2P bridge, we should note briefly that there are two other types of bridges that serve specific roles as illustrated in Figure 8-1.

Host-to-PCI Bridge

None of today's popular processor architectures has a PCI bus coming directly off the chip. Rather, each processor defines its own local bus optimized around the specific architecture. External cache and main memory often reside on the local processor bus. Some local busses also support multiple processors.

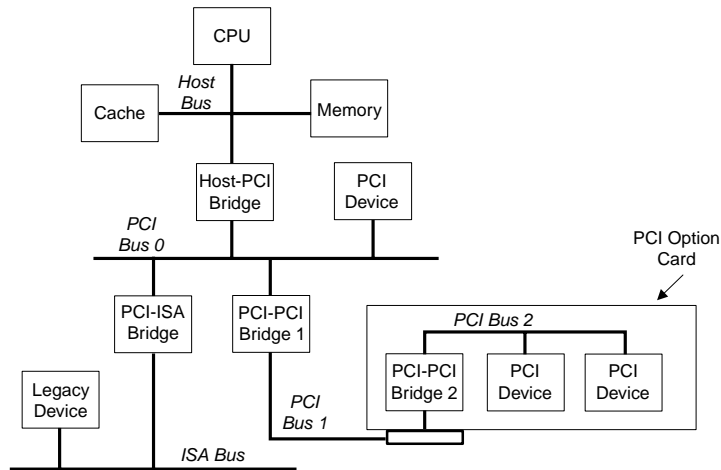


Figure 8-1: PCI bridge hierarchy.

The Host-to-PCI bridge provides the translation from the local processor bus to the PCI. In conventional PC environments, the Host-to-PCI bridge, often referred to as the “North Bridge,” is one element of the chipset and is usually contained in the same chip that manages main memory and the Level 2 cache. To the extent feasible, the architecture of the Host-to-PCI bridge mimics the P2P bridge specification.

PCI-to-Legacy Bus Bridge

Someday, the ISA bus will disappear from PC architecture. Someday income tax forms will be understandable. But for the time being, “legacy” busses such as ISA and EISA are supported through the mechanism of a PCI-to-Legacy Bridge. Like the Host-to-PCI bridge, this is usually an element of the chipset that also incorporates such traditional features as IDE, interrupt and DMA controllers. Legacy bridges often implement subtractive decoding because the cards on the legacy bus aren’t plug-and-play and thus can’t be configured. The PCI-to-ISA bridge is usually referred to as the “South Bridge.”

PCI-to-PCI Bridge

A PCI-to-PCI bridge provides a connection between a *primary interface* and a *secondary interface* (see Figure 8-2). The primary interface is the one electrically “closer” to the host CPU. These are also referred to as the *upstream bus* and the *downstream bus*. Transactions are said to flow downstream when the initiator is on the upstream bus and the target is on the downstream bus. Conversely, transactions flow upstream when the initiator is on the downstream side and the target is on the upstream side.

There is a corresponding symmetry to the structure of the bridge. When transactions flow downstream, the primary interface acts as a target and the secondary interface is the master. When transactions flow upstream, the converse is true. The secondary interface acts as the target and the primary interface is the master.

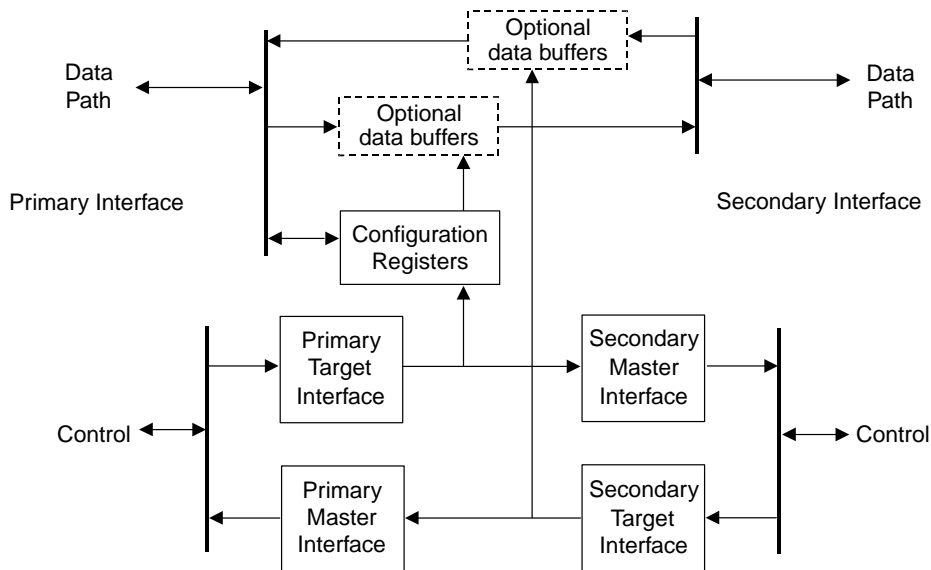


Figure 8-2: PCI bridge structure.

A bridge may, and usually does, include FIFO buffering for posting write transactions and prefetching read data.

One asymmetrical characteristic is that the bridge can only be configured and controlled from the primary interface.

Configuration Address Types

There are two configuration address formats called respectively Type 0 and Type 1. These are distinguished by the LSB of the address where Type 0 is 0 and Type 1 is 1. The difference is that Type 1 includes a device and bus number and Type 0 doesn't (see Figure 8-3). Type 1 represents a configuration transaction directed at a target on another (downstream) bus segment whereas a Type 0 transaction is directed at a target on the bus where the transaction originated. Type 0 transactions are not forwarded across a bridge.

As the Type 1 transaction passes from bridge to bridge, it eventually reaches the one whose downstream bus segment matches the bus number in the transaction. That bridge converts the Type 1 address to a Type 0 and forwards it to the downstream bus where it is executed.

Type 0



Type 1

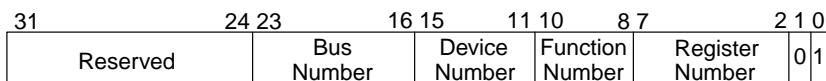


Figure 8-3: Configuration address types.

Configuration Header—Type 1

Figure 8-4 shows the Type 1 Configuration Header defined for the P2P bridge. The first six DWORDs of the Type 1 header are the same as the Type 0. The redefined fields are primarily concerned with identifying bus segments and establishing address windows.

*Optional	31		16 15		0		
	Device ID			Vendor ID		00h	
	Status			Command		04h	
	Class Code			Revision ID		08h	
	BIST*	Header Type	Primary Latency	Cache Line Size		0ch	
	Base Address Registers*					10h 14h	
	Secondary Latency	Subordinate Bus #	Secondary Bus #	Primary Bus #		18h	
	Secondary Status		IO Limit*	IO Base*		1Ch	
	Memory Limit		Memory Base			20h	
	Prefetchable Memory Limit*		Prefetchable Memory Base*			24h	
	Prefetchable Base Upper 32 bits*					28h	
	Prefetchable Limit Upper 32 bits*					2ch	
	IO Limit Upper 16 bits*		IO Base Upper 16 bits*			30h	
	Reserved					34h	
	Expansion ROM Base Address*					38h	
	Bridge Control			Interrupt Pin*	Interrupt Line*		3Ch

Figure 8-4: Configuration space header, Type 1.

The only transactions that a bridge is required to pass through are to 32-bit non-prefetchable memory space using the Memory Base and Limit registers. This space is generally used for memory mapped I/O. Optionally the bridge may support transactions to I/O space, either 64 K or 4 Gbytes using the I/O Base and Limit registers. It may also support prefetchable transactions to 32- or 64-bit address space using the Prefetchable Base and Limit registers.

Secondary Status Register. This register reports status on the secondary or downstream bus and, with the exception of one bit is identical to the Status Register. Bit 14 is redefined from `SIGNALLED_SYSTEM_ERROR` to `RECEIVED_SYSTEM_ERROR` to indicate that `SERR#` has been detected asserted on the Secondary Bus.

Secondary Latency Timer. Defines the timeslice for the secondary interface when the bridge is acting as the initiator.

The Type 1 header may have one or two Base Address Registers if the bridge implements features that fall outside the scope of the P2P Bridge specification. Likewise, it may have an Expansion ROM Base Address Register if, for example, it requires its own initialization code.

Bus Hierarchy and Bus Number Registers

As illustrated in Figure 8-5, there is a very specific strategy for numbering the bus segments in a large, hierarchical PCI system. The topology is a tree with the CPU and host bus at the root. The secondary interface of the Host/PCI bridge is always designated bus 0. The busses of each branch are numbered sequentially.

The three bus number registers provide the information necessary to route configuration transactions appropriately.

Primary Bus Number. Holds the bus number of the primary interface.

Secondary Bus Number. Holds the bus number of the secondary interface.

Subordinate Bus Number. Holds the bus number of the highest numbered bus downstream from this bridge

A bridge ignores Type 0 configuration addresses unless they are directed at the bridge device from the primary interface. A bridge

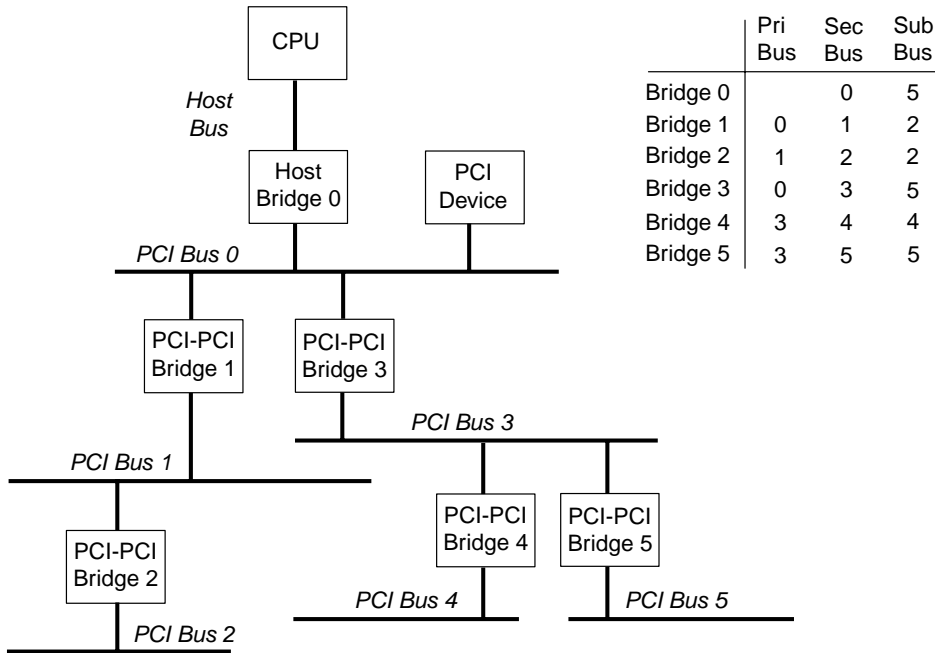


Figure 8-5: Bus number registers.

claims and passes downstream a Type 1 configuration address if the bus number falls within the range of busses subordinate to the bridge. That is, a bridge passes through a Type 1 address if the bus number is greater than the secondary bus number and less than or equal to the subordinate bus number. When a Type 1 address reaches its destination bus, that is the bus number equals the secondary bus register, it is converted to a Type 0 address and the bridge executes the transaction on the secondary interface.

As an example using the topology depicted in Figure 8-5, consider a configuration write directed to a target on bus number 4. Bridge 0 forwards the transaction to bus 0 as a Type 1 because the bus number is in range but is not the secondary bus number. Bridge 1 ignores the transaction because the bus number is not in range. As a result, bridge 2 never sees the transaction. Bridge 3 passes the transaction

downstream because the bus number is in range but not the secondary bus. Bridge 4 recognizes that the transaction is destined for its secondary bus and converts the address to a Type 0. Finally, bridge 5 ignores the transaction because the bus number is out of range.

Configuration transactions are not passed upstream unless they represent Special Cycle requests and the destination bus is not in the downstream range. If the destination bus is the primary interface, the bridge executes the Special Cycle.

A Type 1 configuration write to Device 1Fh, Function 7, Register 0 is interpreted as a Special Cycle Request. The bridge converts a Type 1 configuration write detected on the primary interface to a Special Cycle if the bus number equals the secondary bus number. A Type 1 configuration write detected on the secondary interface is converted to a Special Cycle if the bus number matches the Primary Bus number.

Address Filtering—the Base and Limit Registers

Once the system is configured, the primary function of the bus bridge is to act as an address filter. Memory and I/O addresses appearing on the primary interface that fall within the windows allocated to downstream busses are claimed and passed on. Addresses falling outside the windows on the primary bus are ignored.

Conversely, addresses on the secondary bus that fall within the downstream windows are ignored while addresses outside the windows are passed upstream. See Figure 8-6.

There are three possible address windows each defined by a pair of base and limit registers. Addresses within the range defined by the base and limit registers are in the window. The three possible windows are:

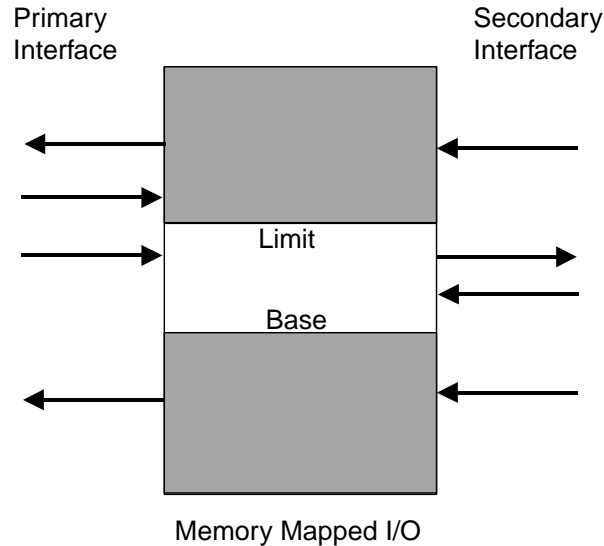


Figure 8-6: Address filtering with base and limit registers.

- Memory
- I/O
- Prefetchable Memory

Memory Base and Limit

32-bit memory space is the only one that the bridge is required to recognize. The upper twelve bits of the 16-bit Memory Base and Limit registers become the upper 12-bits of the 32-bit start and end addresses. Thus the granularity of the memory window is 1 Mbyte. Example:

Memory Base = 5550h
Memory Limit = 5560h

This defines a 2 Mbyte memory mapped window from 55500000h to 556FFFFFh.

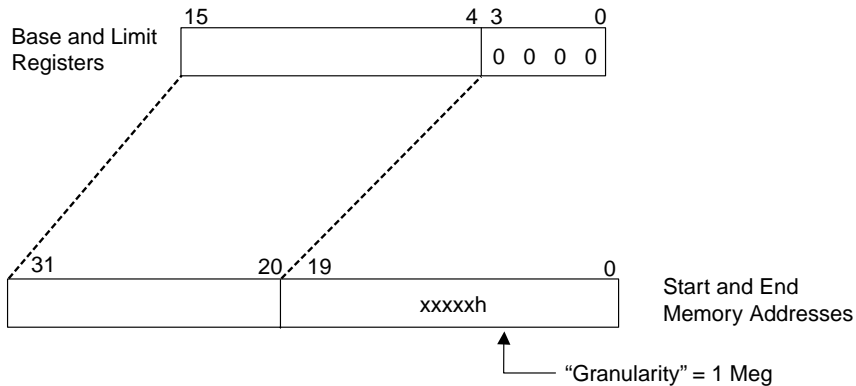


Figure 8-7: Memory base and limit registers.

I/O Base and Limit

A bridge may optionally support a 16-bit or 32-bit I/O address window (or it may not support I/O addressing at all). The low digit of the 8-bit I/O Base and Limit registers indicates whether the bridge supports 16- or 32-bit I/O addressing. The high digit becomes the high digit of a 16-bit address or the fourth digit of an 8-digit 32-bit address. The high order four digits of a 32-bit I/O address come from the I/O Base and Limit Upper 16 bits registers.

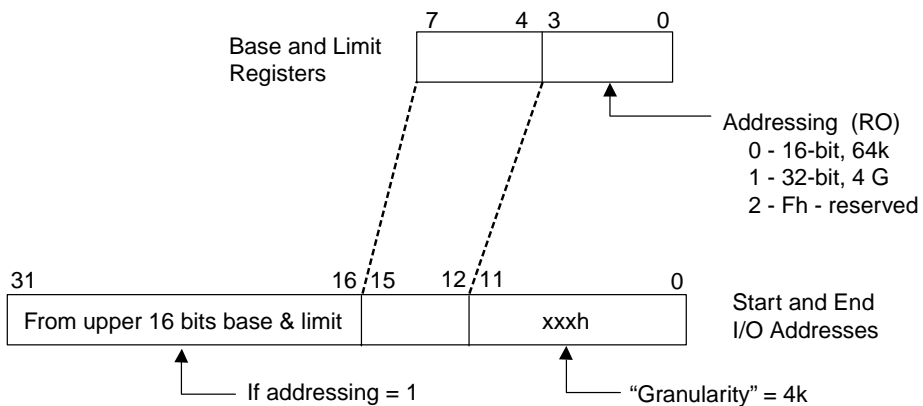


Figure 8-8: I/O base and limit registers.

Prefetchable Base and Limit

The prefetchable memory window is the only one that can be a 64-bit address. The low digit indicates whether the address space is 32 bits or 64 bits. If it is a 64-bit space, the upper 32 bits come from the Prefetchable Base and Limit, Upper 32 Bits. Again the granularity is 1 Mbyte.

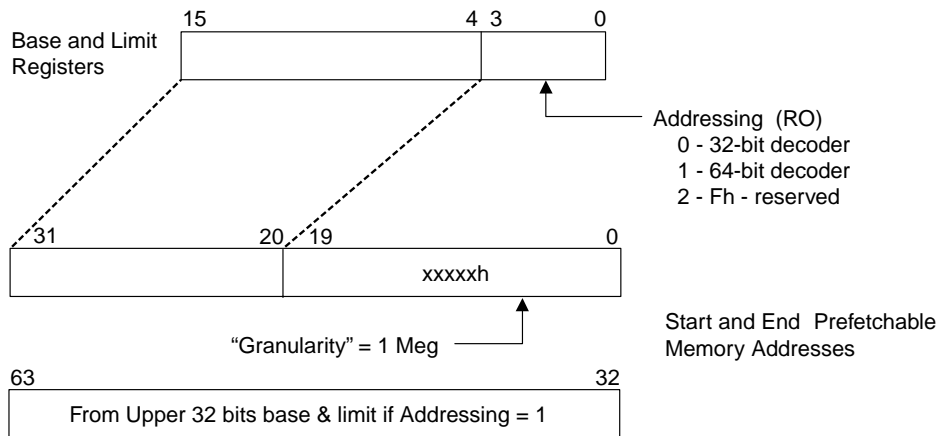


Figure 8-9: Prefetchable base and limit registers.

Prefetching and Posting to Improve Performance

Under certain circumstances the bridge is allowed to prefetch read data in the interest of improving performance. Data for a Memory Read Line or Memory Read Multiple command originating on either side of the bridge may always be prefetched. Data for a Memory Read command originating on the primary bus may be prefetched if it is in the prefetchable memory range, that is, the range defined by the Prefetchable Base and Limit registers if they exist.

A memory read originating on the secondary bus can be *assumed* to reference main memory and thus may be safely prefetched.

However, if the bridge does make this assumption, there must be a way to turn it off through a device-specific bit in configuration space. Note that I/O space is never prefetchable.

Under certain circumstances the bridge may *post* write data, meaning that it may accept and internally queue up write data before passing it on to the target on the other side. The definition of a posted transaction is one that completes on the originating bus before it completes on the destination bus. There are a couple of precautions to observe to make sure this works correctly.

The first rule is that the bridge must flush any write buffers to the target before accepting a read transaction. If the read were from a location that had just been written to, the initiator would get “stale” data if the buffers weren’t flushed first. If a bridge posts write data, it must be able to do so from both bus segments simultaneously. Stated another way, the bridge must have separate posted write buffers for both directions and not rely on flushing the buffer in one direction before accepting posted data in the other direction. Otherwise a deadlock can occur.

Interrupt Handling Across a Bridge

With respect to a bridge, interrupts are for all practical purposes sideband signals. Specifically, the INTx signals from the downstream bus segment are not routed through the bridge. This leads to an interesting problem illustrated in Figure 8-10.

Consider a mass storage controller, for example, on the downstream bus segment that has been instructed to write a block of data into the host’s main memory. Upon completing the write, the controller asserts an interrupt to signal completion. The question is: when the host sees the interrupt, is the data block in main memory?

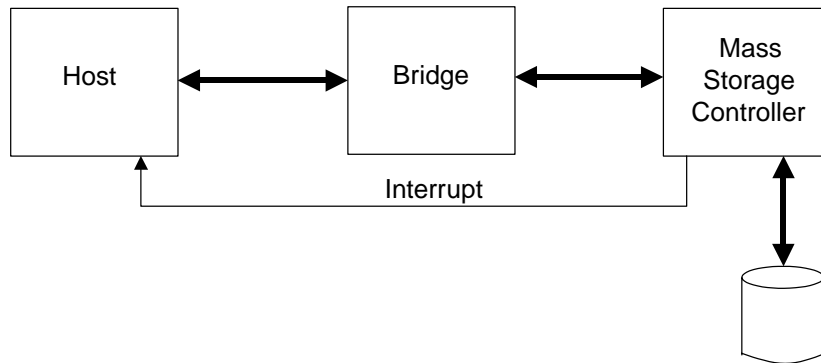


Figure 8-10: Interrupt handling across a bridge.

Chances are it isn't because the bridge most likely posted the write transaction. The nature of posting means that the storage controller saw the transaction completed, and asserted the interrupt, before the bridge completed the write to main memory.

The specification suggests three possible solutions to this problem:

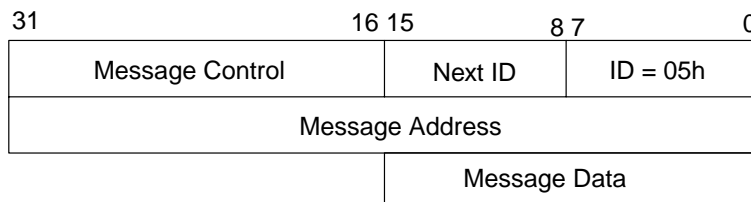
1. The system hardware can guarantee that all posting buffers are flushed before interrupts are delivered to the processor. This seems highly unlikely because it is outside the scope of the specification and would require additional hardware.
2. The interrupting device can perform a read of the data it just wrote. This flushes the posting buffers. This is a reasonable solution but, again, requires additional intelligence in the device.
3. The device driver can cause posting buffers to be flushed simply by reading any register in the interrupting device. Very likely the driver needs to read a register anyway and so the cost of this solution is virtually zero.

The Message Signaled Interrupt

The Message Signaled Interrupt capability introduced with Rev. 2.2 is another viable approach to solving this problem. The idea here is that a device can request service by sending a specific “message” to a specific destination address. This solves the interrupt ordering problem because the message is just another PCI bus transaction and therefore observes all the ordering rules that apply to bus transactions. In the scenario described above, the interrupt message would not reach the processor until the write data block had reached main memory.

MSI is implemented as an optional Capability. Figure 8-11 shows the layout of the MSI Capability structure. There are two formats depending on whether the device supports 64-bit addressing through the DAC. If it does, then it must implement the 64-bit version of the Message Address. Message Address references a DWORD and so the low order two bits are zero.

32-bit Address



64-bit Address

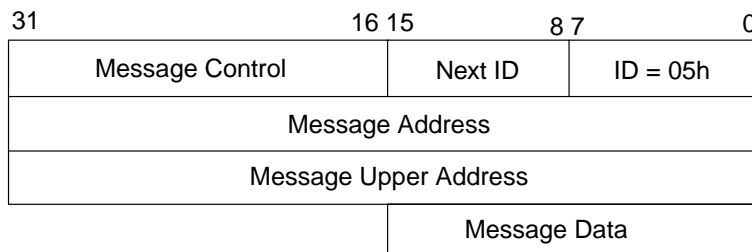


Figure 8-11: Message signaled interrupt (MSI) capability.

An MSI transaction involves a DWORD write of the Message Data field to the destination specified by the Message Address. Message Data is only two bytes so the upper two bytes of the DWORD are zero.

The Message Control Register provides system software control over the MSI process.

Bit

- 7 (RO) 1 = 64-bit address capability
- 6–4 Number of messages allocated. Less than or equal to the number of messages requested by bits 3:1.
- 3–1 (RO) Number of messages requested. System software reads this field to determine how many messages to allocate to this device.
- 0 (RO) 1 = MSI capability enabled.
0 = signal interrupts using INTx

The number of messages requested and allocated is in powers of two as follows:

Encoding	# of Messages
000	1
001	2
010	4
011	8
100	16
101	32

The values 6 and 7 are reserved. This is a mechanism for allocating multiple interrupts to a device. However, the system software has the option of allocating fewer interrupt messages to a device if there aren't enough to go around.

A device generates multiple messages by modifying the low order bits in the Message Data. Thus, if a device has been allocated four messages, these are distinguished by the value in the low order two bits of the Message Data field.

Bridge Support for VGA—Palette “Snooping”

Two issues come up with respect to PCI support of VGA-compatible devices. The first is ISA-compatible addressing. If a VGA device is located downstream of a PCI bridge, then the bridge must positively decode the range of memory and I/O addresses normally used by VGA independent of the address windows allocated by the configuration software. The VGA address ranges are:

- Memory: A0000h to BFFFFh
- I/O 3B0h to 3BBh and 3C0h to 3DFh

The VGA Enable bit in the Bridge Control Register controls whether or not the bridge positively decodes these ranges.

The other issue is known as “palette snooping” and is illustrated in Figure 8-12. The problem is that additional non-VGA devices such as graphics accelerators need to know the contents of the VGA’s palette registers. When both devices reside on the same bus segment

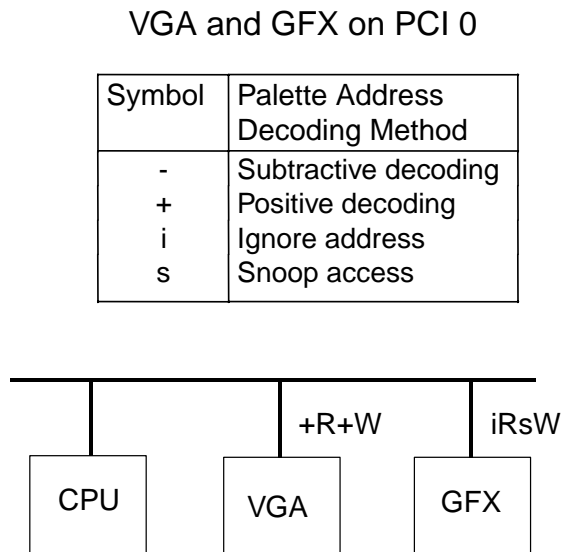


Figure 8-12: Palette “snoop” scenario.

as shown here, the VGA positively decodes both reads and writes to the palette registers. The GFX ignores read accesses but “snoops” writes. That is, when it detects a write to a palette register address, it latches the data but does not respond as a normal target would.

The ability of a device to snoop palette writes is controlled by the VGA Palette Snoop bit of the device’s Command Register and the Snoop Enable bit of the Bridge Control Register.

Things get more complicated when the two devices happen to be on opposite sides of a PCI bridge. Figure 8-13 illustrates a pair of scenarios involving a subtractive bridge. The upstream device must snoop the palette writes in order to give the bridge a chance to subtractively decode the transaction. The downstream device then positively decodes the writes and, if necessary, the reads.

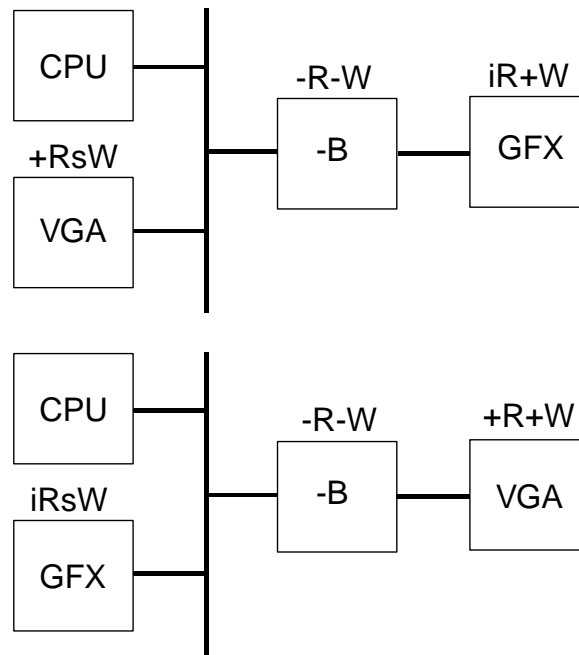


Figure 8-13: Palette “snoop” across a subtractive bridge.

Figure 8-14 illustrates the case of devices coupled across a positive decoding bridge. Again the downstream device positively decodes the writes and the upstream device snoops them. When the GFX is downstream, the bridge's Snoop Enable is set to 1 and VGA Enable is 0 causing the bridge to ignore reads and positively decode writes. When the VGA is downstream, VGA Enable is set to 1 to positively decode both reads and writes.

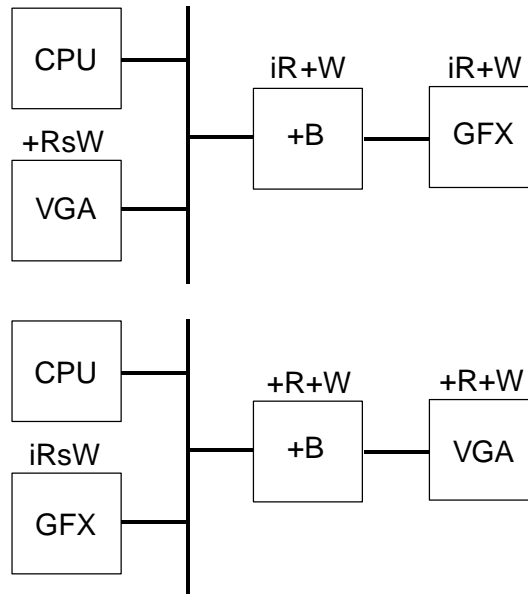


Figure 8-14: Palette “snoop” across a positive bridge.

Resource Locking

In any multi-master configuration there are inevitably occasions when one master needs exclusive (also called atomic or uninterrupted) access to a specific resource. Operations such as test-and-set or read-modify-write must be atomic to be useful. PCI defines a very clever locking mechanism that provides exclusive access to a specific

target or resource without interfering with accesses to other targets. That is, the *resource* is locked, not the bus.

With revision 2.2 of the specification, the lock mechanism is restricted to bridges and only in the downstream direction. Only the host-to-PCI bridge can initiate a locked transaction on behalf of its host processor. A PCI-to-PCI bridge simply passes the LOCK# signal downstream. All other devices are required to ignore the LOCK# signal. To quote the specification, "...the usefulness of a hardware-based locking mechanism has diminished and is only useful to prevent a deadlock or to provide backward compatibility." Really!!!

Backward compatibility refers to the hardware locking mechanism of the EISA bus. A PCI-to-EISA bridge may be the target of a locked transaction initiated by the host processor. A host-to-PCI bridge may honor a locked transaction to main memory initiated by a master on the EISA bus, but only if the PCI-to-EISA bridge resides on the same bus segment as the host bridge (LOCK# can't be propagated upstream).

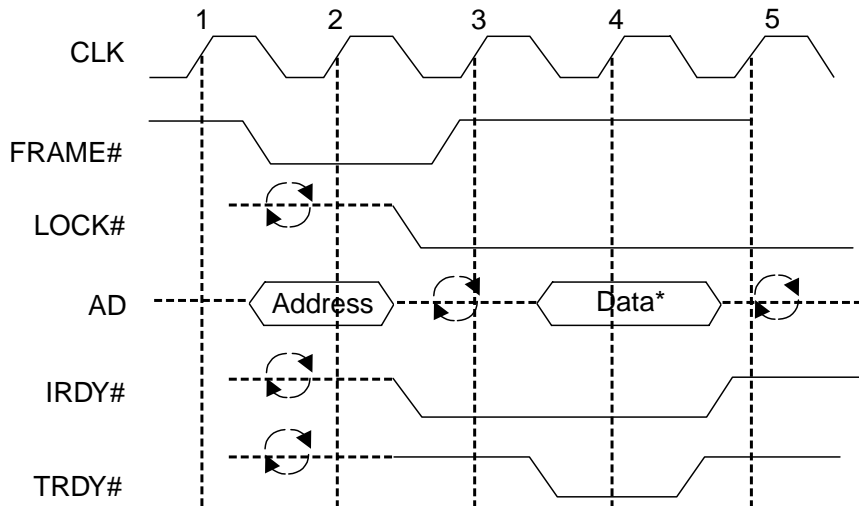
A master that requires exclusive access must first determine that the locking mechanism (the LOCK# signal) is available. The master doesn't assert its REQ# until it detects both FRAME# and LOCK# deasserted. However, while it is waiting for its GNT#, another master may claim the lock mechanism in which case this master deasserts its REQ# to wait for LOCK# to again become available.

The master asserts LOCK# when it finally acquires the bus and begins its transaction.

The master asserts LOCK# in the clock cycle following the assertion of FRAME#, i.e. immediately after the address phase (see Figure 8-15). The first data phase of a locked transaction must be a read. The target recognizes that it is being locked because:

- It was not locked prior to this transaction AND
- LOCK# is asserted during the data phase.

Note by the way that the lock does not take effect until the first data phase is complete. If the target retries the transaction before the first data phase, the master must release LOCK# and try again. Once the first data phase completes, the master keeps LOCK# asserted until the operation completes or an error condition causes an early termination.



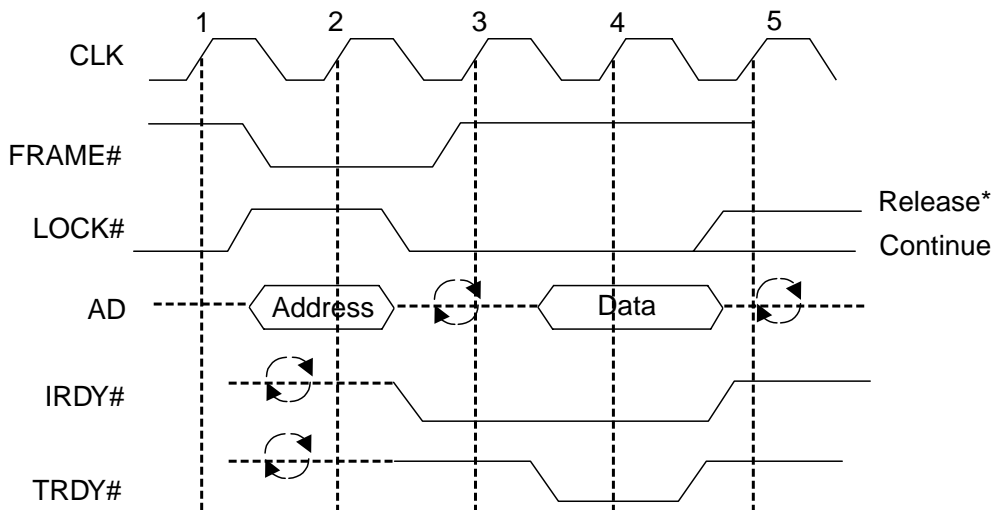
*First transaction must be a read

Figure 8-15: First lock cycle.

Once a master has established a lock, it can release the bus allowing other agents to carry out data transfers, but not with the device that has been locked. Figure 8-16 shows what happens when the master owning the lock executes a subsequent transaction to the locked device.

Clock

- 2 The master *deasserts* LOCK# during the address phase.
This is how the locked target knows its being accessed by the master owning the lock. Only the device asserting LOCK# can release it.
- 3 and 4 The transaction proceeds normally.
- 5 If this is the last transaction in the locked series, the master releases LOCK#.



*Target unlocks when it detects FRAME# and LOCK# deasserted

Figure 8-16: Subsequent lock transactions.

If a locked target sees LOCK# asserted during the address phase, a master other than the one owning the lock is attempting to access the locked target (Figure 8-17). In this case the target executes a retry abort.

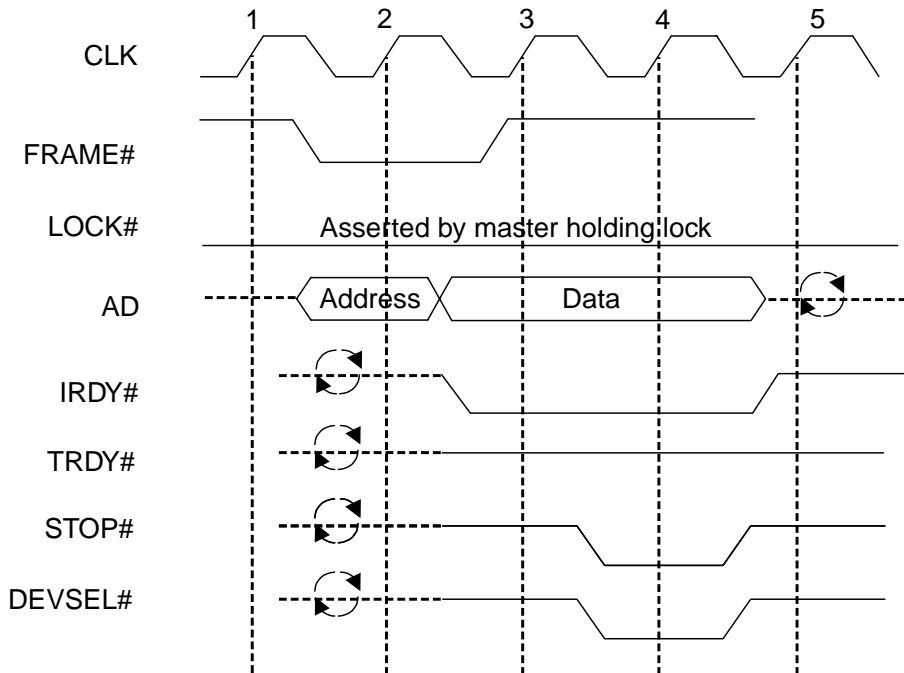


Figure 8-17: Accessing a locked target.

Summary

Bridging is the mechanism that allows a PCI system to expand beyond the electrical limits of a single bus segment. Bridges also serve to interface the host processor to PCI (host-to-PCI bridge) and to interface PCI to legacy busses (PCI-to-ISA bridge).

Once configured, the primary job of a PCI-to-PCI bridge is to act as an address filter, accepting transactions directed at agents downstream of it and ignoring transactions that fall outside of its address windows.

Bridges are allowed to prefetch read data and post write data provided they observe rules to prevent deadlocks and avoid reading stale data. Write posting can create a problem for interrupts because the interrupt may arrive at the host processor before the associated

data buffer is written to memory. The Message Signaled Interrupt capability solves this problem by treating interrupts as bus transactions rather than as separate signals. The interrupt transactions are subject to the same ordering rules as data transfers so that things happen in the right order.

Under rare circumstances, a master is allowed to lock a target for exclusive access. The PCI locking mechanism locks the resource and not the bus so that transactions to targets that are not locked may proceed.

CHAPTER 9

CompactPCI

CompactPCI is just an industrial version of the same PCI bus found in most contemporary PCs. It is electrically compatible with PCI and uses the same protocol. For reliability and ease of repair it is based on a passive backplane rather than the PC motherboard architecture. It utilizes Eurocard mechanics, made popular by VME, and a shielded pin-and-socket connector with 2mm pin spacing.

Perhaps its most interesting feature is that it supports up to eight slots per bus segment rather than the four slots typically found in conventional PCI implementations. This is due to the low capacitance of the connector and extensive simulations that were done in the course of developing the CompactPCI spec.

CompactPCI supports both 32- and 64-bit implementations at up to 33 MHz clock frequency for the full eight slots and 66 MHz over a maximum of five slots.

Why CompactPCI?

Advances in desktop PCs have a way of “migrating” into the world of industrial computing. In all cases the motivation is to leverage the efficiencies of scale resulting from the high volumes inherent in the desktop world. So it is with CompactPCI.

A wide range of reasonably priced PCI silicon is available for use in CompactPCI devices. VME silicon can't begin to match the volume of PCI and so remains generally more expensive.

The same considerations apply to software. Popular operating systems and applications already support PCI, particularly with respect to Plug and Play configurability.

Finally, the ability to swap boards in a running system (Hot Swap) is much further developed in CompactPCI than it is in other industrial busses.

CompactPCI is suitable for virtually any application involving industrial computing—process control, scientific instrumentation, environmental monitoring, etc. Three particular application areas

- Telephony
- Avionics
- Machine Vision

are particularly well suited to CompactPCI implementations.

The telephony industry is attracted by the low cost since they have a large number of channels to implement. They also like the high availability that comes from Hot Swap and it turns out that the 2 mm connector is already widely used in the industry.

With up to 64 bits in a 3U chassis, “compact” is the key word for avionics along with high performance.

Machine vision applications require the high throughput provided by PCI in a rugged industrial package.

Specifications

CompactPCI is embodied in a set of specifications maintained by the PCI Industrial Computer Manufacturer's Group (PICMG)

made up of companies involved in various aspects of industrial computing.

PCI Industrial Computer Manufacturers' Group
301 Edgewater Place, Suite 220
Wakefield, MA 01880
(781) 224-1100

www.picmg.org

The specifications currently maintained by PICMG include:

- CompactPCI Specification, Rev. 3.0 (September '99)
- CPCI Computer Telephony Spec., Rev. 1.0 (April '98)
- CPCI Hot Swap Specification, Rev. 1.0 (August '98)
- PCI-ISA Passive Backplane, Rev. 2.0

The basic CompactPCI Specification relies heavily on the PCI specification for electrical and protocol definitions.

Mechanical Implementation

The most obvious difference between PCI and CompactPCI is in mechanical implementation.

Card

CompactPCI mechanics are based on IEEE Standard 1101.10, commonly known as Eurocard. The basic card size is 160 mm by 100 mm (see Figure 9-1). This is a “3U” card corresponding to 3 “units” of front panel height. The front panel is actually 128.5 mm high. CompactPCI also uses a 6U board that has the same depth but is 233 mm high.

The 3U board requires an ejector handle at the bottom. The 6U board requires two ejectors, one at the top and one at the bottom.

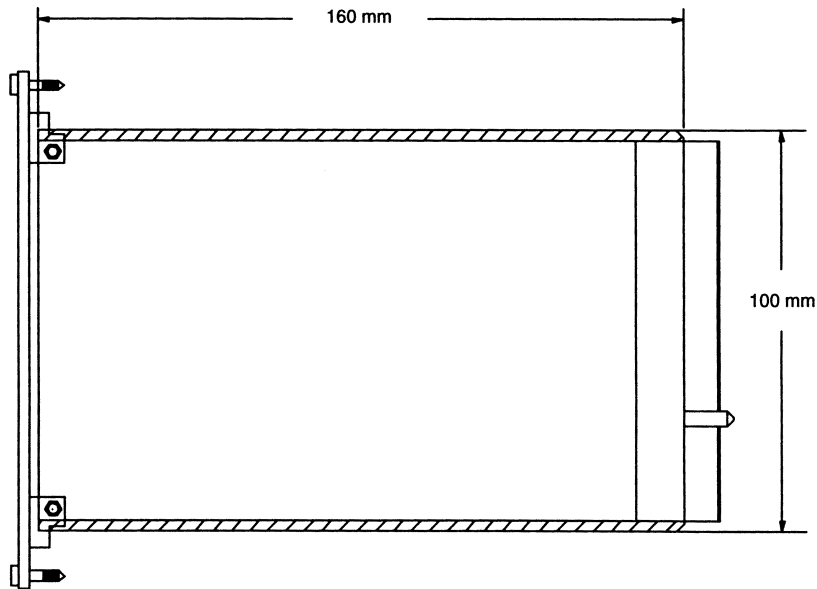


Figure 9-1: 3U Compact PCI card.

Backplane

Figure 9-2 shows a typical 3U backplane *segment* with eight slots. Each segment has exactly one system slot that may be located at either end of the segment. The system slot provides PCI's central resource functionality including the arbiter, clock distribution and required pull-up resistors. A physical backplane may consist of more than one segment. *Capability glyphs* provide visual indication of each slot's capability. The triangle identifies the system slot; the circle identifies peripheral slots.

Each slot has two numbers: a physical slot number and a logical slot number. Physical slot numbers range from 1 to N where N is the total number of slots in the backplane. Slot 1 is at the upper left-hand corner of the backplane. The physical slot number is indicated in the slot's compatibility glyph.

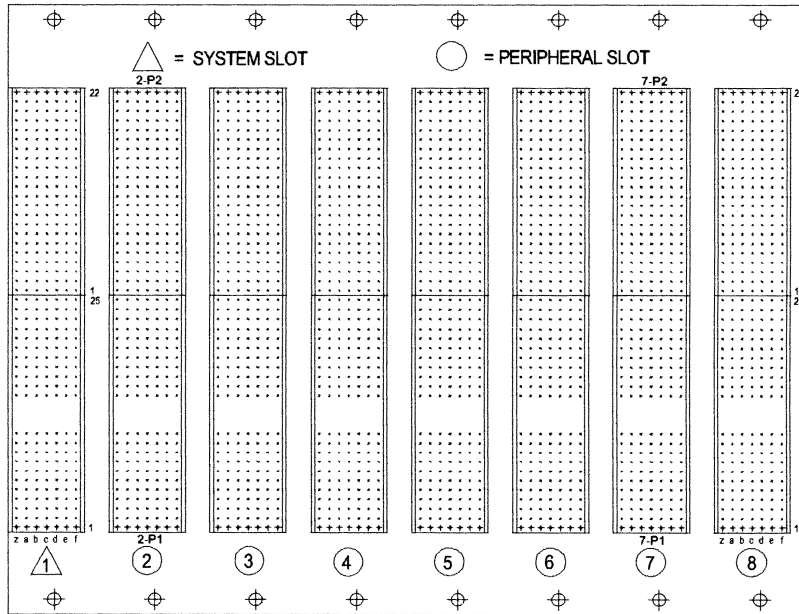


Figure 9-2: Typical 3U backplane segment with eight slots.

The logical slot number identifies a slot's relationship to the segment's system slot. The system slot is logical slot 1 and the peripheral slots are logical slots 2 through 8 in order.¹

The logical slot number defines which address bit the IDSEL pin is connected to and which REQ#/GNT# pair the slot uses. The connectors are also identified with respect to logical slot number in the form x-Py where x is the slot number and y is the connector number. For example, connector 2 in logical slot 5 would be identified as 5-P2.

¹ The specification text never explicitly says that logical slots proceed in numerical order starting from the system slot but the backplane drawings clearly infer it.

Other topologies besides the linear arrangement shown here are allowed. The only catch is that all the simulations assumed a linear topology with 0.8 inch board-to-board spacing. Any other topology must be simulated to verify conformance with PCI specs.

Connector

The basic CompactPCI pin-and-socket connector is organized as 47 rows of pins each (see Figure 9-3). The pins are on the backplane; the sockets are on the modules. Three of the rows are taken up by a keying mechanism that distinguishes 3.3 volt signaling from 5 volt signaling. That leaves 220 pins for power and signaling. A sixth outside column provides ground shielding. A seventh optional column on the other side also provides ground shielding.

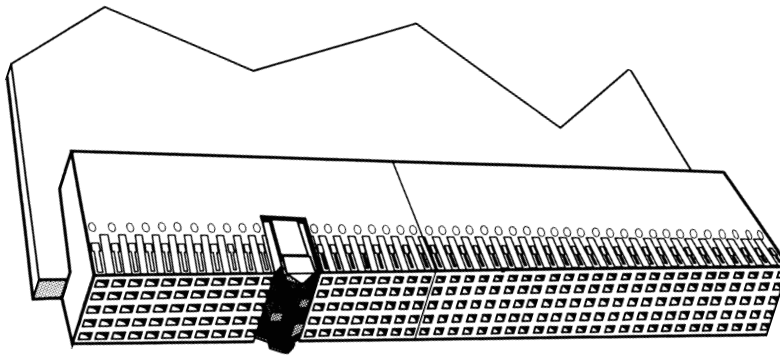


Figure 9-3: 2 mm pin and socket connector.

The connector is called “hard metric” meaning that the pin spacing is 2 mm, not 2.54 mm.

The 220-pin connector on the 3U core module is logically divided into two parts, J1 and J2, each 110 pins. J1 holds the basic 32-bit PCI bus as well as the connector key. J2 supports the 64-bit extension as well as the system slot functions. Optionally, J2 can be used for application I/O.

The extended 6U board adds three more connectors, J3 to J5 which are primarily intended for rear-panel I/O. J4 and J5 can also be used for things like a second CompactPCI bus, STD 32 or VME. The Telephony specification makes use of J4 and J5 (see Figure 9-4).

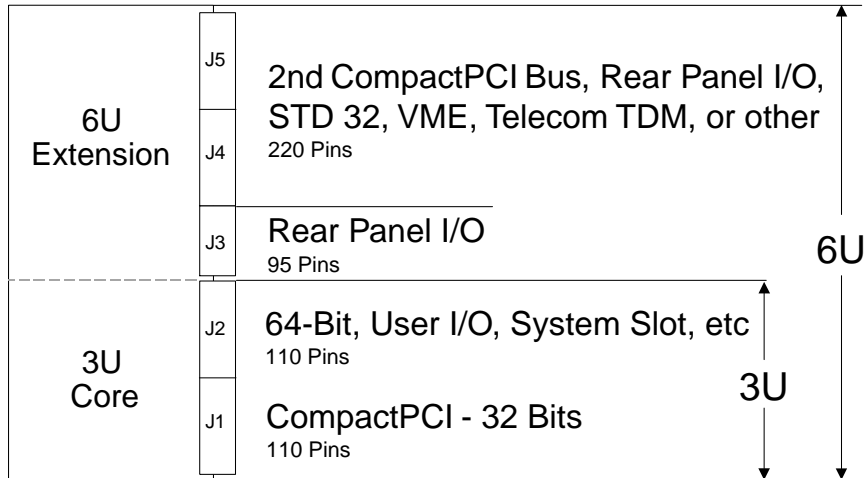


Figure 9-4: Compact PCI connector allocation.

Front and Rear Panel I/O

The front panel of a CompactPCI module may hold connectors for connection to external system elements. Alternatively, I/O connections may be made through the rear of the module on connectors J2/P2 through J5/P5. A recent addition to the 1101 specification, designated 1101.11, provides a standardized mechanism for rear-panel I/O in both the 3U and the extended 6U configuration (see Figure 9-5). The pins of P2 to P5 extend through both sides of the backplane allowing a “rear panel transition module” to be plugged into the back side.

Mechanically, the rear panel transition module is virtually a mirror image of the front side Compact PCI module. It is “typically”

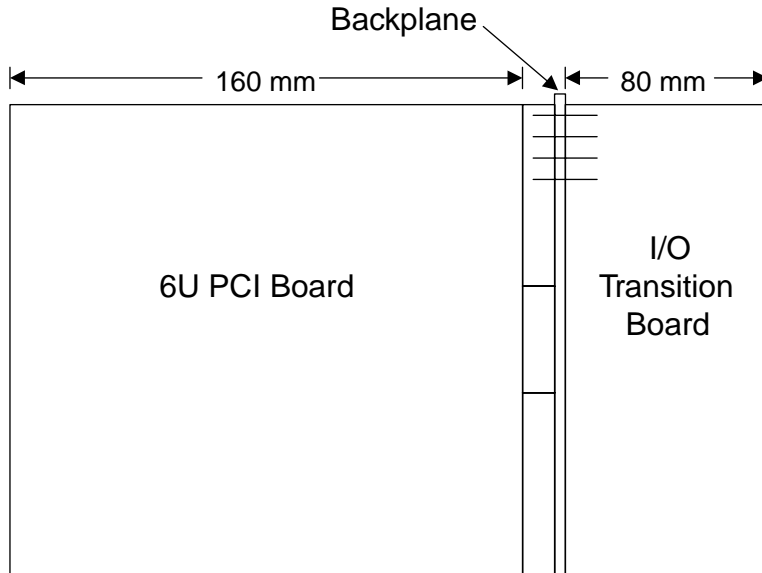


Figure 9-5: Rear panel I/O.

80 mm deep and “should” use the same panels, card guides, ejector handles, etc. The transition module may incorporate signal conditioning circuitry, which may include active components. Power for the signal conditioning circuitry may come from the designated power pins on P1 and P2 or may be supplied through the I/O pins.

The advantage to rear-panel I/O is that the module can be easily exchanged without having to undo and reconnect a bunch of cables. It also gives the front of the rack a neater, more professional appearance.

Electrical Implementation

The electrical differences between conventional PCI and CompactPCI involve some additional signals, routing of point-to-point and interrupt signals and design rules for boards and backplanes derived from the simulations.

Additional Signals

CompactPCI defines several additional signals not found in conventional PCI.

- PRST# *Push Button Reset*, PRST# may be used to reset the System Slot which would in turn reset the rest of the system by asserting PCI RST#. PRST# can be generated by a mechanical switch or pushbutton so the System Slot board is responsible for debouncing it as well as pulling it up.
- DEG# *Power Supply Derating Signal*. Assertion of this optional low-true signal indicates the power supply is derating its output, probably due to overheating. The system board must provide a pull-up.
- FAL# *Power Supply Fail Signal*. Assertion of this optional low-true signal indicates the power supply has failed. The system board must provide a pull-up.²
- SYSEN# *System Slot Identification*. This pin is grounded at the system slot and left open at all peripheral slots. A board that is capable of operating in either system or peripheral mode can use this signal to determine what type of slot it is plugged into.
- ENUM# *Enumeration*. Used by Hot Swap-capable cards to indicate either:
- The board has just been inserted
 - The board is about to be removed

² The specification is rather vague about the DEG# and FAL# signals. In particular, it doesn't say anything about relative timing. It would be nice, for example, if the FAL# signal were asserted a few milliseconds before the supply actually failed to give the host processor some time to do something about it.

	ENUM# tells the host processor to <i>enumerate</i> the system to determine which card is about to change state. See the next chapter on Hot Plug and Hot Swap.
BD_SEL#	<i>Board Select</i> . Also part of Hot Swap, this is one of two “short” pins on the backplane. When a board contacts this pin during a hot insertion, it is ready to be configured.
HEALTHY#	<i>Healthy</i> . This optional signal is used only in the High Availability model of Hot Swap. It allows a board to communicate to the system that it is functioning within tolerance and is ready to be configured.
GA[4::0]	<i>Geographic Addressing</i> . Allows a board to identify which physical slot it is plugged into. The GA pins are either grounded or left open at each slot to generate the binary numbers shown in Table 10-1. Boards that use this feature must pull these signals up with 10k resistors. Geographic addressing is required for backplanes that implement 64 bits and is optional for 32-bit backplanes.
IPMB_PWR, IPMB_SCL & IPMB_SDA	<i>System Management Bus</i> . These pins are reserved for implementing system management functions like board identification, environmental and voltage monitoring, etc. They are in the process of being defined by PICMG 2.9, <i>CompactPCI System Management Specification</i> .
INTP & INTS	<i>Legacy IDE interrupts</i> . Interrupt signals that should be connected to IRQ14 and IRQ15 respectively at the host processor. This provides a “compatibility mode” of operation for hard disks located on the CompactPCI bus.

Table 9-1: Geographic addressing.

Slot	J2-A22 GA4	J2-B22 GA3	J2-C22 GA2	J2-D22 GA1	J2-E22 GA0
1	GND	GND	GND	GND	Open
2	GND	GND	GND	Open	GND
3	GND	GND	GND	Open	Open
4	GND	GND	Open	GND	GND
5	GND	GND	Open	GND	Open
6	GND	GND	Open	Open	GND
7	GND	GND	Open	Open	Open
8	GND	Open	GND	GND	GND
9	GND	Open	GND	GND	Open
10	GND	Open	GND	Open	GND
11	GND	Open	GND	Open	Open
12	GND	Open	Open	GND	GND
13	GND	Open	Open	GND	Open
14	GND	Open	Open	Open	GND
15	GND	Open	Open	Open	Open
16	Open	GND	GND	GND	GND
17	Open	GND	GND	GND	Open
18	Open	GND	GND	Open	GND
19	Open	GND	GND	Open	Open
20	Open	GND	Open	GND	GND
21	Open	GND	Open	GND	Open
22	Open	GND	Open	Open	GND
23	Open	GND	Open	Open	Open
24	Open	Open	GND	GND	GND
25	Open	Open	GND	GND	Open
26	Open	Open	GND	Open	GND
27	Open	Open	GND	Open	Open
28	Open	Open	Open	GND	GND
29	Open	Open	Open	GND	Open
30	Open	Open	Open	Open	GND

Slots 0 and 31 are reserved.

Signal Routing

Conventional PCI makes no rules about the mapping of slots to REQ#/GNT# pairs or IDSEL. However CompactPCI specifies a mapping to logical slot numbers which may or may not correspond to physical slot numbers as shown in Table 9-2.

Table 9-2: Point-to-point signal routing.

Logical Slot	REQ#	GNT#	IDSEL
2	REQ0#	GNT0#	AD31
3	REQ1#	GNT1#	AD30
4	REQ2#	GNT2#	AD29
5	REQ3#	GNT3#	AD28
6	REQ4#	GNT4#	AD27
7	REQ5#	GNT5#	AD26
8	REQ6#	GNT6#	AD25

On the system slot, REQ0# and GNT0# utilize the pins on J1 normally used for REQ# and GNT#. All other REQ# and GNT# signals originate on P2 of the system slot.

The current specification requires that the system slot provide seven individual clock signals such that each peripheral slot in an 8-slot backplane has its own clock. Unlike earlier revisions, the precise mapping of clock sources on the system slot to clock sinks on peripheral slots is not specified in Rev. 3.0. Earlier revisions mandated only five clock sources from the system slot and provided for logical slots 2 and 3, and 4 and 5 to share clock signals. Subsequent simulation revealed that clock sharing would not be acceptable in a Hot Swap environment.

Interrupt routing in CompactPCI mandates the rotating “braided” routing that is recommended in the PCI specification (see Figure 9-6). In this way, each of the first four slots gets a unique interrupt for its INTA# pin. Interrupt sharing is not avoided entirely of course since the rotation repeats for the next four slots.

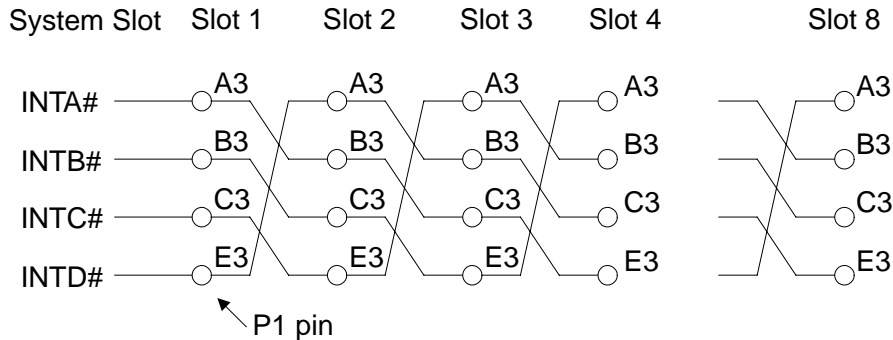


Figure 9-6: Required interrupt routing.

Backplane Design Rules

In the course of developing the CompactPCI specification, extensive simulations were done to verify conformance with the basic PCI electrical specifications. Pinout was optimized with respect to common mode noise and crosstalk as well as to allow easy hookup to the “preferred” signal ordering defined in the PCI specification for peripheral chips.

Several configurations were analyzed using both best and worst case buffers. These were:

- Fully loaded
- “Moderately” loaded
- Lightly loaded

The simulation results led to recommendations and rules for backplane and adapter card design.

The PCI specification has no requirement for the impedance of an unloaded motherboard. However the tighter electrical requirements of Compact PCI require that an unloaded backplane have an impedance of 65 ohms $\pm 10\%$.

Simulation revealed that a lightly loaded 8-slot configuration with a system slot board and a peripheral board loaded adjacent to the system slot using the strongest case drivers had a problem owing to the long unterminated stub presented by the unloaded connectors. This was solved with a fast Schottky diode termination at the far end of the backplane trace or on a termination board plugged into the farthest slot (see Figure 9-7).

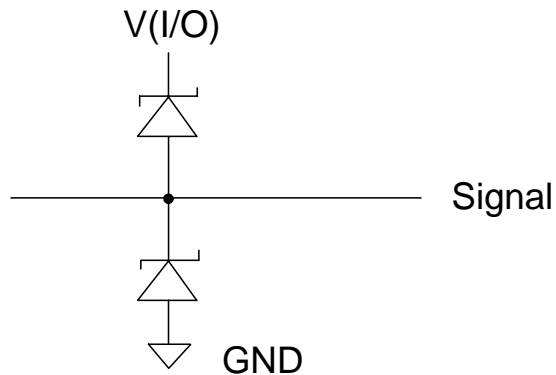


Figure 9-7: Backplane termination for lightly loaded case.

Board Design Rules

As shown in Figure 9-8, all CompactPCI boards must provide a 10 ohm series termination resistor for all PCI signals except, CLK, REQ#, GNT# and the JTAG signals. The resistor must be located no more than 0.6 inches from the connector pin. The trace length requirements are more “generous” than the PCI specification but include the series termination resistor.

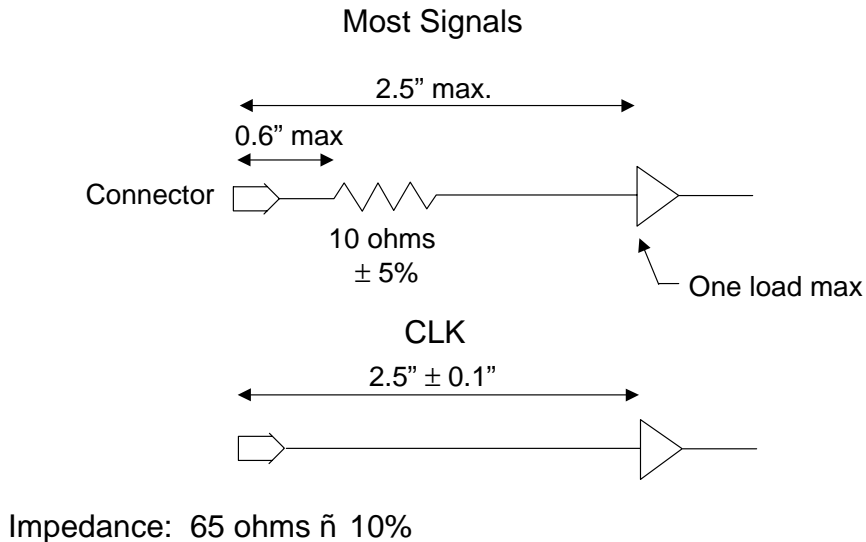


Figure 9-8: Board design rules.

The CLK signals require series termination resistors at their source on the system board “sized according to the output characteristics of the clock buffer”. The GNT# signals must be series terminated at the driver with an appropriately sized resistor. Likewise, REQ# *should* be series terminated on any board that drives it.

Like the backplane, adapter board impedance is more carefully specified in CompactPCI. Characteristic impedance is required to be 65 ohms ±10%.

CompactPCI Bridging

A standard 19-inch rack can, in theory, accommodate 21 or 22 slots at a 0.8 inch pitch. To control this many slots from a single host computer, you must bridge up to three 7- or 8-slot backplane segments. There are several approaches bridging standard backplanes. The obvious approach is a dual-wide module that plugs into the last peripheral slot of one backplane and into the adjacent system slot of

the next. Although this uses up two slots, it may be preferable to the alternatives in very high availability environments.

One alternative is a dual-wide module that plugs on to the rear of the backplane using the rear-panel I/O area. This leaves the front-side slots available for functional modules. Whether the bridge module plugs into the front or rear of the backplane, in both cases it is said to be “perpendicular” to the backplane. Another alternative, called a “pallet bridge”, is a board that plugs over the P1 and P2 pins on the rear of the backplane, *parallel* to the backplane. The advantage to rear-mounted bridges, whether perpendicular or parallel is that they don’t use any slots. On the other hand, they are difficult to replace should the need arise.

Figure 9-9 illustrates graphically how two segments may be bridged using either a front-plugging module or a rear-plugging pallet board. The host CPU resides in the system slot of Segment A, which is the “upstream” segment for the bridge while Segment B is the downstream segment. In the case of a rear-mounted bridge, the

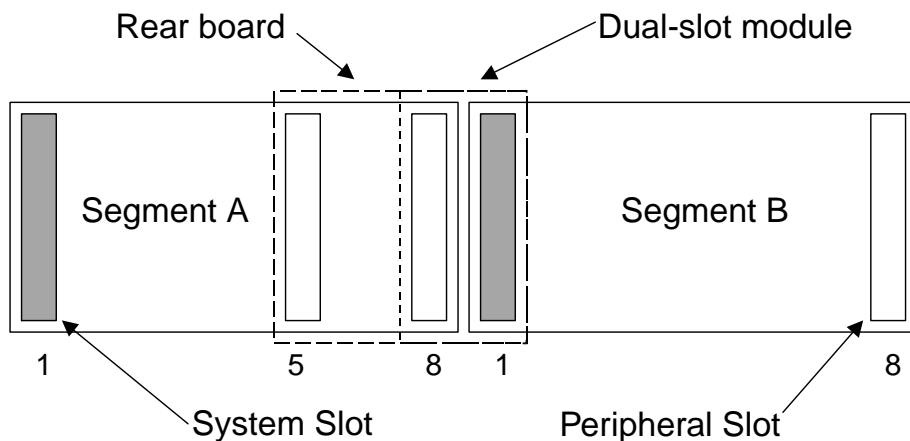


Figure 9-9: Bridging two segments using either a front-plugging module or a rear-plugging pallet board.

system slot in Segment B may be used for a peripheral card. Note that the physical size of the PCI bridge chip dictates that the pallet bridge board span several slots.

The configuration in the previous slide could be easily extended to accommodate a third Segment C. However, the problem with that approach is that transactions targeted at Segment C would have to pass through two bridges incurring latency in each one. It would be preferable to position the host processor so that it could bridge directly to each of the other segments.

Figure 9-10 shows a solution to that problem utilizing pallet bridge boards. The host processor resides in the system slot of Segment B and bridges directly to Segments A and C. Note that Segment A must have its system slot on the right and that two different bridge boards are required—one that bridges from right to left and another that bridges from left to right. In practice, the same PC board can be used for both forms with different mounting locations for the connectors.

The same strategy can be implemented with front-loading bridge modules. At least one vendor (Teknor) currently offers a dual-wide SBC that incorporates the bridge function.

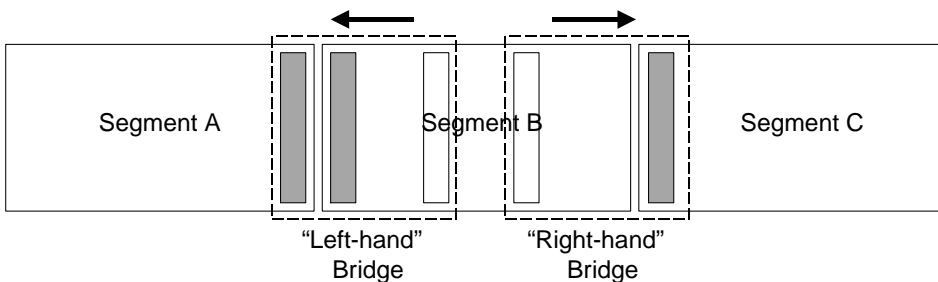


Figure 9-10: CPCI bridging of three segments.

Summary

CompactPCI is an industrial implementation of the PCI bus. It uses a passive backplane and standardized Eurocard mechanics. The use of low-capacitance connectors allows up to eight PCI slots per backplane segment.

CompactPCI defines additional signals beyond the basic PCI protocol. Among the features provided by these extra signals are: system slot identification, system enumeration and geographical addressing. Every board requires series termination of the bus signals.

CHAPTER 10

Hot Plug and Hot Swap

In high-availability, mission-critical environments, it is useful (in many cases absolutely essential) to be able to swap system components while the system is running. Attempting to do this in a system that has not taken Hot Pluggability into account will very likely result in component damage and system disruption.

Two approaches to Hot Pluggability have been developed. The PCISIG invented Hot Plug for conventional PCI cards. PICMG created Hot Swap for CompactPCI. In some ways these approaches complement each other and in other ways they contrast.

PCI Hot Plug

Hot Plug is defined in the *PCI Hot Plug Specification* Rev. 1.0 dated October 1997. The primary objective of Hot Plug is “to enable higher availability of file and application servers by standardizing key aspects of the process of removing and installing PCI adapter cards while the system is running”. In an effort to expedite market acceptance of Hot Plug by making virtually any PCI card Hot Pluggable, the specification puts the burden of hardware changes on the platform vendor. Specifically, the Hot Plug environment requires that each slot have:

- Power switches such that each board can be independently powered up and down.
- Bus isolation switches that electrically isolate the slot from the bus while a board is being inserted or removed.
- An independent RST# signal.
- A way of drawing an operator's attention to a specific slot, an "attention indicator", probably an LED. There may also be a slot state indicator to show whether the slot is on or off. The state indicator may be combined with the attention indicator.
- Ability to read the PRSNT[1:2]# signals while the board is isolated from the bus.
- Ability to read M66EN while the board is isolated from the bus.

Hot Plug follows what may be termed a "no surprises" strategy. This means that before inserting or removing a board, the operator must inform the operating system of his intentions and wait until the system notifies him that it is OK to proceed.

Hot Plug System Components

Figure 10-1 shows the elements added to a system to support Hot Plug. These include:

- *Hot Plug Controller.* Provides hardware control of the power and bus isolation switches, individual RST#s and attention indicators. Monitors PRSNT[1:2]# and M66EN.
- *Hot Plug System Driver.* Software interface to the Hot Plug controller. Implements the Hot Plug primitives described below.

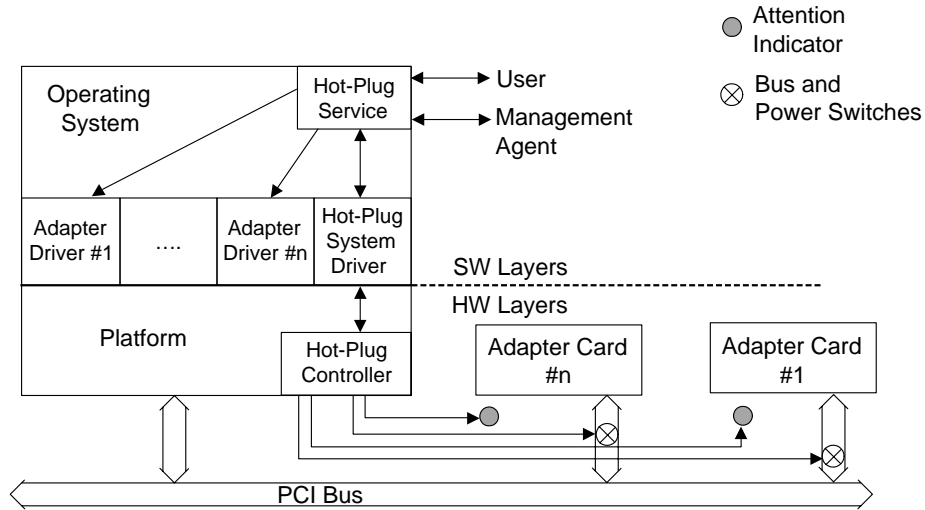


Figure 10-1: Hot Plug system components.

- **Hot Plug Service** Provides the interface to the user that allows the user to communicate insertion events to the system. Also interacts with adapter drivers to quiesce and activate the driver in response to insertion events.

Hot Plug Insertion

This is the sequence of events that occurs when a board is inserted into a Hot Plug environment. We start with the assumption that unoccupied slots are not powered, are isolated from the bus and that $RST\#$ is asserted.

1. The operator inserts the board in the slot.
2. The operator notifies the operating system that the board has been inserted in a specific slot
3. The Hot Plug Service notifies the Hot Plug System Driver to turn on the board. In turn, the Hot Plug System Driver directs the Hot Plug Controller to do the following:

- Power up the slot
 - Deassert RST# and connect the slot to the bus, in either order.
 - Change the optional slot state indicator to show that the slot is on.
4. The Hot Plug Service notifies the operating system that a new board has been inserted. Elements of the operating system and/or platform-dependent software then proceed to:
 - Configure the board
 - Load the adapter driver or create a new instance of the driver
 - Start the driver instance
 5. The Hot Plug Service notifies the operator that the board is ready.

Hot Plug Removal

This is the sequence of events that occurs when a board is removed from a Hot Plug environment:

1. The operator informs the Hot Plug Service of his desire to remove a specific board.
2. The Hot Plug Service notifies the operating system to “quiesce” the corresponding adapter driver instance. This means that the driver will complete the transaction currently in process and not accept any more transactions. When the current transaction is complete, it places the board in a state that will not generate interrupts or bus master activity.

3. The Hot Plug Service notifies the Hot Plug System Driver to turn off the slot. In turn, the Hot Plug System Driver directs the Hot Plug Controller to:
 - Assert RST# and isolate the slot from the bus, in either order.
 - Power down the slot
 - Change the optional slot state indicator to show that the slot is off.
4. The Hot Plug Service notifies the operator that the slot is off.
5. The operator removes the board.

Hot Plug Primitives

The Hot Plug Service is normally supplied by the operating system vendor while the Hot Plug System Driver is normally supplied by the platform vendor. The Hot Plug Primitives define what information must pass between these two elements. The primitives are defined only in terms of information passed in and information returned. The actual programming interface is operating system dependent. The operating system vendor may choose to split each primitive into multiple operations in the interest of efficiency.

Query Hot Plug System Driver

Parameters passed:	None
Parameters returned:	Set of logical slot identifiers controlled by this Hot Plug System Driver

This is the mechanism for each Hot Plug System Driver to report the set of logical slots that it controls.

Set Slot Status

Parameters passed:	Logical slot identifier New state {off, on} New Attention Indicator state {normal, attention}
Parameters returned:	Completion status {successful, wrong frequency, insufficient power, insufficient configuration resources, power failure, general failure}

This request controls the state of a hot plug slot and its associated Attention Indicator. For purposes of this primitive, a slot has only two states: on or off. In the on state the slot is powered and connected to the bus. In the off state it is not powered, isolated from the bus and RST# is asserted.

If the request fails, the Hot Plug System Driver should leave the slot in the off state unless otherwise indicated. Possible failures include:

- *Wrong Frequency.* A 33 MHz board was plugged into a bus segment operating at 66 MHz.
- *Insufficient Power.* By reading PRSNT[2::1], the Hot Plug System Driver has determined that there is not enough power left to turn on this slot.
- *Insufficient Configuration Resources.* If the Hot Plug System Driver is responsible for running the configuration routine, it may return this error if there are not enough resources available to configure the board. The slot may be left on if the operating system can tolerate a partially configured board.

- *Power Failure.* A power fault, i.e. short, was detected in the slot.
- *General Failure.* Any condition not otherwise covered.

Query Slot Status

Parameters passed:	Logical Slot identifier
Parameters returned:	Slot state {on, off}
	Board power requirement {not present, low, medium, high}
	Board frequency capability {33 MHz, 66 MHz, insufficient power}
	Slot frequency {33 MHz, 66 MHz}

This request returns the state of a hot plug slot and any board that may be plugged in. The Hot Plug System Driver determines a board's frequency capability either by reading M66EN or the 66 MHz CAPABLE bit in the Configuration Header. The driver will return an indication of insufficient power if it must read the Configuration Header but is unable to turn on the slot due to insufficient power.

Asynchronous Notification of Slot Status Change

Parameters passed:	Logical slot identifier
Parameters returned:	None

This primitive is used by the Hot Plug System Driver to notify the Hot Plug Service of an unsolicited change in the status of a slot such as a run-time power fault or a new board installed in a previously empty slot. This is not required for normal Hot Plug insertion and removal because these operations must follow “orderly procedures.” However, this primitive is very useful in Hot Swap as we’ll see shortly.

Expansion ROM

Intel x86 code contained in on-board expansion ROMs is generally designed to execute at boot time before the operating system is loaded. Attempting to execute this code at run time when the board is plugged into a running system may result in serious errors. It is up to the operating system vendor to specify whether or not expansion ROM code is executed during a hot insertion. If it is not, the board vendor must supply an alternate means to accomplish the same function, perhaps by incorporating it into the device driver.

CompactPCI Hot Swap

Hot Swap is defined by the *CompactPCI Hot Swap Specification*, Rev. 1.0 dated August 1998. Hot Swap builds on the architecture defined by Hot Plug but takes exactly the opposite tack in that the burden of support is placed on CompactPCI boards rather than the platform. This makes perfect sense in that the platform is in fact a passive backplane. The principal objectives of Hot Swap are:

- Allow “orderly insertion & extraction of boards” without powering down
- Provide for system reconfiguration and fault recovery with no down time
- Isolate faulty boards so system can continue in presence of a fault

The other key point that distinguishes Hot Swap from Hot Plug is the ability of the system to automatically detect an insertion “event”. This doesn’t mean that a Hot Swap capable operating system can tolerate surprises, but rather that the impending occurrence of an insertion event can be communicated to the operating system automatically.

Hot Swap Processes

Hot Swap can be described in terms of three processes. These processes can be described further as a procession of states. Each succeeding state is dependent on the success of the preceding state. The processes are described below in terms of board insertion where the order is:

1. Physical Connection
2. Hardware Connection
3. Software Connection

Board extraction operates in the reverse order:

1. Software Disconnection
2. Hardware Disconnection
3. Physical Extraction

Physical Connection

This is the process of actually inserting or removing the board. This process is embodied in the notion of “pin staging” or different pin lengths that are intended to make physical connection at different times. The first physical element to make contact as a board is inserted is the electrostatic card guide. Its purpose is to discharge any static accumulation that may have built up on the inserted board. Nevertheless, the specification cautions that “Normal ESD protection should be used when hot swapping boards.”

The longest pins—the first to make contact—are called the “Early Voltages”. These comprise two each +5V and +3.3V, the VIO pins and several grounds. The objective is to provide power for the PCI interface independent of the “backend,” application logic. At this stage, all of the PCI bus lines are *precharged* to approximately one volt to minimize the capacitive effects of attaching the lines to

Table 10-1: Pin staging.

Long Pins (first to engage)	Two each: +5 volts, +3.3 volts, Vio Six Gnd
Short Pins (last to engage)	BDSEL#, IDSEL
Medium Pins	Everything else

the active bus. Note that there is no guarantee as to what order these pins make contact. The only guarantee is that they will make contact before the next set of pins.

The medium length pins—the next to make contact—constitute all of the PCI bus signals. By the time they make contact they have been charged up to a voltage level that will not disturb operations on the bus.

Finally, the board contacts the two short pins, BDSEL# and IDSEL. The board pulls BDSEL# high with a pullup resistor. On the backplane this signal is either grounded or controlled by a High Availability platform.

The primary obligation of a Hot Swap board is to make a distinction between *Early Power* and *Back End Power*. Early Power is provided by long pins and is intended to power the PCI interface silicon so as to precharge all PCI bus lines to about 1 volt. Early power is limited to two amps.

Back end power is provided by all those power pins that are *not* long but rather medium. This is what provides power to the application logic after the PCI interface has stabilized. Even though the back end power pins are medium length, the board itself must control switching of back end power based on the assertion of BDSEL#.

Hardware Connection

This is the process of getting the board ready to configure. The board is connected to the PCI bus and the backend application logic is powered up. In the Basic and Full Hot Swap models this process happens automatically by virtue of contacting the BDSEL# pin. In the High Availability model BDSEL# is controlled by software through the Hot Swap Controller.

Software Connection

The Software Connection process begins with the deassertion of RST#. First, system software assigns resources to the board and initializes the board's Configuration Header. Next the device driver and other supporting software are loaded and/or instantiated. The board is now ready to be used.

Hot Swap Models

Hot Swap defines three levels of Hot Swap functionality as shown in Table 10-2. These are differentiated mainly in how the hardware and software connection processes are carried out. Basic Hot Swap is the simplest in terms of its impact on both boards and backplanes and, not surprisingly, has the least capability. The Basic Model operates much like Hot Plug in that the operator must interact with

Table 10-2: Hot Swap models.

System Type	Hardware Connection	Software Connection
Basic Hot Swap	Automatic in HW	Manually by Operator
Full Hot Swap	Automatic in HW	Controller (Automatic) by Software
High Availability	Controlled by SW	Controller (Automatic) by Software

the system to effect software connection and disconnection and the functions must be performed in the correct sequence for proper system operation.

Full Hot Swap provides facilities that automatically notify the system software that a board is either being plugged in or removed. This allows the software connection process to be automated.

High Availability adds software control of the hardware connection process in order to detect and, hopefully, isolate faulty boards. Each model builds on the facilities of the preceding simpler one.

The three models lead to several definitions of both platforms and boards as shown in Figure 10-2. The Hot Swap architecture is designed to allow all combinations of platforms and boards to interoperate. The system model is determined by the features of the lowest common denominator.

Platforms come in three flavors:

- Non-Hot Swap platforms lack any or all of the elements required to support Hot Swap.

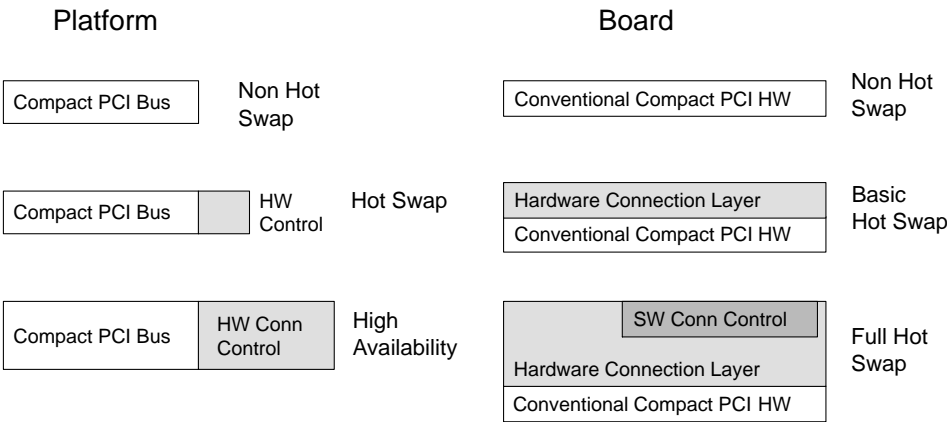


Figure 10-2: Hot Swap interoperability.

- Hot Swap platforms contain all the required Hot Swap elements.
- High Availability (HA) platforms contain the required Hot Swap elements plus a platform-specific implementation for Hardware Connection Control

Likewise, boards come in three flavors:

- Non-Hot Swap boards don't have a Hardware Connection Layer.
- Basic Hot Swap boards have the Hardware Connection Layer.
- Full Hot Swap boards add the Software Connection Control resources.

The various combinations of platforms and boards lead to the set of system configurations shown in Table 10-3. The Hot Swap specification layers on top of the basic *Compact PCI Specification*, providing backward compatibility and allowing Hot Swap to operate in a conventional platform. This configuration does not support Hot Swap.

Table 10-3: System configurations.

Platform Type	Board Type	System
Non-Hot Swap	Non-Hot Swap	Conventional <i>Compact PCI</i>
	Basic Hot Swap	
	Full Hot Swap	
Hot Swap	Non-Hot Swap	Conventional <i>CompactPCI</i>
	Basic Hot Swap	Basic Hot Swap System
	Full Hot Swap	Full Hot Swap System
High Availability	Non-Hot Swap	Conventional <i>CompactPCI</i>
	Basic Hot Swap	High Availability System
	Full Hot Swap	

A Hot Swap platform can have a mixture of Hot Swap and Non-Hot Swap boards. The Non-Hot Swap elements are of course not Hot Swappable but otherwise function normally. The Hot Swap boards are swappable. Note that HA functionality is a function of the platform and not the boards.

The specification cautions that mixing Basic and Full Hot Swap boards can create an environment that “could be confusing to the operator. If some boards configure automatically, and some require operator intervention, the operator may incorrectly insert (or extract) a board.”

Figure 10-3 shows the overall architectural model encompassing both hardware and software. Note the Hot Plug Service and Hot Plug System Driver. These are essentially the same elements defined by PCI Hot Plug.

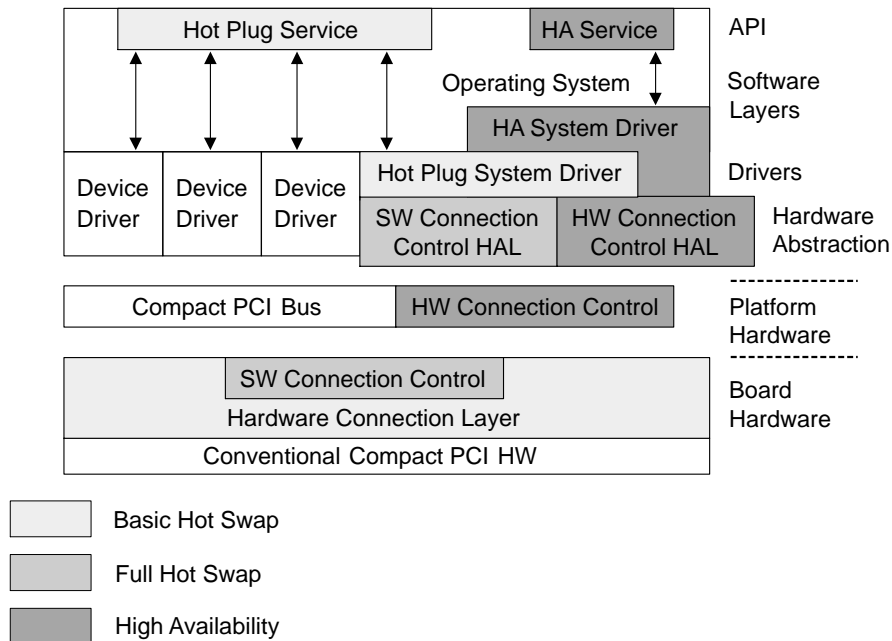


Figure 10-3: Hot Swap system architecture.

Resources for Full Hot Swap

The Software Connection process for Full Hot Swap requires several additional resources on both the board and the platform.

Handle Switch and Status LED

A full Hot Swap board has a switch activated by the lower ejector handle as shown in Figure 10-4. On insertion the switch changes state when the board is fully seated and the ejector handle is locked. On extraction, the switch changes state as soon as the handle is unlocked and before any movement of the board. The change in state of the switch is used to assert the ENUM# signal as described below.

System software lights the LED when it is safe to remove the board. This LED is *blue* and is also located near the lower ejector handle.

ENUM# Signal

The ENUM# signal is asserted to indicate a board insertion or extraction *event*. This tells the system software to *enumerate* the bus to determine the source of the event and what type of event (insertion or extraction) it is. ENUM# is controlled by the ejector handle switch. On insertion, ENUM# is asserted when the handle is locked after the board is fully inserted. On extraction, it is asserted when the handle is unlocked and before any movement of the board.

In response to ENUM#, the system software reads the Hot Swap Control/Status Register (CSR) to determine which board caused the enumeration event and what kind of event it is. For an insertion event the system activates the software connection process for the inserted board. For an extraction event the system activates the

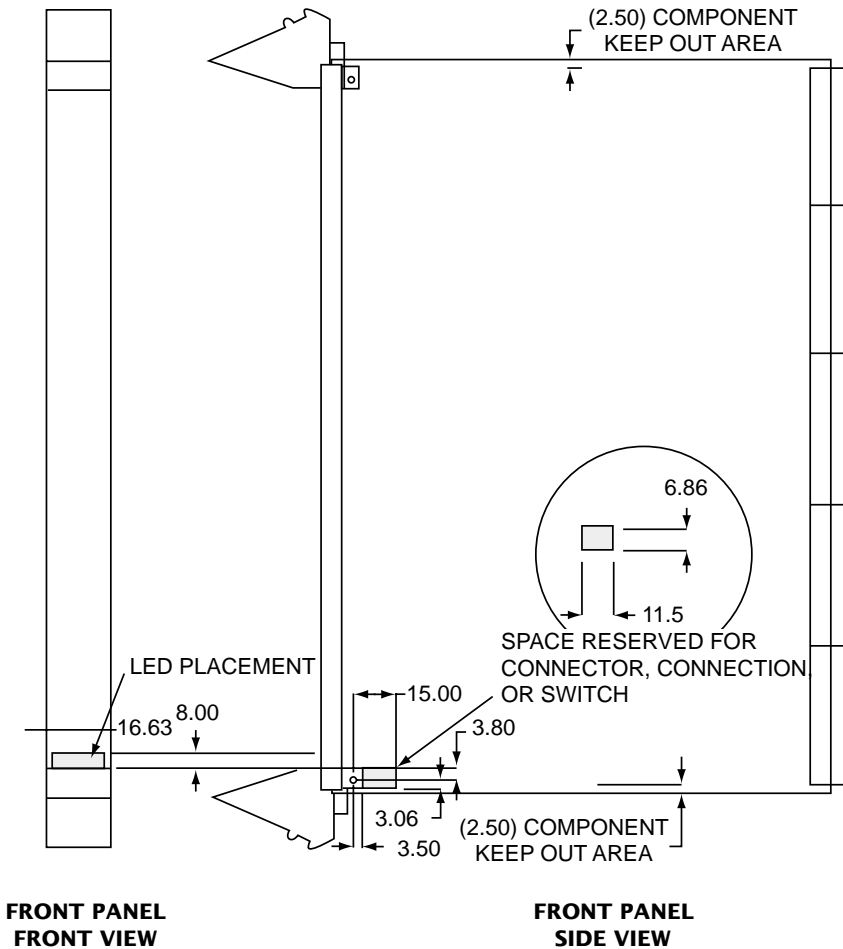


Figure 10-4: Hot Swap board with handle switch and status LED.

software disconnection process. When that process is complete, i.e. the board is “quiesced,” the system will illuminate the Status LED to inform the operator that it is safe to remove the board. The operator must not remove the board until the Status LED is lit.

The system may poll ENUM# but it is highly recommended that response to ENUM# be interrupt driven.

Hot Swap Control/Status Register

Figure 10-5 shows the Hot Swap Control/Status Register (HS_CSR). Two control and status bits are used by the software to identify the nature of an ENUM event. The INS bit indicates that the board has been inserted. The EXT bit means the board is about to be extracted. The assertion of either bit causes ENUM# to be asserted. When the Hot Swap driver identifies the event it writes a one to the appropriate bit (INS or EXT) to clear it. LOO (LED On/Off) controls the Status LED and EIM masks the assertion of ENUM#.

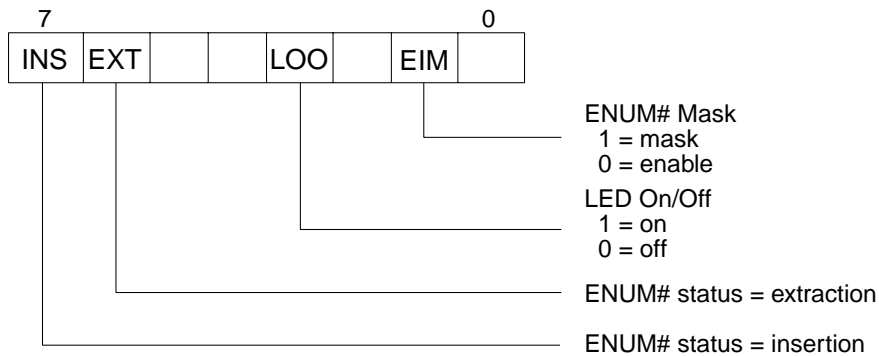


Figure 10-5: Hot Swap control/status register.

The preferred implementation of the HS_CSR, supported by “Hot Swap friendly” silicon, is as an Extended Capability using the Extended Capability Pointer in the Configuration Header. Figure 10-6 shows the Capability List entry.

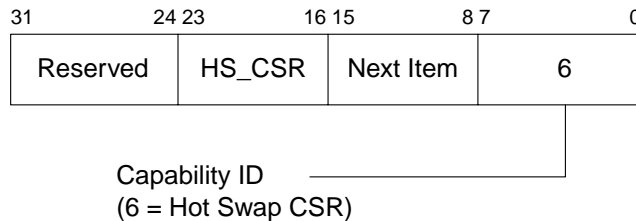


Figure 10-6: HS_CSR capabilities list entry.

Resources for High Availability

The additional features of the High Availability model are supported by a set of three radial signals that connect each slot to a *Hot Swap Controller* (HSC). The connection to the HSC, indeed the very location of the HSC, is considered outside the scope of the specification, that is it is platform-dependent. The three radial signals are: BD_SEL#, HEALTHY# and RST#.

BD_SEL# is used to control power to the back end logic on the board. It is pulled up to Vio with a 1.2 K resistor on the board. Back end power is applied when BD_SEL# is asserted.

In a platform without hardware connection control, BD_SEL# is simply tied to ground (see Figure 10-7). In fact, the pin is called out as GND in earlier revisions of the *Compact PCI Specification*. In this case back end power is turned on as soon as the short BD_SEL# pin makes contact.

In a HA platform the HSC pulls BD_SEL# down with a relatively high value resistor. So when no board is inserted, the HSC sees BD_SEL# as low. Upon insertion, the board's pullup overcomes the weak pulldown of the HSC and drives BD_SEL# high or unasserted

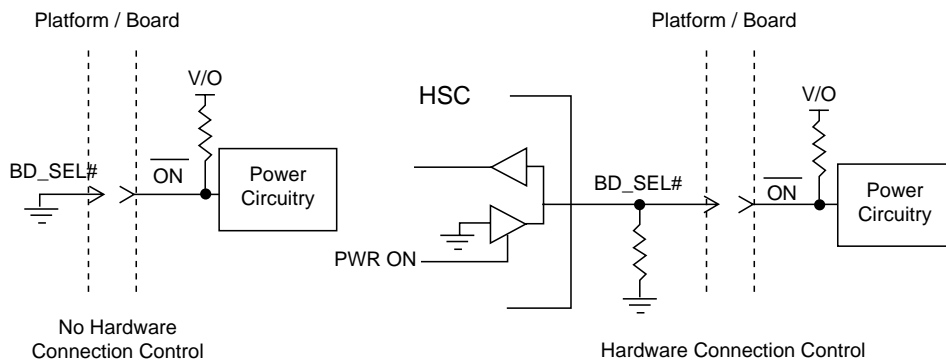


Figure 10-7: Handling of the BD_SEL# signal.

thus signaling its presence. When the HSC decides that it is appropriate to apply backend power, it drives BD_SEL# low.

HEALTHY# is an output from the board's power isolation circuitry and is asserted when back end power is within tolerance ($\pm 5\%$ according to the *Compact PCI Specification*). The assertion of HEALTHY# may also depend on other conditions being met, such as successfully completing a POST. This signal is not used on platforms without hardware connection control but all Hot Swap boards are required to implement it (see Figure 10-8).

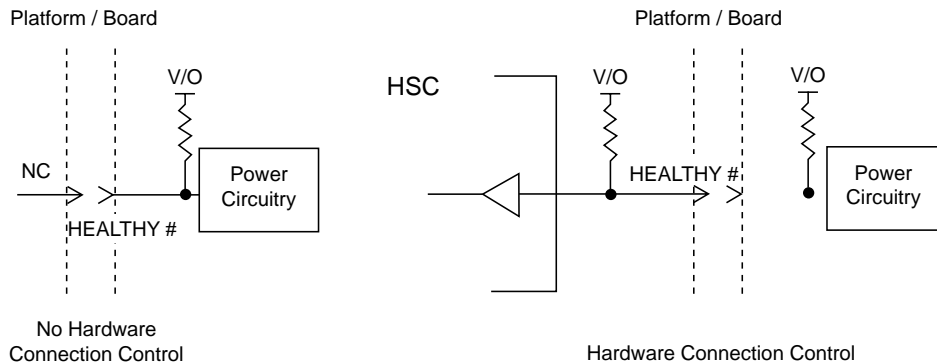


Figure 10-8: Handling of the HEALTHY# signal.

The HSC uses the assertion of HEALTHY# as the indication to deassert RST# to the board. Note that HEALTHY# may be deasserted at any time that the board determines it is not healthy. In response to seeing HEALTHY# deasserted, the HSC could notify the operating system of a faulty board and then attempt to isolate it by asserting RST# and deasserting BD_SEL#.

The specification suggests a weak pullup on HEALTHY# so the signal is not floating in non-HA platforms.

In a platform without hardware connection control, RST# is simply bussed to all slots and driven by the Host CPU in the system

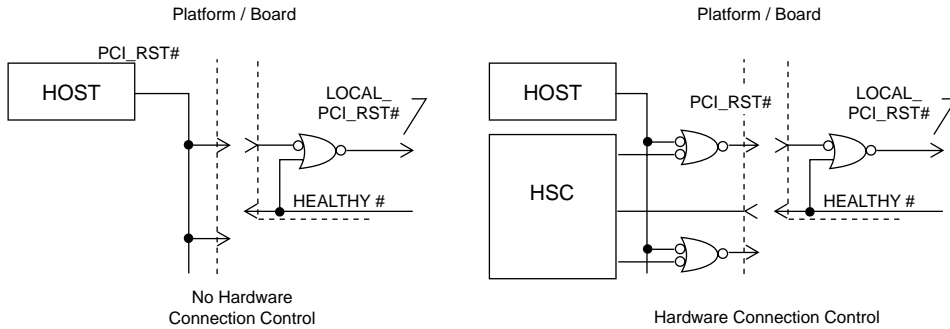


Figure 10-9: Handling of the RST# signal.

slot. In HA platforms, RST# may be a radial signal from the HSC in which case it must be the OR of the system host's reset and the slot-specific reset generated by the HSC. In any case, the board must keep its LOCAL_PCI_RST# asserted until HEALTHY# is asserted (see Figure 10-9).

Summary

The ability to change boards while the system is running is crucial to high-availability, mission-critical environments. Hot Plug, developed by the PCI SIG, and Hot Swap, developed by PICMG, provide solutions to this problem.

Hot Plug places the burden of supporting live insertion on the platform so that virtually any PCI board is Hot Pluggable. Support for live insertion includes bus isolation and power switches on the motherboard for each slot. The operator must notify the system of his desire to insert or extract a board and wait for confirmation before taking the action. The Hot Plug Service provides the interface to the operator while the Hot Plug System Driver controls the platform resources. A set of Hot Plug primitives defines the essence of an API between these two elements.

Hot Swap builds on the Hot Plug model but places the burden of support on the board with only minor modifications to the back-plane. Hot Swap also includes a mechanism to automatically detect an insertion or extraction event, simplifying the operator's task.

The specification defines three models of Hot Swap operation:

- *Basic*. Operates much like Hot Plug. The operator must notify the system before taking any action.
- *Full*. Provides for automatic detection of insertion and extraction events. This allows the software connection process to proceed without operator intervention.
- *High Availability*. Adds software control of the hardware connection process. A board is taken out of reset and allowed to operate only after it has confirmed that it is "healthy."

Class Codes

<i>Class/ Subclass</i>	<i>Programming Interface</i>
Class 00	Device predates class code definitions
00	Non-VGA devices
01	VGA devices
Class 01	Mass storage controllers
00	SCSI controller
01	IDE controller
xx	See Note 1
02	Floppy disk controller
03	IPI bus controller
04	RAID controller
Class 02	Network controllers
00	Ethernet
01	Token Ring
02	FDDI
03	ATM
04	ISDN
Class 03	Display controllers
00	VGA/8514
01	VGA-compatible
02	8514-compatible
01	XGA
02	3-D controller
Class 04	Multimedia devices
00	Video
01	Audio
02	Computer telephony

Note

- IDE Programming interface:
 - Bit 0 Operating mode (primary)
 - Bit 1 Programmable indicator (primary)
 - Bit 2 Operating mode (secondary)
 - Bit 3 Programmable indicator (secondary)
 - Bit 7 Master IDE device

Class / Subclass	Programming Interface
Class 05	Memory controllers
00	RAM
01	Flash
Class 06	Bridge devices
00	Host bridge
01	ISA bridge
02	EISA bridge
03	MCA bridge
04	PCI to PCI bridge
	00 PCI to PCI bridge
	01 Supports subtractive decode
05	PCMCIA bridge
06	NuBus bridge
07	Cardbus bridge
08	RACEway bridge
Class 07	Simple communication controllers
00	00 Generic XT-compatible serial controller
	01 16450-compatible serial controller
	02 16550-compatible serial controller
	03 16650-compatible serial controller
	04 16750-compatible serial controller
	05 16850-compatible serial controller
	06 16950-compatible serial controller
01	00 Parallel Port
	01 Bi-directional parallel port
	02 ECP 1.X compliant parallel port
	03 IEEE 1284 controller
	FE IEEE 1284 target device
02	Multiport serial controller
03	00 Generic modem
	01 Hayes compatible, 16450 interface (2)
	02 Hayes compatible, 16550 interface (2)
	03 Hayes compatible, 16650 interface (2)
	04 Hayes compatible, 16750 interface (2)

Note

2. First BAR (10h) maps appropriate compatible register set. Registers can be either memory or I/O mapped

Class / Subclass	Programming Interface
Class 08	Generic system peripherals
00	Interrupt controllers
	00 Generic 8259
	01 ISA PIC
	02 EISA PIC
	03 I/O APIC (3)
01	DMA controllers
	00 Generic 8237
	01 ISA DMA
	02 EISA DMA
02	Timers
	00 Generic 8254
	01 ISA system timer
02	EISA system timer (two timers)
03	Real-time clock
	00 Generic RTC
	01 ISA RTC
04	Generic PCI Hot-Plug controller
Class 09	Input devices
00	Keyboard controller
01	Digitizer (pen)
02	Mouse controller
03	Scanner controller
04	Gameport
	00 Generic
	02 See note 4
Class 0A	Generic docking station
Class 0B	Processors
00	386
01	486
02	Pentium
10	Alpha
20	Power PC
30	MIPS
40	Co-processor

Note

- First BAR (10h) requests minimum 32 bytes non-prefetchable space. Base+0 = I/O Select, Base+10h = I/O Window. See Intel 82420/82430 *PCIs et EISA Bridge Databook* (#290483-003) for more details
- “Legacy” game port. Byte at offset 01h aliases to byte at offset 00h

Class / Subclass	Programming Interface
Class 0C	Serial bus controllers
00	IEEE 1394
00	Firewire
10	Open HCI specification
01	ACCESS.bus
02	SSA
03	USB
00	Universal Host Controller specification
10	Open HCI specification
80	No specific programming interface
FE	USB device, not controller
04	Fibre Channel
05	System Management Bus
Class 0D	Wireless controllers
00	iRDA controller
01	Consumer IR controller
10	RF controller
Class 0E	Intelligent I/O controllers
00	
xx	I2O Architecture Specification 1.0
1.	Message FIFO at offset 40h
Class 0F	Satellite communication controllers
00	TV
01	Audio
02	Voice
03	Data
Class 10	Encryption/decryption
00	Network & computing en/decryption
10	Entertainment en/decryption
Class 11	Data acquisition & signal processing
00	DPIO modules

Note

- For all classes except 00, subclass 80h means “other”.

APPENDIX B

Connector Pin Assignments

PCI Connector

Pin	Side B (2)	Side A
1	–12V	TRST#
2	TCK	+12V
3	Gnd	TMS
4	TDO	TDI
5	+5V	+5V
6	+5V	INTA#
7	INTB#	INTC#
8	INTD#	+5V
9	PRSNT1#	Reserved
10	Reserved	+Vio (1)
11	PRSNT2#	Reserved
12	3.3V: Keyway	
13	5V: Gnd	
14	Reserved	3.3Vaux
15	Gnd	RST#
16	CLK	+Vio (1)
17	Gnd	GNT#
18	REQ#	Gnd
19	+Vio (1)	PME#
20	AD[31]	AD[30]
21	AD[29]	+3.3V
22	Gnd	AD[28]
23	AD[27]	AD[26]
24	AD[25]	Gnd

Pin	Side B (2)	Side A
25	+3.3V	AD[24]
26	C/BE[3]	IDSEL
27	AD[23]	+3.3V
28	Gnd	AD[22]
29	AD[21]	AD[20]
30	AD[19]	Gnd
31	+3.3V	AD[18]
32	AD[17]	AD[16]
33	C/BE[2]	+3.3V
34	Gnd	FRAME#
35	IRDY#	Gnd
36	+3.3V	TRDY#
37	DEVSEL#	Gnd
38	Gnd	STOP#
39	LOCK#	+3.3V
40	PERR#	Reserved
41	+3.3V	Reserved
42	SERR#	Gnd
43	+3.3V	PAR
44	C/BE[1]	AD[15]
45	AD[14]	+3.3V
46	Gnd	AD[13]
47	AD[12]	AD[11]
48	AD[10]	Gnd

PCI Connector *(continued)*

Pin	Side B (2)	Side A
49	M66EN	AD[09]
50	3.3V: Gnd	
51	5V: Keyway	
52	AD[08]	C/BE[0]
53	AD[07]	+3.3V
54	+3.3V	AD[06]
55	AD[05]	AD[04]
56	AD[03]	Gnd
57	Gnd	AD[02]
58	AD[01]	AD[00]
59	+Vio (1)	+Vio (1)
60	ACK64#	REQ64#
61	+5V	+5V
62	+5V	+5V
KEYWAY, 64 Bit Spacer		
63	Reserved	Gnd
64	Gnd	C/BE[7]
65	C/BE[6]	C/BE[5]
66	C/BE[4]	+Vio (1)
67	Gnd	PAR64
68	AD[63]	AD[62]
69	AD[61]	Gnd
70	+Vio (1)	AD[60]

Pin	Side B (2)	Side A
71	AD[59]	AD[58]
72	AD[57]	Gnd
73	Gnd	AD[56]
74	AD[55]	AD[54]
75	AD[53]	+Vio (1)
76	Gnd	AD[52]
77	AD[51]	AD[50]
78	AD[49]	Gnd
79	+Vio (4)	AD[48]
80	AD[47]	AD[46]
81	AD[45]	Gnd
82	Gnd	AD[44]
83	AD[43]	AD[42]
84	AD[41]	+Vio (1)
85	Gnd	AD[40]
86	AD[39]	AD[38]
87	AD[37]	Gnd
88	+Vio (1)	AD[36]
89	AD[35]	AD[34]
90	AD[33]	Gnd
91	Gnd	AD[32]
92	Reserved	Reserved
93	Reserved	Gnd
94	Gnd	Reserved



Compact PCI Connectors – P2

Pin	A	B	C	D	E
22	GA[4]	GA[3]	GA[2]	GA[1]	GA[0]
21	CLK6 (3)	Gnd	Res (4)	Res	Res
20	CLK5 (3)	Gnd	Res	Gnd	Res
19	Gnd	Gnd	Res	Res	Res
18	Bus Res	Bus Res	Bus Res	Gnd	Bus Res
17	Bus Res	Gnd	PRST#	REQ6# (3)	GNT6# (3)
16	Bus Res	Bus Res	DEG#	Gnd	Bus Res
15	Bus Res	Gnd	FAL#	REQ5# (3)	GNT5# (3)
14	AD[35]	AD[34]	AD[33]	Gnd	AD[32]
13	AD[38]	Gnd	+Vio (1)	AD[37]	AD[36]
12	AD[42]	AD[41]	AD[40]	Gnd	AD[39]
11	AD[45]	Gnd	+Vio (1)	AD[44]	AD[43]
10	AD[49]	AD[48]	AD[47]	Gnd	AD[46]
9	AD[52]	Gnd	+Vio (1)	AD[51]	AD[50]
8	AD[56]	AD[55]	AD[54]	Gnd	AD[53]
7	AD[59]	Gnd	+Vio (1)	AD[58]	AD[57]
6	AD[63]	AD[62]	AD[61]	Gnd	AD[60]
5	C/BE[5]	Gnd	+Vio (1)	C/BE[4]	PAR64
4	+Vio (1)	Bus Res	C/BE[7]	Gnd	C/BE[6]
3	CLK4 (3)	Gnd	GNT3# (3)	REQ4# (3)	GNT4# (3)
2	CLK2 (3)	CLK3 (3)	SYSEN#	GNT2# (3)	REQ3# (3)
1	CLK1 (3)	Gnd	REQ1# (3)	GNT1# (3)	REQ2# (3)

Compact PCI Connectors – P1

Pin	A	B	C	D	E
25	+5V	REQ64#	ENUM#	+3.3V	+5V
24	AD[01]	+5V	+V _{io} (1)	AD[00]	ACK64#
23	+3.3V	AD[04]	AD[03]	+5V	AD[02]
22	AD[07]	Gnd	+3.3V	AD[06]	AD[05]
21	+3.3V	AD[09]	AD[08]	M66EN	C/BE[0]
20	AD[12]	Gnd	+V _{io} (1)	AD[11]	AD[10]
19	+3.3V	AD[15]	AD[14]	Gnd	AD[13]
18	SERR#	Gnd	+3.3V	PAR	C/BE[1]
17	+3.3V	SCL(5)	SDA(5)	Gnd	PERR#
16	DEVSEL#	Gnd	+V _{io} (1)	STOP#	LOCK#
15	+3.3V	FRAME#	IRDY#	BDSEL#	TRDY#
14 – 12 KEYWAY					
11	AD[18]	AD[17]	AD[16]	Gnd	C/BE[2]
10	AD[21]	Gnd	+3.3V	AD[20]	AD[19]
9	C/BE[3]	IDSEL	AD[23]	Gnd	AD[22]
8	AD[26]	Gnd	+V _{io}	AD[25]	AD[24]
7	AD[30]	AD[29]	AD[28]	Gnd	AD[27]
6	REQ#	Gnd	+3.3V	CLK	AD[31]
5	Bus Res	Bus Res	RST#	Gnd	GNT#
4	PWR(5)	HLTHY#	+V _{io} (1)	INTP	INTS
3	INTA#	INTB#	INTC#	+5V	INTD#
2	TCK	+5V	TMS	TDO	TDI
1	+5V	–12V	TRST#	+12V	+5V

Notes

- V_{io} is +5V in 5V signaling environments and +3.3V in 3.3V signaling environments
- Side B = Component Side, Side A = Solder Side
- System slot only.
- “Res” = Reserved, “Bus Res” = Reserved and bussed to all slots in the segment.
- Power Management Bus, defined by PICMG 2.9, *Compact PCI System Management Specification*
-  = Long pin
 = Short pin

Index

- Address Filtering, 132–135
- AD[31:0], 37–39
- Arbitration:
 - defined, 22
 - fairness, 25–26
 - latency in, 27
 - latency timer, 28
 - two competing masters, 22–23
- Agent, 25
- Base address register (BAR), 103–107
- BIOS:
 - operating modes, 116
 - services, 118–119
- Bridging:
 - address filtering, 132–133
 - bus number registers, 130–132
 - compact PCI, 162–164
 - configuration address types, 128–130
 - hierarchies, 125–128
 - host to PCI, 125–126
 - interrupt handling, 136–137
 - PCI to legacy bus, 126
 - PCI to PCI, 127
 - prefetching, 106–107, 135–136
 - posting, 136
 - resource locking, 142–146
 - VGA palette “snooping”, 140–142
- Bus:
 - defined, 6–7
 - multiplexed, 7–8
 - non-multiplexed, 8
 - performance parameters, 8–9
- Bus parking, 26
- C/BE[3:0], 32–37
- Capabilities list, 110–111
- Central resource, 20
- Commands, PCI bus, 32–34
- Compact PCI:
 - additional signals found in, 156–157
 - board design rules, 161–162
 - bridging, 162–164
 - defined, 148
 - front and rear panel I/O, 154–155
 - Hot Swap, 173–185
 - mechanical details, 150–154
 - specifications, 149–150

- Configuration space:
 - accessing, 93–94
 - header types, 95–97
- DAC command, 65–66
- DEVSEL:
 - subtractive decoding, 40
 - timing, 39–41
- DWORD, 20
- Electrical characteristics:
 - 3.3 volt, 77–80
 - 5 volt, 72–76
 - DC, 75, 77
 - AC, 75, 79
 - reflected wave switching, 69–70
 - timing, 81–84, 87–88
- Error detection and reporting:
 - PAR, 51
 - PERR, 51
 - SERR, 53–54
- Expansion ROM base address
 - register, 107–108
- Extensions to PCI:
 - 64-bit, 62–63
 - 66 MHz, 85–88
- Firewire bus, 7
- FRAME, 15
- General Purpose Interface Bus (GPIB), 7
- Hot Plug:
 - defined, 166–167
 - insertion, 168–169
 - primitives, 170–73
 - removal, 169–170
 - system components, 167–168
- Hot Swap:
 - basic, 176–177
 - CSR, 180
 - defined, 173–174
 - event enumeration, 180–181
 - full, 177
 - hardware connection, 176
 - high availability, 177
 - Hot Swap Controller (HSC), 183
 - physical connection, 174–175
 - resources for, 180–185
 - software connection, 176
 - system architecture, 179
 - system configuration, 178
- Industry Standard Architecture (ISA) bus, 9
- Initiator, 20
- Interrupt handling:
 - INT_x, 57
 - interrupt acknowledge command, 59–60
 - message signaled interrupt, 138–139
- IRDY, 15
- I/O space, 38, 106

- Latency:
- acquisition, 27
 - arbitration, 27
 - bandwidth vs. latency, 28–30
 - defined, 27
 - initial target, 27
 - timer, 28
- Master, 20
- Mechanical characteristics:
- CompactPCI, 150–154
 - PCI, 88–90
- Memory space:
- prefetchable, 32, 106–107
- PCI Industrial Computer Manufacturers Group (PICMG), 149–150
- Peripheral Component Interconnect (PCI) bus:
- commands, 32–34
 - definitions, 20
 - electrical characteristics, 70–80
 - features, 11–12
 - mechanical characteristics, 88–90
 - signal groups, 13–18
 - signal types, 18–20
 - Special Interest Group (PCISIG), 12–13
 - timing specifications, 81–84
- Plug and Play Configuration:
- Base Address Registers (BAR), 103–106
- capabilities list, 110–11
 - command register, 97–99
 - configuration address space, 93–103
 - configuration header, 95–103
 - configuration transactions, 93–94
 - expansion ROM, 107–110
 - identification registers, 96–97
 - latency timer, 101–102
 - status register, 99–100
 - Vital Product Data (VPD), 111–115
- Prefetching read data, 135–136
- Posting write data, 136
- Read/write transactions, 34–45
- Reflected wave switching, 69
- Resource locking:
- LOCK, 15
- Sideband signals, 19–20
- Signaling environments:
- 3.3 volt, 77–80
 - 5 volt, 72–76
- STOP, 15
- Target, 20
- Timing specifications, 81–84
- Transactions:
- defined, 20

Transactions (*continued*):

read/write, 34–45
termination–master, 45
termination–target, 45–51

Universal Serial Bus (USB), 7

Vital Product Data (VPD), 111–115

VESA Local Bus, 10–11

64-bit operation:

AD[63:32], 63
ACK64, 62
C/BE[7:4], 62
PAR64, 63
REQ64, 62

66 MHz operation:

M66EN, 85

Demystifying Technology™ series

Technical publications by engineers, for engineers.

Video Demystified, Second Edition

A Handbook for the Digital Engineer

by Keith Jack

INCLUDES WINDOWS/MAC CD-ROM. Completely updated edition of the “bible” for digital video engineers and programmers.

1-878707-23-X \$59.95

NEW!

Short-range Wireless Communication

Fundamentals of RF System Design and Application

by Alan Bensky

INCLUDES WINDOWS CD-ROM. A clearly written, practical tutorial on short-range RF wireless design. The CD-ROM contains a number of useful Mathcad worksheets as well as a full searchable version of the book.

1-878707-53-1 \$49.95

NEW!

PCI Bus Demystified

by Doug Abbott

NEW! INCLUDES WINDOWS CD-ROM with full searchable version of the text. This concise guide covers PCI fundamentals, for both hardware and software designers, including the new PCI Hot-Plug Specification and new features of the PCI BIOS spec.

1-878707-54-X \$49.95

NEW!

Telecommunications Demystified

A Streamlined Course in Digital (and Some Analog) Communications for E.E. Students and Practicing Engineers

by Carl Nassar

NEW! INCLUDES WINDOWS CD-ROM. A straightforward and readable introduction to the theory, math, and science behind telecommunications. The CD-ROM contains useful Matlab tutorials and a full searchable version of the book.

1-878707-55-8 \$59.95

Digital Signal Processing Demystified

by James D. Broesch

INCLUDES WINDOWS 95/98 CD-ROM. A readable and practical introduction to the fundamentals of digital signal processing, including the design of digital filters.

1-878707-16-7 \$49.95

Digital Frequency Synthesis Demystified

by Bar-Giora Goldberg

INCLUDES WINDOWS CD-ROM. An essential reference for electronics engineers covering direct digital synthesis (DDS) and PLL frequency synthesis. The accompanying CD-ROM contains useful design tools and examples, and a DDS tutorial.

1-878707-47-7 \$49.95

Bebop to the Boolean Boogie

An Unconventional Guide to Electronics

Fundamentals, Components, and Processes

by Clive “Max” Maxfield

The essential reference on modern electronics, written with wit and style. Worth the price for the glossary alone!

1-878707-22-1 \$35.00

Modeling Engineering Systems

PC-Based Techniques and Design Tools

by Jack W. Lewis

INCLUDES WINDOWS CD-ROM. Teaches the fundamentals of math modeling and shows how to simulate any engineering system using a PC spreadsheet.

1-878707-08-6 \$29.95

Fibre Channel, Second Edition

Connection to the Future

by the Fibre Channel Industry Association

A concise guide to the fundamentals of the popular ANSI Fibre Channel standard for high-speed computer interconnection.

1-878707-45-0 \$16.95

Visit www.LLH-Publishing.com for great technical print books, eBooks, and more!