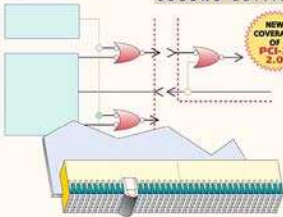




Demystifying Technology Series™
BY ENGINEERS FOR ENGINEERS

PCI BUS DEMYSTIFIED

SECOND EDITION



DOUG ABBOTT



PCI Bus Demystified

PCI Bus Demystified

SECOND EDITION

By Doug Abbott



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier



Newnes is an imprint of Elsevier
200 Wheeler Road, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Copyright © 2004, Elsevier Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>), by selecting "Customer Support" and then "Obtaining Permissions."



Recognizing the importance of preserving what has been written, Elsevier prints its books on acid-free paper whenever possible.

Library of Congress Cataloging-in-Publication Data

Abbott, Doug.
PCI bus demystified / by Doug Abbott.
p. cm.
Includes index.
ISBN 0-7506-7739-2
1. PCI bus (Computer bus) I. Title.

TK7895.B87A22 2004
004.6'4—dc22

2003069126

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

For information on all Newnes publications
visit our website at www.newnespress.com

04 05 06 07 08 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

To Susan:

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

To Brian:

*Budding actor/musician/lighting designer,
future pilot and all around neat kid.
Thanks for keeping me young at heart.*

Contents

Introduction	xi
Intended Audience	xii
The Rest of this Book	xiii
Chapter 1: Introducing the Peripheral Component Interconnect (PCI) Bus	1
So What is a Computer Bus?	1
Bus Taxonomy	3
What's Wrong with ISA and Attempts to Fix It	4
The VESA Local Bus	4
Introducing PCI	5
Features	6
The PCI Special Interest Group	6
PCI Signals	7
Signal Groups	8
Signal Types	12
Sideband Signals	13
Definitions	13
Summary	13
Chapter 2: Arbitration	15
The Arbitration Process	15
An Example of Fairness	17
Bus Parking	18
Latency	18
Summary	22

Chapter 3: Bus Protocol	23
PCI Bus Commands	23
Basic Read/Write Transactions	25
Transaction Termination—Initiator	33
Transaction Termination—Target	33
Error Detection and Reporting	38
Summary	40
Chapter 4: Optional and Advanced Features	41
Interrupt Handling	41
“Special” Cycle	44
64-bit Extensions	46
System Management Bus (SMB)	49
Summary	49
Chapter 5: Electrical and Mechanical Issues	51
A “Green” Architecture	51
Signaling Environments—3V and 5V	54
5 Volt Signaling Environment	55
3.3 Volt Signaling Environment	58
Timing Specifications	62
66 MHz PCI	67
Mechanical Details	72
Summary	74
Chapter 6: Plug and Play Configuration	77
Background	77
Configuration Address Space	78
Configuration Header—Type 0	80
Base Address Registers (BAR) (0x10 to 0x24)	88
Expansion ROM (0x30)	91
Capabilities List	94
Vital Product Data	96
Summary	99
Chapter 7: PCI Bridging	101
Bridge Types	101
Configuration Address Types	103
Configuration Header—Type 1	104

Bus Hierarchy and Bus Number Registers	105
Address Filtering—the Base and Limit Registers	107
Prefetching and Posting to Improve Performance	110
Interrupt Handling Across a Bridge	110
Bridge Support for VGA—Palette “Snooping”	116
Resource Locking	118
Summary	121
Chapter 8: System Configuration and the PCI BIOS	123
Who Configures the System?	123
System Configuration—An Overview	124
PCI BIOS	128
BIOS Services	130
Summary	135
Chapter 9: CompactPCI	137
Introduction—Why CompactPCI?	137
Mechanical	138
Electrical	144
CompactPCI Bridging	149
Summary	151
Chapter 10: Hot Plug and Hot Swap	153
PCI Hot Plug	153
CompactPCI Hot Swap	158
Summary	168
Chapter 11: Introduction to PCI-X	171
Why PCI-X	171
What’s New	171
Backward Compatibility	173
Chapter 12: PCI-X Protocol	175
PCI-X Transactions	176
PCI-X Commands	179
Split Transactions	184
Error Checking and Correcting	188
Source Synchronous Data Transfers	192
Device ID Messages	193

16-bit Bus	196
Summary	197
Chapter 13: PCI-X Configuration and Initialization	199
Configuration Transaction Timing	199
Configuration Address	200
Configuration Attributes	202
Configuration Header Registers	203
PCI-X Capabilities List Item	204
Mode 2 Configuration Space	211
Initialization	213
Summary	215
Chapter 14: PCI-X Electrical and Mechanical Features	217
Signal Categories	217
3.3 Volt Signaling Environment	218
Timing Parameters	223
1.5 Volt Signaling Environment	224
PCIXCAP and MODE2	232
Mechanical	233
Summary	234
Appendix A: Class Codes	235
Appendix B: Connector Pin Assignments	241
Index	245

Introduction

Computer technology continues to advance with breathtaking speed in accordance with Moore's¹ Law, among other things. A new computer is nearly obsolete as soon as you take it out of the box. Since Intel first defined the PCI bus back in 1992, memory bandwidth requirements in virtually all computer systems, whether they be high-end servers or home PCs for game playing, have increased by orders of magnitude. What started out as a 32-bit, 33 MHz bus with a bandwidth of 132 Mbytes per second was first expanded to 64 bits and then to 66 MHz. The maximum bandwidth of a 64-bit, 66 MHz system is thus 528 Mbytes per second. This insatiable demand for bandwidth led to the development of PCI-X, an extension to the PCI standard that boosts maximum bandwidth to 4 Gigabytes per second.

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 systems demand high-performance interconnects that also allow devices to be changed or upgraded with a minimum of effort and technical knowledge by the end user.

In response to this need, PCI (Peripheral Component Interconnect) and a subsequent major enhancement, PCI-X, have emerged as the dominant mechanism for interconnecting the elements of modern, high-performance computer systems. It is a

¹ Proposed by Gordon Moore of Intel way back in 1965. Moore observed that the number of transistors that can be economically put on a single chip doubles approximately every 18 months. Every once in a while someone says that Moore's Law is reaching the end of the line, but Intel expects it to hold at least through the end of the decade.

well thought out standard with a number of forward-looking features that should keep it relevant into the foreseeable future. Originally conceived as a mechanism for interconnecting peripheral components on a motherboard, PCI has evolved into at least a half dozen different physical implementations and form factors directed at specific market segments yet all using the same basic bus protocol. In the form of Compact PCI, it is the foundation of modern telecommunications infrastructure. 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 the enhancements defined by PCI-X push maximum bandwidth to 4 gigabytes.
- *Configurability.* PCI offers the ability to configure a system automatically, relieving the user of the tedious 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 both PCI and PCI-X. It is not a replacement for the specifications nor does it go into that level of detail. Think of it as a “companion” to the specifications. If you're designing boards or systems using off-the-shelf PCI interface silicon, this book together with the silicon vendor's data

sheets should be sufficient for your needs. On the other hand, if your goal is to design PCI silicon, motherboards or backplanes, you will undoubtedly need to reference the specifications for additional detail.

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

The book is roughly divided into two parts. The first part, consisting of chapters 1 through 10 introduces the basic concepts of PCI. The second part, consisting of chapters 11 through 14, describes the enhancements introduced with PCI-X.

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: 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 8: Looks at the software issues of configuring a PCI system and describes the PCI BIOS, a platform-independent API for accessing PCI’s configuration space.

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

Chapter 10: Looks 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 11: Introduces the protocol enhancements defined in the PCI-X addendum to the PCI specification.

Chapter 12: Describes the PCI-X protocol enhancements.

Chapter 13: Describes PCI-X enhancements and modifications to Configuration Space and transactions.

Chapter 14: Covers the electrical signaling aspects of PCI-X

Introducing the Peripheral Component Interconnect (PCI) Bus

Let's start with a little history. 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. Digital Equipment Corp. (DEC) was an early and very successful proponent of this strategy.

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.

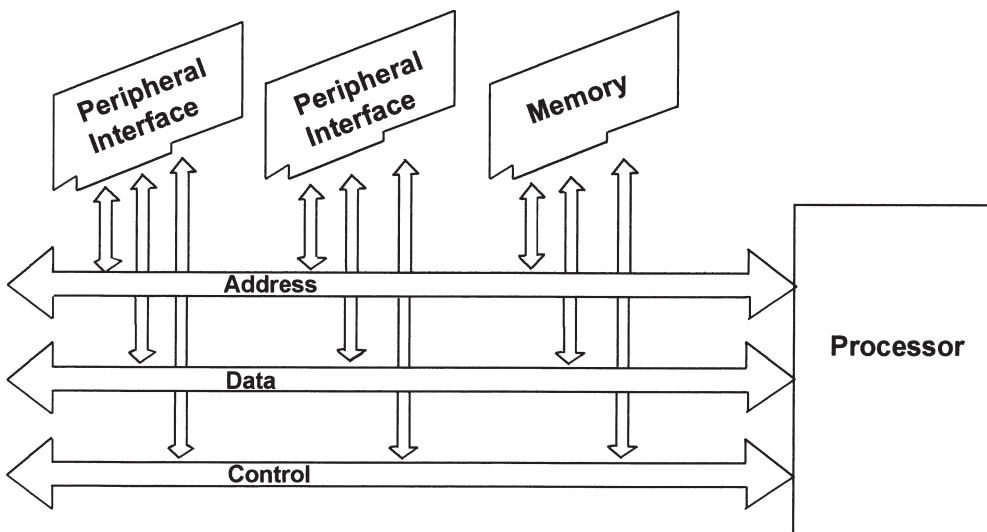


Figure 1-1: Functional diagram of a computer bus.

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 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 bandwidth 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.

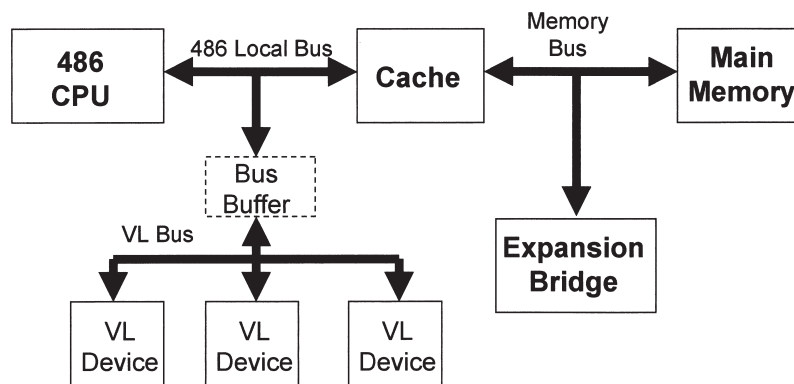


Figure 1-2: Functional diagram of the VL Bus.

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 his 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 Revision 2.1 in the first quarter of 1995. Revision 2.2 was a major rewrite of the entire specification released in February 1999. The current Revision, 3.0, released in August 2002, is more "evolutionary" in nature.

Features

PCI implements a set of forward-looking features that should keep it relevant for the foreseeable future:

- The maximum theoretical transfer rate of the base configuration is 132 Mbytes/sec. Extensions to the base PCI specification can boost this by a factor of four to 528 Mbytes/sec. The PCI-X enhancements extend potential bandwidth to over 4 gigabytes per second.
- 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. Revision 2.3 removes support for 5 volt only cards.

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
- and so forth

The specifications currently include:

- PCI Local Bus Spec., Rev. 3.0
- 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
- PCI-X Protocol Addendum to the PCI Spec., Rev. 2.0
- PCI Express Spec., Rev. 1.0a

Copies of the specifications may be ordered from:

PCI Special Interest Group www.pcisig.com
5440 SW Westgate Dr., #217
Portland, OR 97221
503-291-2569 FAX: 503-297-1090

All of the specifications are available in PDF format on a single CD-ROM.

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.

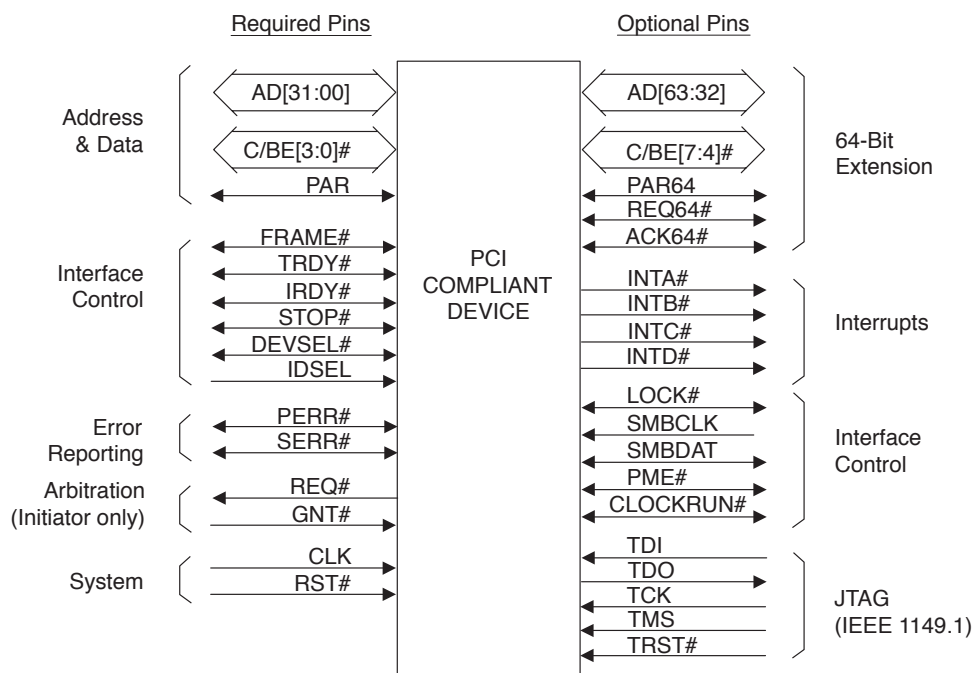


Figure 1-3: Signals defined in the PCI standard.

Signal Groups

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

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 deasserted, the transaction is in its final data phase or has completed. (s/t/s)

DEVSEL# *Device Select* indicates that a device has decoded its address as the target of the current transaction. (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 transaction¹. (s/t/s)

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

LOCK# (optional) Indicates an *atomic* operation to a bridge that may require multiple transactions to complete. The use of LOCK# in bridges is permitted but strongly discouraged as it may result in data overruns for streaming data and communication devices. It is expected that LOCK# will eventually be removed from the PCI specification.

¹ An early contributor to the original PCI specification refers to the set of signals including FRAME#, DEVSEL#, IRDY#, TRDY# and STOP# as the “five brothers.”

SMBCLK (optional) System Management Bus clock. (o/d)

SMBDAT (optional) System Management Bus data. (o/d)

Arbitration (master only)

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. Deassertion 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]#², CLKRUN#, M66EN, 3.3Vaux

Of course, a signal that is “input only” must originate somewhere and that somewhere is generally the central resource (see definitions below).

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#, SMBCLK, SMBDAT

² Although the specification calls the PRSNT# signals input only, this author believes they are really outputs because the information is being communicated from the add-in card to the motherboard.

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. The next chapter will describe the arbitration mechanism that regulates orderly access to the bus by multiple masters.

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#** signal indicating that this master is next in line to use the bus.

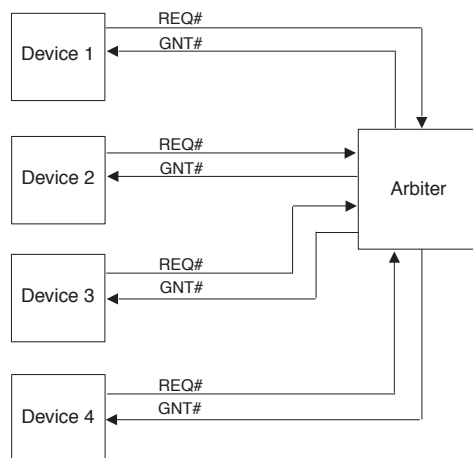


Figure 2-1: Arbitration.

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# deasserted.

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

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.

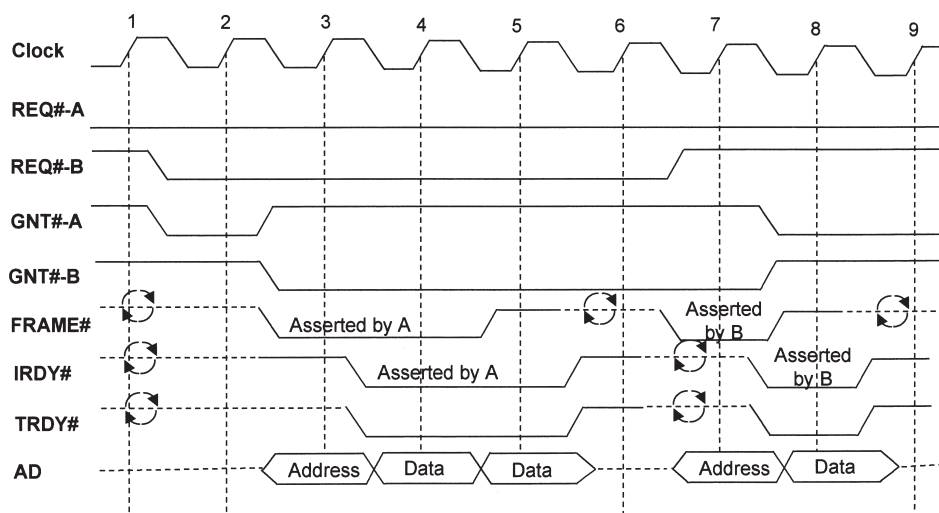


Figure 2-2: Arbitration timing.

Arbitration is “hidden,” meaning that arbitration for the next transaction occurs at the same time as, or in parallel with, the current transaction. This means that the arbitration process doesn’t take any additional 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, in fairness it must wait until Device B has executed its 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.

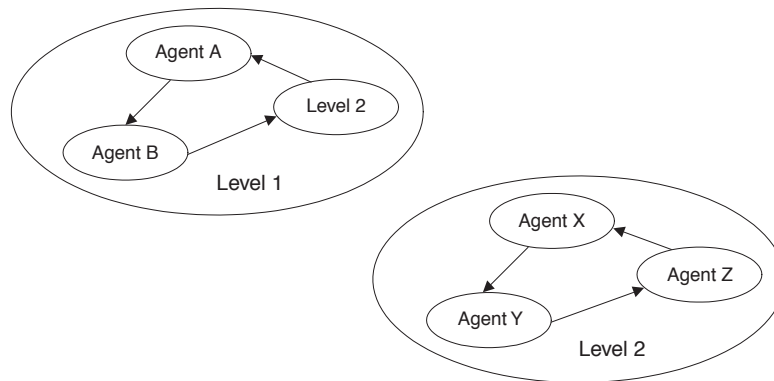


Figure 2-3: Example of fairness in arbitration.

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 Y 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.

The result is that every agent gets a chance at the bus. A higher priority agent can't "starve" a lower priority agent for bus cycles.

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.

The agent on which the bus is parked must drive its AD[31::0], C/BE#[3::0] and PAR signals.

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.

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 simultaneously 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 target(s) of the transaction 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.

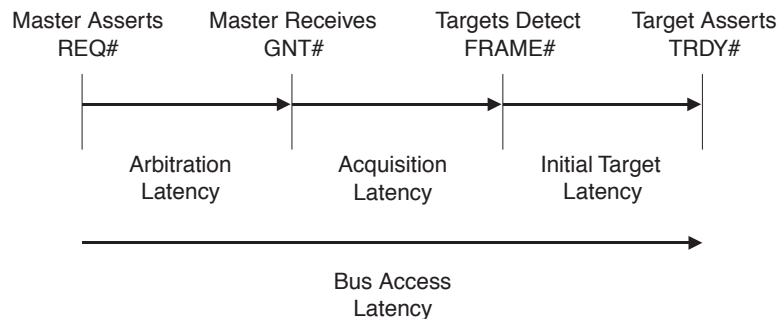


Figure 2-4: Bus access latency.

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#** signal 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.

The arbiter should leave the current master's **GNT#** asserted if no other **REQ#** signals are asserted. Thus, if no one else needs to use the bus, the current master may continue beyond its Latency Timer value.

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 8 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 (μs)
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

$$\text{Latency} = \text{total clocks} * 0.030 \mu\text{s}$$

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 transaction 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. Nevertheless, 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.

The next chapter will describe the basic bus protocol.

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,” meaning that reads from memory space have no “side effects.” In contrast, a read from I/O space may have the side effect of, for example, resetting the Data Ready bit in a status register. If such a register were prefetched but not actually read, data could be lost.

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 initiator intends to read most of, if not all of the full current cache line. The target may gain some performance advantage by knowing that it is expected to supply up to an entire cache line. When an initiator 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 as soon as possible.

Table 3-1: PCI command types.

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

Memory Write and Invalidate is semantically identical to Memory Write with the addition that the initiator 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 initiator 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 initiator 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 initiator 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.

This is also a *turnaround cycle*: In a read transaction, the initiator 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 initiator 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 initiator latches the data bus.
- 7 The initiator has deasserted **IRDY#** indicating that it is not ready for the next data element. This is another wait cycle.
- 8 The initiator 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 initiator 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 shows the details of a typical write transaction where data moves from the initiator 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 initiator can drive data onto the AD bus during clock 3.

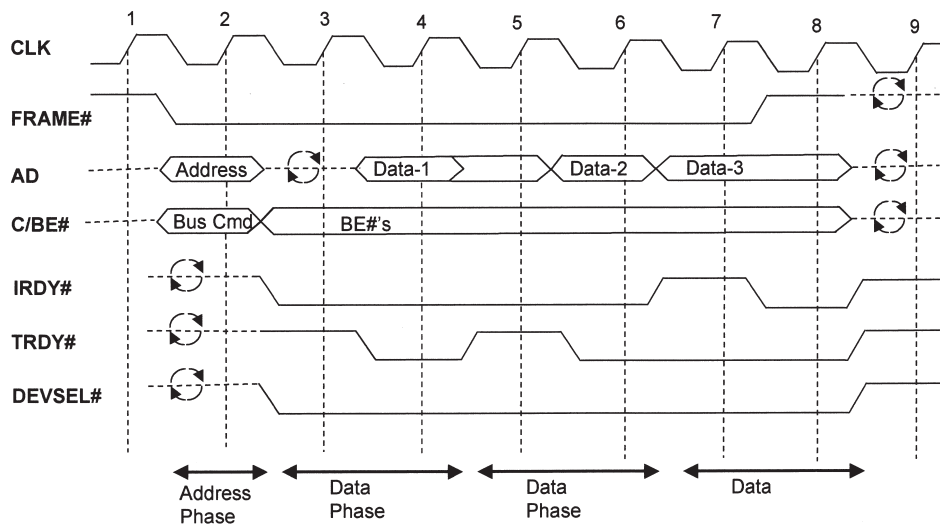


Figure 3-1: Timing diagram for a typical read transaction.

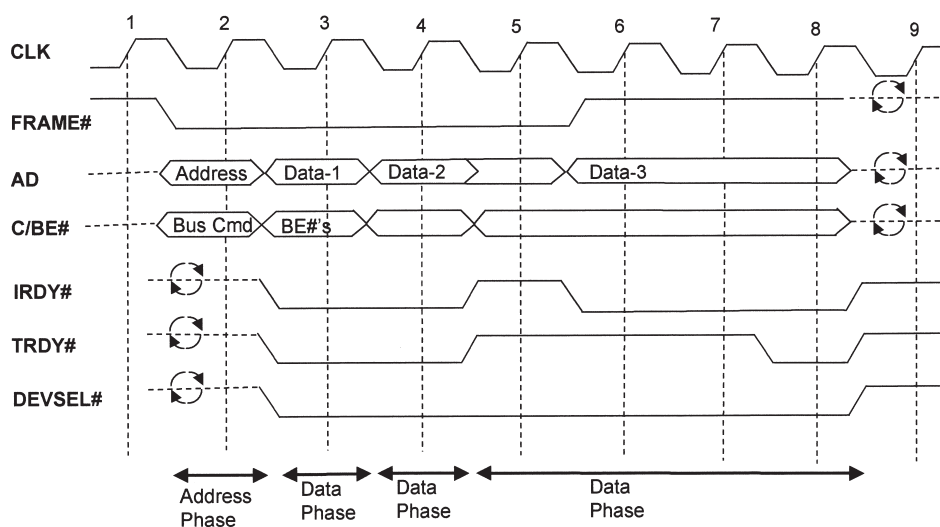


Figure 3-2: Timing diagram for a typical write transaction.

Byte Enable Usage

During the data phases of a transaction, the C/BE# signals indicate which *byte lanes* convey meaningful data. The initiator 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 initiator is free to use any contiguous or noncontiguous 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] is not needed during the address phase of a memory transaction and can be used for something else. Specifically, AD[1:0] indicates how the target should advance the address during a multidata 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.

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.

Table 3-2: AD[1:0] for memory transfers.

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.

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 for I/O transfers

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 initiator 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).

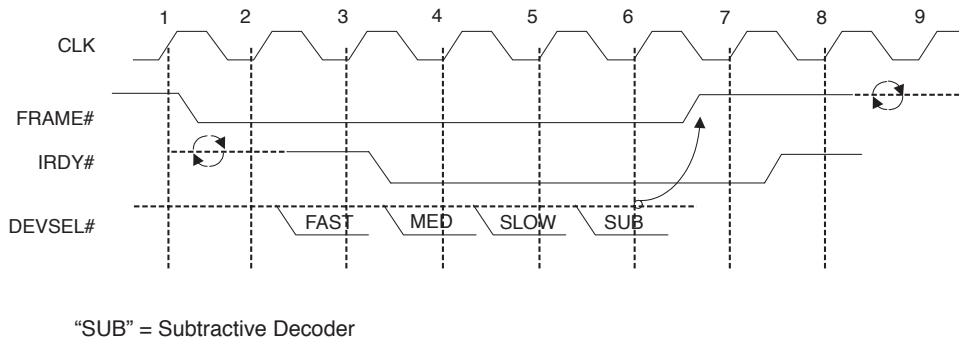


Figure 3-3: DEVSEL# timing.

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, and so forth. The strategy is that if no agent claims the transaction on this bus segment, then it's 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 initiator asserts **FRAME#** only when all four driver groups are on. Data can likewise be stepped. The example here is a write cycle so the initiator asserts **IRDY#** only when all four driver groups have switched to the current data item.

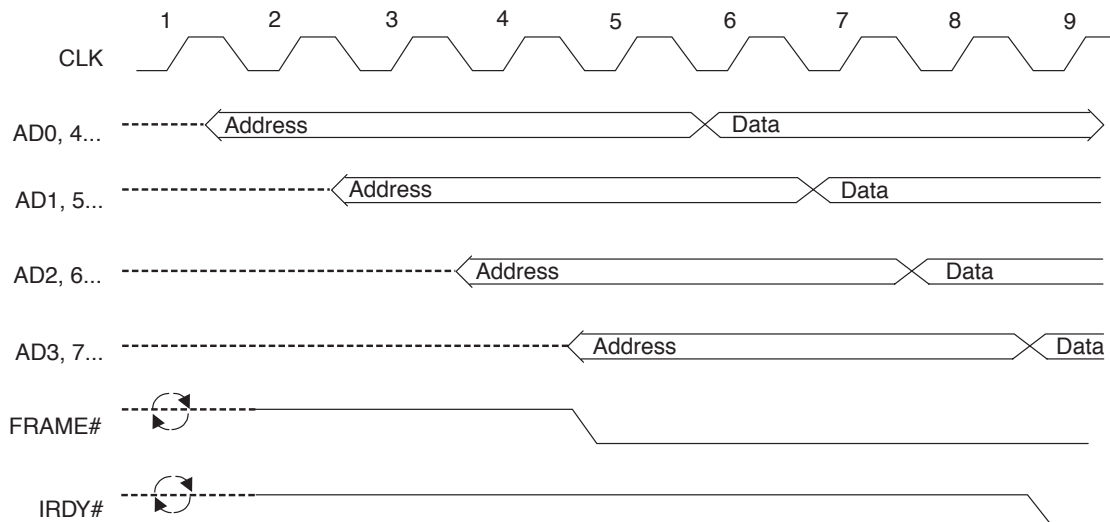


Figure 3-4: Address/Data stepping.

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

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, initiator 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 initiator 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 initiator 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.

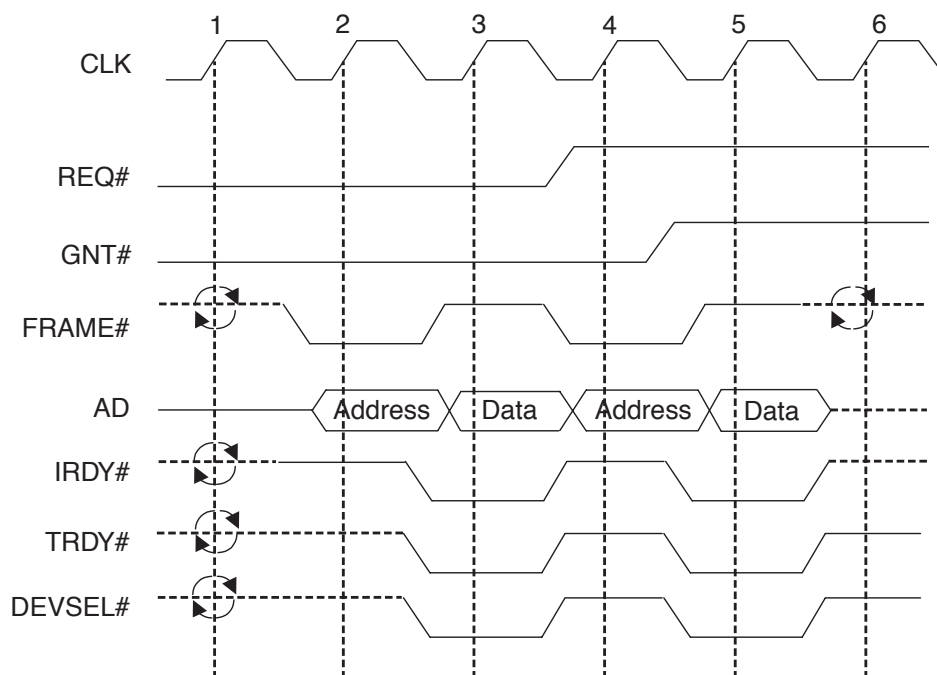


Figure 3-5: Fast back-to-back timing.

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—Initiator

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

Initiator Preempted

If another agent requests use of the bus and the current initiator’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 initiator 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 initiator deasserts **FRAME#** to terminate the transaction with no data transferred. In the case of a retry, the initiator is obligated to retry the exact same transaction at some time in the future.

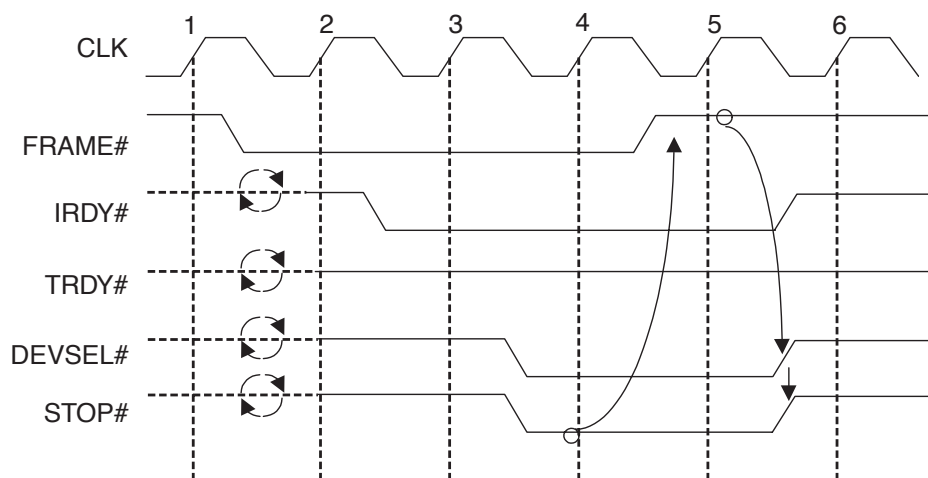


Figure 3-6: Target retry.

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 initiator and begins executing the transaction internally. This allows the bus to be used by other masters while the target is busy. 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 initiator 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 exact 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} clocks¹.

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 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 initiator 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 initiator 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.

¹ At 33 MHz, this works out to about two minutes.

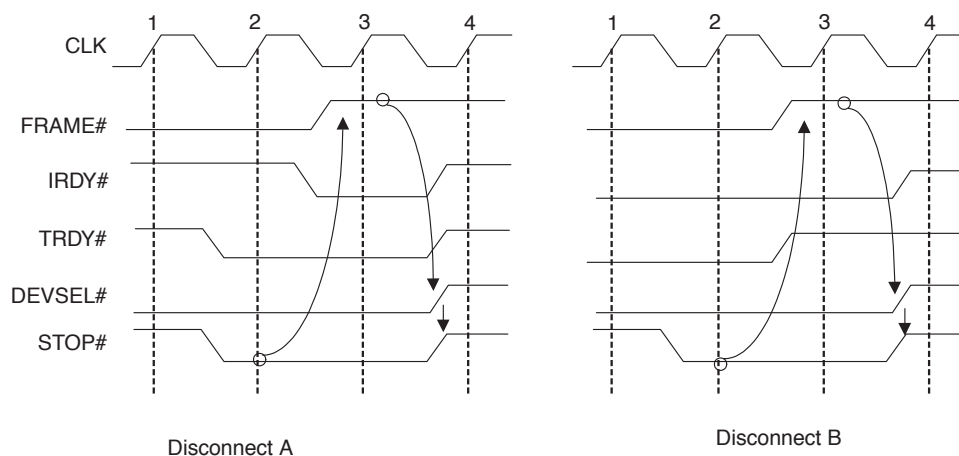


Figure 3-7: Target disconnect—with data.

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

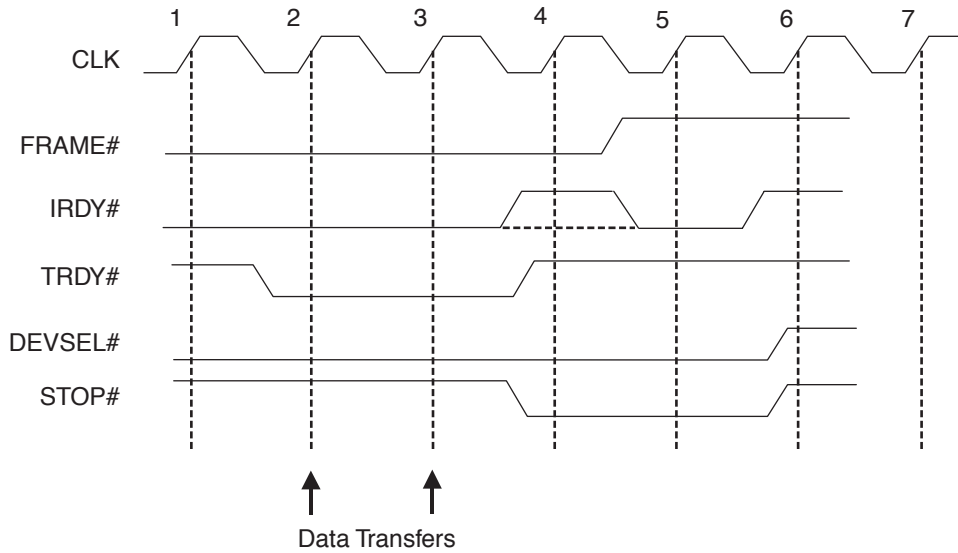


Figure 3-8: Target disconnect—without data.

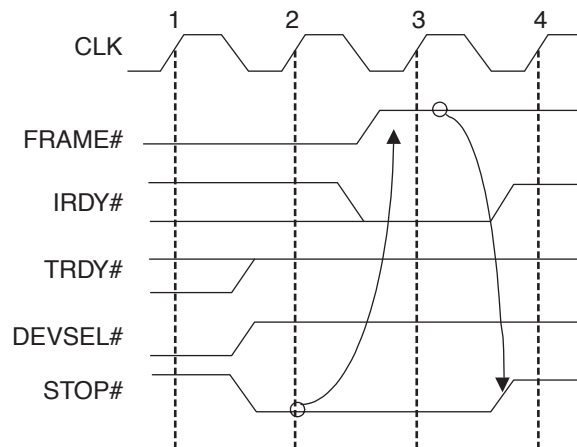


Figure 3-9: Target abort.

Error Detection and Reporting

Parity Generation and 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.” In other words, it’s not a big deal if a sound card drops a bit now and then.

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 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 initiator 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 Initiator 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 turn-around is required on either **AD** or **PAR**.

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

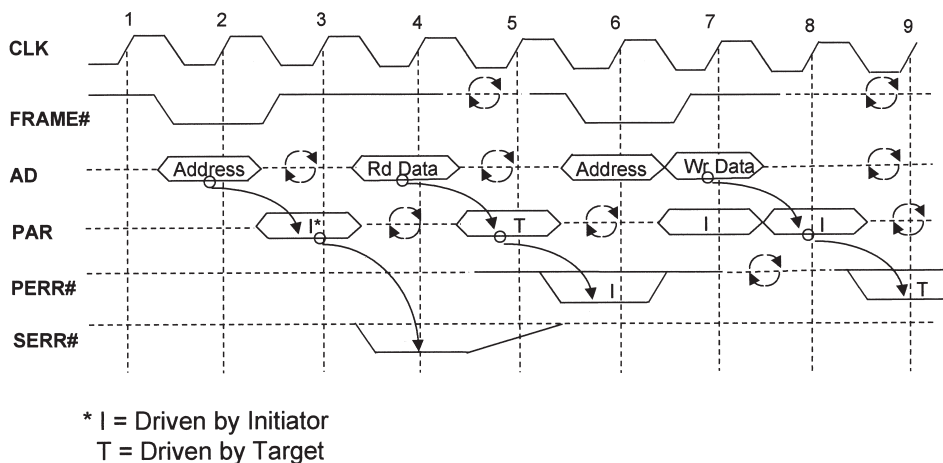


Figure 3-10: Timing diagram for parity generation and detection.

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 initiator/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

that detects a parity error during the address phase asserts **SERR#** and sets the **SIGNALED 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 okay
- Claim the transaction and terminate with Target-Abort
- Don’t claim the transaction and let the initiator 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 nonmaskable 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.

The next chapter describes advanced features of the PCI protocol.

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#. *Multifunction* 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 multifunction 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-low¹ 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

¹ By contrast, interrupts on the ISA bus are specified as assertion-high and edge-sensitive. As a result, it is very difficult to share interrupts on the ISA bus.

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.

Table 4-1: Suggested interrupt routing.

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

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.

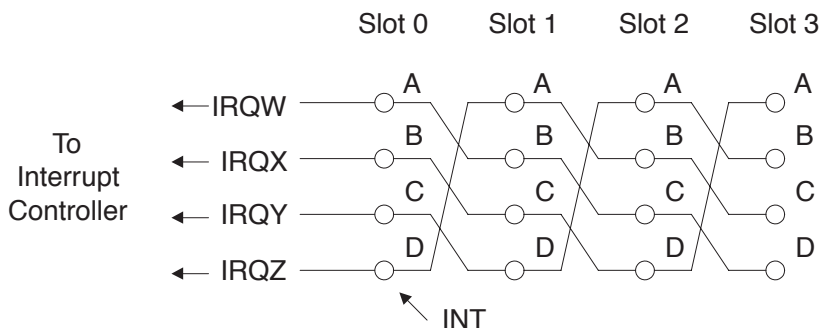


Figure 4-1: Suggested interrupt routing.

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. 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 in response to an interrupt 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 (Advanced Programmable Interrupt Controller).

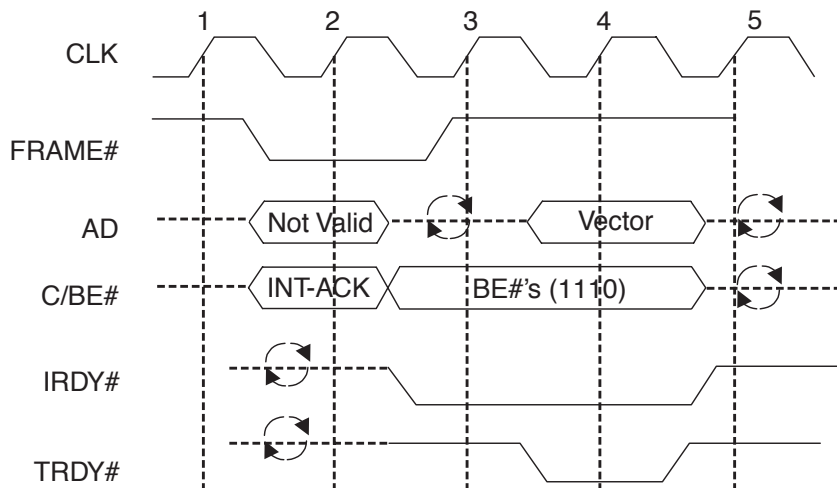


Figure 4-2: Interrupt acknowledge cycle.

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.

Note, incidentally, that it is not necessary to utilize the PCI bus to communicate the interrupt vector from the APIC to the processor. In most cases, this communication is handled within the chip set.

“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.

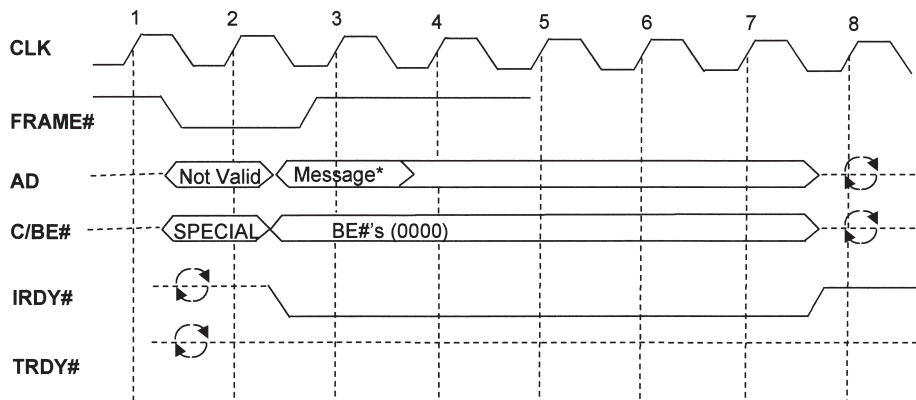
Figure 4-3 shows the timing of a Special Cycle.

Table 4-2: Defined message types.

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.

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 to 7 Master waits to time out with a Master Abort.



*Message must be latched on first cycle of IRDY

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

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 interoperate 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 4-5 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 8 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 4 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 at the beginning of the transaction.

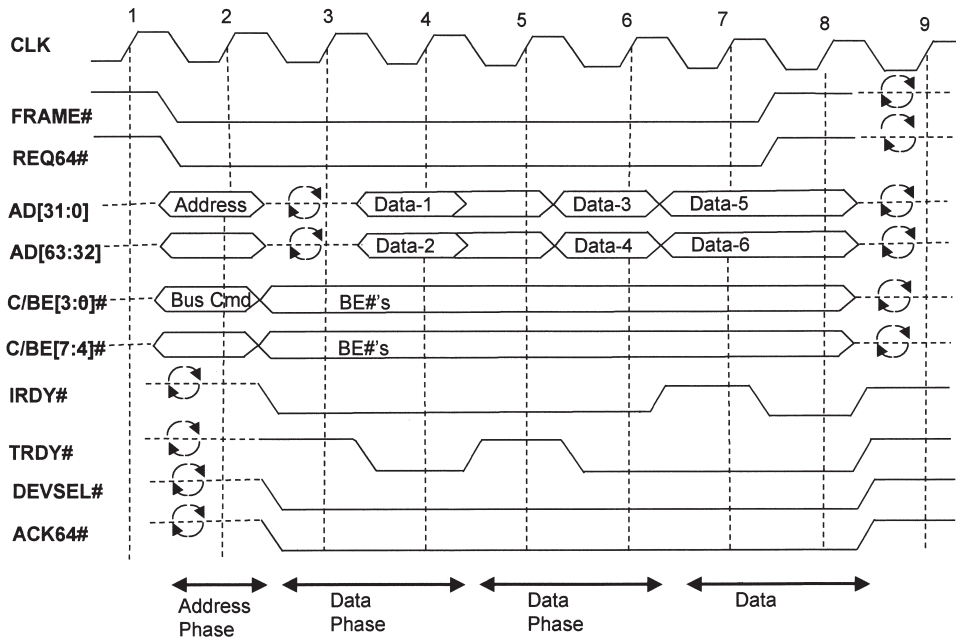


Figure 4-4: Timing diagram of a 64-bit transaction.

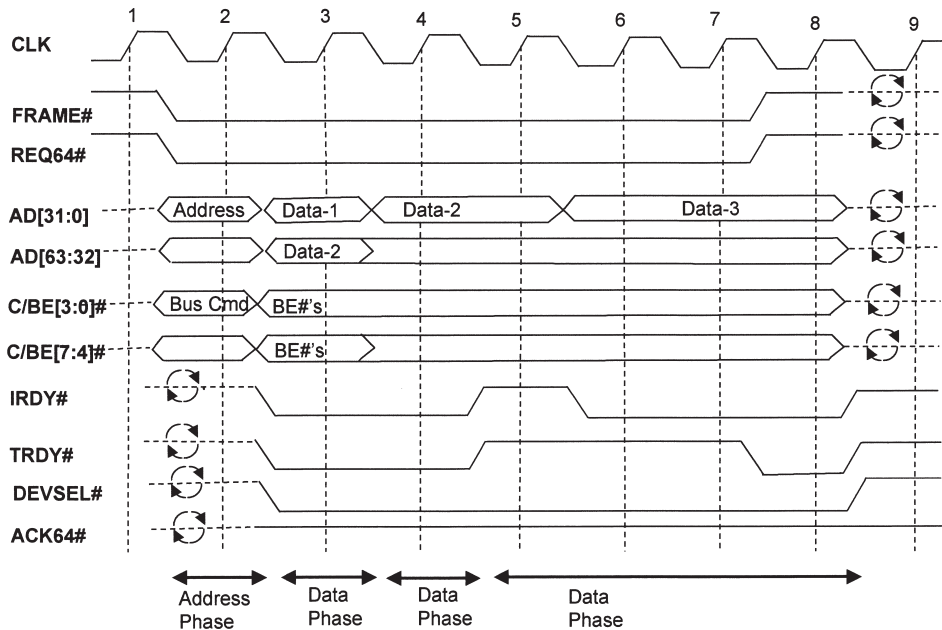


Figure 4-5: Execution of a write transaction from a 64-bit master to a 32-bit target.

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 (DAC) command. 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 Cycle 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].

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.

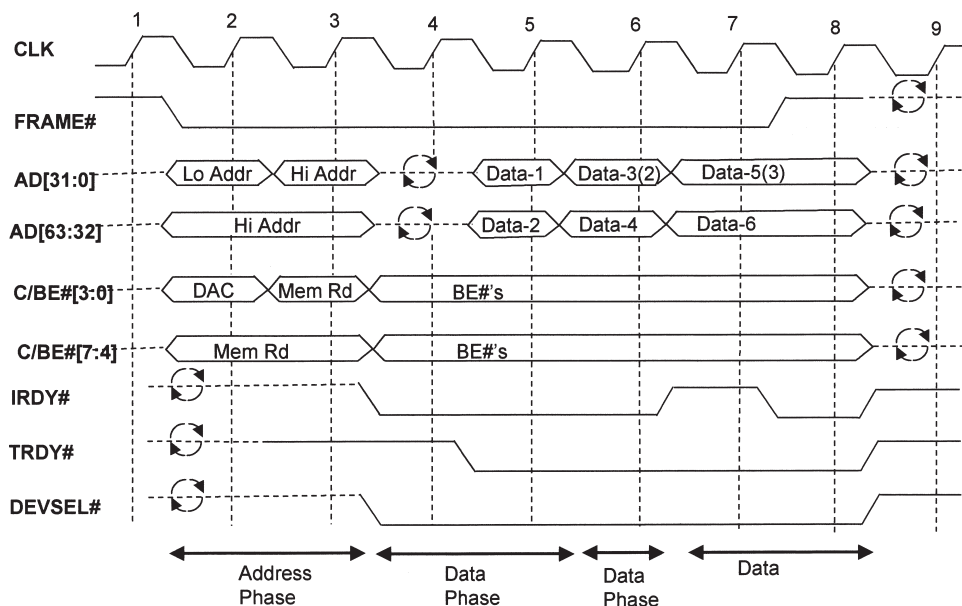


Figure 4-6: Dual Address command used in 64-bit addressing.

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

System Management Bus (SMB)

Revision 3.0 of the PCI specification assigns a pair of pins, **SMBCLK** and **SMBDAT**, to support the System Management Bus (SMB). This is a bit-serial bus based on I²C that provides a mechanism to communicate various forms of management information among devices on the bus. SMB is another mechanism for addressing the requirement for sideband signals that aren't part of the PCI specification.

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#**. Multifunction 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 Cycle command (DAC) provides a way for 32-bit agents to access a 64-bit memory address space.

The next chapter describes the electrical signaling environment of PCI, and mechanical issues related to add-in connectors.

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, such as the Unibus and Qbus, 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.

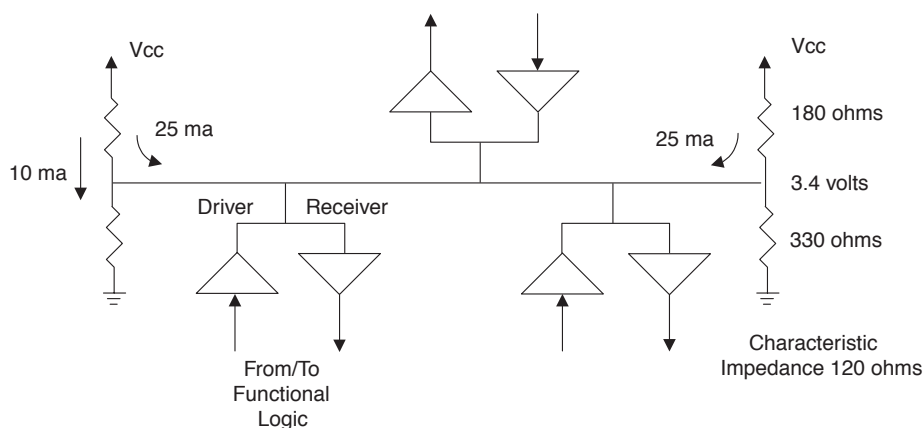


Figure 5-1: “Traditional” bus—incident wave switching.

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! And that’s just for 32 bits. A 64-bit bus would almost double that requirement.

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.

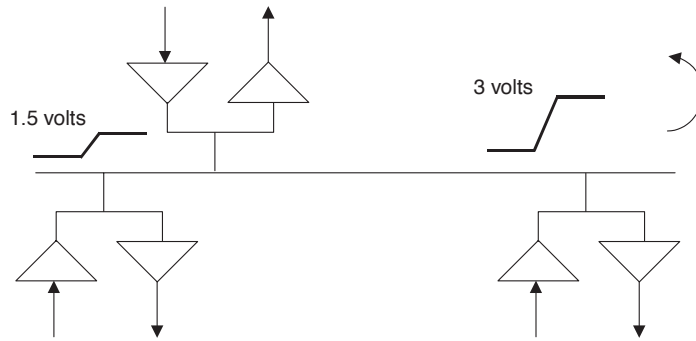


Figure 5-2: Reflected wave switching.

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 K ohm in the 5V signaling environment and 8.2 K ohm in the 3V signaling environment.

Signaling Environments—3V and 5V

Historically, most computer busses have used 5 volt TTL-compatible signaling levels. However, the need for lower power consumption and higher speed is driving a trend toward 3.3 volt logic, particularly in portable and mobile environments. Unfortunately, these two logic families don't mix well together, so PCI has developed separate electrical specifications for each signaling environment. The spec specifically says that the signaling environments cannot be mixed. All devices on a given PCI bus segment must use the same 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 components on the board. A component designed to operate off of 3.3V power may be capable of operating in a 5V signaling environment and vice-versa.

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 “V_{io}.” A universal board powers its PCI transceivers from the V_{io} pins. The motherboard connects the V_{io} pins to the power rail corresponding to the system's signaling environment.

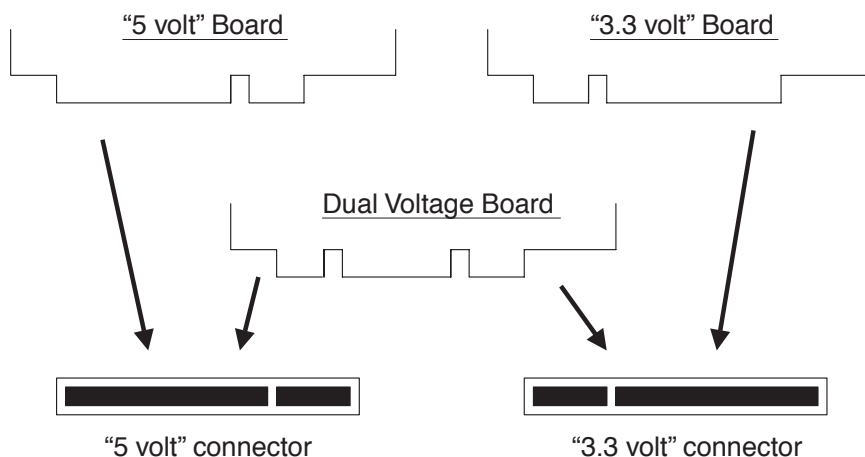


Figure 5-3: 3.3V vs. 5V keying.

It has been the intention of the PCI-SIG since at least Revision 2.2 to migrate the PCI bus to a 3.3V only environment. With Revision 3.0, this migration is complete. The current spec removes support for the 5V-only expansion board and removes the 5V key in the motherboard connector. The V_{io} pins are now designated +3.3V (I/O). Nevertheless, for historical completeness, we'll describe both the 5V and 3.3V environments.

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 Revision 3.0 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 bidirectional buffers with tri-state outputs.
2. Signals without pull-up resistors must have 3 mA low output current. Signals requiring pull-up must have 6 mA; the latter include: FRAME#, TRDY#, IRDY#, DEVSEL#, STOP#, SERR#, PERR#, LOCK#, and, when used, AD[63::32], C/BE[7::4]#, PAR64, REQ64#, and ACK64#.
3. Absolute maximum pin capacitance for a PCI input is 10 pF (except for CLK, SMBDAT and SMBCLK) 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, and so forth).
4. Lower capacitance on this input-only pin allows for nonresistive coupling to AD[xx].
5. This is a recommendation, not an absolute requirement. The actual value should be provided with the component data sheet.

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

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.

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

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 maximums should be met by design. In order to facilitate component testing, a maximum current test point is defined for each side of the output driver.
4. This parameter is to be interpreted as the cumulative edge rate across the specified range, rather than the instantaneous rate at any point within the transition range. The specified load (Figure 5-5) is optional; i.e., the designer may elect to meet this parameter with an unloaded output. However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline).

There are three interesting points on the V/I curves of Figure 5-4:

- *DC Drive Point:* This is the steady state operating point. Note that the current requirements in steady state are quite modest. The voltage at the DC drive points comes from V_{oh} and V_{ol} in Table 5-1 above.
- *AC Drive Point:* Defines the minimum instantaneous current curve required to switch the bus line with a single reflection. This defines the driver's maximum drive requirement. From the DC steady state, the current at the AC drive point must be reached within the output delay time, T_{val} (see Table 5-7).
- *Test Point:* Defines the maximum allowable instantaneous current curve in order to limit switching noise and avoid signal integrity problems.

3.3 Volt Signaling Environment

The 3.3 volt environment is based on V_{cc} -relative switching voltages. The key voltages are specified as fractions of the supply voltage rather than as absolute numbers. This approach is optimized for a CMOS environment. The intent is that components connect directly together, whether on the motherboard or an expansion board, without any external buffers or other “glue.”

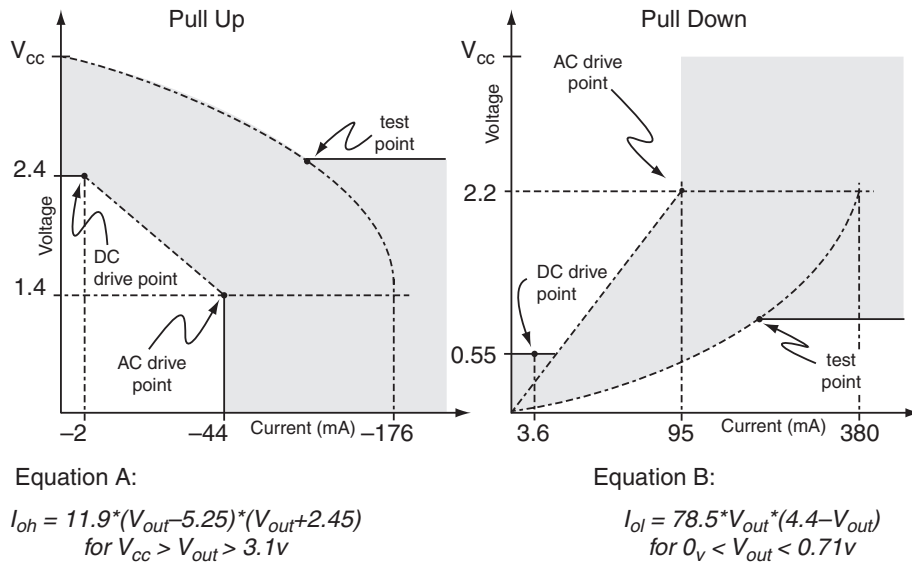


Figure 5-4: Characteristic V/I curves for a PCI driver in the 5V signaling environment.

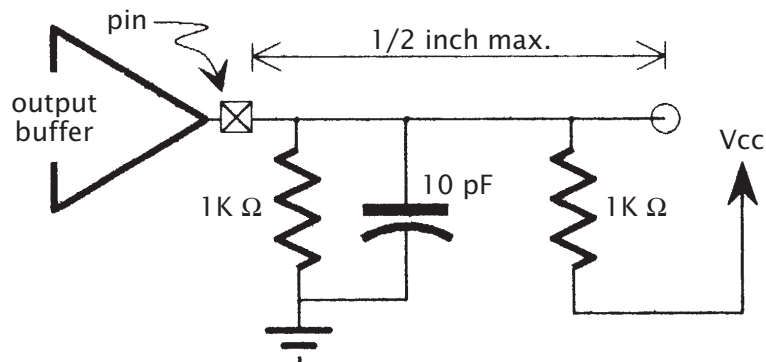


Figure 5-5: Specified load for output rise and fall slew rate measurements.

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 bidirectional buffers with tri-state outputs.

3. Absolute maximum pin capacitance for a PCI input is 10 pF (except for CLK, SMBDAT and SMBCLK) with an exception granted to motherboard-only devices 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, and so forth).
4. Lower capacitance on this input-only pin allows for nonresistive 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 5-5 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 specification does not apply to CLK and RST# which are system outputs. “Switching Current High” specifications are not relevant to SERR#, PME#, INTA#, INTB#, INTC#, and INTD# which are open drain outputs.
2. Maximum current requirements must be met as drivers pull beyond the first step voltage. Equations defining these maximums (C and D) are provided with the respective diagrams in Figure 5-6. The equation-defined maximums should be met by design. In order to facilitate component testing, a maximum current test point is defined for each side of the output driver.
3. This parameter is to be interpreted as the cumulative edge rate across the specified range, rather than the instantaneous rate at any point within the transition range. The specified load is optional (see Figure 5-5); i.e., the designer may elect to meet this parameter with an unloaded output. However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline). Rise slew rate does not apply to open drain outputs.

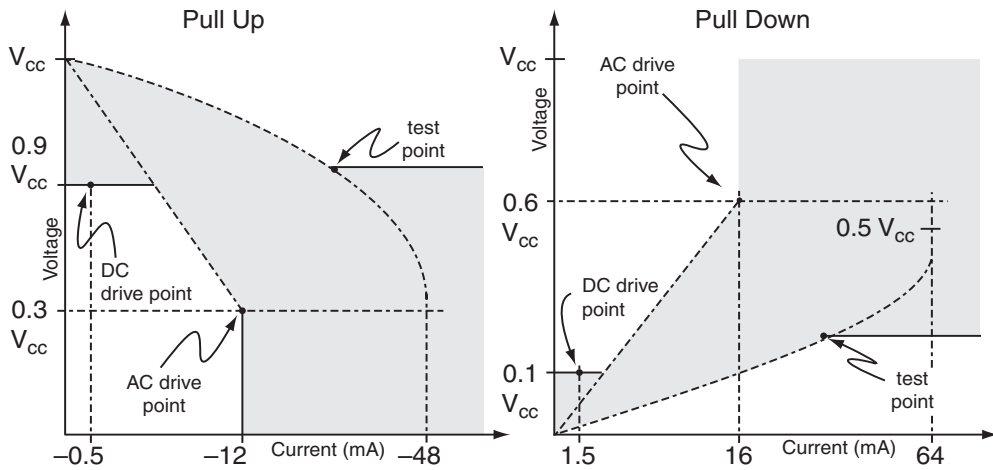
Table 5-4: AC specifications for 3.3V signaling.

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

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.



Equation C:

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

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

Equation D:

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

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

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

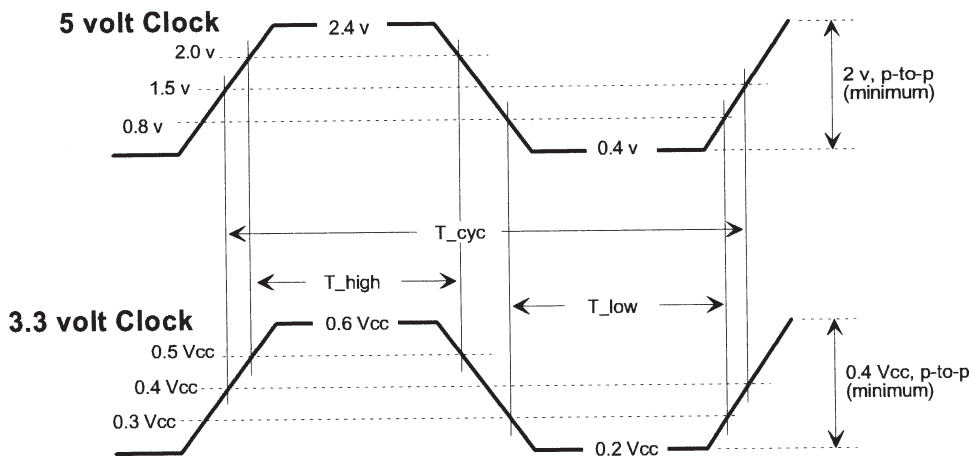


Figure 5-7: Clock waveform and required measurement points.

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:
 - a. 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

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	
T_{pvrh}	Power valid to RST# high	100		ms	

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.

For parts compliant to the 3.3V signaling environment:

- b. 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# has a setup of 10; REQ# has a setup of 12. All other signals are bussed.
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.

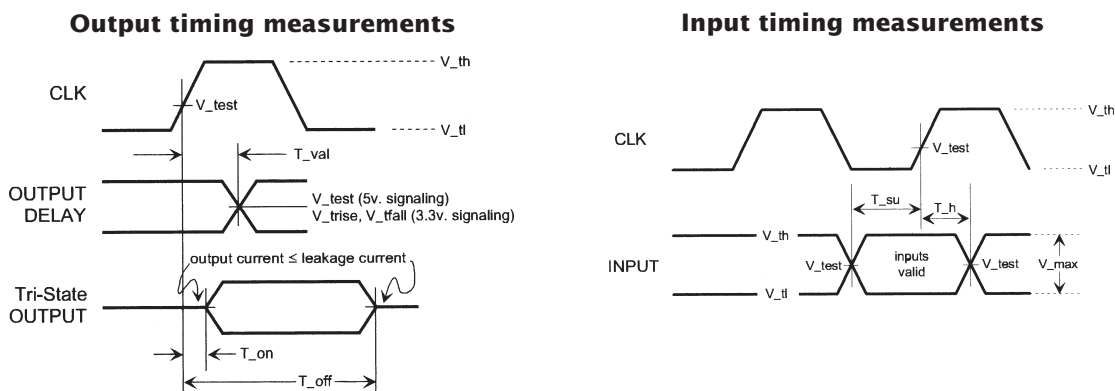


Figure 5-8: Input and output timing measurement conditions.

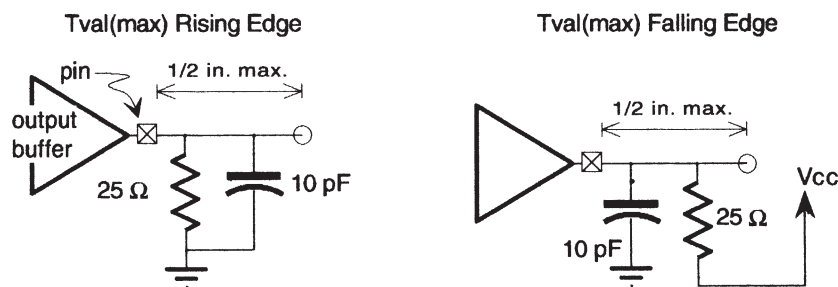


Figure 5-9: Load circuits for 3.3V slew measurements.

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	

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. Although 66 MHz is still defined and supported in PCI, the preferred alternative for clock rates above 33 MHz is PCI-X (see Chapter 11 and beyond). 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.

AC Specifications

Table 5-8 shows the AC specifications for PCI 66. While 33 MHz PCI bus drivers are defined by their V/I curves, 66 MHz PCI output buffers are specified in terms of their AC and DC drive points, timing parameters, and slew rate. The minimum AC drive

point defines an acceptable first step voltage and must be reached within the maximum T_{val} time. The maximum AC drive point limits the amount of overshoot and undershoot in the system. The DC drive point specifies steady state conditions. The minimum slew rate and the timing parameters guarantee 66 MHz operation. The maximum slew rate minimizes system noise. This method of specification provides a more concise definition for the output buffer.

Table 5-8: PCI 66 AC specifications.

Symbol	Parameter	Condition	Min	Max	Units	Notes
AC Drive Points						
$I_{oh(AC)}$	Switching Current High	$V_{out} = 0.7V_{cc}$		$-32V_{cc}$	mA	
		$V_{out} = 0.3V_{cc}$	$-12V_{cc}$		mA	1
$I_{ol(AC)}$	Switching Current Low	$V_{out} = 0.18V_{cc}$		$38V_{cc}$	mA	
		$V_{out} = 0.6V_{cc}$	$16V_{cc}$		mA	1
DC Drive Points						
V_{oh}	Output High Voltage	$I_{out} = -500\text{ }\mu\text{A}$	$0.9V_{cc}$		V	2
V_{ol}	Output Low Voltage	$I_{out} = 1500\text{ }\mu\text{A}$		$0.1V_{cc}$	V	2
Clamp Currents						
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 Rates						
$slew_r$	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

Notes for Table 5-8

1. In conventional PCI 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. In PCI-X devices, REQ# and GNT# must have full size drivers. This specification does not apply to CLK and RST# which are system outputs.

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

2. These DC values come from Table 5-3 and are included for completeness.
3. This parameter is to be interpreted as the cumulative edge rate across the specified range, rather than the instantaneous rate at any point within the transition range. The specified load (Figure 5-5) is optional; i.e., the designer may elect to meet this parameter with an unloaded output. However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline). Rise slew rate does not apply to open drain outputs.

Clock Specification

Table 5-9 shows the clock specifications for 66 MHz operation. Not surprisingly, the numbers are roughly 1/2 the same values for 33 MHz operation as shown in Table 5-5.

Table 5-9: 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
Spread Spectrum					
f_{mod}	Modulation frequency	30	33	kHz	
f_{spread}	Modulation spread	–1	0	%	

Notes for Table 5-9

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.

Spread-Spectrum Clocking

Spread spectrum clocking is a technique for reducing unwanted, in some cases unlawful, EMI. If a clock signal is oscillating at a single frequency, then all of the radiated energy appears at that frequency. Suppose instead we modulate the clock frequency over a relatively small range, say a few percent. The total radiated energy remains the same but now it is spread over the range of frequencies encompassed by the modulation. So the radiated energy at any specific frequency is lower.

66 MHz PCI supports spread-spectrum clocking as shown by the last two rows of Table 5-9. Modulation frequency refers to how rapidly the clock frequency changes. Modulation spread defines the maximum permitted frequency deviation. So, for example, at 66 MHz the clock frequency may vary between 65.340 MHz and 66.0 MHz, and this variation may occur at a rate between 30 and 33 KHz. Note that the modulation spread is not allowed to exceed 66 MHz as reinforced by Note 4.

Timing Parameters

Table 5-10 shows those timing parameters that change from 33 MHz to 66 MHz.

Table 5-10: 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 5-10

1. See the output timing measurement conditions in Figure 5-8.
2. Minimum times are evaluated with the 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 bussed.
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.

The timing relationship between 33 MHz and 66 MHz is illustrated graphically in Figure 5-10.

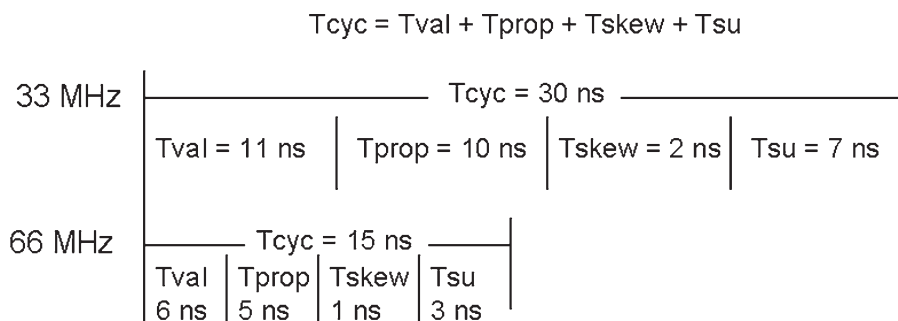


Figure 5-10: Timing relationship between 33 MHz and 66 MHz.

Mechanical Details

Connector

PCI expansion cards utilize a connector derived from the connector used by IBM's Micro Channel® (see Figure 5-11). 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 cards¹.

The 64-bit extension, built into the same connector molding, extends the total number of pins to 184 as shown in Figure 5-12. Note that the 64-bit connector requires two different implementations to accommodate signaling environment keying.

The connector pinout is given in Appendix B.

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. Three basic form factors are defined: standard length, short length and low profile.

¹ But remember that the 5V signaling environment is no longer supported.

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 $2.5" \pm 0.1"$.

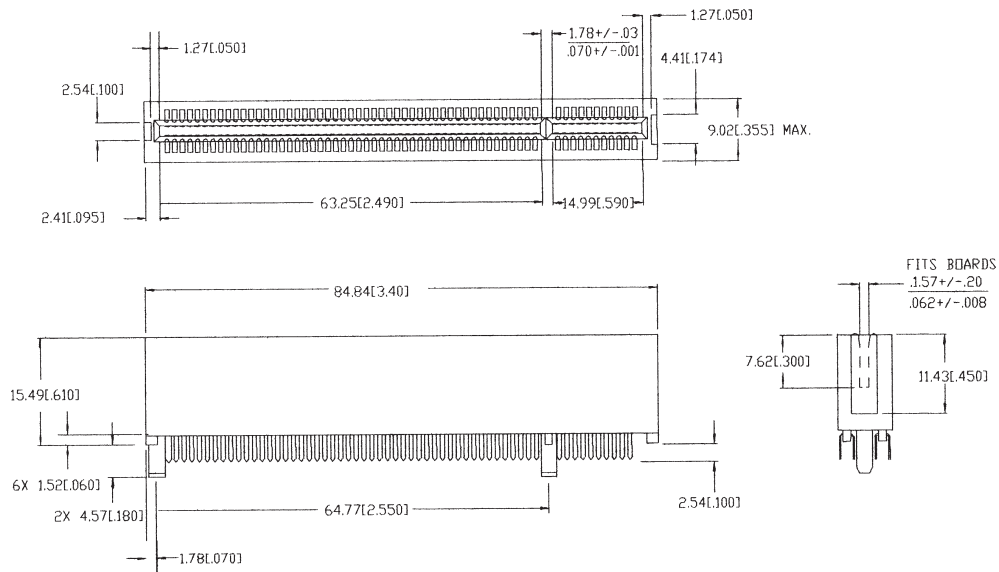


Figure 5-11: 32-bit PCI expansion card connector.

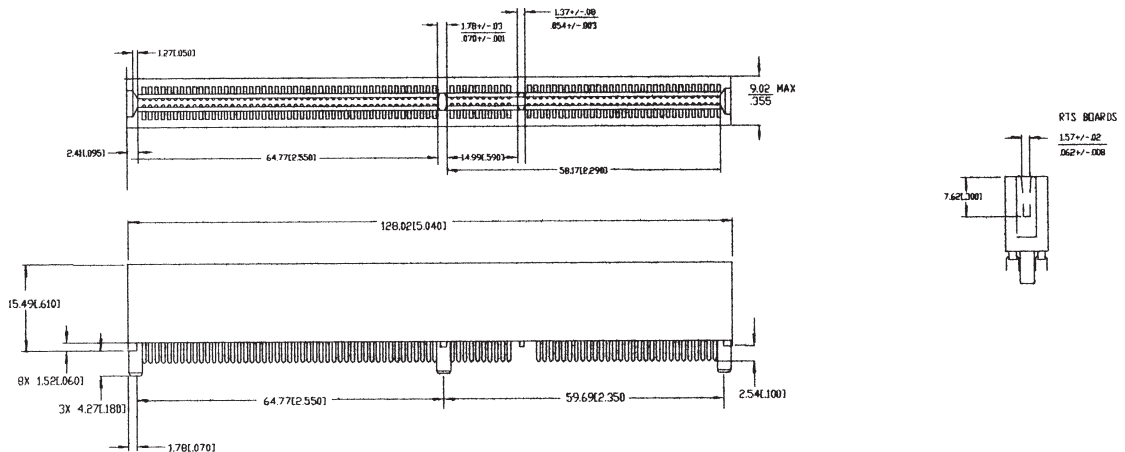


Figure 5-12: 64-bit PCI expansion card connector.

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 5-13. This contributes to shorter, more consistent stub lengths.

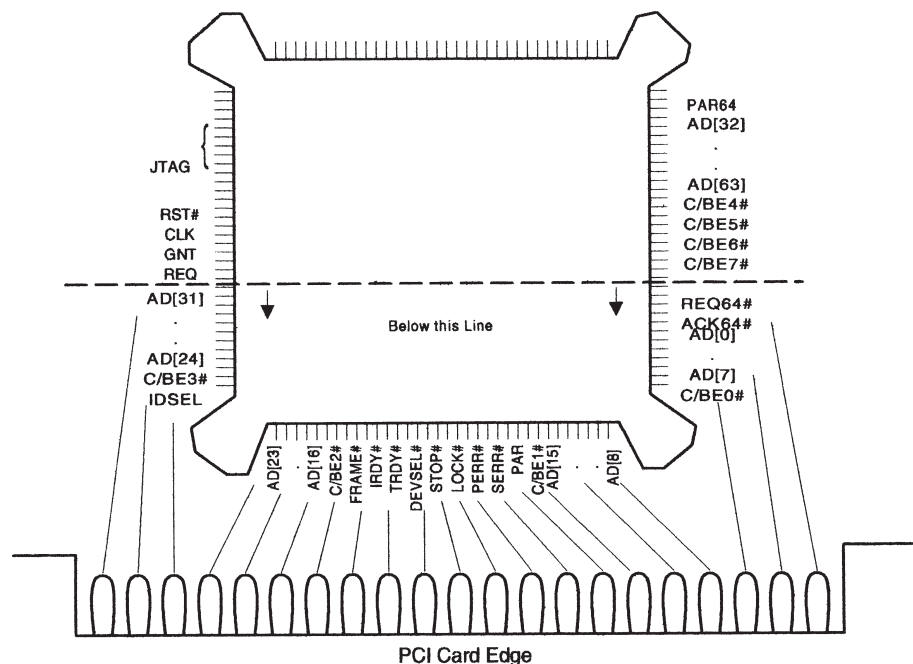


Figure 5-13: Suggested pinout for PQFP PCI component.

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. The specification provides 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.

The next chapter covers Plug-and-Play configuration.

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,” system 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. Your typical Silicon Valley geek can cope with these problems, but Aunt Martha, who only wants to surf the Web, definitely can't. Just as the computer bus solved the problem of mass marketing minicomputers, Plug-and-Play configuration helped turn the PC into a consumer product.

In the PC world, various device types like 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.

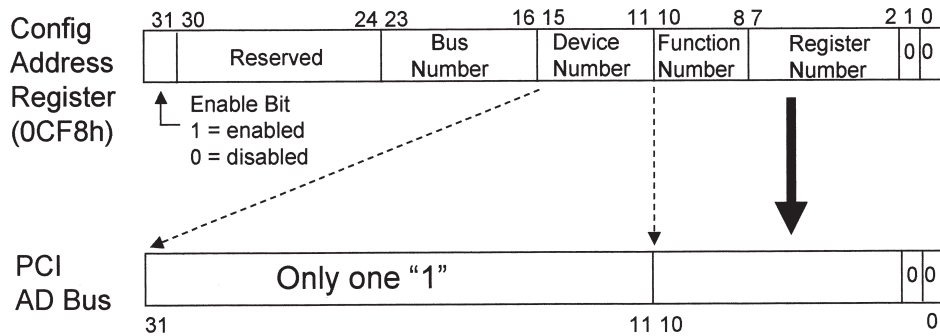


Figure 6-1: x86 Configuration Address.

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 PCI bus connector then has its IDSEL pin connected to exactly one of AD[31:11] as shown in Figure 6-2. Thus, when accessing Configuration Space, a device is selected by its physical location.

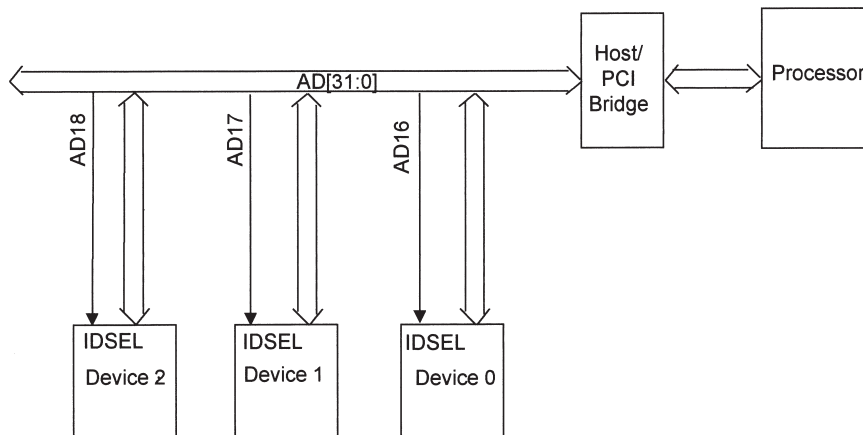


Figure 6-2: Asserting IDSEL.

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.

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.

Header Type (0xE)

The value of this byte defines the layout of the predefined header starting at byte 10h. Currently, three different header types are defined: 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 multifunction device.

Identification Registers

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

- *Vendor ID (0x0)*: 16 bits. Identifies the vendor of the device. More specifically, it identifies the vendor of the PCI silicon. Vendor ID codes are assigned by the PCI SIG.
- *Device ID (0x2)*: 16 bits. Identifies the device. This value is assigned by the vendor.
- *Revision ID (0x8)*: 8 bits. 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 (0x2C)*: 16 bits. Identifies the vendor of a functional PCI device. This is where board vendors can distinguish themselves from PCI silicon vendors.
- *Subsystem Device ID (0x2E)*: 16 bits. Assigned by the vendor to identify a functional PCI device. Can also be used to identify individual functions in a multifunction device.

The Class Code (0x9) 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, and so forth. A network controller can be Ethernet, token ring and so forth.
- *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.

Appendix A gives a complete list of the currently defined Class Codes.

Command Register (0x4)

The 16-bit read/writable Command Register provides coarse control over a device's ability to generate and respond to PCI cycles.

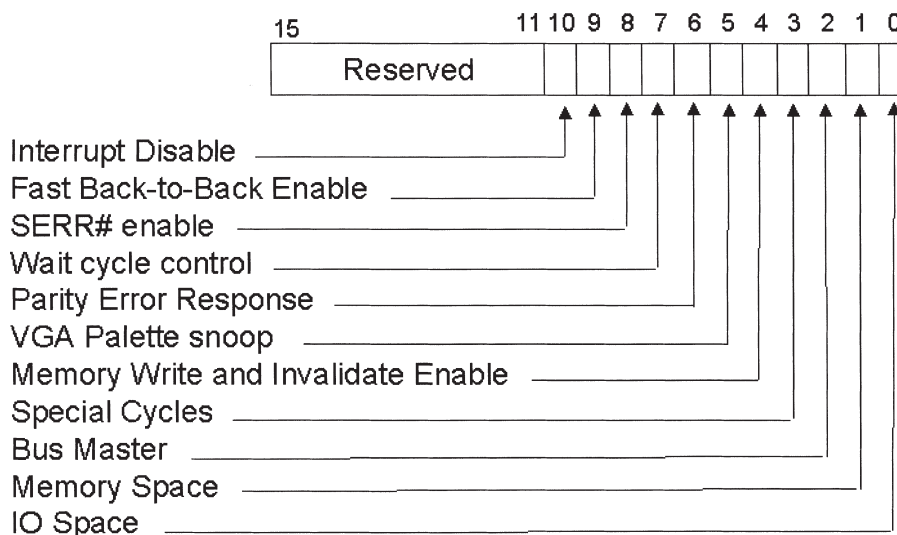


Figure 6-4: Configuration Command register.

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.
- 10 When 1, prevents the device from asserting an INTx# signal. When 0, the device is allowed to assert INTx#.

Note that writing all zeros to this register effectively disconnects the device from the PCI bus for all accesses except configuration cycles. The state of the register following RST# is all zero. Not all bits are required to be implemented depending on the device's functionality. For example, a device that doesn't use I/O space need not implement bit 0 and will return 0 in this bit when the command register is read.

Status Register (0x6)

The 16-bit 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.

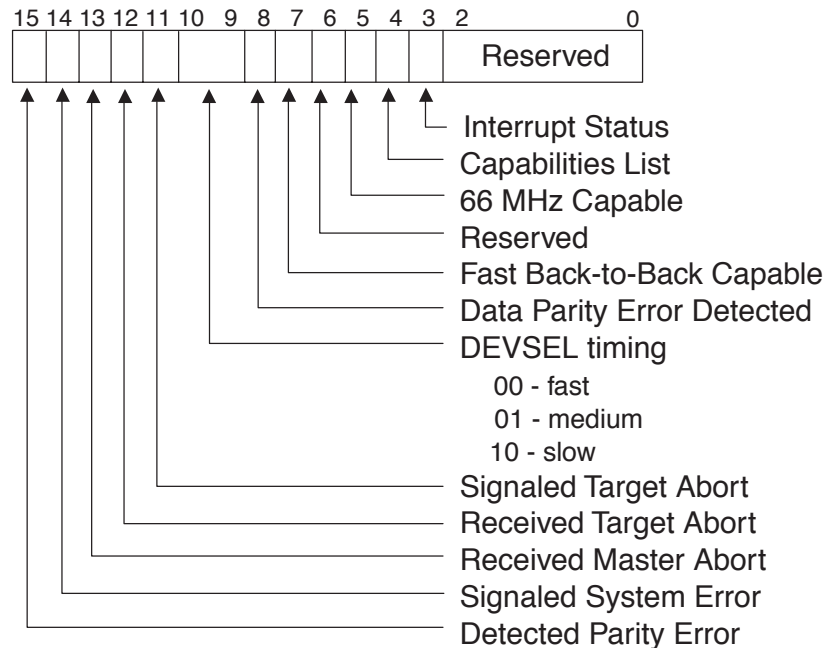


Figure 6-5: Configuration Status register.

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

- 3 RO. 1 = device has asserted its interrupt source. If Command Register bit 10 is 0, assert the appropriate INTx signal. Setting Command Register bit 10 to 1 does not change the state of this bit.
- 4 RO. 1 = extended capabilities pointer exists.
- 5 RO. 1 = device is capable of 66 MHz operation.
- 6 RO. Reserved. This bit was previously used to indicate that the device supported “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) (0xF)

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.

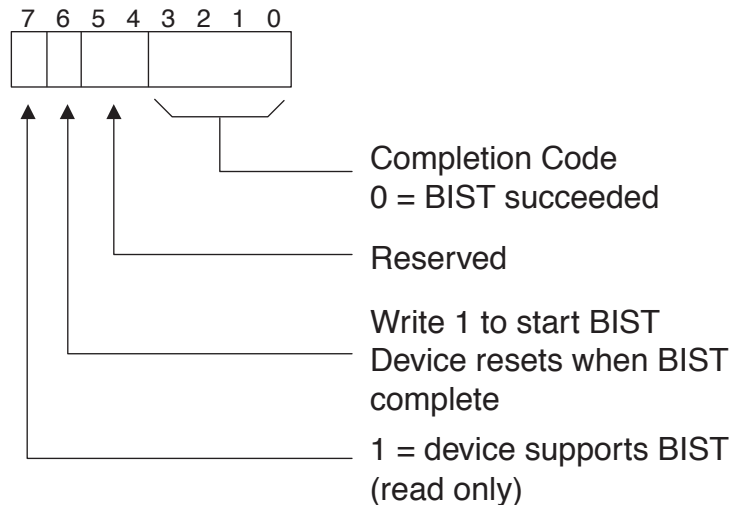


Figure 6-6: Built-in Self-test (BIST) register.

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 (0xD)

The Latency Timer is required to 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. RST# sets the Latency Timer to zero.

Recall from Chapter 2 that the value in the Latency Timer register is transferred to a down counter when a master asserts **FRAME#** to claim the bus. The counter is decremented by the bus clock. If the counter reaches zero before the transaction completes AND the device's **GNT#** signal is deasserted, it must terminate the transaction to allow another device to access the bus.

The Latency Timer may be read-only (and really doesn't mean much anyway) if the master never bursts more than two data phases.

Cache-Line Size (0xC)

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. The specification suggests that masters that implement the advanced read commands, Memory Read Line and Memory Read Multiple, should take advantage of this register to optimize their use of the read commands. **RST#** sets this register to zero.

Cardbus CIS Pointer (0x28)

Optional, 32 bits. 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 2.10 of the PCMCIA specification.

Capabilities Pointer (0x34)

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) (0x3F)

The specification says that this optional read-only 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.

If the device has no pressing requirement for maximum latency, it should return 0 in this register.

Min_Gnt (Minimum Grant) (0x3E)

This optional read-only register indicates how long the master would like to retain bus ownership when it initiates a transaction. Like Max_Lat, values of Min_Gnt are in increments of 250 ns or eight clocks at 33 MHz. If the device has no minimum grant requirement it should return 0.

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

Example of setting MIN_GNT and MAX_LAT

Consider a 100 Mbps fast Ethernet controller with a 64-byte buffer for each transfer direction. The optimal use of these buffers is to treat each one as two 32-byte ping-pong buffers. Each 32-byte buffer has eight DWORDs of data to transfer, requiring eight data phases on the PCI bus. Eight data phases translate to roughly 250 nanoseconds at 33 MHz, so the MIN_GNT value for this device is 1.

At roughly 10 MB/sec, the device will need to empty or fill a 32-byte buffer every 3.2 microseconds. This corresponds to a MAX_LAT value of 12 ($12 * 0.25 = 3$ microseconds).

Interrupt Pin (0x3D)

This read-only register tells which interrupt pin the device/function uses. The values are:

0. Device/function doesn't use an interrupt pin.
1. INTA#
2. INTB#
3. INTC#
4. INTD#

Recall from Chapter 1 that a single-function device is limited to using INTA#. Multi-function devices may use any combination of INTx signals.

Interrupt Line (0x3C)

This read/write register is used to communicate interrupt routing information. During configuration, POST software writes a value into this register that tells which input of the system's interrupt controller the device's interrupt pin is actually connected to. The device itself doesn't use this information. It's stored here for use by the device driver to determine which interrupt vector to attach to.

Values in this register are architecture-specific. For x86-based PCs, the values correspond to IRQ numbers 0 to 15 of the standard dual 8259 configuration. The value FFh means "unknown" or "no connection." Other values are reserved.

Base Address Registers (BAR) (0x10 to 0x24)

The Base Address Registers are the very essence of Plug-and-Play configurability. They 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, nonconflicting 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 Address 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 of x86 processors. Although this is no longer supported, "System software should recognize this encoding and handle appropriately." Bit 3 identifies prefetchable memory.

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

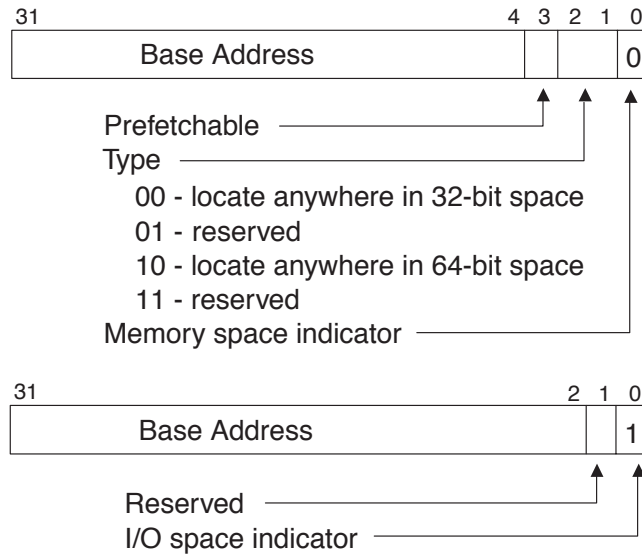


Figure 6-7: Base Address register.

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	0xFFFF00008
3. Mask off the lower four read-only bits	0xFFFF00000
4. Take the 1's complement	0x000FFFFF
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. First, in the PC architecture, I/O space is limited and highly fragmented making it potentially difficult to allocate I/O space. Second, I/O space is assumed to have read side effects and is thus 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.” What this means is 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 (0x30)

Expansion ROM offers the opportunity to embed device-specific initialization and run-time code directly on the board. 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. Each image may, for example, contain the same code but for a different processor

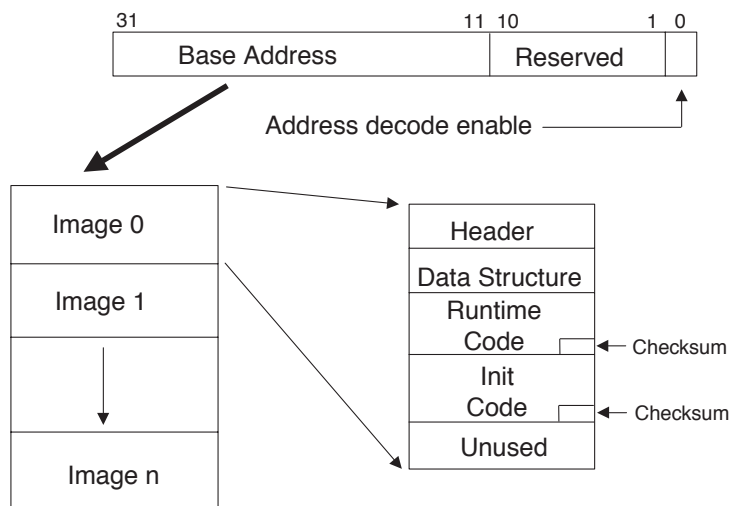


Figure 6-8: Expansion ROM Base Address register.

architecture. 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 portion of the code can be discarded after it is executed, thus reducing memory requirements.

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.

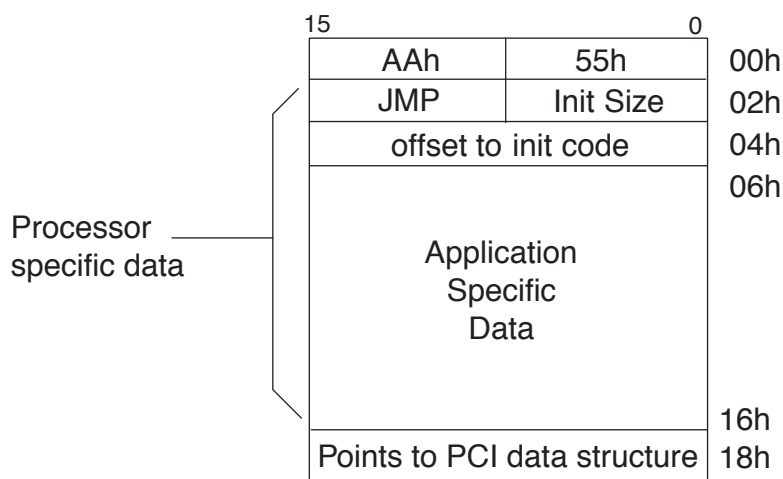


Figure 6-9: ROM image header.

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 3 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 4 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.

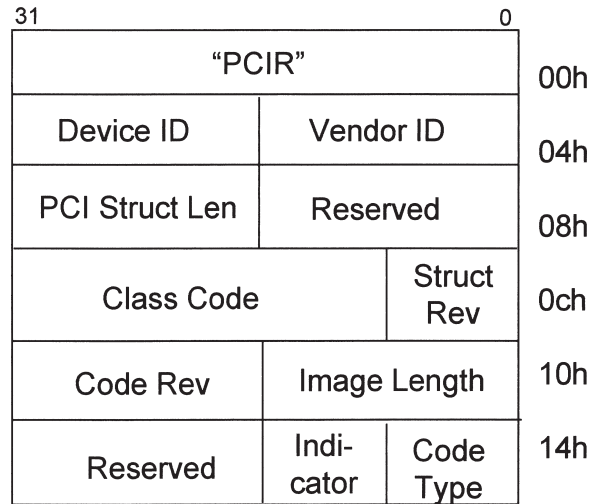


Figure 6-10: PCI data structure.

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.

Power On Self-Test (POST) and Expansion ROMs

Here are the steps that POST code goes through to execute the initialization code in an expansion ROM:

1. Test for expansion ROM Base Address Register.
2. Map expansion ROM to unused area of memory address space and enable it.
3. Validate the expansion ROM (ROM signature, ID fields match).

4. Find appropriate image and copy it into RAM using Init Size field.
5. Disable expansion ROM.
6. Leave RAM writable and call initialization code.
7. Adjust RAM size based on the Init Size field, which may have been modified by the initialization code.
8. Set the RAM area to read only.
9. Go on to the next device.

Capabilities List

Figure 6-11 shows the *Capabilities List*, a mechanism first introduced 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 0x34 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.

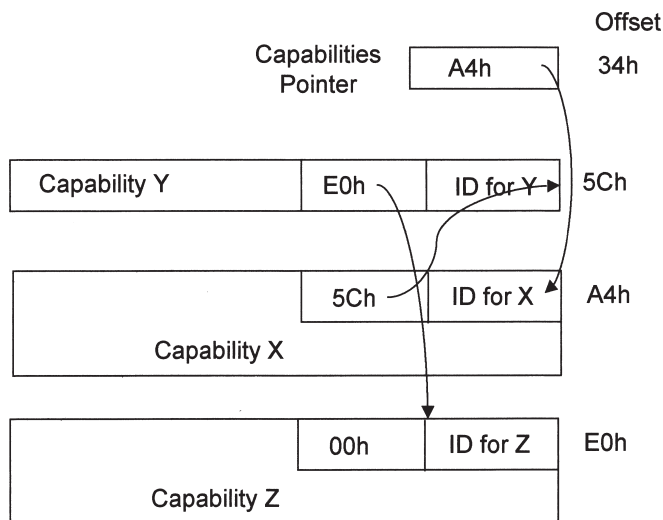


Figure 6-11: Capabilities list.

Table 6-1 lists the capabilities that are currently defined. Unless indicated otherwise, referenced documents are available from the PCI SIG.

Table 6-1: Assigned capability IDs.

ID (hex)	Capability
00	Reserved.
01	Power Management Interface. Documented in <i>PCI Power Management Interface Specification</i> .
02	AGP. Identifies a device supporting Accelerated Graphics Port features. See <i>Accelerated Graphics Port Specification</i> available at www.agpforum.org .
03	VPD. Device supports Vital Product Data as described below.
04	Slot Identification. Identifies a bridge that provides external expansion capabilities. See <i>PCI to PCI Bridge Architecture Specification</i> .
05	Message Signaled Interrupts. See Chapter 7.
06	CompactPCI Hot Swap. See Chapter 10.
07	PCI-X. See Chapter 13.
08	HyperTransport. Refer to the <i>HyperTransport I/O Link Specification</i> available at www.hypertransport.org .
09	Vendor specific. Layout is undefined except that the byte following the “next” field is a length field indicating the number of bytes in the capability structure including the ID and Next bytes.
0A	Debug Port.
0B	CompactPCI Central Resource Control. See <i>PICMG 2.13 Specification</i> available from www.picmg.com .
0C	PCI Hot Plug. See Chapter 10.
0E	AGP 8x.
0F	Secure Device.
10	PCI Express.
11	MSI-X. Identifies an optional extension to basic MSI capability. See Chapter 7.
12-FF	Reserved.

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 forth. 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 4 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.

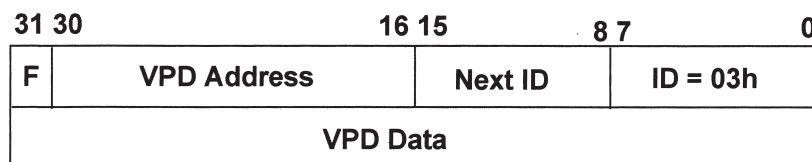


Figure 6-12: VPD capability.

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

Figure 6-13: VPD information field.

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”

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.

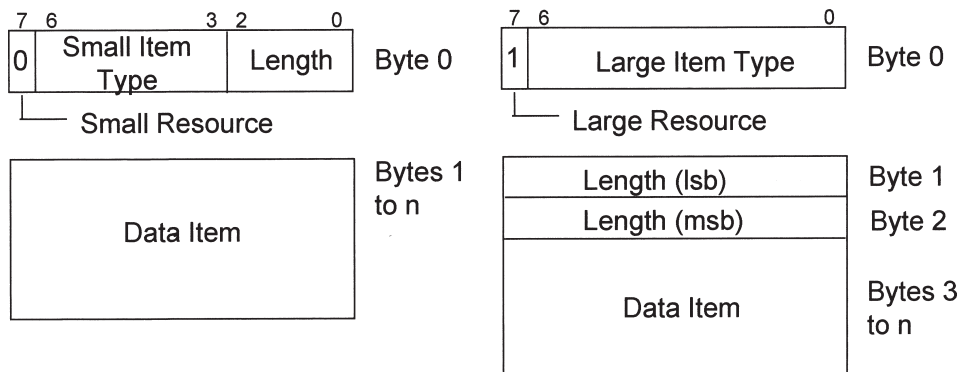


Figure 6-14: Resource data tags.

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.
- 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 4 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

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.

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 predefined header that provides all of the functionality and information 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.

The next chapter describes the PCI bridge.

CHAPTER 7

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 7-1.

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.

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.

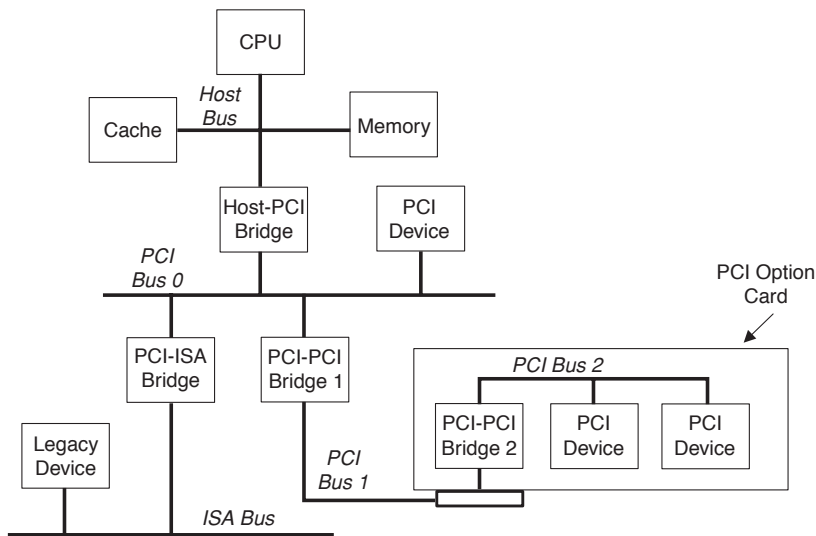


Figure 7-1: PCI bridge hierarchy.

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 7-2). 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 and operation 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.

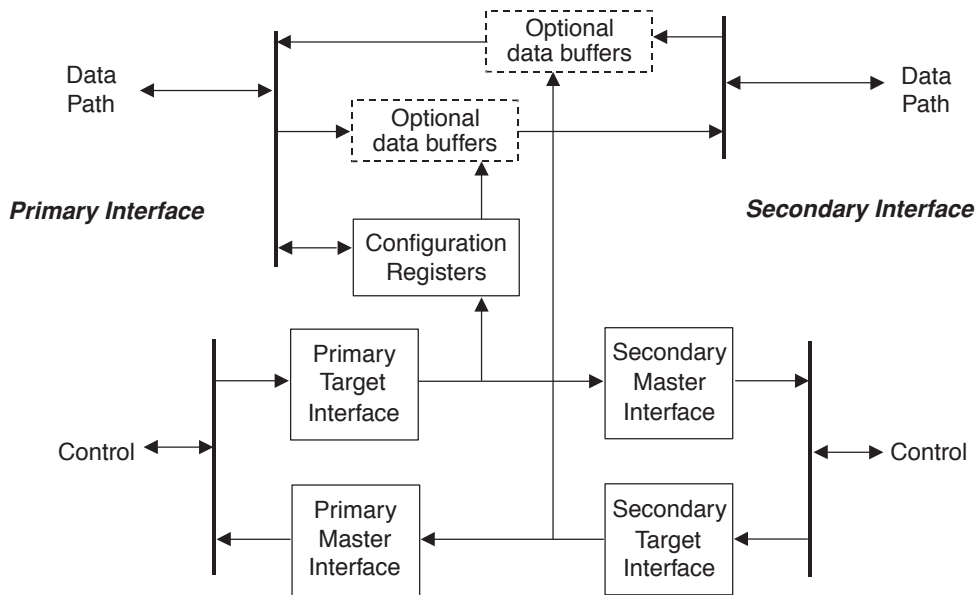


Figure 7-2: PCI bridge structure.

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 Type 0 and Type 1, respectively. 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 7-3). 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

31	11	10	8	7	2	1	0
Reserved				Function Number	Register Number		0 0

Type 1

31	24	23	16	15	11	10	8	7	2	1	0
Reserved			Bus Number		Device Number		Function Number		Register Number		0 1

Figure 7-3: Configuration address types.

Configuration Header—Type 1

Figure 7-4 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.

	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
*Optional	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 7-4: 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 7-5, 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 (upstream) interface.

Secondary Bus Number: Holds the bus number of the secondary (downstream) 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 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

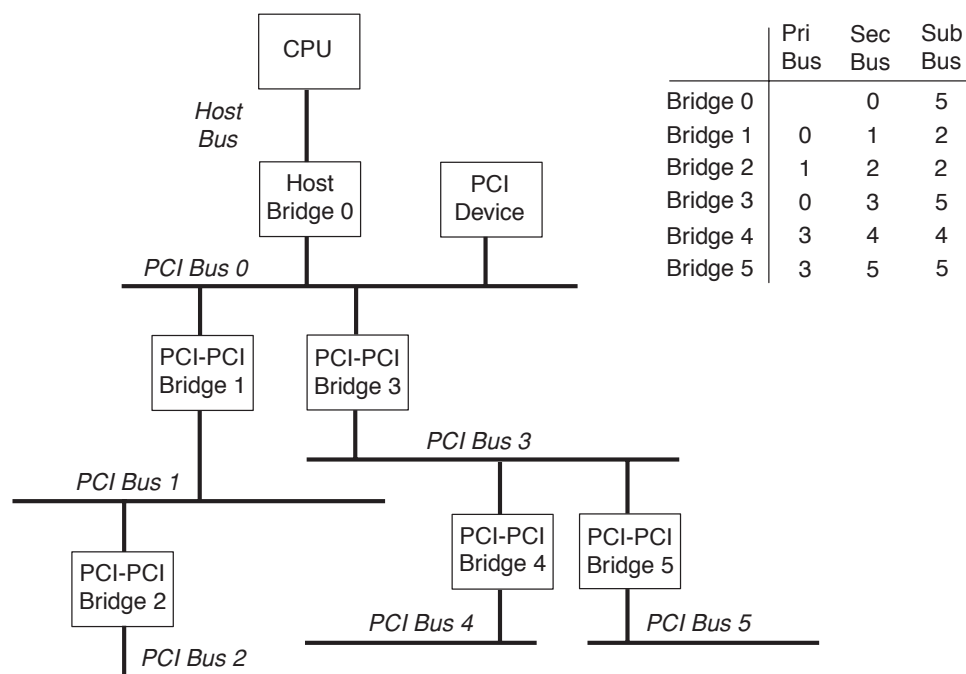


Figure 7-5: Bus number registers.

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 7-5, 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 7-6.

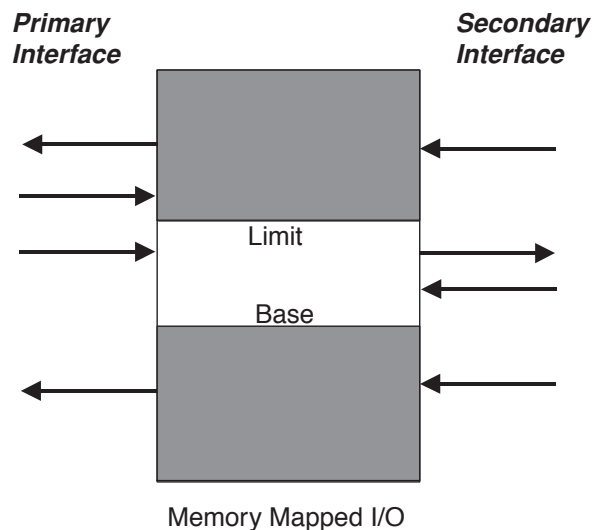


Figure 7-6: Address filtering with base and limit registers.

Note that the address windows of a given bridge must include the windows allocated to any downstream bridges in that branch, i.e., all subordinate bridges. This means that the address ranges for all the devices on all the bridges in a branch must be allocated contiguously.

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:

- 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.

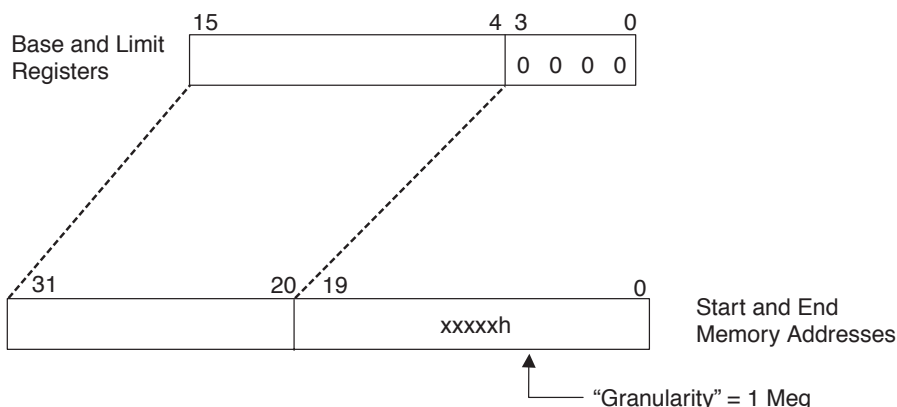


Figure 7-7: Memory base and limit registers.

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 7-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 7-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.

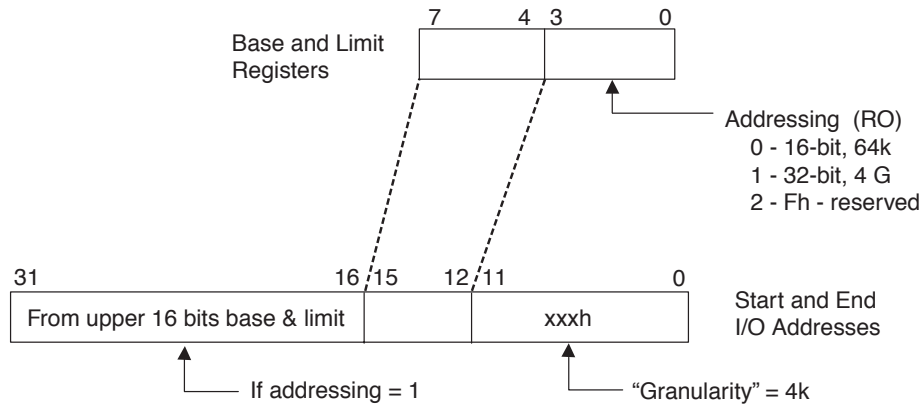


Figure 7-8: I/O base and limit registers.

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.

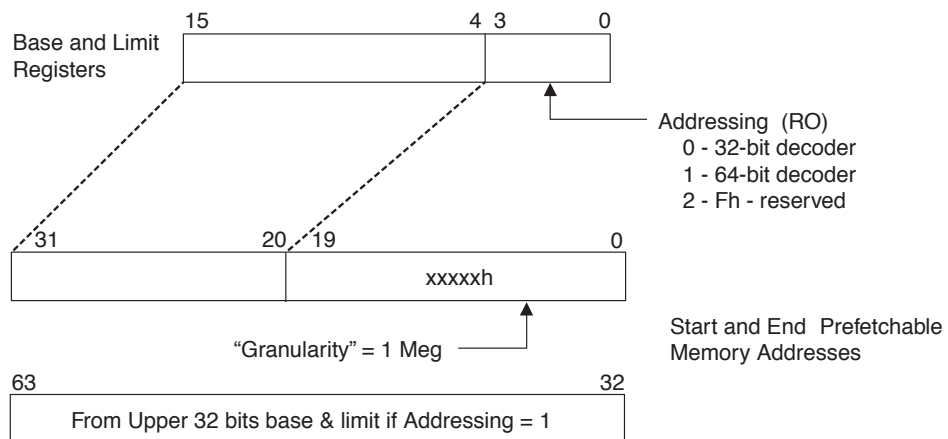


Figure 7-9: Prefetchable base and limit registers.

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. Secondly, 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 7-10.

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?

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.

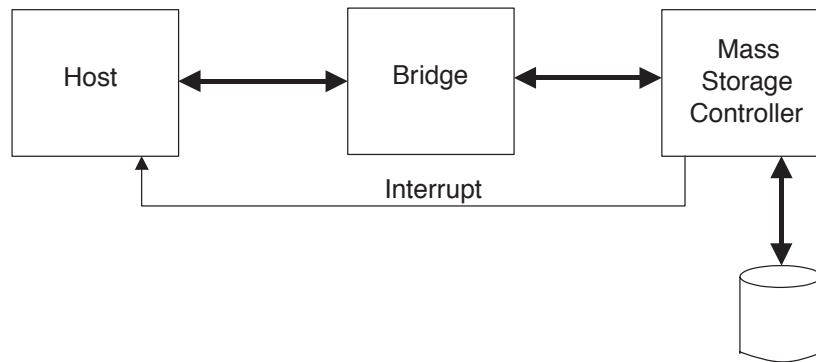


Figure 7-10: Interrupt handling across a bridge.

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 (MSI) 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. The address is presumed to be part of the host processor’s interrupt logic. It may, for example, map to a specific IRQ line or level. The message data identifies the source of the interrupt and may be thought of as an interrupt “vector.”

MSI 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. On the other hand, being just another transaction, the MSI is subject to latency delays caused by

transactions originating in other devices. We cannot guarantee interrupt latency with MSI.

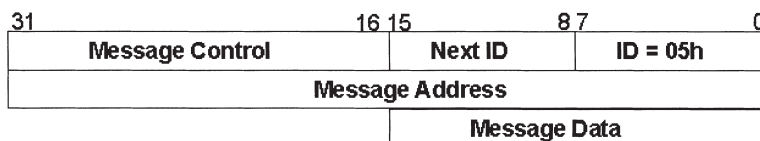
MSI is implemented as an optional Capability. Figure 7-11 shows the layout of the MSI Capability structure. There are three formats:

- 32-bit message address.
- 64-bit message address. If a device supports 64-bit addressing through the DAC then it must implement this, or the following format.
- 64-bit message address with optional per-vector mask and pending bits.

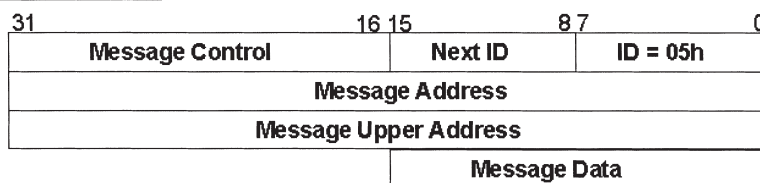
Message Address references a DWORD and so the low order two bits are zero. Each function in a multifunction device has its own MSI capability structure if it supports MSI.

The Message Address and Message Data fields are writable and are set during configuration. An MSI transaction involves a DWORD write of the Message Data field

32-bit Address



64-bit Address



64-bit Address and Per Vector Masking

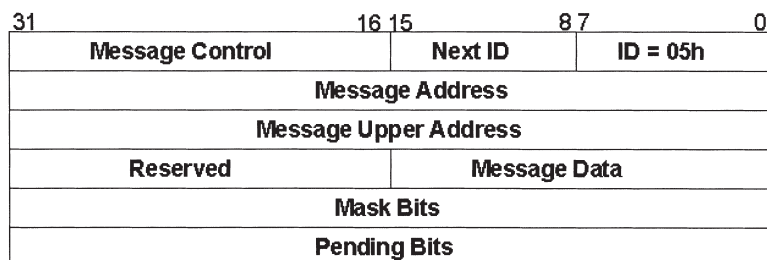


Figure 7-11: Message signaled interrupt (MSI) capability.

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

15-9 Reserved. Read as 0.

8 (RO) 1 = Function supports per-vector masking. The mask and pending registers are present.

7 (RO) 1 = 64-bit address capability.

6-4 Number of messages allocated. Written by configuration software. 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 function.

0 1 = MSI capability enabled.

0 = signal interrupts using INTx.

Note that this bit is intended for system configuration software only. A device driver should not use this bit as a way to mask a function's interrupt sources.

The MSI capability and the INTx signals are mutually exclusive. If the MSI capability is enabled, a function must not assert an INTx signal.

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 function. However, the system software has the option of allocating fewer interrupt messages to a function if there aren't enough to go around.

This in turn means the function, and its associated device driver, must be prepared to deal with the possibility that it won't get all the vectors it asked for. Multiple interrupt sources within the function will have to share a vector.

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

Per-Vector Masking

This is an optional feature of MSI introduced in Revision 3.0 of the specification. In a function that supports multiple vectors, it is generally useful to be able to independently mask individual interrupt sources to prevent them from generating messages at inopportune times. The Mask Bits and Pending Bits registers both implement as many bits as the number of messages (vectors) requested by bits 1 to 3 of the Message Control register, starting from bit 0. The bit positions in these registers map to the low order bits in the Message Data register.

When a Mask Bit is set to one, the corresponding interrupt source is not allowed to generate interrupt messages. The function sets the corresponding Pending Bit to signify that a service request is pending. When software subsequently clears the Mask Bit, and the Pending Bit is set, the function sends the appropriate message and clears the pending bit when the message is sent.

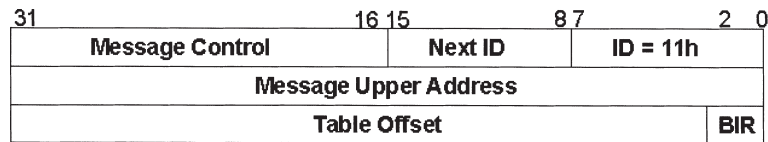
The Mask and Pending bits registers also allow for a “polled” mode of operation. System software could set all Mask bits to one and then periodically poll the Pending Bits register. Upon finding a Pending bit set, the software services the corresponding event whereupon the function resets the Pending bit.

MSI-X

This is another optional Capability introduced in Revision 3.0 that substantially extends the Message Signaled Interrupt concept. MSI-X allows for a larger number of interrupt vectors per function and allows each vector to have independent address and data values as specified in a table in Memory Space. A function is permitted to have both an MSI and an MSI-X capability, but only one of them may be enabled at a time. System behavior is undefined if both capabilities are enabled simultaneously.

Figure 7-12 illustrates the Capability structure for MSI-X. Instead of Address and Data registers, MSI-X uses one of the function’s BARs to allocate space in memory for a table of vectors each consisting of an address and data. The 3-bit read-only BAR Indicator Register (BIR) indicates which of the six possible BARs points to the vector table. The read-only Table Offset is the offset from the base to the start of the vector table. A vector is 8 bytes (4-byte Address, 4-byte Data) and so the offset is 8-byte aligned. The Message Upper Address register allows for a 64-bit message address. Note that all vectors use the same upper address value. If this register is set to zero, then 32-bit addressing is used.

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



Message Address	Message Data	Entry 0	Base
Message Address	Message Data	Entry 1	Base + 1*8
Message Address	Message Data	Entry 2	Base + 2*8
Message Address	Message Data	Entry N - 1	Base + (N - 1)*8

Figure 7-12: Extended message signaled interrupt (MSI) capability.

Bit

15 1 = MSI-X capability enabled.

0 = signal interrupts using INTx or MSI if that capability exists and is enabled.
Note that this bit is intended for system configuration software only. A device driver should not use this bit as a way to mask a function's interrupt sources.

14-11 Reserved.

10-0 (RO) Table Size. Number of entries in the vector table. This is encoded as $N - 1$. So, for example, a value of 3 in the Table Size register represents a table with four entries. The maximum table size is 2048 entries.

Figure 7-13 shows an entry in the vector table. The Address field specifies the low 32 bits of the target of the Message Signaled write transaction. The Data field is the data delivered by that transaction. Since the address is referencing a DWORD target, the low order two bits are used as mask and pending bits. These behave the same as the optional mask and pending bits under MSI. Note that mask and pending are not optional in MSI-X.

The motivation behind MSI-X is to provide a better mechanism for dealing with the case where the system software allocates fewer vectors than the function requested. With MSI, it's up to the device to figure out how to use the allocated vectors. With MSI-X, system software/device driver will put something into every vector table

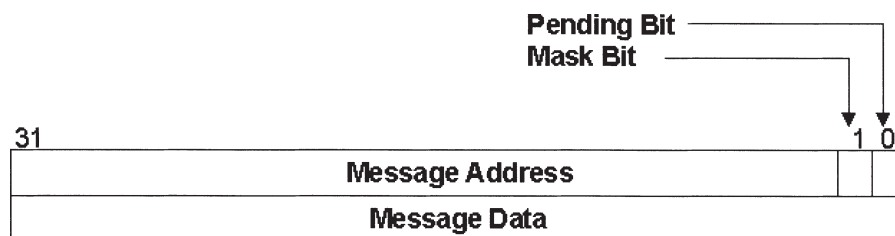


Figure 7-13: MSI-X vector table entry.

entry. If the system must allocate fewer vectors than requested, the device driver can decide which table entries will share a vector by writing the same address and/or data values to them. The function doesn't even know it has fewer vectors than it wanted.

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 7-14. 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 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 7-15 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.

VGA and GFX on PCI 0

Symbol	Palette Address Decoding Method
-	Subtractive decoding
+	Positive decoding
i	Ignore address
s	Snoop access

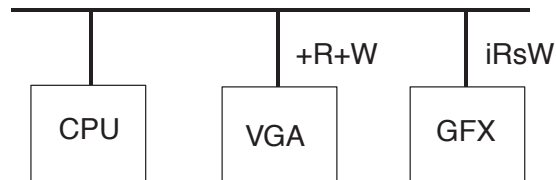


Figure 7-14: Palette “snoop” scenario.

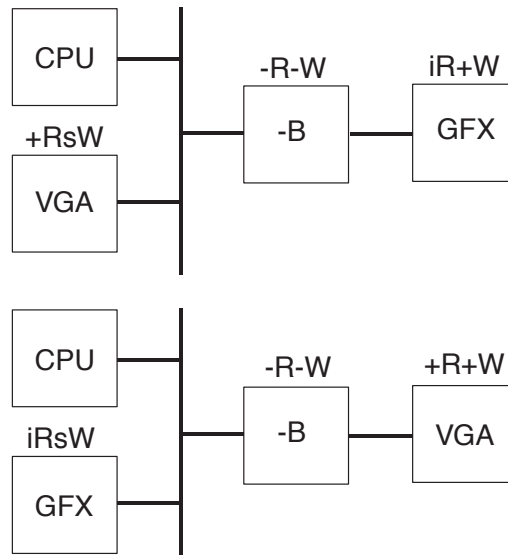


Figure 7-15: Palette “snoop” across subtractive bridge.

Figure 7-16 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.

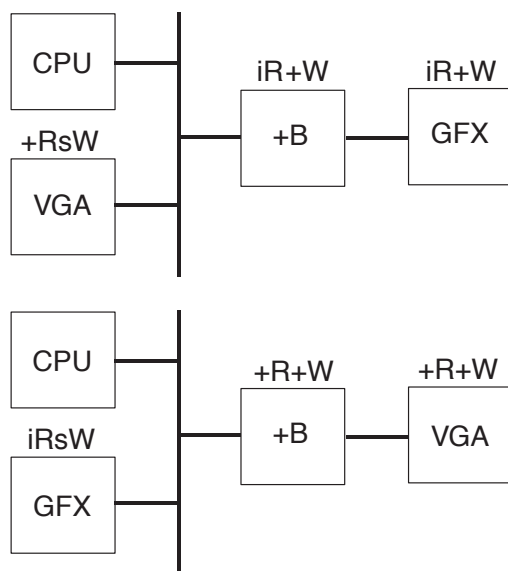


Figure 7-16: Palette “snoop” across a positive bridge.

Resource Locking

In any multimaster 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!

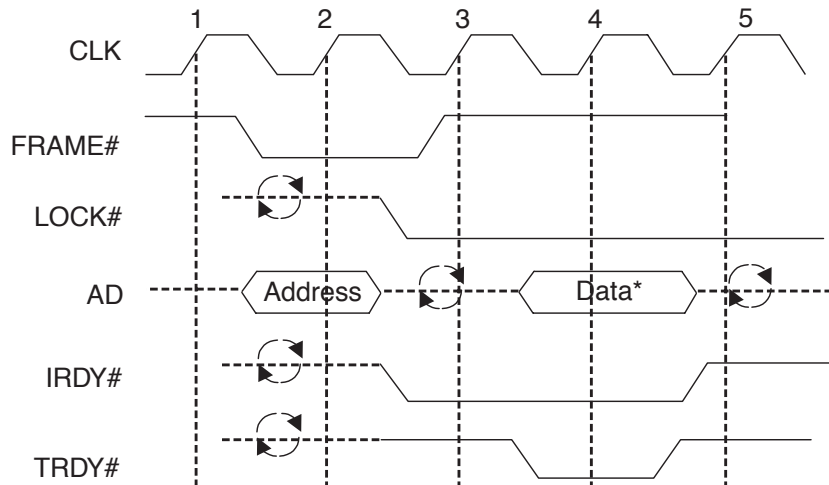
Backward compatibility refers to the hardware locking mechanism of the EISA bus. A PCI-to-EISA bridge may be the target of a locked transaction initiated by the host processor. A host-to-PCI bridge may honor a locked transaction to main memory initiated by a master on the EISA bus, but only if the PCI-to-EISA 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 7-17). 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.



*First transaction must be a read

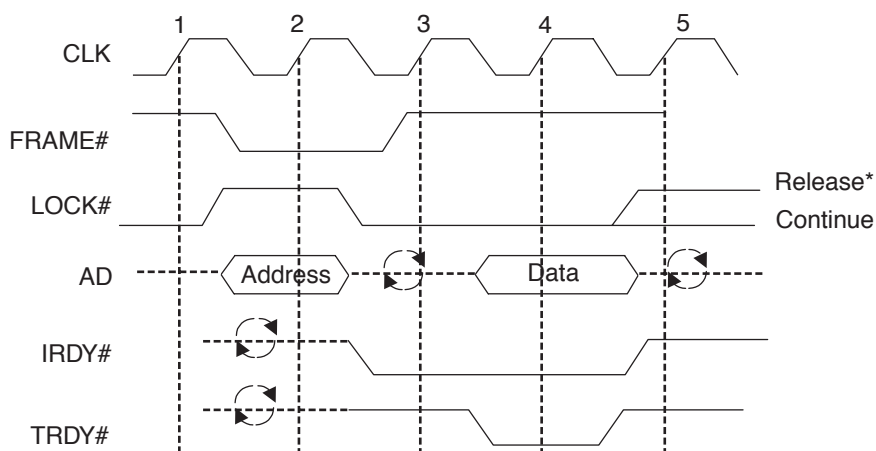
Figure 7-17: First lock cycle.

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.

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 7-18 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 7-18: 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 7-19). In this case, the target executes a retry abort.

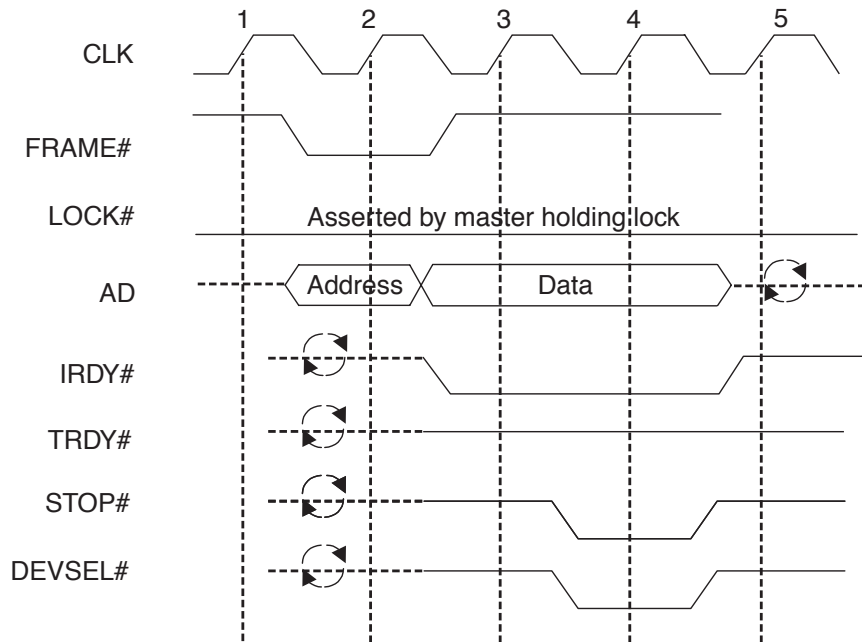


Figure 7-19: Accessing a locked target.

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.

The next chapter describes the PCI BIOS, a platform-independent means of accessing Configuration Space.

System Configuration and the PCI BIOS

Now that we've seen how Plug-and-Play configuration works and how a large system may be built up from multiple bus segments via bridging, it's time to consider the software implications of configuring a PCI system. This is commonly known as “enumerating the bus” and basically consists of traversing the entire bus hierarchy starting at segment 0 to determine:

- How many P2P bridges are present and their relative hierarchy
- How many devices are connected to each bus segment
- What resources each device requires.

Having determined the sum total of required resources—memory space, prefetchable memory space, I/O space, interrupts—the software traverses the hierarchy again allocating resources to devices. Resources for all bridges and devices “behind,” downstream, of a given bridge are allocated contiguously.

Who Configures the System?

Obviously, system configuration is something that must happen fairly soon after power is turned on. At the very least, the BIOS requires access to the screen and keyboard to output its messages and to provide for BIOS configuration. It also requires access to a disk drive to boot the operating system. So the BIOS must include some minimum level of configurability.

In fact, contemporary BIOSes have the ability to completely configure the PCI bus hierarchy and store the resulting configuration in a nonvolatile structure called *Extended System Configuration Data (ESCD)*. Each time the system is powered up, the BIOS compares the current set of installed devices against the ESCD. If it's the same,

as it usually is, then the ESCD configuration can just be loaded without having to completely enumerate the bus.

But, having said that the BIOS *can* configure the PCI hardware doesn't necessarily mean that it will. Later Windows operating systems incorporate full Plug-and-Play capability and are referred to as PnP operating systems. Even if the BIOS does the configuration, the OS may go ahead and do its own configuration when it is started. The BIOS will typically have a parameter called "PnP OS Installed." If this parameter is set to "Yes," the BIOS will skip PCI configuration beyond the minimum necessary to boot the operating system.

After the system is configured, regardless of who does it, the device drivers are started. Each device driver is responsible for finding its device(s) and determining the configuration of that device by reading its Configuration space. The PCI BIOS API, described below, is one mechanism for doing that.

System Configuration—An Overview

This section is a brief overview of PCI configuration software presented in the form of pseudo-code. It is primarily derived from the FreeBIOS project on SourceForge¹. The objective is simply to provide a feel for what goes into the process of auto-configuring a PCI system.

Data Structures

Figures 8-1 and 8-2 show the principal data structures. **pci_dev** defines a PCI function. All functions in the system are linked together through the **next** field. All functions on a bus segment are linked together through the **sibling** field.

pci_bus defines a bus segment and is associated with the bridge for which the segment is the secondary bus. All **pci_bus** structures are linked together through the **next** field starting with **pci_root**, which describes bus segment 0. All bus segments subordinate to this one are linked together through the **children** field.

¹ www.sourceforge.net. Search for "BIOS." At least some of the code in FreeBIOS comes from Linux. Unlike Linux, this code is commented and the symbols are real words.

```
struct pci_dev {
    struct pci_bus    *bus;           /* bus this device is on */
    struct pci_dev    *sibling;       /* next device on this bus */
    struct pci_dev    *next;          /* chain of all devices */

    unsigned int      devfn;          /* encoded device & function index */
    unsigned short     vendor;
    unsigned short     device;
    unsigned int       class;          /* 3 bytes: (base, sub, prog-if) */

    unsigned int       hdr_type;       /* PCI header type */
    unsigned int       irq;            /* irq generated by this device */
    unsigned long      base_address[6];
    unsigned long      size[6];
    unsigned long      rom_address;
};
```

Figure 8-1: pci_dev data structure.

```
struct pci_bus {
    struct pci_bus    *parent;         /* parent bus this bridge is on */
    struct pci_bus    *children;       /* chain of P2P bridges on this bus */
    struct pci_bus    *next;           /* chain of all PCI busses */
    struct pci_dev    *self;           /* bridge device as seen by parent */
    struct pci_dev    *devices;        /* devices behind this bridge */

    unsigned char      number;          /* bus number */
    unsigned char      primary;         /* number of primary bridge */
    unsigned char      secondary;       /* number of secondary bridge */
    unsigned char      subordinate;     /* max number of subordinate buses */

    unsigned long      mem_sz, premmem_sz, io_sz; /* resource requirements for this
                                                    and all subordinate buses computed
                                                    by compute_resources() */

    u32 membase, memlimit;              /* resource windows */
    u32 premmembase premmemlimit;
    u32 iobase, iolimit;
};
```

Figure 8-2: pci_bus data structure.

Scanning the system

Listing 8-1 shows `pci_scan_bus()`, the function that scans the entire system building up a list of device/functions and a hierarchical bus tree. `pci_scan_bus()` is called recursively for each P2P bridge encountered. For each function encountered `pci_scan_bus()` determines, among other things, the size requirement for each Base Address Register.

```
unsigned int pci_scan_bus (struct pci_bus *bus)
/*
  Scans the bus segment pointed to by 'bus'. Returns maximum bus number
  allocated after scanning all subordinate busses. Called recursively for each
  bus segment.
*/
{
    unsigned int devfn, max = bus->secondary;
    struct pci_dev *dev, **bus_last = &bus->devices;
    struct pci_bus *child;

    for (devfn = 0; devfn < 0xff; devfn++) {
        if (function is present) {
            dev = kmalloc (sizeof (pci_dev));
            Fill dev with device information

            Get base address registers and sizes
            Link into list of functions
            Link into list of functions for this bus
        }
    }
    for (dev = bus->devices; dev; dev = dev->sibling) {
        if (function is a bridge) {
            child = kmalloc (sizeof (pci_bus));
            Fill child with bus information

            child->next = bus->children;
            bus->children = child;
            child->self = dev;
            child->parent = bus;

            child->number = child->secondary = ++max;
            child->primary = bus->secondary;
            child->subordinate = 0xff;
            /* Now we can scan all subordinate buses i.e. the bus behind the bridge */
            max = pci_scan_bus (child);
            child->subordinate = max;
            Set bus number fields
        }
    }
    return max;
}

struct pci_bus pci_root;

void pci_init (void)
{
    memset (&pci_root, 0, sizeof (pci_root));
    pci_root.subordinate = pci_scan_bus (&pci_root);
}
```

Listing 8-1: pci_scan_bus()

```
void compute_allocate_io (struct pci_bus *bus)
{
    int i;
    struct pci_bus *curbus;
    struct pci_dev *curdev;
    unsigned long io_base base = bus->iobase;

    /* First, walk all the bridges. When you return, grow the limit of the current bus
       since sub-busses need IO rounded to 4096 */
    for (curbus = bus->children; curbus; curbus = curbus->children) {
        curbus->iobase = io_base;
        compute_allocate_io (curbus);
        io_base = round (curbus->iolimit, IO_BRIDGE_ALIGN);
    }
    /* Walk through all the devices on current bus and compute IO address space.*/
    for (curdev = bus->devices; curdev; curdev = curdev->sibling) {
        for (i = 0; i < 6; i++) {
            if (curdev->base_address[i] is I/O) {
                curdev->base_address[i] = io_base;
                io_base += round (curdev->size[i], IO_ALIGN);
            }
        }
    }
    bus->iolimit = round (io_base, IO_BRIDGE_ALIGN) - 1;
}

void compute_allocate_resources (struct pci_bus *bus)
/*
This is a one-pass process. We first compute all the IO, then memory, then prefetchable
memory. This is only called once passing in pci_root as the argument.
*/
{
    compute_allocate_io (bus);
    compute_allocate_mem (bus);

    // now put the prefetchable memory at the end of the memory
    bus->prefmembase = round (bus->memlimit, ONEMEG);
    compute_allocate_prefmem (bus);
}

void pci_configure()
{
    pci_init();
    pci_root.membase = PCI_MEM_START;
    pci_root.prefmembase = PCI_MEM_START;
    pci_root.iobase = PCI_IO_START;

    compute_allocate_resources (&pci_root);
    // now just set things into registers ... we hope ...
    assign_resources (&pci_root);
}
```

Listing 8-2: pci_configure(), compute_allocate_resources() and compute_allocate_io()

Once the entire system is enumerated, we can allocate resources with the function `compute_allocate_resources()` shown in Listing 8-2. This function is only called once passing in `pci_root` as its argument. `compute_allocate_resources()` calls three different functions to allocate the three different PCI spaces: IO, memory and prefetchable memory. `compute_allocate_io()` is shown in Listing 8-2. The other two are similar. The three allocation functions are called recursively for each bus segment. The function `round()` rounds its first argument up modulo its second argument, which must be a power of two.

When `compute_allocate_resources()` returns, every function has resources allocated to it in its `pci_dev` structure, and every bridge has base and limits set in its `pci_bus` structure. Nothing has been written to Configuration Space yet. That's the job of `assign_resources()`. It first walks through the list of busses setting the base and limit registers. Then it walks through the list of device/functions setting all of the Base Address Registers. The entire configuration process is encapsulated in the function `pci_configure()`.

PCI BIOS

It is entirely possible for device drivers to access the Configuration Space directly using the mechanism described in Chapter 6. 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 8-3. 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.

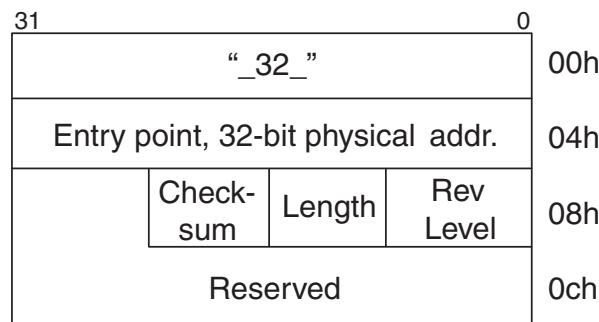


Figure 8-3: BIOS32 Service Directory.

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

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

ENTRY

AX B101h

EXIT

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

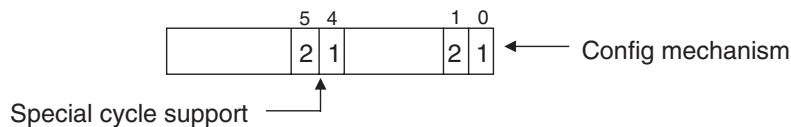
EDX "PCI "

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



Find PCI Device/Class

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)

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.

Generate Special Cycle

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

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.

Read Configuration Register (Byte, Word, Dword)

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

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.

Write Configuration Register (Byte, Word, Dword)

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

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.

Get Interrupt Routing Options

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

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.

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 8-4. 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 8-4: 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 specification 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

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

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.

Summary

System configuration may be done by the BIOS or by the operating system. Configuration involves building a linked list of all devices and a hierarchical tree of all bus segments. Each device/function is scanned to determine its resource requirements. Resources are then allocated recursively starting with the “farthest out” bridge and working back to the root.

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.

The next chapter describes Compact PCI, an industrial-strength version of PCI.

CHAPTER 9

CompactPCI

CompactPCI is just an industrial strength version of the same PCI bus found in any contemporary personal computer. 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 2 mm 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. Extensive simulations were done in the course of developing the CompactPCI specification to verify that it could indeed support eight slots.

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.

Introduction—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 enormously 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, and so forth. Three particular application areas are particularly well suited to CompactPCI implementations. They are:

- Telephony
- Avionics
- Machine Vision

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), which is made up of companies involved in various aspects of industrial computing.

PCI Industrial Computer Manufacturers’ Group
c/o Virtual, Inc.
401 Edgewater Place, Suite 600
Wakefield, MA 01880
(781) 246-9318 www.picmg.org

Table 9-1 lists the specifications currently approved and maintained by PICMG.

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

Mechanical

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

Specification Number	Revision	Description
2.0	3.0	CompactPCI Core Specification
2.1	2.0	CompactPCI Hot Swap
2.2	1.0	VME64x on CompactPCI
2.3	1.0	PMC on CompactPCI
2.4	1.0	IP on CompactPCI
2.5	1.0	CompactPCI Computer Telephony
2.7	1.0	6U CompactPCI Dual System Slot Specification
2.9	1.0	CompactPCI System Management
2.10	1.0	Keying of CompactPCI Boards and Backplanes
2.11	1.0	CompactPCI Power Interface
2.12	2.0	Hot Swap Infrastructure Interface
2.14	1.0	CompactPCI Multicomputing Specification
2.15	1.0	PCI Telecom Mezzanine/Carrier Card Specification
2.16	1.0	CompactPCI Packet Switching Backplane Specification
2.17	1.0	CompactPCI StarFabric Specification
2.20	1.0	CompactPCI Serial Mesh Backplane Specification

Table 9-1: PICMG specifications.

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.

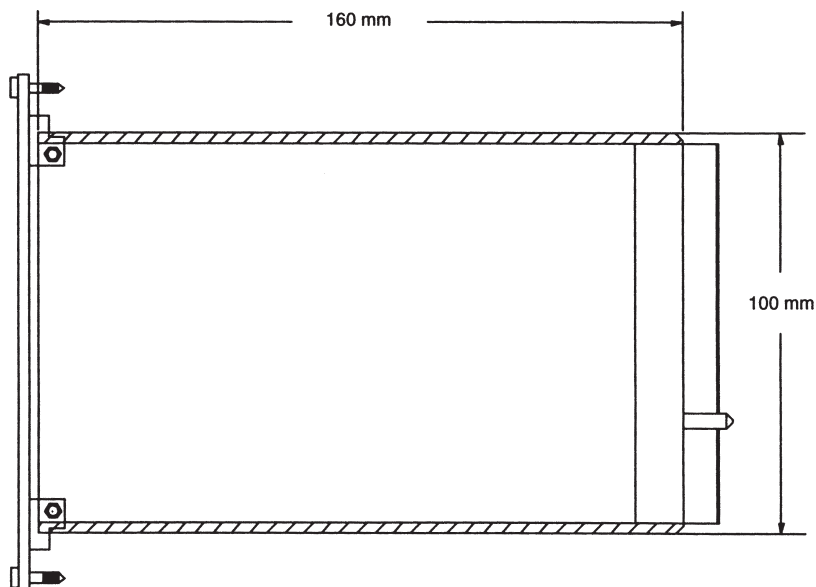


Figure 9-1: 3U Compact PCI card.

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.

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 order¹. 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

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.

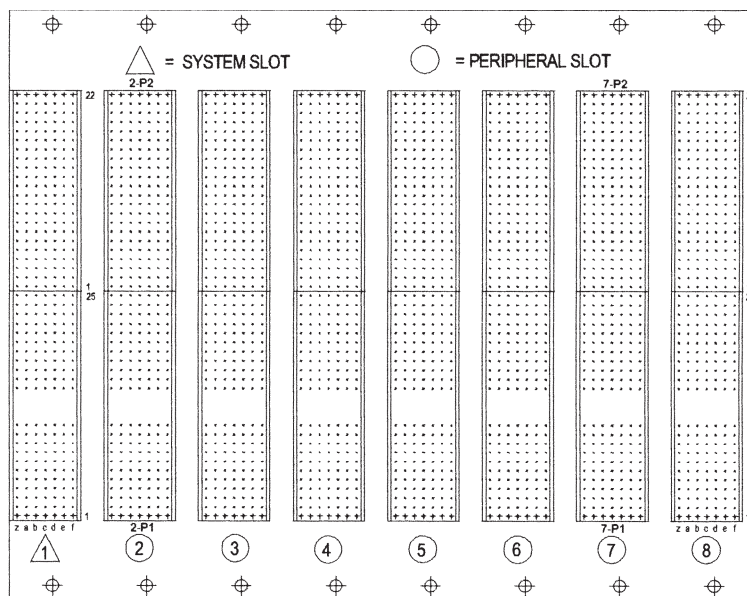


Figure 9-2: Typical 3U backplane segment with eight slots.

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.

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 electrical specifications.

Connector

The basic CompactPCI pin-and-socket connector is organized as 47 rows of 5 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.

The connector is called “hard metric,” meaning that the pin spacing is 2 mm, not 2.54 mm (0.1 inch).

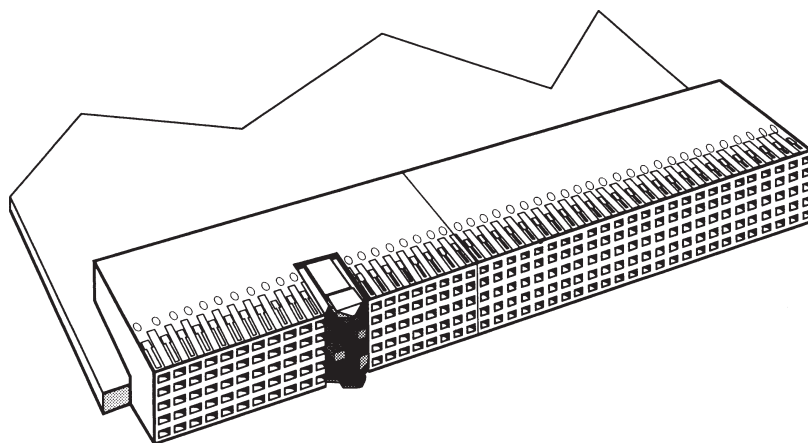


Figure 9-3: 2 mm pin and socket connector.

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).

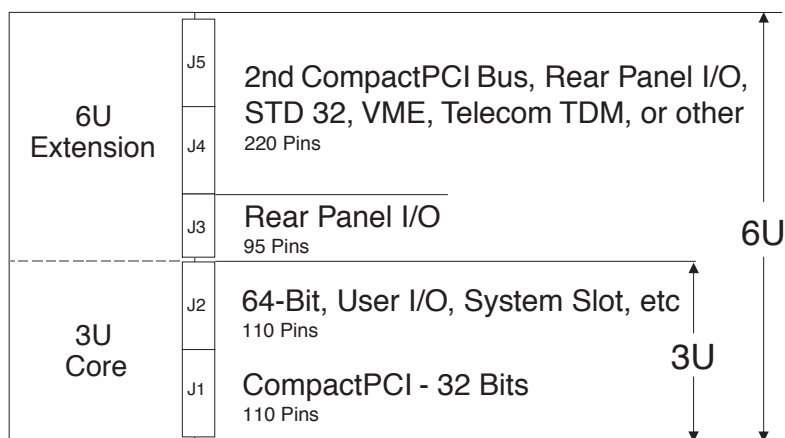


Figure 9-4: CompactPCI connector allocation.

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” 80 mm deep and “should” use the same panels, card guides, ejector handles, and so forth. 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.

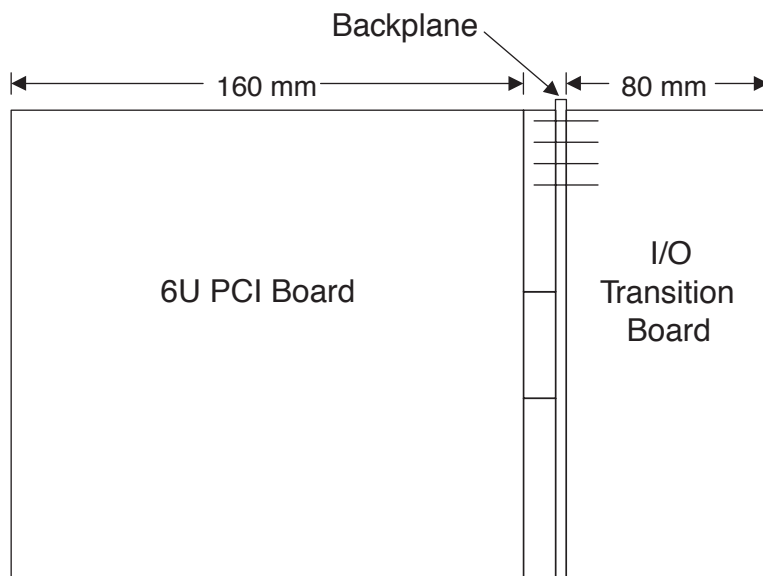


Figure 9-5: Rear panel I/O.

Electrical

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.

Table 9-2: CompactPCI Additional Signals.

PRST#	<i>Push Button Reset.</i> 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#	<i>Power Supply Derating Signal.</i> 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#	<i>Power Supply Fail Signal.</i> Assertion of this optional low-true signal indicates the power supply has failed. The system board must provide a pull-up. ²
SYSEN#	<i>System Slot Identification.</i> 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#	<i>Enumeration.</i> Used by Hot Swap-capable cards to indicate either: <ul style="list-style-type: none">■ 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 9-3. 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 & PMB_SDA	<i>System Management Bus</i> . These pins are reserved for implementing system management functions like board identification, environmental and voltage monitoring, and so forth. 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.

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 9-4.

Table 9-3: 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.

Table 9-4: 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.

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.

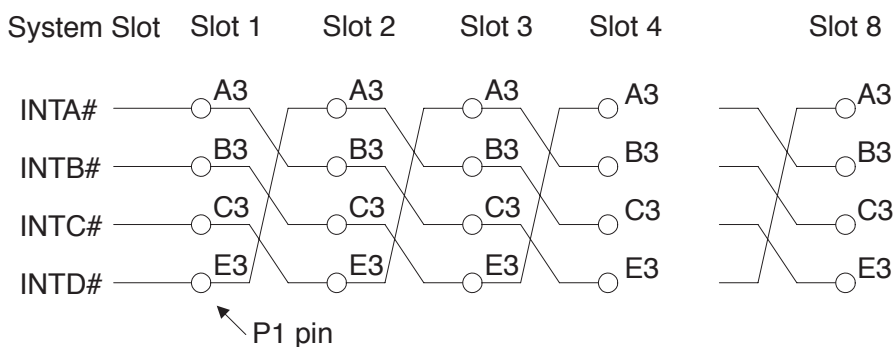


Figure 9-6: Required interrupt routing.

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).

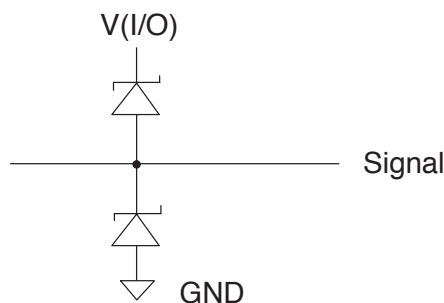


Figure 9-7: Backplane termination for lightly loaded case.

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.

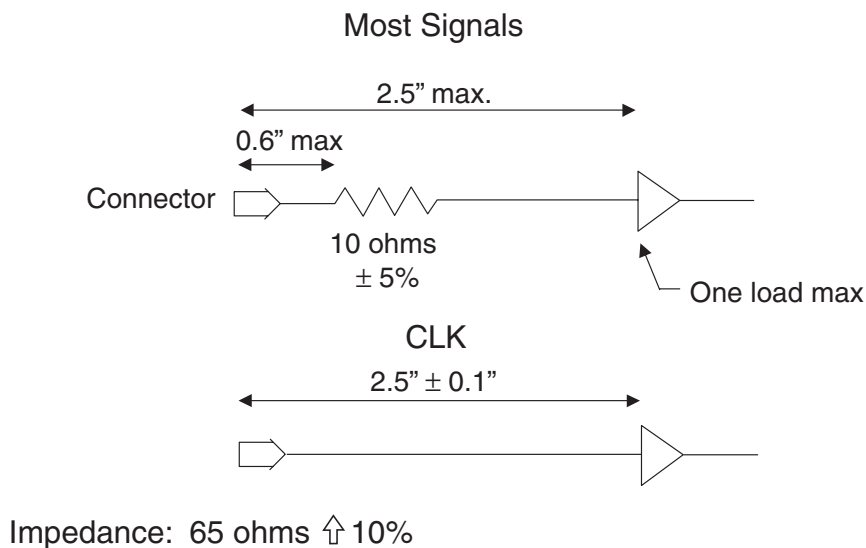


Figure 9-8: Board design rules.

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 to 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.

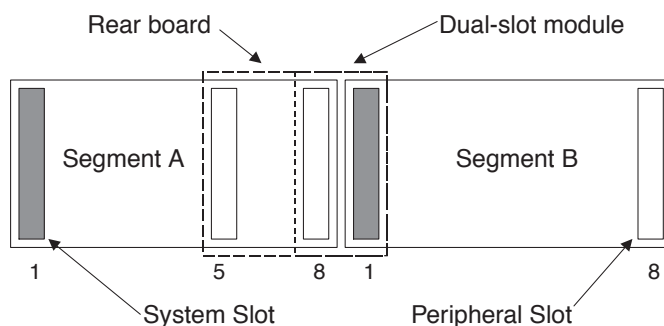


Figure 9-9: Bridging two segments using either a front-plugging module or a rear-plugging pallet board.

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 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 Figure 9-9 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.

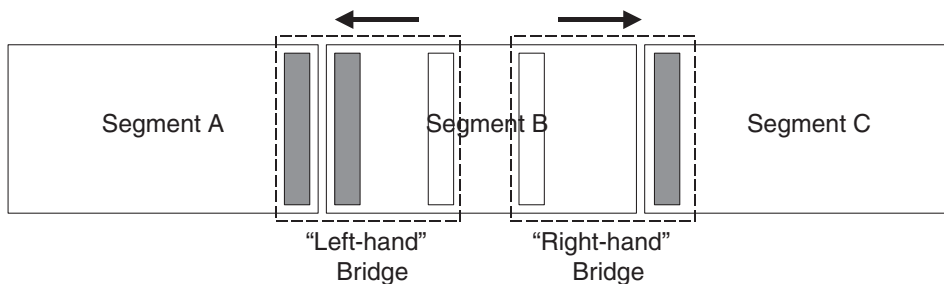


Figure 9-10: CPCI bridging of three segments.

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

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.

The next chapter looks at Hot Plug and Hot Swap, which are mechanisms that allow boards to be inserted or removed from a running system.

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.
- *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.

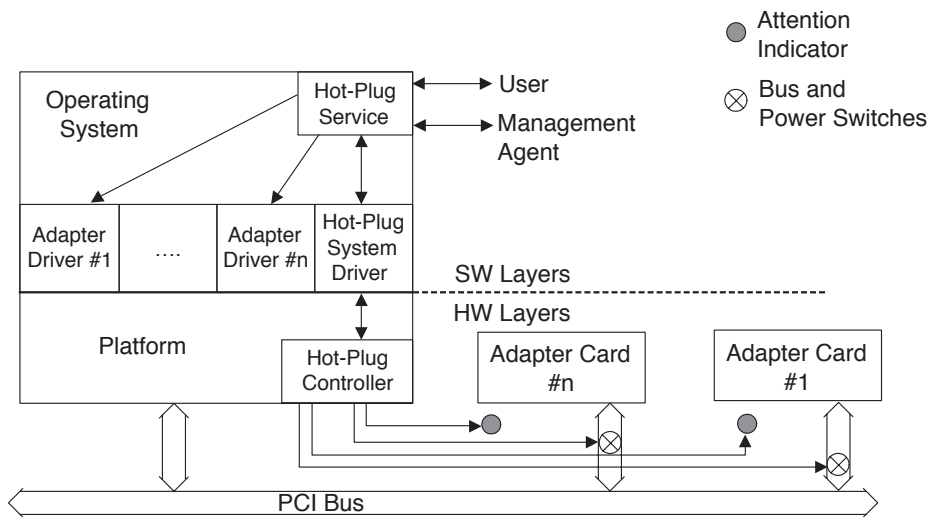


Figure 10-1: Hot Plug system components.

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 the 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 onboard 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 the 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.”

Table 10-1: Pin staging.

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

The longest pins—the first to make contact—are called the “Early Voltages.” These comprise two each +5V and +3.3V, the V_{io} 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 1 volt to minimize the capacitive effects of attaching the lines to 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 pull-up 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 the system to effect software connection and disconnection, and the functions must be performed in the correct sequence for proper system operation.

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

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.
- 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.

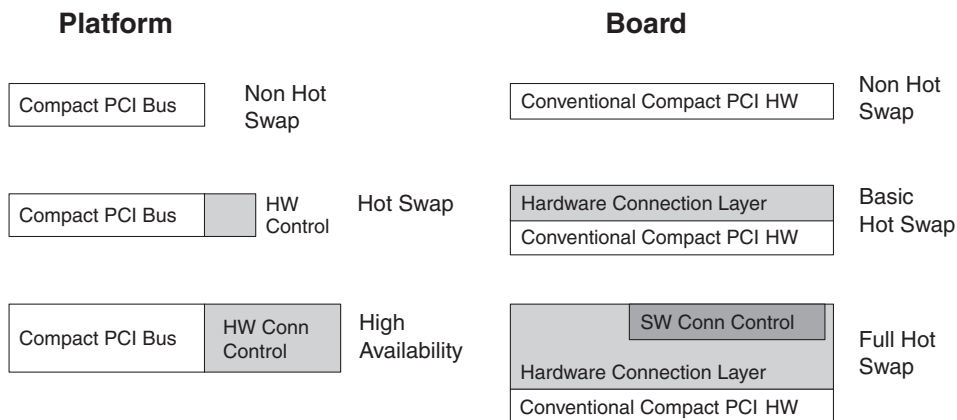


Figure 10-2: Hot Swap interoperability.

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

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	

Compact PCI Specification, providing backward compatibility and allowing Hot Swap to operate in a conventional platform. This configuration does not support Hot Swap.

A Hot Swap platform can accommodate 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 High Availability (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.

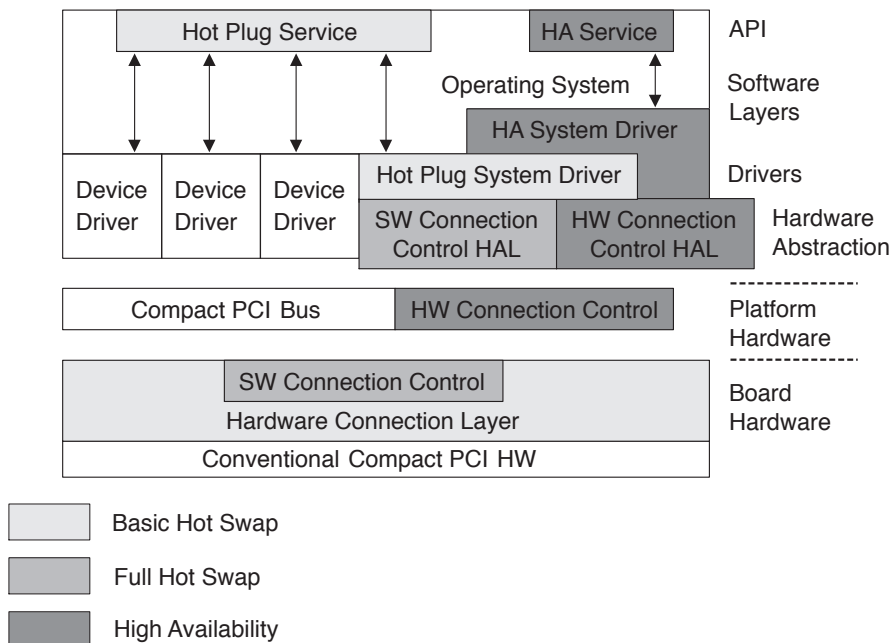


Figure 10-3: Hot Swap system architecture.

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.

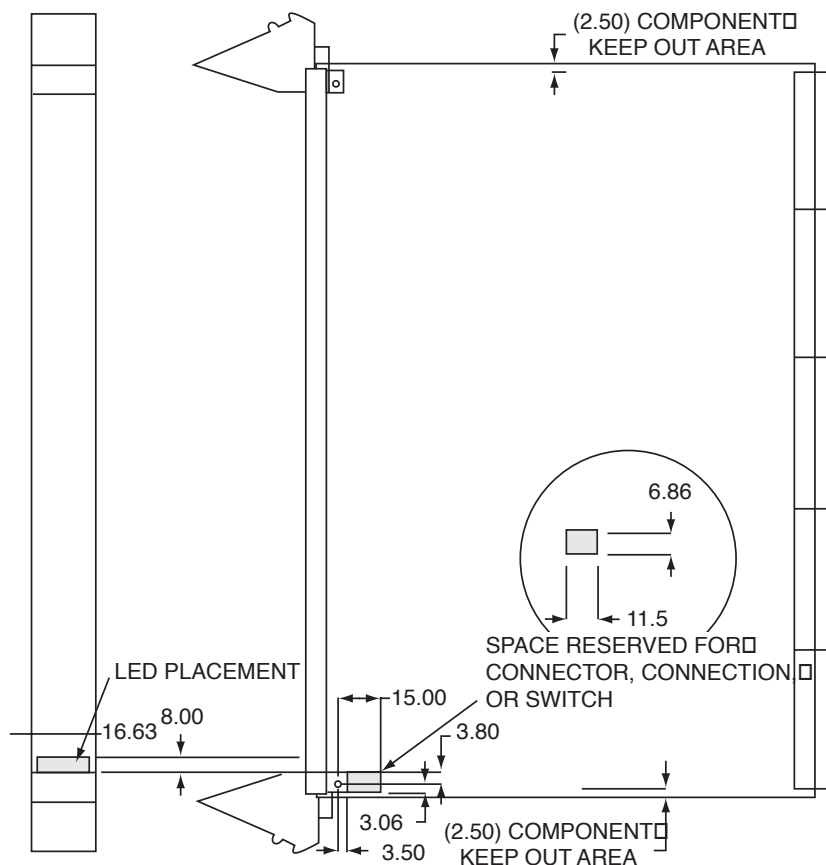


Figure 10-4: Hot Swap board with handle switch and status LED.

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 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

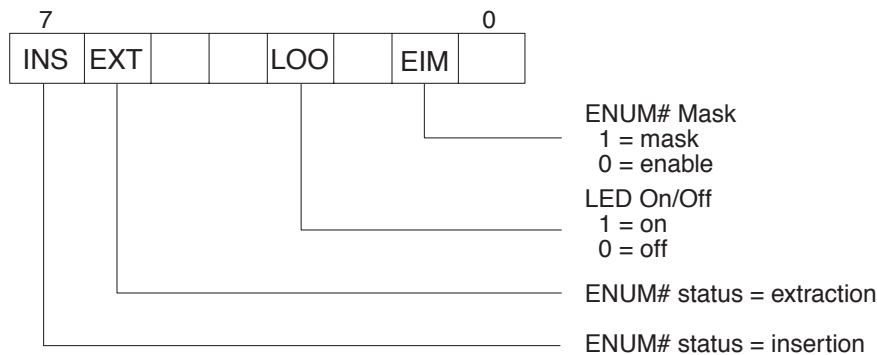


Figure 10-5: Hot Swap control/status register.

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#**.

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.

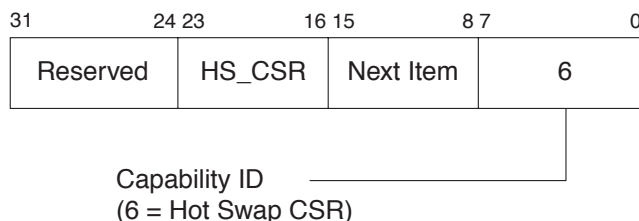


Figure 10-6: HS_CSR capabilities list entry.

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 V_{io} 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 pull-up overcomes the weak pull-down of the HSC and drives **BD_SEL#** high or unasserted, thus signaling its presence. When the HSC decides that it is appropriate to apply backend power, it drives **BD_SEL#** low.

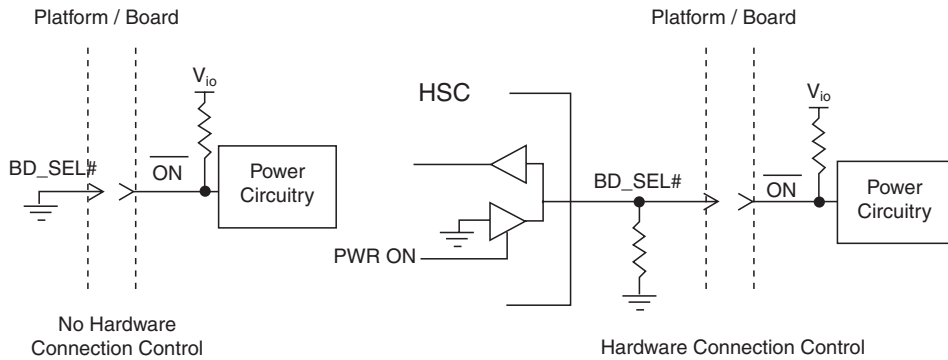


Figure 10-7: Handling of the BD_SEL# signal.

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).

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#.

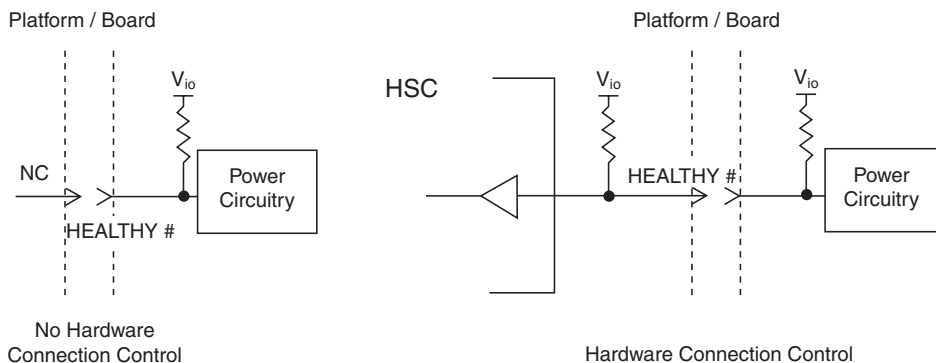


Figure 10-8: Handling of the HEALTHY# signal.

The specification suggests a weak pull-up 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 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).

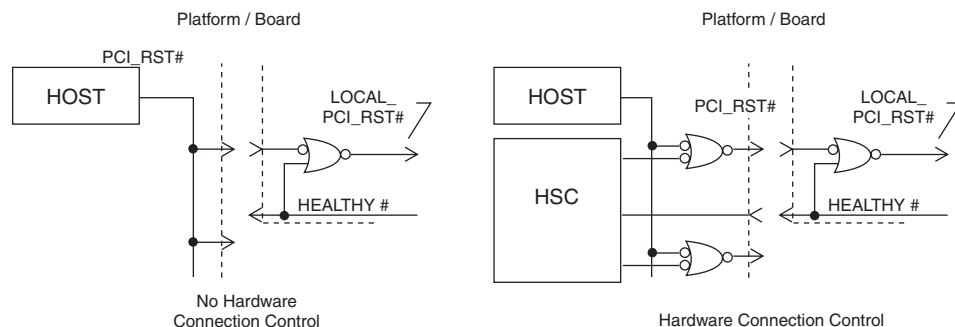


Figure 10-9: Handling of the **PCI_RST# signal.**

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 the 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 backplane. 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.”

The next chapter introduces PCI-X.

CHAPTER 11

Introduction to PCI-X

PCI-X is officially considered to be an addendum to the PCI Local Bus Specification¹. Revision 1.0 was introduced in 1999 to be followed by Revision 2.0 in 2002.

Why PCI-X

When PCI was first introduced in 1992, its maximum bandwidth of 132 Mbytes per second matched well with the processors and peripherals of the day. Today's faster I/O technologies, such as Gigabit Ethernet, Ultra-3 SCSI and Fibre Channel require higher bandwidth system busses to operate at peak efficiency.

Revision 2.0 of PCI-X supports a maximum bandwidth of over 4 Gbytes per second, enough to provide some “headroom” for a few years anyway. Figure 11-1 compares bandwidths for several popular interconnect mechanisms. PCI-X is the clear leader.

What's New

Revision 1.0 PCI-X enhancements fall into two basic categories: higher clock frequency—133 MHz—and enhancements to the protocol to make it more efficient for large block transfers. This yields roughly a gigabyte of bandwidth for a 64-bit bus.

The protocol enhancements include:

- A *split cycle*. This is a substitute for the Retry protocol in PCI. A target that can't complete a transfer within the maximum latency notifies the initiator

¹ It's a pretty big addendum. Revision 2.0 of the PCI-X protocol is 388 pages, versus 344 pages for Revision 3.0 of the PCI specification. Then there are another 137 pages of electrical and mechanical specifications.

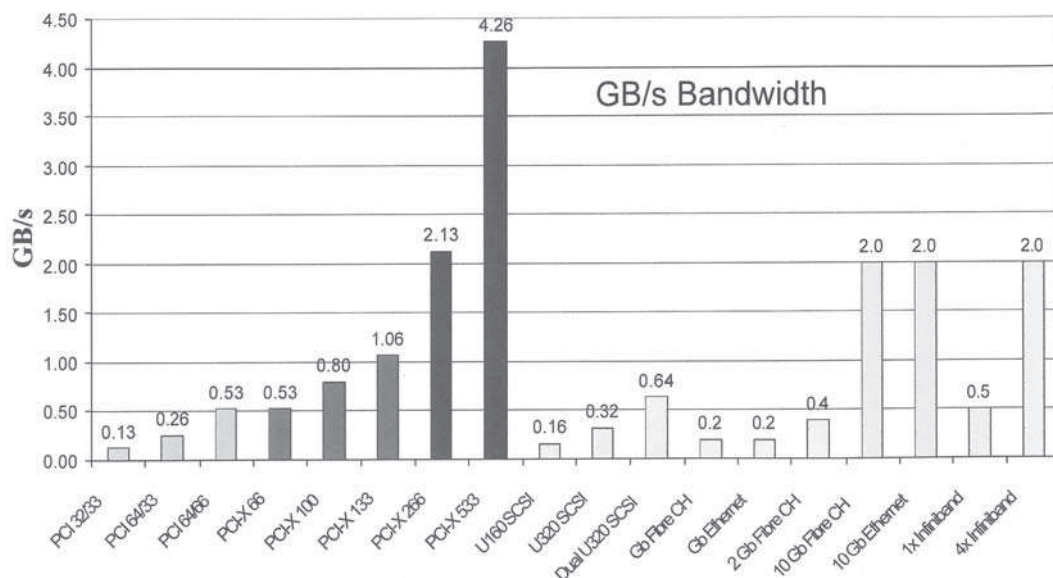


Figure 11-1: Bandwidth comparison of contemporary busses.

with a *Split Completion* termination. Later, the target initiates a transaction back to the original initiator to complete the original transaction.

- An *attribute phase*. All PCI-X transactions include an attribute phase immediately after the address phase of the transaction. The attribute phase includes:
 - Information about the initiator.
 - The length of the transaction.
 - Information on transaction ordering and cacheability.
- Restricted wait state and disconnect rules for more efficient use of the bus.
 - Initiators are not permitted to insert wait state.
 - Targets may insert wait states only on the initial data phase.
 - Disconnects are permitted only on naturally aligned 128-byte boundaries. This encourages the use of longer burst lengths. Targets are permitted to disconnect after the first data phase where longer bursts aren't required.

Revision 2.0 gets to 4 Gbytes/second transfer rate by introducing the notion of *source synchronous* clocking whereby two or four data phases occur during each clock cycle.

To accomplish this feat, the agent driving data on the bus drives a pair of phased strobe signals.

Additional enhancements include:

- 1.5V signaling levels to support source synchronous mode and to improve signal integrity.
- Error Correcting Codes (ECC) that can detect and correct single bit errors.
- Expanded Configuration Space of 4096 bytes per device/function.
- A low pin-count 16-bit interface for embedded applications.
- A device ID messaging transaction for peer-to-peer communication.

Nomenclature

PCI-X defines two modes of operation:

- Mode 1: One data transfer per clock cycle at 66 or 133 MHz. This is known as *common clock sampling*.
- Mode 2: Two or four data transfers per clock cycle at 133 MHz. This is source synchronous mode.

Each of these two modes can operate at two different data rates. This yields four different operating modes in all:

- PCI-X 66: Mode 1 at 66 MHz
- PCI-X 133: Mode 1 at 133 MHz
- PCI-X 266: Mode 2, two transfers per clock cycle
- PCI-X 533: Mode 2, four transfers per clock cycle

Backward Compatibility

Over the past decade, vendors and end users have invested substantial amounts of money and effort in systems based on PCI. Recognizing this substantial investment, the PCI SIG has always stressed backward compatibility when developing enhancements to PCI.

In the same way that 33 MHz PCI devices can coexist with 66 MHz devices, PCI-X devices coexist amiably with conventional PCI devices. Furthermore, the extent and nature of the enhancements in PCI-X Revision 2.0 make devices conforming to this

revision “different” from devices built to Revision 1.0. This is the essential distinction between Mode 1 and Mode 2 devices.

The operational mode of a bus segment is determined by the lowest common denominator. The bus bridge for a segment determines the capabilities of all devices on the segment by monitoring a pair of bus signals during initialization time (see Chapter 13). If all devices are PCI-X capable, the bridge initializes the segment in PCI-X Mode 1 at either 66 MHz or 133 MHz or Mode 2 at 133 MHz. Otherwise, the segment is initialized in conventional PCI mode at either 33 MHz or 66 MHz.

Figure 11-2 is an interoperability matrix between conventional PCI and both Modes of PCI-X.

Systems		Conventional PCI			Add-in Cards PCI-X Mode 1		PCI-X Mode 2	
		33 MHz (5V I/O)	33 MHz (3.3V I/O or Universal)	66 MHz (3.3V I/O or Universal)	PCI-X 66 (3.3V I/O or Universal)	PCI-X 133 (3.3V I/O or Universal)	PCI-X 266 (3.3V I/O)	PCI-X 533 (3.3V I/O)
Con- ventional PCI	PCI-33 (5V I/O)	PCI-33 (5V I/O)	PCI-33 (5V I/O)	PCI-33 (5V I/O)	PCI-33 (5V I/O)	PCI-33 (5V I/O)	Not Supported	Not Supported
	PCI-33		PCI-33	PCI-33	PCI-33	PCI-33	PCI-33	PCI-33
	PCI-66		PCI-33	PCI-66	a) PCI-33 b) PCI-66	a) PCI-33 b) PCI-66	a) PCI-33 b) PCI-66	a) PCI-33 b) PCI-66
PCI-X Mode 1	PCI-X 66		PCI-33	a) PCI-33 b) PCI-66	PCI-X 66	PCI-X 66	PCI-X 66	PCI-X 66
	PCI-X 133		PCI-33	a) PCI-33 b) PCI-66	PCI-X 66	PCI-X 133	PCI-X 133	PCI-X 133
PCI-X Mode 2	PCI-X 266		PCI-33	a) PCI-33 b) PCI-66	PCI-X 66	PCI-X 133	PCI-X 266	PCI-X 266
	PCI-X 533		PCI-33	a) PCI-33 b) PCI-66	PCI-X 66	PCI-X 133	PCI-X 266	PCI-X 533



Most popular cases

Figure 11-2: Interoperability matrix.

CHAPTER 12

PCI-X Protocol

The basic thrust of PCI-X is to improve performance through enhancements to the PCI protocol. The objective is to make the transfer of large blocks of data more efficient. These enhancements include:

- Provision for registered outputs and inputs such that data can be clocked directly out of a register and directly into a register. This requires that a device be allowed up to two clocks to respond to a change.
- Tighter restrictions on wait states for both the initiator and target, and rules about when a transaction is allowed to disconnect.
- PCI-X transactions include information about the identity of the initiator of a transaction as well as the total number of bytes remaining to be transferred. This allows for...
- The *Split Transaction*, which replaces the Delayed Transaction of PCI. A target that can't complete a transaction within the maximum latency responds with a *Split Completion*, effectively telling the initiator "I can't do this right now, I'll get back to you." Later, when the target is ready, it initiates the completion transaction back to the original initiator using information provided in the attribute phase of the original request. When executing a Split Completion, the target is referred to as the *Completer*.

Nevertheless, the fundamental protocol rules remain essentially the same. The essential nature and meaning of the "five brothers," **FRAME#**, **DEVSEL#**, **IRDY#**, **TRDY#** and **STOP#**, remains with only a few minor changes to accommodate the protocol enhancements.

PCI-X Transactions

Comparing Conventional PCI with PCI-X

Figures 12-1 and 12-2 illustrate respectively a typical PCI write transaction and the corresponding PCI-X transaction. Let's first review the PCI write transaction.

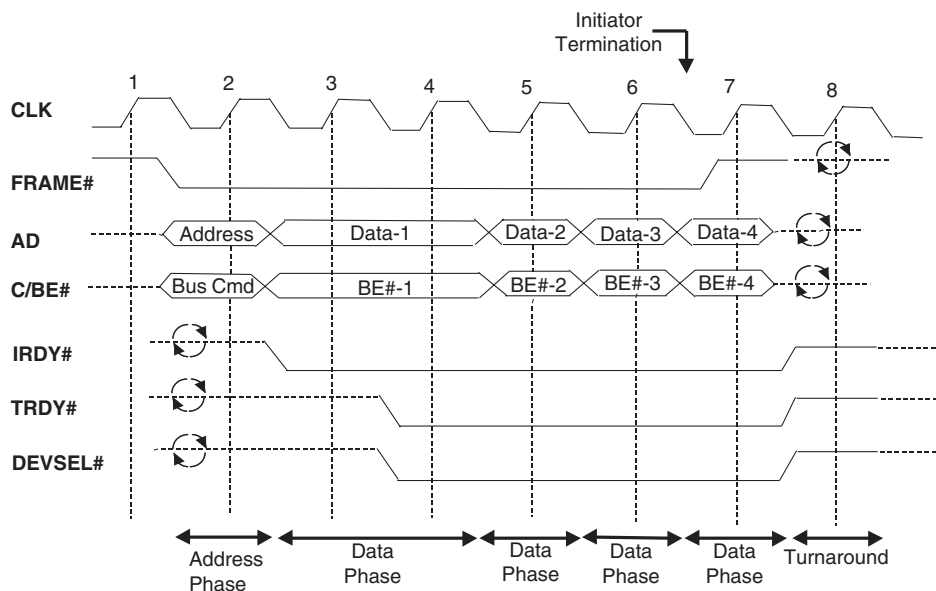


Figure 12-1: Conventional PCI write transaction.

Clock

- 1 The bus is idle and most signals are tri-stated. The initiator for the upcoming transaction has received GNT# and asserts FRAME#. It also places the target address on the AD bus and the bus command on the C/BE bus.
- 2 Address Phase. All targets detect the beginning of a transaction and sample the address on the AD bus and the command on the C/BE bus. One target recognizes that it has been selected.
- 3 The initiator may now place write data on the AD bus and assert the corresponding byte enables on the C/BE bus along with IRDY#.
- 4 The target requires an extra clock to respond by asserting DEVSEL# and TRDY#. In this case both IRDY# and TRDY# remain asserted for the entire transaction resulting in four data phases in four contiguous clocks.

- 7 The initiator negates **FRAME#** signifying that this is the last data phase. The initiator must leave **IRDY#** asserted for at least one clock cycle beyond the negation of **FRAME#** to guarantee the turnaround cycle between transactions.
- 8 The target negates **DEVSEL#** and **TRDY#**. The initiator negates **IRDY#**. This is the turnaround cycle that guarantees no contention on the bus. Note that because this is a write transaction, no turnaround cycle is required between the address phase and the first data phase.

Now, let's compare this with the corresponding PCI-X transaction illustrated in Figure 12-2.

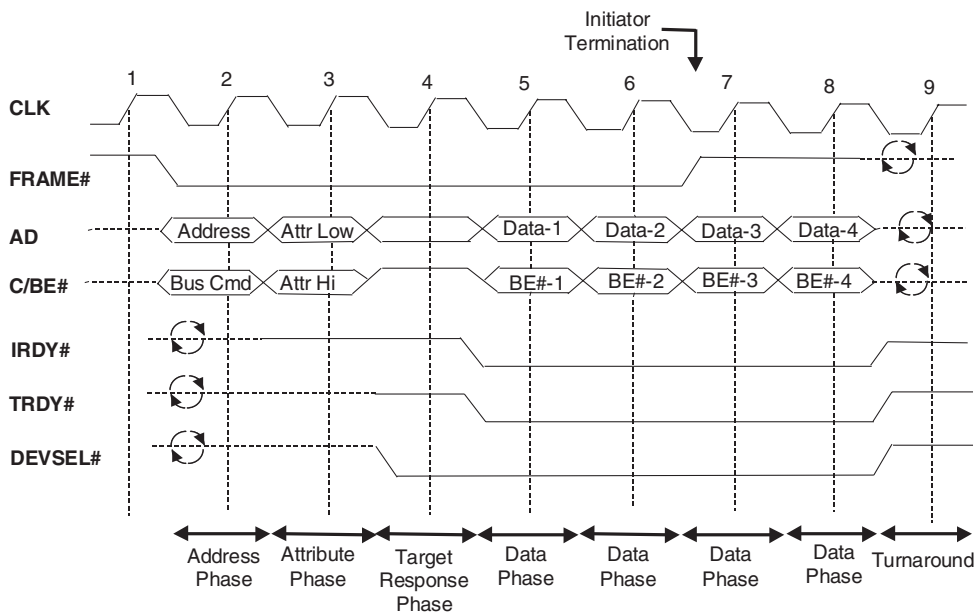


Figure 12-2: PCI-X write transaction.

Clock

- 1 The bus is idle as before, and the initiator begins the transaction by placing the target address on the **AD** bus, the command on the **C/BE#** bus and asserting **FRAME#**.
- 2 Address Phase, same as before.
- 3 Attribute Phase. Here's the first difference between PCI and PCI-X. The Attribute Phase is always the clock immediately following the Address Phase.

It provides additional information about the transactions, primarily the identity of the initiator.

- 4 Target Response Phase. The time required for the target to claim the transaction by asserting **DEVSEL#**. This may be one or more clock cycles. The Target Response Phase also exists in the PCI protocol but is not explicitly identified.
- 5 First Data Phase. The data phase protocol is the same as PCI, i.e., a data transfer occurs whenever both **IRDY#** and **TRDY#** are asserted. The difference is that once data transfer has started, neither agent is allowed to insert wait states by negating its ready signal.
- 7 The initiator signals the end of the transaction by negating **FRAME#** *two* clock cycles before the end.
- 9 Turnaround cycle as in PCI.

Perhaps the most interesting thing to notice about this comparison is that the PCI-X transaction takes one clock cycle *more* to transfer the same amount of data. The PCI protocol takes seven clocks to do four data phases while the PCI-X protocol requires eight. But PCI-X is optimized for large block transfers where the one extra clock for the Attribute Phase is amortized over a large number of data phases. More significantly, performance is gained by restricting wait states and eliminating dataless Retry transactions in favor of the Split Transaction.

DEVSEL# Timing

The relationship of **DEVSEL#** to **FRAME#** remains essentially unchanged between PCI and PCI-X, but the terminology changes a little. Table 12-1 compares **DEVSEL#** timing and terminology for both PCI and PCI-X.

Notes for Table 12-1

1. If the transaction uses a dual address cycle, Decode Speed is measured from the second Address Phase.
2. Decode A is not supported in ECC mode to allow time to check the ECC bits of the Address.

In deference to the higher clock frequency, PCI-X allows an extra clock cycle for subtractive decoding.

Table 12-1: DEVSEL# Timing.

Decode Speed (clocks after Address Phase) ¹	PCI	PCI-X
1	Fast	Not Supported (Attribute Phase)
2	Medium	Decode A ²
3	Slow	Decode B
4	Subtractive	Decode C
5	Master Abort	N/A
6	N/A	Subtractive
7	N/A	Master Abort

PCI-X Commands

Table 12-2 lists PCI-X commands along with the corresponding PCI command for the same C/BE encoding. As in PCI, initiators must not generate commands identified as “Reserved,” and targets must not respond to them by asserting DEVSEL#.

Notes for Table 12-2

1. For all commands except Dual Address Cycle, the command appears on C/BE[3:0]# during the address phase. For transactions with dual address cycles, C/BE[3:0]# contains the Dual Address Command during the first address cycle and the transaction command during the second address cycle. For 64-bit transactions, C/BE[7:4]# contain the transaction command during both addresses phases.
2. This command is reserved by the PCI SIG for future use. Current initiators must not generate the command. Current targets must treat it as if it were a Memory Read Block.
3. This command is reserved by the PCI SIG for future use. Current initiators must not generate the command. Current targets must treat it as if it were a Memory Write Block.
4. These commands require different timing. See Chapter 13.

Table 12-2: PCI vs. PCI-X Command Encoding.

Encoding	PCI Command	PCI-X Command	Length	Byte Enable Usage	Notes 1
0000b	Interrupt Acknowledge	Interrupt Acknowledge	DWORD	attr	
0001b	Special Cycle	Special Cycle	DWORD	attr	
0010b	I/O Read	I/O Read	DWORD	attr	
0011b	I/O Write	I/O Write	DWORD	attr	
0100b	Reserved	Reserved	N/A	N/A	
0101b	Reserved	Device ID Message	Burst	none	
0110b	Memory Read	Memory Read DWORD	DWORD	attr	
0111b	Memory Write	Memory Write	Burst	data phase	
1000b	Reserved	Alias to Memory Read Block	Burst	none	2
1001b	Reserved	Alias to Memory Write Block	Burst	none	3
1010b	Configuration Read	Configuration Read	DWORD	attr	4
1011b	Configuration Write	Configuration Write	DWORD	attr	4
1100b	Memory Read Multiple	Split Completion	Burst	none	
1101b	Dual Address Cycle	Dual Address Cycle	N/A	N/A	1
1110b	Memory Read Line	Memory Read Block	Burst	none	
1111b	Memory Write and Invalidate	Memory Write Block	Burst	none	

Burst vs. DWORD Transactions

PCI-X commands are divided into two categories based on transaction length. DWORD transactions are a single data phase conveying 32 bits or less of data. REQ64# must not be asserted. Burst transactions may be any length from 1 to 4096 bytes. The requested burst length is conveyed in the Attribute Phase. Burst transactions may be 64 bits wide. That is, the initiator may, at its option, assert REQ64#.

Burst transactions for which the initiator is the source of data are called *Burst Push* transactions. Burst Push transactions include the commands:

Memory Write

Memory Write Block

Alias to Memory Write Block

Split Completion

Device ID Message

Note that burst transactions only apply to memory space. I/O and configuration space transactions are always a single 32-bit data phase.

PCI-X treats Byte Enables differently than PCI. For DWORD transactions, the byte enable information is conveyed in the Attribute Phase rather than during the data phase as in PCI. For burst transactions, except for Memory Write, byte enables are not used at all. The transaction begins with the starting address and proceeds to, but not including, the starting address plus the byte count conveyed in the Attribute Phase. All bytes between the starting and ending addresses are considered valid. So, for example, if the starting address is 0x1002 and the byte count is 0x17, bytes 0x1000 and 0x1001 would not be transferred and the last byte transferred would be at address 0x1018.

The one exception is the Memory Write command. This is a burst transaction that nevertheless conveys byte enable information on the C/BE bus during the data phases in the same manner as PCI.

Allowable Disconnect Boundaries

In the conventional PCI protocol, a target is allowed to terminate the transaction (a *disconnect*) at any time by asserting **STOP#**. The initiator will respond by negating **FRAME#** at the earliest opportunity. However, in PCI-X, burst transactions are only allowed to terminate on an *Allowable Disconnect Boundary* (ADB), a naturally-aligned modulo 128 address. This restriction applies to the initiator as well as the target.

The target can signal disconnect, by asserting **STOP#**, at any time during the transaction, but data transfer continues to the next ADB. At that point, the initiator negates **FRAME#** and **IRDY#**, and the target negates **DEVSEL#**, **TRDY#** and **STOP#**.

The rationale behind the ADB is that devices like host bridges, which normally incorporate or make use of cache memory, operate more efficiently if transactions are aligned on cache-line boundaries. The Split Transaction allows a large block transfer to be split into multiple sub-transfers. Requiring that these sub-transfers be naturally aligned may result in substantial efficiency gains.

Another consequence of ADBs is that many transactions are atomic on PCI-X that are potentially non-atomic on PCI. Consider, for example, transferring a 64-bit pointer on a 32-bit bus. In PCI, this is potentially a non-atomic operation because the target could disconnect after the first DWORD. In PCI-X this transaction is atomic, provided that the 64-bit quantity does not cross an ADB.

That portion (or all) of a transaction or buffer that fits between two adjacent ADBs is called an *ADB Delimited Quantum (ADQ)*. For example, if a transaction starts between two ADBs, crosses one ADB, and ends before reaching the next ADB, the transaction includes two ADQs of data. Such a transaction fits in two buffers inside a device that divides its buffers on ADBs.

There are exceptions to the ADB requirement. The target is allowed to disconnect immediately after the first data phase regardless of where it occurs. This is called a *Single Data Phase Disconnect*. The target may also signal abort at any time, causing the transaction to terminate immediately.

Attributes and the Attribute Phase

Attributes convey additional information about a transaction. The Attribute Phase is always a single clock immediately following the Address Phase. Attributes are conveyed on AD[31:0] and C/BE[3:0]. The upper busses of 64-bit devices (AD[63:32] and C/BE[7:4]) are reserved and driven high during the Attribute Phase.

There are four different attribute formats—three for requesters and one for completers. This section describes the attribute formats for burst and DWORD transactions. The attribute format for Type 0 configuration transactions is described in Chapter 13. The completer attribute format is described later in this chapter under Split Transactions.

Figures 12-3 and 12-4 illustrate the attributes for a DWORD transaction and a burst transaction, respectively. Most of the attribute fields are used to convey the identity of the initiator and the byte count in the case of a burst transaction. This information is of value to the target if it chooses to split the transaction. Later, when the target completes the transaction, it will use this information to identify the original initiator.

The fields *Requester Bus Number* and *Requester Device Number* represent respectively the bus segment number and logical device number assigned to the requesting device at configuration time. By contrast, *Requester Function Number* is assigned by the

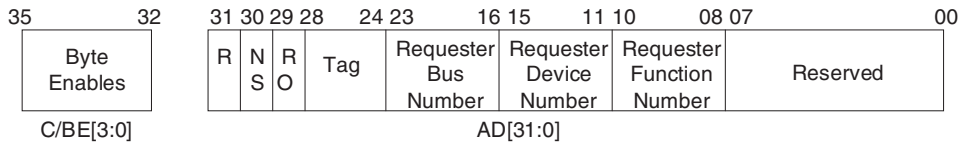


Figure 12-3: DWORD transaction attributes.

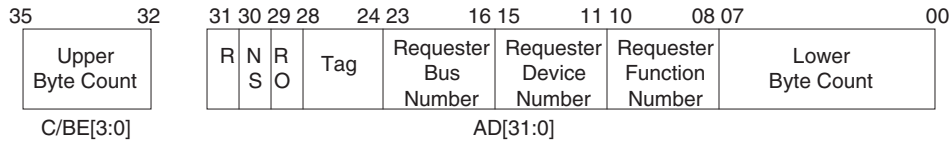


Figure 12-4: Burst transaction attributes.

device's designer and does not require any initialization. These three fields taken together become the *Requester ID*.

The *Upper Byte Count* and *Lower Byte Count* convey the number of bytes the initiator intends to transfer in a burst transaction. A value of zero represents 4096 bytes. It may take several transactions to transfer the total byte count because either the initiator or target disconnects the transaction prematurely. This collection of transactions is called a *Sequence*. On subsequent transactions in a *Sequence*, the initiator updates the byte count fields to reflect the number of bytes remaining to be transferred.

The Lower Byte Count field is reserved for DWORD transactions while the Upper Byte Count field, on the C/BE bus, conveys the byte enables for the subsequent single data phase.

The *Tag* field uniquely identifies up to 32 Sequences from a single initiator. Multiple sequences may be simultaneously in progress across the bus. The Tag field allows targets to identify the sequence to which a specific transaction belongs. It's also necessary for the Split Completion transaction.

Finally, there are three one-bit fields in the Attribute Phase:

- **R:** Reserved. Requesters must set this bit to 0. Completers should ignore it except for parity checking.
- **NS:** No Snoop. When a requester sets this bit, it is guaranteeing that the locations between the starting and ending addresses, inclusive, are not stored in any cache in the system. There is a potential performance advantage from

not snooping the cache if we know for certain that the subject locations are not cached. Note that PCI-X does not require cache coherency for addresses accessed by PCI-X requestors, but this bit can be useful in systems that do support it.

- **RO: Relaxed Ordering.** A requestor sets this bit to tell the target, and any intervening bridges, that it is not necessary to maintain strict ordering of this transaction sequence relative to other transactions. For a read transaction, this means that Split Completions need not stay in strict order with respect to writes moving in the same direction. A write transaction with this bit set need not stay in strict order with respect to other writes moving in the same direction. This attribute only applies to memory space transactions that are not Message Signaled Interrupts.

Split Transactions

Terminology

The agents participating in a conventional PCI transaction are designated as the *initiator* and the *target*. The initiator is the agent that initiates the transaction, the target is the agent selected by the initiator to respond, i.e., the target of the transaction. PCI-X introduces two new terms necessary to identify the agents participating in a Split Transaction. The *requester* is the agent that first introduces a transaction onto the PCI-X bus. When requesting the transaction, it is the initiator.

The *completer* is the agent that ultimately completes the transaction. During the initial request phase, the completer is the target. Later, when it is ready to complete the transaction, the completer becomes the initiator and the original requester becomes the target.

Split Response

A target that is unable to complete a transaction in a timely manner responds to the requestor with a *split response*, as illustrated in Figure 12-5. The target claims the transaction normally by asserting DEVSEL#. But, in what would normally be the first data phase with IRDY# and TRDY# asserted, the target *negates* DEVSEL#. Thus, no data transfer occurs, and the requester is notified that the target will complete the transaction later.

The target may terminate any transaction except memory write with a Split Response. That is, memory reads, interrupt acknowledge, all I/O and all configuration

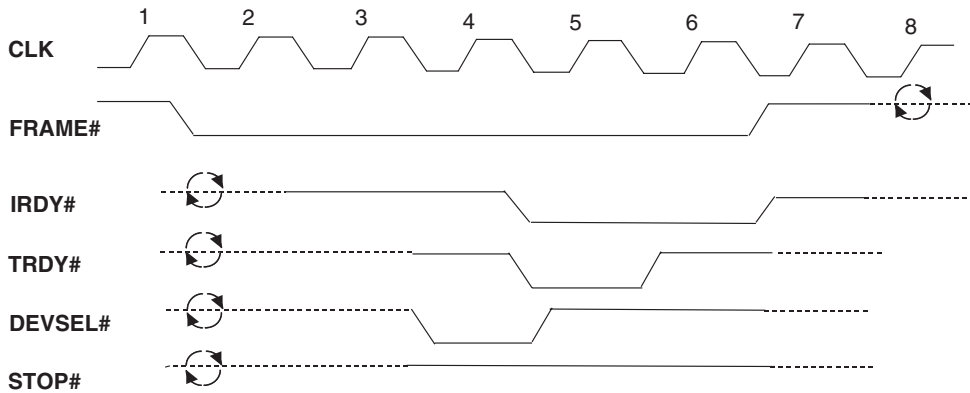


Figure 12-5: Split response.

transactions may be terminated with Split Response. For a read transaction, the target must drive the AD bus to all 1s.

Note, by the way, that the Split Transaction does not entirely eliminate the need for the Retry transaction termination. A device is designed to handle only so many Split Transactions simultaneously, perhaps only one. If it receives another request while in the midst of completing a previous request, it will respond with Retry.

Split Completion

When the completer is ready to complete the transfer, it initiates a transaction using the Split Completion command and identifies the requester in the address. Thus, the requester becomes the target of the split completion transaction.

Split Completion Address

Figure 12-6 shows the address format for a Split Completion transaction. The target of a Split Completion transaction is a specific PCI-X *device* and not an element in the memory address space. So the address, consisting of the Bus, Device and Function numbers of the original requester, is copied directly from the attribute phase of the transaction that was terminated by a Split Response.

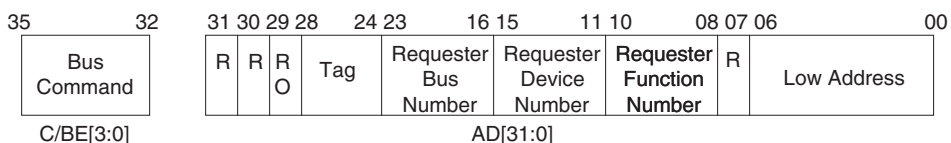


Figure 12-6: Split completion address.

The Tag field is also copied from the attributes of the original transaction. This identifies a particular transaction sequence and allows a requesting agent to have several split transactions outstanding. The Low Address field is copied from the address of the original requesting transaction, but only under certain circumstances:

- The request was a burst read
- This is the first Split Completion of a sequence and...
- This is not a Split Completion Message (see Split Completion Attributes below)

If the original request was a write or a DWORD read, the Low Address field is set to zero. The completer may choose to split the response to a burst read request into multiple transactions terminating on ADBs. In this case, the first Split Completion carries the low-order seven bits of the address, but subsequent responses have the Low Address field set to zero because those transactions always begin on a 128-byte boundary.

Finally, the Relaxed Ordering bit is copied from the attributes of the request transaction to the address of the Split Completion.

Split Completion Attributes

Figure 12-7 shows the format of the attribute phase for a Split Completion. The address information in the attribute phase—Bus, Device and Function numbers—always refers to the transaction initiator. So in the case of a Split Completion, it refers to the completer. The Byte Count fields are copied from the corresponding fields of the requesting transaction's attributes. However, if this is a response to a burst read, and the completer chooses to split the response into multiple transactions, the Byte Count is modified to reflect the actual number of bytes in this transaction. In this case, the completer also sets the Byte Count Modified (BCM) bit.

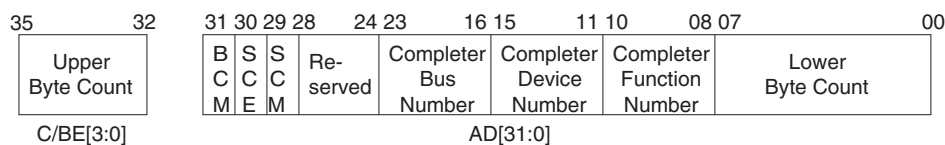
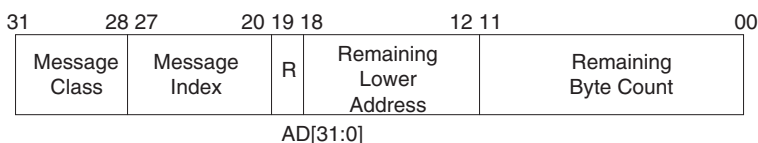


Figure 12-7: Completer attributes.

Requesters need not observe the Byte Count Modified bit. It is there primarily to aid devices that snoop transactions such as bus analyzers.

There are two other attribute bits that deserve special attention: *Split Completion Message* (SCM) and *Split Completion Error* (SCE). In response to a read request, the completer usually, but not always, replies with the requested data. But suppose the completer encounters an error condition that prevents the delivery of data. It needs to notify the requester of the error condition. Likewise, in response to a split write, either to I/O or Configuration space, the requester needs to know that either the transaction succeeded or there was an error. This is the purpose of the Split Completion Message.

Figure 12-8 shows the format of a Split Completion Message. Messages fall into classes identified by the *Message Class* field:



Within each class, the *Message Index* field identifies the exact message. For the Write Completion class there is only one index, 0. All others are reserved. The Completer Error class defines the following Message Indices:

If the byte count is out of range, the completer must initiate Split Completions with read data up to its upper address boundary and then disconnect. Then it replies with the Split Completion Message signaling Byte Count Out of Range. Note that a requester should be smart enough to know the address range of the target it selects. So, Byte Count Out of Range indicates a serious error condition in either the requester or the completer. This should be reported back to the requester's host.

For the Device-Specific Error class, the low four bits of the Message Index are available to the device to identify specific error conditions.

The Remaining Lower Address and Remaining Byte Count fields are useful only in burst read transactions and are primarily of value to PCI-X bridges to help them manage buffer space reserved for Split Transactions. The Remaining Lower Address field conveys the low seven address bits of the first byte *not* previously transferred in this transaction sequence. For DWORD transactions, this field is set to zero. The Remaining Byte Count field indicates the number of bytes that have not been transferred. For DWORD transactions, this field is set to four, even for Write Completion messages where in fact the data has been successfully transferred.

Error Checking and Correcting

The protocol enhancements described up until now were originally defined in Version 1.0 of the PCI-X Addendum. Devices that only support these features of PCI-X are said to operate in *Mode 1*. The remainder of this chapter describes protocol enhancements subsequently introduced in Version 2.0 of the Addendum. Devices that support these additional features are said to operate in *Mode 2*.

Parity Mode

All PCI-X devices are required to generate and check parity in essentially the same manner as conventional PCI. To briefly summarize the parity requirements:

- Even parity is computed across the AD and C/BE# busses and driven on PAR (AD[31::0] and C/BE[3::0]) and PAR64 (AD[63::32] and C/BE[7::4] for 64-bit devices) by the device that drives the AD bus.
- PAR and PAR64 are valid one clock cycle after the cycle for which parity is computed.
- The device checking parity does so in the clock cycle after the cycle in which PAR and PAR64 are valid.

- A device that detects a parity error in the Address or Attribute phases of a transaction asserts **SERR#** two clock cycles after **PAR** and **PAR64** are valid and sets the Detected Parity Error bit in its Status Register.
- A device that detects a parity error for a data phase asserts **PERR#** (if enabled) two clock cycles after **PAR** and **PAR64** are valid.

ECC (Error Correction Code) Mode

PCI-X Version 2.0 introduced an optional 7- or 8-bit Error Correction Code. Seven bits are used for a 32-bit bus and eight bits are necessary for a 64-bit bus. ECC is optional when PCI-X is operating in Mode 1 and required when it is operating in Mode 2. The ECC covers the **AD** bus during all phases of all transactions and the **C/BE** bus during the Address and Attribute phases of all transactions and all phases of Memory Writes.

Generating the ECC Value¹

The ECC can correct a single bit error and detect a double bit error. It can also detect triple or quadruple errors within a single “nibble” (4 bits). The bits covered by the ECC, **AD** bus and perhaps **C/BE** bus, together with the calculated ECC value are called a “codeword.” A valid codeword (one in which all bits are correct) will have at least three “1” bits and odd parity, and at least two “0” bits. This is useful in detecting situations where all bits are stuck at 0 or stuck at 1.

Each bit on the **AD** and **C/BE** busses has a unique ECC “signature” associated with it as shown in Table 12-3 for a 32-bit bus. Fundamentally, the 7-bit ECC value is computed by XORing the signatures for each bit in the data value that is a “1.”

Example:

One data phase of a burst memory read transaction contains the value 19h on the **AD** bus. So, **AD** bits 4, 3 and 0 are “1.” Taking the ECC signatures for those bits:

AD [4]	68h
AD [3]	4Ch
AD [0]	49h

¹ This is a brief summary of the ECC algorithm. The full description in the specification covers 20 pages.

Table 12-3: Bit-wise ECC Signatures.

Line	Code	ECC Check Bits E6-E0							Line	Code	ECC Check Bits E6-E0						
		6	5	4	3	2	1	0			6	5	4	3	2	1	0
AD[31]	2Ch		X		X	X			AD[15]	45h	X				X		X
AD[30]	4Ah	X				X		X	AD[14]	51h	X		X				X
AD[29]	1Ah			X		X		X	AD[13]	43h	X					X	X
AD[28]	29h		X		X			X	AD[12]	61h	X	X					X
AD[27]	5Eh	X		X		X	X	X	AD[11]	25h		X			X		X
AD[26]	6Bh	X	X			X		X	AD[10]	31h			X	X			X
AD[25]	2Ah		X			X		X	AD[9]	13h				X		X	X
AD[24]	3Bh		X	X		X		X	AD[8]	5Bh	X			X		X	X
AD[23]	64h	X	X				X		AD[7]	46h	X				X	X	
AD[22]	26h		X			X	X		AD[6]	32h			X	X		X	
AD[21]	3Eh		X	X		X	X	X	AD[5]	23h			X			X	X
AD[20]	15h			X			X		AD[4]	68h	X	X			X		
AD[19]	34h		X	X			X		AD[3]	4Ch	X			X	X		
AD[18]	54h	X		X			X		AD[2]	52h	X			X			X
AD[17]	37h		X	X			X	X	AD[1]	62h	X	X				X	
AD[16]	6Eh	X	X			X	X	X	AD[0]	49h	X				X		X
C/BE[3]	3Dh		X	X		X	X	X	C/BE[1]	58h	X			X	X		
C/BE[2]	19h			X		X		X	C/BE[0]	5Dh	X			X		X	X

And XORing those three values, we get 6Dh. This is the value that goes on the seven ECC lines.

It would be easy enough to devise a software algorithm that would sequentially scan the AD bus and XOR into the ECC value the signature for each bit that is set to “1.” But PCI-X requires that the ECC value be computed in *one clock cycle*. For each of the seven columns in Table 12-3 representing an ECC check bit, the X’s in that column represent inputs to a “tree” of 2-input XOR gates. Note that no column has more than 19 X’s. So, each of the seven ECC check bits is generated by a tree of 2-input XOR gates. The specification claims that this algorithm, covering AD[31::0] and C/BE#[3::0] and eliminating common sub-expressions, can be implemented in fewer than 80 gates.

Checking the ECC Value

The device checking the ECC value does the exact same computation over the AD bus and, if required, the C/BE bus. The resulting check bits are XOR’ed with the check bits in the codeword and the result is called the *syndrome*. If the syndrome is zero, the protected data is correct. If the syndrome is non-zero and it matches one of the signatures in Table 12-3, it represents a correctable single-bit error. Simply invert the bit with the matching signature and the data is correct.

The check bits themselves are also potentially correctable. If the syndrome contains a single “1,” it represents a single-bit error in the corresponding check bit. Finally, if the syndrome has more than one “1” and does not match the signatures in Table 12-3, it represents a non-correctable multiple bit error.

64 Bits and Phase Protection

The strategy outlined above can be extended to 64 bits with another table of signatures for AD[63::32] and C/BE#[7::4]. This table has eight check bits where the eighth bit is the XOR of all 36 bits in the table, i.e., every entry in the eighth column has an X.

But there’s more. Each phase of a transaction has an ECC signature that is combined with the signatures for the data bits. There are six different signatures for data phases depending on the Address phase value of AD[6::3]. This is a way of detecting that the two agents in a transaction have gotten out of sync with each other. A syndrome that matches a phase protection signature is uncorrectable.

Implementing ECC

Timing for ECC mode is essentially the same as for parity mode. The computed signature appears on the ECC bus, ECC[7::0] one clock cycle after the phase for which it is computed. The syndrome is computed by the receiving agent one clock cycle after that. If the syndrome reveals an uncorrectable error, either PERR# or SERR# are asserted in the following clock cycle according to the same rules as for parity mode.

Of the eight ECC check bits, four are allocated from previously reserved pins (see Appendix B) and four share pins with existing uses as follows:

ECC[0]	PAR
ECC[1]	ACK64#
ECC[6]	REQ64#
ECC[7]	PAR64

It makes sense to reassign PAR and PAR64 since we simply don’t report parity on the bus in ECC mode. But REQ64# and ACK64# are another issue. PCI-X supports the conventional PCI policy of dynamically negotiating 64-bit transfers but in ECC mode the protocol is a little different. ECC data is not valid until the second clock of a PCI-X transaction. REQ64# can be asserted by a 64-bit initiator during the (first)

Address phase but must be released in the next clock cycle so it can be used for ECC data. This means that targets must latch REQ64# during the first clock cycle.

ACK64# on the other hand, can't be shared because it is normally asserted by the target coincident with DEVSEL#. Here's where the protocol differs. A 64-bit target responds to REQ64# by asserting STOP# for one clock cycle at the same time it asserts DEVSEL#.

Source Synchronous Data Transfers

Perhaps the most significant new feature in Version 2.0 is the notion of *Source Synchronous Burst Push* transactions. Up to this point in the book, everything we've talked about is synchronized by the clock signal. PCI-X 2.0 refers to this as *common clock sampling*. One obvious consequence of common clock sampling is that there is exactly one data transfer per clock cycle. At 133 MHz and 64 bits, that amounts to roughly one Gbyte/sec transfer rate. The way PCI-X gets to 2 Gbytes and 4 Gbytes is by introducing multiple data *subphases* within each clock cycle as illustrated in Figure 12-9. PCI-X 266 has two data subphases per clock cycle while PCI-X 533 has four.

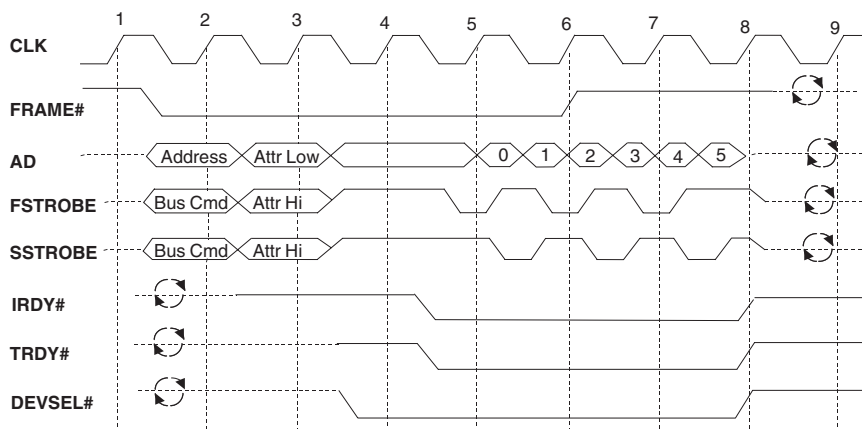


Figure 12-9: Source synchronous burst write.

Source synchronous transfers apply only to burst-push transactions, i.e., burst writes, where the initiator generates strobe signals used by the target to clock the data subphases. All other transactions in Mode 2, as well as the Address and Attribute phases of burst-push transactions, use common-clock sampling. During a source synchronous transfer, the C/BE# signals are reassigned as pairs of strobes during data

phases (see Table 12-4). **FSTROBE** (first strobe) and **SSTROBE** (second strobe) are 180 degrees out of phase such that the first data subphase is clocked by the rising edge of **FSTROBE** and the second subphase is clocked by the rising edge of **SSTROBE**.

Table 12-4: Source Synchronous Strobe Allocation.

Group Number	Data Strobe	Connector Signal Name	Source Synchronous Signaling Group
0	FSTROBE[0]	C/BE [0]#	AD[15::0], PAR/ECC[0] ACK64#/ECC[1], REQ64#/ECC[6]
	SSTROBE[0]	C/BE [1]#	
1	FSTROBE[1]	C/BE [2]#	AD[31::16], ECC[5::2]
	SSTROBE[1]	C/BE [3]#	
2	FSTROBE[2]	C/BE [4]#	AD[47::32]
	SSTROBE[2]	C/BE [5]#	
3	FSTROBE[3]	C/BE [6]#	AD[63::48], PAR64/ECC[7]
	SSTROBE[3]	C/BE [7]#	

The initiator uses a phase-locked loop to generate an internal clock at some multiple of the PCI bus clock, at least four. Both strobes are driven high during the clock cycle after the Attribute phase. The first falling edge of **FSTROBE** occurs nominally 3/4 of a clock cycle from the rising clock edge following the Attribute phase. The first falling edge of **SSTROBE** occurs a half clock cycle later. Meanwhile, data is driven on the **AD** bus such that it is stable at the rising edges of **FSTROBE** and **SSTROBE**.

In PCI-X 266, the strobes toggle at the same frequency as the bus clock, 133 MHz, while the **AD** bus toggles at twice the clock frequency. In PCI-X 533, the strobes toggle at twice the bus clock frequency and the **AD** bus toggles at four times the clock frequency. In PCI-X 533, **FSTROBE** clocks the first and third data subphases while **SSTROBE** clocks the second and fourth subphases.

When operating in Mode 2, Category 1 signals use 1.5V signaling (see Chapter 14). In Mode 1 or conventional PCI mode Category 1 signals use 3.3V signaling. All other signals always use 3.3V signaling.

Device ID Messages

Conventional PCI has no means for one device/function to communicate explicitly with another device/function other than in Configuration Space, which serves a very specific and limited function. The Device ID Message (DIM) command provides a

mechanism for an initiator to send an arbitrary message directly to a target device/function outside of the address spaces—Memory, I/O, Configuration—defined for the bus. The target of a DIM command is called the Completer.

Device ID Messages are burst-push transactions. The initiator is the source of the data. The message may be any length up to 4096 bytes.

Support of the DIM command is optional for non-bridge PCI-X devices. If a device that does not support the DIM command is the target of a Device ID Message, it doesn't respond and the transaction terminates with Master Abort.

Address/Attributes

Figure 12-10 shows the layout of the Address phase of a DIM command. *Message Class* identifies one of 16 message types. Currently the value zero is called “Vendor Defined,” and all other values are reserved by the PCI SIG². The field *Class Specific* is available to further classify a message. If the message has a 64-bit address, then the upper 32 bits are also part of the Class Specific field. The Completer address fields identify the target of this message. The remaining 2 bits will be dealt with below.

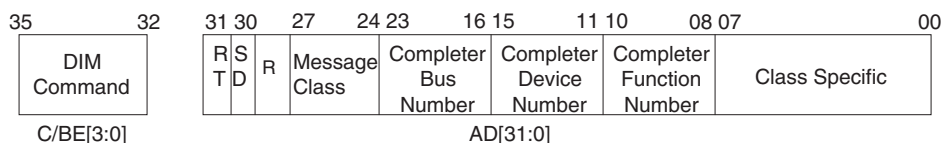


Figure 12-10: Device ID message address.

Figure 12-11 shows the layout of the Attribute phase of a DIM command. It is almost identical to the Attribute phase of other burst transactions (see Figure 12-4). Bit 30, the No Snoop bit in the burst attribute is reserved here. Bit 31 that is reserved in the burst attribute is now defined as *Initial Request*. It is set if this is the first transaction of a sequence and cleared if this is the continuation of a sequence previously disconnected by either the requestor, completer or an intervening bridge.

² The lack of any defined message types gives the impression that Device ID Messaging is a solution in search of a problem. It's probably a useful tool, but no one it seems has an immediate application for it.

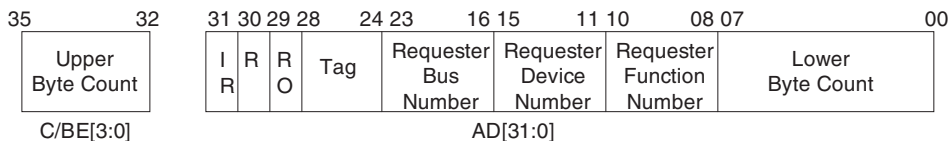


Figure 12-11: Device ID message attributes.

Explicit vs. Implicit Routing

Bit 31 of the Address phase sets the *route type* for this Device ID Message. Route type 0 means that the message explicitly addresses a device using the Completer address fields. Bridges route explicit Device ID Messages based on the Completer bus number as follows:

- If the Device ID Message appears on the primary bus segment and the Completer bus number is between the bridge’s secondary bus number and subordinate bus number inclusive, the bridge forwards the transaction downstream. This is like Configuration Space transactions.
- If the Device ID Message appears on the secondary bus segment, and the Completer bus number is *not* between the bridge’s secondary bus number and subordinate bus number inclusive, the bridge forwards the transaction upstream to the primary bus segment.

Implicit transactions are always routed upstream from the secondary bus segment to the primary bus segment until the transaction reaches the host bridge.

Forwarding only occurs if both the primary and secondary bus segments are operating in PCI-X mode. Mode 2 PCI-X bridges are required to forward Device ID Messages. Forwarding is optional in Mode 1 bridges.

Silent Drop

Bit 30 of the Address phase is called *Silent Drop*. If a Completer receives a Device ID Message with the Silent Drop bit set to 0 and the Message Class is unsupported or reserved, it responds with Target Abort. If Silent Drop is set to 1, the Completer accepts the entire message without complaint.

Likewise, if a bridge finds that it can’t forward a Device ID Message because the destination bus segment is operating in conventional PCI mode, if Silent Drop is 0 it responds with Target Abort. If Silent Drop is 1, the bridge accepts the message and throws it away.

16-bit Bus

Recognizing that PCI-X may be useful in embedded applications where size and cost are significant considerations, PCI-X 2.0 defines a 16-bit interface. There are a number of conditions and restrictions on the 16-bit interface:

- It only operates in Mode 2. Mode 2 devices are required to support the 16-bit bus.
- Devices designed exclusively for 16-bit embedded applications need not implement features that aren't required for Mode 2 operations such as parity generation. Category 1 signals on these devices need not support 3.3V signaling
- Bridges are not permitted to connect a 16-bit bus to a subordinate bus. Stated another way, bridges do not support the 16-bit bus on the primary or upstream side.
- Bus width is 16 bits only. There is no dynamic bus width negotiation protocol.
- As the 16-bit interface is optimized for embedded applications, no add-in card connector is specified and no status bit in Configuration Space defines this characteristic. Beyond the basic protocol, much of the 16-bit interface is implementation-defined.

In addition to the normal bus control signals, 16-bit PCI-X uses:

AD[31::16]

C/BE#[3::2]

ECC[5::2]

The protocol is quite simple. Everything that occurs in a single phase on a 32-bit bus requires two subphases or clock cycles on the 16-bit bus. The low-order 16 data bits and two low-order C/BE# bits are transferred in the first subphase, and the high-order bits are transferred in the second subphase. The 16-bit bus uses the same 7-bit ECC algorithm as the 32-bit bus with an additional bit for improved detection of uncorrectable errors.

Summary

PCI-X constitutes an extensive set of enhancements to the basic PCI protocol. These include:

- Revised command protocol emphasizing a distinction between DWORD and burst transactions.
- An Attribute phase in each transaction to identify the initiator of the transaction.
- Split transactions whereby a request is completely decoupled from the response. PCI-X Split Transactions replace conventional PCI's Retry.
- An error correction protocol that allows single bit errors to be corrected automatically.
- Source synchronous data transfers that yield up to 4 Gbytes of bandwidth. Source synchronous transfers implement either two or four data subphases per clock cycle.
- Device ID Messaging that allows a device to talk directly to another device outside of the normal bus address spaces. The target of such a transaction may be explicitly specified by bus, device and function number or implicitly as the host bridge.
- A 16-bit interface optimized for embedded applications.

The next chapter looks at configuration and initialization issues in PCI-X.

CHAPTER 13

PCI-X Configuration and Initialization

The PCI-X protocol has a number of implications for configuration transactions and how a system is initialized.

Configuration Transaction Timing

In most PCI implementations, a device's IDSEL signal is connected to one of the upper AD lines on the backplane. To minimize DC loading on the AD line, the connection is made through a series resistor. The downside is that the capacitive load of the IDSEL input must be charged through this resistor and that takes a finite time. With PCI-X running at 133 MHz, it can take more than one clock for IDSEL to be recognized as asserted.

PCI-X deals with this by requiring that the initiator of a configuration transaction assert the address on the AD bus four clocks *before* asserting FRAME#. This is illustrated in Figure 13-1.

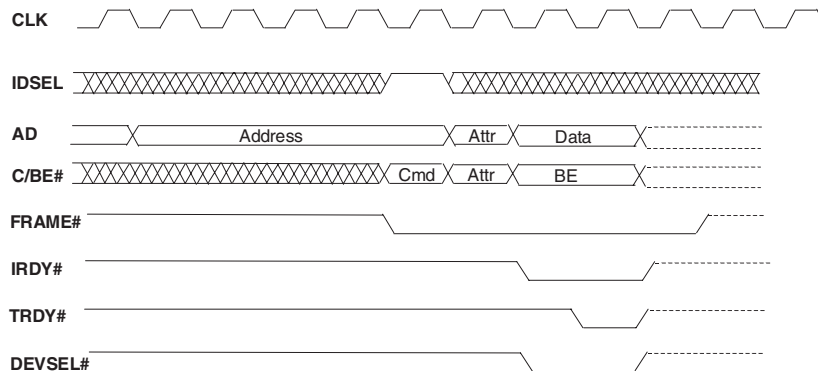


Figure 13-1: PCI-X configuration transaction timing.

The initiator is allowed to assert the address on the AD bus as soon as it recognizes its GNT# signal asserted. However, until it asserts FRAME# four clocks later, the bus is technically idle. Meanwhile, the arbiter may remove GNT# in favor of a higher priority device. If that happens, if for example GNT# is removed before clock *n*, the current initiator must float the AD bus within two clocks of seeing GNT# negated and try again.

Bus segments capable of Mode 2 operation are not allowed to connect IDSEL to an AD line because they use different signaling protocols. The bridge on a Mode 2 bus segment must provide separate connections for IDSEL.

Configuration Address

In conventional PCI, there is no need for a device to know its physical address, i.e., its device and bus numbers. These numbers are relevant only to the configuration software. However, PCI-X requires that a device convey its physical address in the attribute phase of any transaction that it initiates. The registers that hold this physical address are “hidden,” that is, they are not explicitly visible in Configuration Space (but see below, PCI-X Status Register). Instead they are loaded automatically from information in the address and attribute phases of every configuration transaction.

In conventional PCI, there are two “types” of configuration transaction—Type 0 and Type 1. A Type 1 configuration transaction crosses one or more bridges until it reaches the bus segment conveyed in the address. The bridge then converts it to a Type 0 transaction as shown in Figure 13-2a. PCI-X also has Type 0 and Type 1 configuration transactions, which are converted according to the same strategy. The difference, as shown in Figure 13-2b, is that the Device Number carries over into the Type 0 address so that it can program the target device’s Device Number register. The Device Number register is programmed each time a device is addressed by a Type 0 configuration transaction.

There’s one more wrinkle. PCI-X Mode 2 increases the Configuration Space to 4096 bytes per function (see below, Mode 2 Configuration Space). The additional four bits for the configuration register address are carried in bits 24 to 27 of the Configuration Address as shown in Figure 13-2c. These are passed through in the Type 1 to Type 0 translation, leaving only 8 bits to be used to drive IDSEL. But Mode 2 system boards are not allowed to tie IDSEL to an AD bus line. The bridge must use separate drivers for IDSEL, so the third column in Table 13-1 is of no consequence anyway.

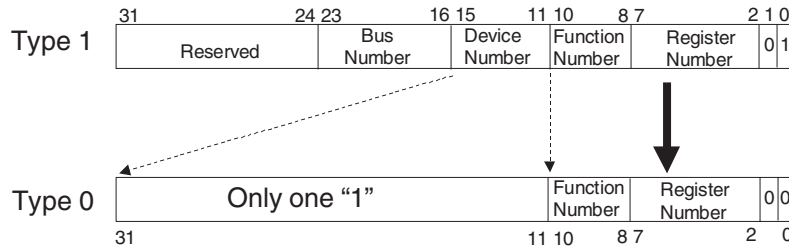


Figure 13-2a: Converting PCI configuration address from Type 1 to Type 0.

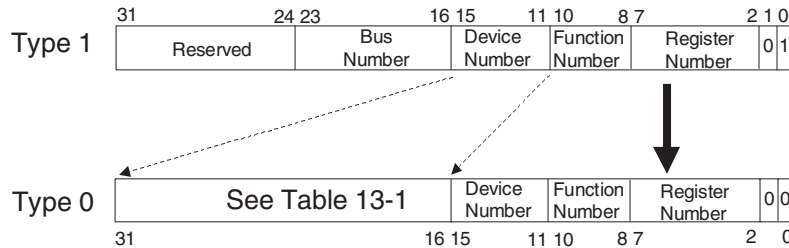


Figure 13-2b: Converting PCI-X Mode 1 configuration address from Type 1 to Type 0.

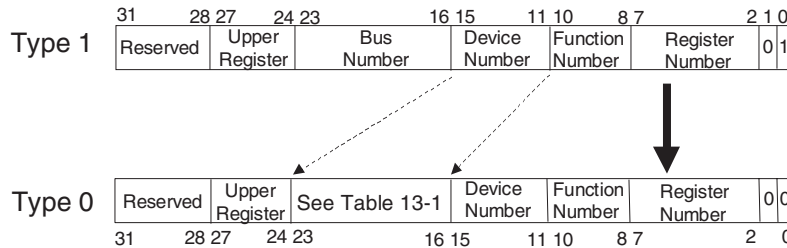


Figure 13-2c: Converting PCI-X Mode 2 configuration address from Type 1 to Type 0.

The PCI specification does not define a mapping between the Device Number field and the upper AD bits used to drive IDSEL. That is left up to the system implementer. PCI-X, however, defines a specific mapping as shown in Table 13-1.

Table 13-1: Mapping device number to upper AD lines.

Device Number	AD[31::16] Mode 1	AD[23::16] Mode 2
0	0000 0000 0000 0001b	0000 0001b
1	0000 0000 0000 0010b	0000 0010b
2	0000 0000 0000 0100b	0000 0100b
3	0000 0000 0000 1000b	0000 1000b
4	0000 0000 0001 0000b	0001 0000b
5	0000 0000 0010 0000b	0010 0000b
6	0000 0000 0100 0000b	0100 0000b
7	0000 0000 1000 0000b	1000 0000b
8	0000 0001 0000 0000b	0000 0000b
9	0000 0010 0000 0000b	0000 0000b
10	0000 0100 0000 0000b	0000 0000b
11	0000 1000 0000 0000b	0000 0000b
12	0001 0000 0000 0000b	0000 0000b
13	0010 0000 0000 0000b	0000 0000b
14	0100 0000 0000 0000b	0000 0000b
15	1000 0000 0000 0000b	0000 0000b
16 to 31	0000 0000 0000 0000b	0000 0000b

Configuration Attributes

The format of the Attribute phase of a Type 0 configuration transaction differs from the normal format as shown in Figure 13-3. The difference is the use of bits 0 to 7 as the Secondary Bus Number. This is the bus number of the segment on which the target device resides and is used to program the target's Bus Number register.

If the configuration transaction originated on the same bus segment as the target, the Secondary Bus Number and Requester Bus Number are the same. However, if the transaction originated on another bus segment as a Type 1 configuration address, the PCI-X bridge that translates the Type 1 address to Type 0 generates the Secondary Bus Number field. In this case, the Requester Bus Number and Secondary Bus Number fields are different.

Note, by the way, that the No Snoop (NS) and Relaxed Ordering (RO) bits are declared as Reserved because they only apply to memory space transactions.

The Attribute phase of a Type 1 configuration transaction is identical to the DWORD format shown in Figure 12-3.

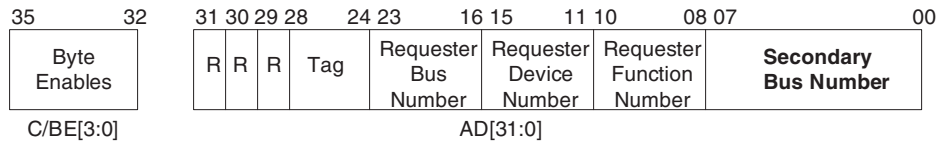


Figure 13-3: Type 0 configuration transaction attributes.

Configuration Header Registers

The 64-byte configuration header defined for conventional PCI remains essentially the same under PCI-X with a few minor modifications.

Command Register

Bit 2 Bus Master: Ignored when initiating a Split Completion transaction.

Bit 4 Memory Write and Invalidate Enable: Ignored. PCI-X doesn't support the Memory Write and Invalidate command.

Bit 6 Parity Error Response: In parity mode, this bit behaves as in conventional PCI. In PCI-X ECC mode, it regulates the assertion of **PERR#** in response to an uncorrectable data error.

Bit 9 Fast Back-to-Back Enable: Ignored. PCI-X doesn't use fast back-to-back timing.

Bit 10 Interrupt Disable: While this bit and the Interrupt Status bit in the Status Register are considered optional, the PCI-X spec says that these bits are "recommended" for Mode 1 devices and "required" for Mode 2 devices. If one of these bits is implemented, then they both must be¹.

Status Register

Bit 3 Interrupt Status: See description of Bit 10 in the Command Register above.

Bit 4 Capabilities List: PCI-X devices, of necessity, include the PCI-X Capabilities item and so this bit must be set to 1.

Bit 7 Fast Back-to-Back Capable: Not meaningful in PCI-X mode because PCI-X devices don't utilize fast back-to-back timing.

Bit 8 Data Parity Error Detected: and,...

¹ The spec doesn't specifically say so, but I assume this means the bits are required if the device is capable of asserting an **INTx#** pin.

Bit 15 Detected Parity Error: In parity mode, these bits behave identically to conventional PCI. In ECC mode, their behavior is essentially the same but in response to uncorrectable data errors.

Bits 9 and 10 DEVSEL Timing: Only describes conventional PCI DEVSEL timing. PCI-X devices may determine their DEVSEL timing based on clock rate and operating mode.

Base Address Registers

All Base Address registers that request memory resources (except the Expansion ROM Base Address register) must support 64-bit addressing using the method defined in conventional PCI. The Prefetchable bit must be set unless the range contains locations with read side effects or locations in which the device does not tolerate write merging. The minimum memory address range requested by a Base Address register is 128 bytes. To conserve address space, the specification recommends that devices request an address range no larger than the smallest integral power of two that is larger than the range actually used by the device.

Latency Timer Register

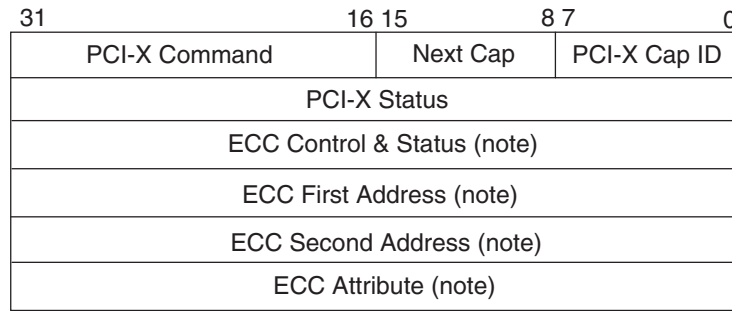
The default value of the Latency Timer in PCI-X mode is 64.

PCI-X Capabilities List Item

Devices that are capable of PCI-X operation, whether or not they're operating in PCI-X mode, include additional control and status registers in the Capabilities List area of Configuration Space. Figure 13-4 shows the layout of the PCI-X Capabilities Item. The Capability ID is 7. Next Cap is a pointer to the next capability item in the list. The ECC registers are only present if the device indicates that it supports ECC either in Mode 2 or both Modes 1 and 2.

PCI-X Command Register

This combination of read-only and read-write bits controls various functions and features of PCI-X. Think of it as an extension of the Command Register in Configuration Header Space. Figure 13-5 shows the layout of the Command Register.



Note: These registers only appear in versions 1 and 2 of the PCI-X Capability. See the PCI-X Command Register description

Figure 13-4: PCI-X capability structure.

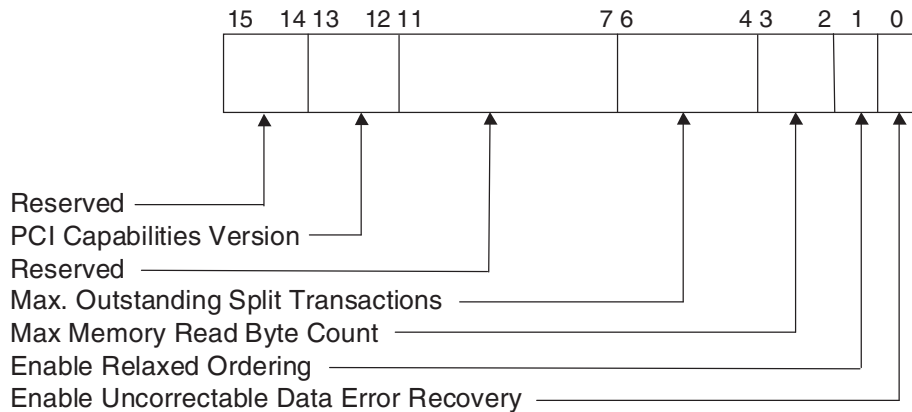


Figure 13-5: PCI-X Command register.

Bit

- 0 The device driver sets this bit to 1 to allow the device to attempt to recover from an unrecoverable data error. If this bit is 0 in PCI-X mode, the device asserts **SERR#** (if enabled) whenever Status Register bit 8 (Data Parity Error) is asserted. Value after **RST#** = 0.

- 1 When set, this allows the device to set the Relaxed Ordering bit in the Requester Attributes of transactions it initiates if strict ordering isn't required. This bit is permitted to be read-only and set to 0 for devices that never allow relaxed ordering. Value after **RST#** (if writable) = 1.

- 3-2 Sets the maximum byte count the device is allowed to use when initiating burst memory read transactions. System configuration software can use these bits to tune system performance. Generally speaking, device drivers should not modify this value.

<i>Value</i>	<i>Maximum Byte Count</i>
0	512
1	1024
2	2048
3	4096

Value after RST# = 0.

- 6-4 Sets the maximum number of Split Transactions the device is allowed to have outstanding at one time as a requester. System configuration software can use these bits to tune system performance. Generally speaking, device drivers should not modify this value.

<i>Value</i>	<i>Maximum Outstanding</i>
0	1
1	2
2	3
3	4
4	8
5	12
6	16
7	32

Value after RST# = Maximum number of Split Transactions the device is designed to have outstanding when the Maximum Memory Read Byte Count field is set to 0 (512 bytes).

- 13-12 Read-only. Indicates the format (size) of the PCI-X Capabilities List item and whether or not the device supports ECC in Mode 1.

Version	ECC Support	Capabilities List Item Size
0	none	8 bytes
1	Mode 2 only	24 bytes
2	Modes 1 and 2	24 bytes
3	reserved	reserved

PCI-X Status Register

Like its counterpart in Configuration Header Space, this register is a combination of read-only fields that convey additional information about the device's capabilities and read-write bits that indicate the device's current status where writing a 1 to a writable bit sets it to 0. Figure 13-6 shows the layout of the Status Register.

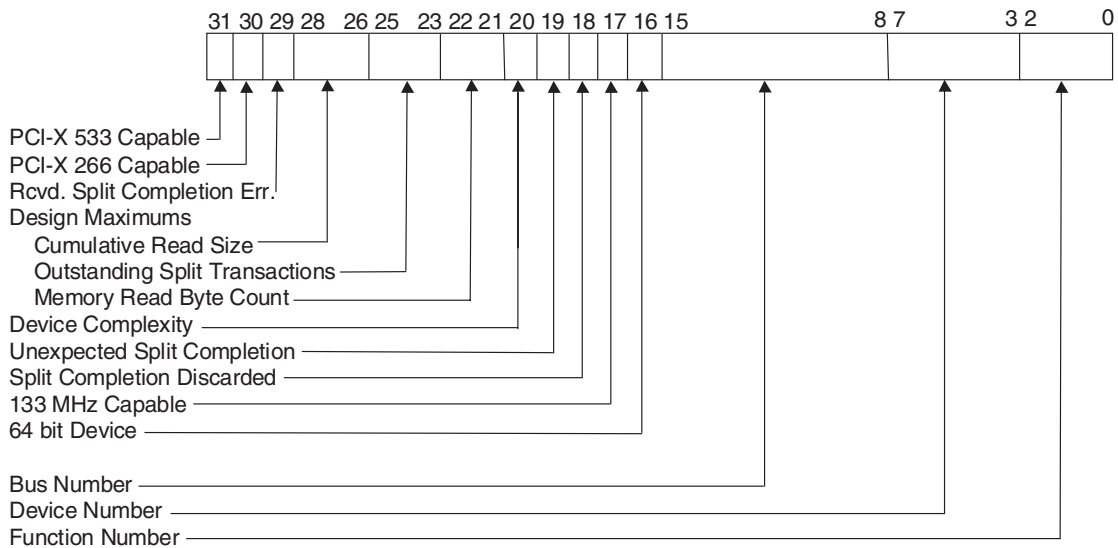


Figure 13-6: PCI-X Status Register.

The low-order 16 bits are read-only versions of the bus number, device number and function number derived from the Configuration Transaction address and attributes. This information can be useful for debugging. The remainder of the Status Register is:

Bit

- 16 Read only. 1 means this device/function has a 64-bit data path all the way to the add-in card connector.

- 17 Read only. 1 means the device's maximum clock frequency is 133 MHz. 0 means the maximum clock frequency is 66 MHz.
- 18 Write 1 to clear. This bit is set if the device discards a Split Completion transaction that it initiated because the requester would not accept it. The requester may, for example, terminate the Split Completion transaction with Master Abort. Setting this bit asserts **SERR#** if enabled.
- 19 Write 1 to clear. This bit is set if the device is the target of a Split Completion transaction that it is not expecting. This may happen because one of the fields in the Split Completion was corrupted—the Requester ID, Tag, or Byte Count for example. If, for example, the Tag field doesn't match an outstanding request, the device has the option of ignoring it by not asserting **DEVSEL#** (Master Abort) or accepting and discarding the entire transaction and setting this bit.
- 20 Read only. 0 indicates a "simple device." 1 indicates a "bridge device." This distinction relates to whether or not a device posts write transactions. Simple devices don't post write transactions, bridge devices do.
- 22-21 Read only. This field indicates the maximum byte count the device is designed to use when initiating one of the burst memory read transactions. System configuration software may use this value in setting the Maximum Memory Read Byte Count field in the PCI-X Command Register.
- 25-23 Read only. This field indicates the maximum number of split transactions that the device is designed to have outstanding at one time as a requester. System configuration software may use this value in setting the Maximum Outstanding Split Transactions field in the PCI-X Command register².

Value	Maximum Outstanding
0	1
1	2
2	3
3	4
4	8
5	12
6	16
7	32

² The specification suggests that this field may be a function of the Maximum Memory Read Byte Count field in the PCI-X Command register and its value must track that field.

28-26 Read only. This field indicates the maximum cumulative size of all burst memory read transactions the device is designed to have outstanding at any one time as a requester³.

Value	Maximum Outstanding	
	ADQs	Bytes
0	8	1 KB
1	16	2 KB
2	32	4 KB
3	64	8 KB
4	128	16 KB
5	256	32 KB
6	512	64 KB
7	1024	128 KB

29 Write 1 to clear. This bit is set if the device receives a Split Completion Message with the Error attribute bit set.

30 Read only. 1 means the device is capable of 266 MHz Mode 2 operation. If this bit is set to 1, bit 17 (133 MHz capable) must also be 1.

31 Read only. 1 means the device is capable of 533 MHz Mode 2 operation. If this bit is set to 1, both bits 17 and 30 must also be 1.

The 64-bit Device and speed “Capable” bits are “intended for use by system management software to assist the user in identifying the best slot for an add-in card.” But in order to make such recommendations, the software must know the characteristics of the slots as well as the add-in cards. That’s beyond the scope of the specification.

ECC Registers

The remaining registers in the PCI-X Capability store information about a transaction in which an ECC error was detected. These registers only occur in versions 1 and 2 of the Capability structure. This information can be useful in identifying a

³ The specification suggests that this field may be a function of the Maximum Memory Read Byte Count field in the PCI-X Command Register and its value must track that field.

potentially failing device. Registers, or fields of registers, that capture data off the bus capture the uncorrected data directly off the bus even if the error can be corrected.

ECC Control & Status Register

Figure 13-7 shows the layout of the ECC Command and Status register. It contains a combination of read-only fields and write-1-to-clear fields.

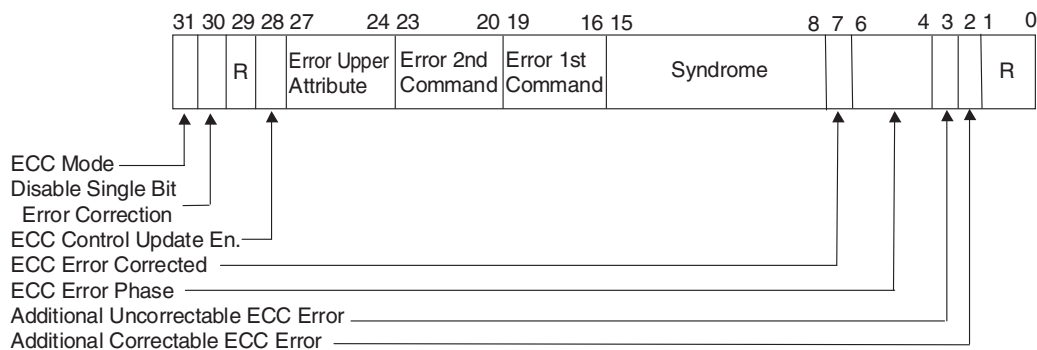


Figure 13-7: PCI-X ECC Control and Status register.

Bit

- 1-0 Reserved.
- 2 Write 1 to clear. Set if device detects a correctable ECC error while error correction is enabled and device is already indicating an error.
- 3 Write 1 to clear. Set if device detects an uncorrectable ECC error while error correction is enabled and device is already indicating an error or if it detects a correctable error while error correction is disabled.
- 6-4 Write 1 to clear. When the device detects an ECC error, correctable or not, this field indicates the phase in which the error was detected as follows:
 - 0 No error
 - 1 First 32 bits of address
 - 2 Second 32 bits of address
 - 3 Attribute phase
 - 4 32- or 16-bit data phase
 - 5 64-bit data phase
 - 6 Reserved

- 7 Reserved
- Writing a 1 to any bit in this field clears the whole field and enables the device to capture the next error.
- 7 Read only. Set to one if the Error Phase field is non-zero and the error was corrected. Zero if no error or error was uncorrectable.
- 15-8 Read only. ECC check bit value for the detected (and possibly corrected) error. Bit 8 corresponds to E0, bit 15 to E7.
- 27-16 Read only. These three fields record the contents of the C/BE# lines for different phases of the erroneous transaction.
- 19-16 First Address phase.
- 23-20 Second Address phase (if present).
- 27-24 Attribute phase.
- 28 Write transient, always reads as 0. If this bit is 1 in the pattern written to the ECC Control and Status register, then bits 30 and 31 are updated. If this bit is 0, bits 30 and 31 are not changed.
- 29 Reserved.
- 30 Read/write. If set to 1, correctable errors are *not* corrected. If 0 and the device is in ECC mode, correctable errors are corrected.
- 31 Read/write in Mode 1, read-only in Mode2. 1 sets EEC mode, 0 sets parity mode.

ECC Address and Attribute Registers

These registers hold the contents of the AD bus for the Address and Attribute phases of a transaction containing an ECC error.

Mode 2 Configuration Space

PCI-X devices capable of Mode 2 operation have 4096 bytes of Configuration Space per function as shown in Figure 13-8. Figure 13-2c shows where the additional 4 address bits go to access this additional space. The first 256 bytes behave identically to Configuration Space in conventional PCI.

Extended Configuration Space can be used either for Extended Capabilities List items or for device-specific purposes outside the scope of the specification. If the

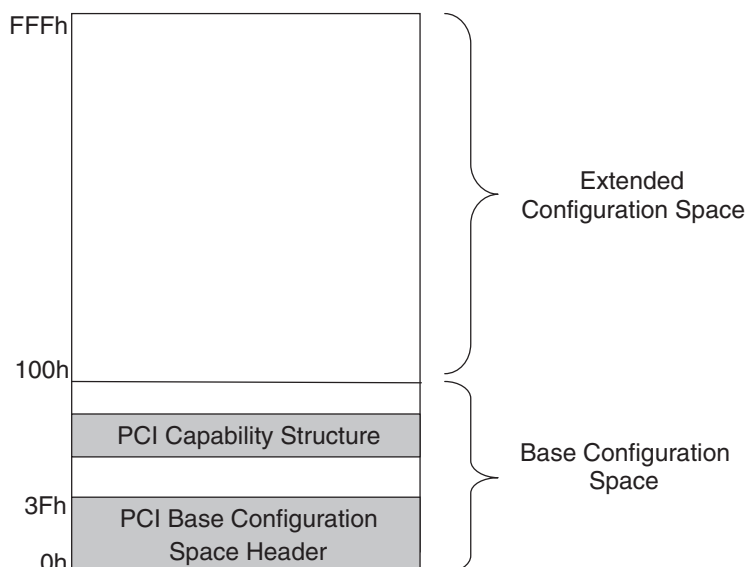


Figure 13-8: Mode 2 extended Configuration Space.

Extended Capabilities List is present, it begins at location 100h. If the DWORD value at location 100h is non-zero, then the list is present. If it is zero, there is no Extended Capabilities List. Figure 13-9 shows the layout of an Extended Capabilities item header. It is basically just an expanded form of the Capabilities item header in conventional PCI.

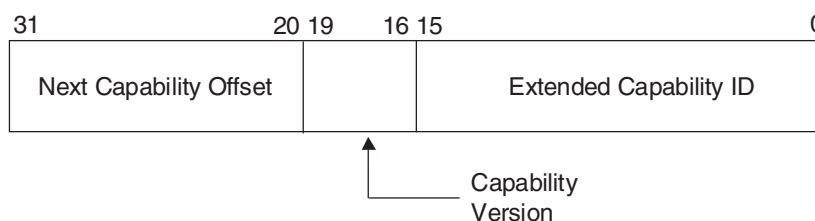


Figure 13-9: Extended Capabilities item header.

The Next Capability Offset is from the beginning of Configuration Space so it is either 0 for the end of the list or a value greater than 100h. Headers are always DWORD aligned so the low-order two bits of the offset are always 0. The specification does not call out any values for either the Capability ID or Version. Presumably, at least one of them must be non-zero to signify a valid header at location 100h.

Initialization

As mentioned back in Chapter 11, PCI-X devices are fully backward compatible with conventional PCI. At boot time, the system determines the capabilities of each bus segment—whether it is capable of PCI-X operation and at what frequency or is limited to conventional PCI. How the system does that is outside the scope of the specification.

The system must also determine the capabilities of each device—whether it is capable of PCI-X Mode 1 or Mode 2 operation or is a conventional PCI device. After all capabilities are determined, the system sets up each bus segment to operate at the level of the least capable device on that segment but not higher than the capability of the bus segment itself.

PCIXCAP and MODE2

The primary mechanism for determining the capability of a device is a pair of signals called **PCIXCAP** and **MODE2**. Conventional PCI defines both of these pins as ground. **PCIXCAP** is connected to 3.3V through a suitable pull-up resistor (see Chapter 14 for details). A conventional PCI card connects **PCIXCAP** to ground. A PCI-X card capable of 133 MHz operation leaves **PCIXCAP** unconnected so it is pulled up to 3.3V. PCI-X 66, 266 and 533 connect **PCIXCAP** to ground through a pull-down resistor as shown in Table 13-2 so that the voltage at the **PCIXCAP** pin is somewhere between 0 and 3.3V.

Mode	Pull-down Resistor value
Conventional PCI	0
PCI-X 66	10 k Ω \pm 5%
PCI-X 133	∞
PCI-X 266	3.01 k Ω \pm 1%
PCI-X 533	1.0 k Ω \pm 1%

Table 13-2: Pull-down Resistors for PCIXCAP.

Systems that don't support Hot Plug are allowed to bus **PCIXCAP** to a single sensing circuit. Hot Plug systems must have a separate sense circuit for each add-in connector.

MODE2 is pulled up to 3.3V through a 1 k Ω \pm 5% resistor. A conventional PCI or PCI-X Mode 1 card connects this pin to ground. A PCI-X Mode 2 card connects this

pin to ground through a $10\text{ k}\Omega \pm 5\%$ resistor in parallel with the gate of an FET (see Chapter 14 for details).

PCI-X Initialization Pattern

The significance of using PCIXCAP in this manner to determine capability is that it can be done while $\text{RST}\#$ is asserted and before any transactions have occurred on the bus. The system determines the capabilities of all devices while $\text{RST}\#$ is asserted, then sets up a PCI-X Initialization Pattern on $\text{PERR}\#$, $\text{DEVSEL}\#$, $\text{STOP}\#$ and $\text{TRDY}\#$

PERR#	DEVSEL#	STOP#	TRDY#	Mode	Error Protection	Min. Clock Freq. (MHz)	Max. Clock Freq. (MHz)
D	D	D	D	conventional 33 MHz	Parity	0	33
D	D	D	D	conventional 66 MHz	Parity	33	66
D	D	D	A	PCI-X Mode 1	Parity	50	66
D	D	A	D	PCI-X Mode 1	Parity	66	100
D	D	A	A	PCI-X Mode 1	Parity	100	133
D	A	D	D	PCI-X Mode 1	ECC	Reserved	Reserved
D	A	D	A	PCI-X Mode 1	ECC	50	66
D	A	A	D	PCI-X Mode 1	ECC	66	100
D	A	A	A	PCI-X Mode 1	ECC	100	133
A	D	D	D	PCI-X 266 (Mode 2)	ECC	Reserved	Reserved
A	D	D	A	PCI-X 266 (Mode 2)	ECC	50	66
A	D	A	D	PCI-X 266 (Mode 2)	ECC	66	100
A	D	A	A	PCI-X 266 (Mode 2)	ECC	100	133
A	A	D	D	PCI-X 533 (Mode 2)	ECC	Reserved	Reserved
A	A	D	A	PCI-X 533 (Mode 2)	ECC	50	66
A	A	A	D	PCI-X 533 (Mode 2)	ECC	66	100
A	A	A	A	PCI-X 533 (Mode 2)	ECC	100	133

Table 13-3: PCI-X Initialization patterns.

as shown in Table 13-3. All devices latch this pattern on the deassertion (rising edge) of **RST#** to determine the mode they are expected to operate in.

Notes for Table 13-3

1. A = asserted, D = deasserted.
2. Conventional PCI distinguishes 33 MHz from 66 MHz with the **M66EN** pin

Summary

PCI-X adds new requirements in terms of configuration and initialization. PCI-X devices need to know their device number so they can communicate that in the Attribute phase of a transaction. Thus, the device number is now conveyed in Type 0 configuration transactions.

PCI-X makes some minor changes to the 64-byte configuration header, mainly in terms of identifying characteristics that are either always present in a PCI-X device or are not supported by PCI-X. A PCI-X Capabilities List item is defined to provide expanded Command and Status registers and ECC data for transactions with errors.

Mode 2 expands the Configuration Space to 4096 bytes per function. This may be used for Extended Capabilities List items or device specific information.

Two pins, **PCIXCAP** and **MODE2**, are used to identify the capabilities of an add-in card at initialization time while **RST#** is asserted. Based on the information from all add-in cards, the system places a PCI-X Initialization Pattern on the bus when **RST#** is deasserted.

The next and final chapter looks at the electrical and mechanical requirements of PCI-X.

CHAPTER 14

PCI-X Electrical and Mechanical Features

We conclude our tour of the world of PCI with a look at the electrical and mechanical aspects of PCI-X.

Signal Categories

PCI-X Mode 2 introduces a new signaling environment that operates at 1.5 volts. To clarify the characteristics and requirements of the various signaling environments, PCI-X categorizes all bus signals as shown in Table 14-1.

Table 14-1: Signal categories.

Category		Supply Voltage	Output Type	Input Terminator	Signals
1	Mode 1	3.3V	Totem Pole	No	AD C/BE# PAR64/ECC[7] REQ64#/ECC[6] ECC[5::2] ACK64#/ECC[1] PAR/ECC[0]
	Mode 2	1.5V	Totem Pole	Yes	
2		3.3V	Totem Pole	No	FRAME# DEVSEL# IRDY# TRDY# STOP# IDSEL PERR# LOCK# REQ# GNT# CLK RST#
3		3.3V	Open Drain	No	SERR# INTx#
4		See PCI 3.0			M66EN PRSNT[1::2]# TCK TDI TDO TMS TRST# SMBCLK SMBDAT
5		See PCI PM 1.1			PME#
6		n/a			PCIXCAP MODE2

Category 1

The nature of these signals depends on which PCI-X mode the device is operating in. In Mode 1, they use 3.3V signaling and common clock timing for the entire

transaction. In Mode 2, they use 1.5V signaling and source synchronous timing for data phases. Other phases of a transaction use common clock timing.

The 1.5V signaling environment uses a differential receiver with an input termination resistor.

Category 2

These basic control signals always operate at 3.3V and use common clock timing.

Category 3

These are the open drain signals. **SERR#** is asserted synchronously with the clock but its deassertion is asynchronous. Both edges of **INTx#** are asynchronous to the clock.

Categories 4 and 5

This is a collection of miscellaneous signals defined in conventional PCI and the Power Management specification.

Category 6

These card identification pins are point-to-point signals between the device and the central resource or hot plug controller.

3.3 Volt Signaling Environment

DC Specifications

The specifications for 3.3 volt signaling in PCI-X are nearly identical to those for conventional PCI. Table 14-2 shows the DC specifications for PCI-X together with the same specs for PCI. The values that differ are **bolded**.

Notes for Table 14-2

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 bidirectional buffers with tri-state outputs.
3. Absolute maximum pin capacitance for a PCI/PCI-X input except for CLK and IDSEL.

Table 14-2: DC Specifications for 3.3V Signaling.

Symbol	Parameter	Condition	PCI-X		PCI		Units	Notes
			Min	Max	Min	Max		
V_{cc}	Supply Voltage		3.0	3.6	3.0	3.6	V	
V_{ih}	Input High Voltage		$0.5V_{cc}$	$V_{cc} + 0.5$	$0.5V_{cc}$	$V_{cc} + 0.5$	V	
V_{il}	Input Low Voltage		-0.5	$0.35V_{cc}$	-0.5	$0.3V_{cc}$	V	
V_{ipu}	Input Pull-up Voltage		$0.7V_{cc}$		$0.7V_{cc}$		V	1
I_{il}	Input Leakage Current	$0 < V_{in} < V_{cc}$		± 10		± 10	μA	2
V_{oh}	Output High Voltage	$I_{out} = -500 \mu A$	$0.9V_{cc}$		$0.9V_{cc}$		V	
V_{ol}	Output Low Voltage	$I_{out} = 1500 \mu A$		$0.1V_{cc}$		$0.1V_{cc}$	V	
C_{in}	Input Pin Capacitance			8		10	pF	3
C_{clk}	CLK Pin Capacitance		5	8	5	12	pF	
C_{IDSEL}	IDSEL Pin Capacitance			8		8	pF	4
L_{pin}	Pin Inductance			15		20	nH	5
I_{Off}	PME# input leakage	$V_o \leq 3.6 V$ V_{cc} off or floating		1		1	μA	6

4. For conventional PCI only, lower capacitance on this input-only pin allows for non-resistive coupling to AD[xx]. PCI-X configuration transactions drive the AD bus 4 clocks before asserting FRAME#.
5. For conventional PCI this is a recommendation, not an absolute requirement. For PCI-X it is a requirement.
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 14-3 summarizes the AC specifications for the 3.3 volt signaling environment, while Table 14-3 shows the AC specifications for PCI 66 reproduced from Chapter 5 for comparison. Figure 14-1 illustrates the corresponding V/I curves for PCI-X with the corresponding PCI curves superimposed.

Table 14-3: PCI-X AC specifications for 3.3V signaling.

Symbol	Parameter	Condition	Min	Max	Units	Notes
$I_{oh(AC)}$	Switching Current High	$0 < V_{cc} - V_{out} \leq 3.6V$		$-74(V_{cc} - V_{out})$	mA	
		$0 < V_{cc} - V_{out} \leq 1.2V$	$-32(V_{cc} - V_{out})$		mA	1
		$1.2V < V_{cc} - V_{out} \leq 1.9V$	$-11(V_{cc} - V_{out}) - 25.2$			1
		$1.9V < V_{cc} - V_{out} \leq 3.6V$	$-1.8(V_{cc} - V_{out}) - 42.7$		mA	1
$I_{ol(AC)}$	Switching Current Low	$0 < V_{out} \leq 3.6V$		$100V_{out}$	mA	
		$0 < V_{out} \leq 1.3V$	$48V_{out}$		mA	1
		$1.3V < V_{out} \leq 3.6V$	$5.7V_{out} + 55$		mA	1
I_{cl}	Low Clamp Current	$-3V < V_{in} \leq -0.8875V$	$-40 + (V_{in} + 1)/0.005$		mA	
		$-0.8875V < V_{in} \leq -0.625V$	$-25 + (V_{in} + 1)/0.015$		mA	
I_{ch}	High Clamp Current	$0.8875V \leq V_{in} - V_{cc} < 4V$	$40 + (V_{in} - V_{cc} - 1)/0.005$		mA	
		$0.625V \leq V_{in} - V_{cc} < 0.8875V$	$25 + (V_{in} - V_{cc} - 1)/0.015$		mA	

Table 14-4: PCI 66 AC specifications.

Symbol	Parameter	Condition	Min	Max	Units	Notes
$I_{oh(AC)}$	Switching Current High	$V_{out} = 0.7V_{cc}$		$-32V_{cc}$	mA	
		$V_{out} = 0.3V_{cc}$	$-12V_{cc}$		mA	1
$I_{ol(AC)}$	Switching Current Low	$V_{out} = 0.18V_{cc}$		$38V_{cc}$	mA	
		$V_{out} = 0.6V_{cc}$	$16V_{cc}$		mA	1
I_{cl}	Low Clamp Current	$-3 < V_{in} \leq -1$	$-25 + (V_{in} + 1)/0.015$		mA	
I_{ch}	High Clamp Current	$V_{cc} + 4 > V_{in} \geq V_{cc} + 1$	$25 + (V_{in} - V_{cc} - 1)/0.015$		mA	

Notes for Table 14-4

1. In conventional PCI 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. In PCI-X devices, REQ# and GNT# must have full size drivers. 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.

Table 14-5: Output Slew Rate for 3.3V Signaling.

Symbol	Parameter	Condition	PCI-X		Conventional PCI		Units	Notes
slew_r	Output Rise Slew Rate	$0.2V_{cc}$ to $0.6V_{cc}$ load	1	6	1	4	V/ns	1
slew_f	Output Fall Slew Rate	$0.6V_{cc}$ to $0.2V_{cc}$ load	1	6	1	4	V/ns	1

Notes for Table 14-5

1. 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. However, adherence to both maximum and minimum parameters is required (the maximum is not simply a guideline). Rise slew rate does not apply to open drain outputs.

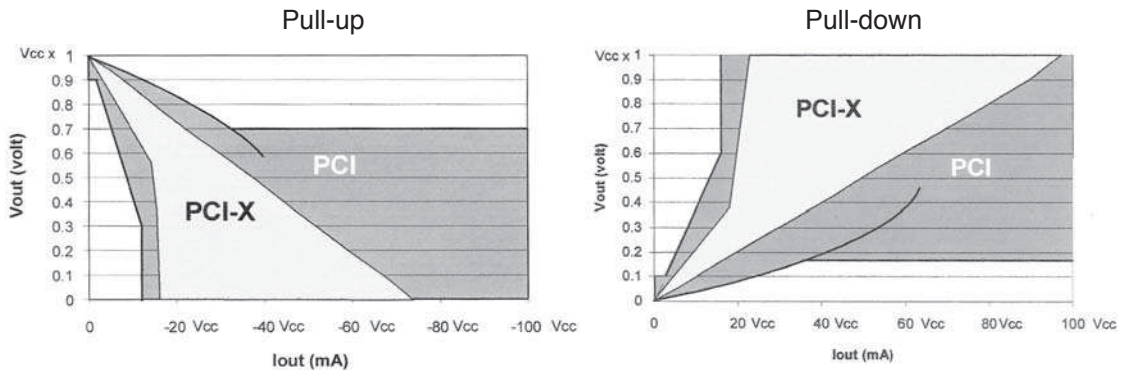


Figure 14-1: Driver V/I curves for 3.3V signaling.

Timing Specifications

Clock

Clock specifications for the 3.3 volt PCI-X environment are essentially identical to 3.3 volt conventional PCI except that V_{il} is higher as shown in Table 14-2. Table 14-6 summarizes the PCI-X clock requirements. For reference, Figure 14-2 duplicates the 3.3 volt portion of Figure 5-7.

Table 14-6: Clock & Reset Specifications.

Symbol	Parameter	PCI-X 133		PCI-X 66		Conv. PCI 66 (ref)		Conv. PCI 33 (ref)		Units	Notes	
		Min	Max	Min	Max	Min	Max	Min	Max			
Clock Jitter class 1												
T _{cyc}	CLK Cycle Time	7.5	20	15	20	15	30	30	∞	ns	1, 3, 4	
T _{high}	CLK High Time	3		6		6		11		ns	4	
T _{low}	CLK Low Time	3		6		6		11		ns	4	
Clock Jitter class 2												
T _{cyc}	CLK Cycle Time	Avg.	7.5	20	15	20	15	30	30	∞	ns	1, 4
		Abs Min.	7.375		14.8		14.8		29.7		ns	1, 3, 4
T _{high}	CLK High Time	2.5		5.5		5.5		10		ns	4	
T _{low}	CLK Low Time	2.5		5.5		5.5		10		ns	4	
Slew Rate												
-	CLK Slew Rate	1.5	4	1.5	4	1.5	4	1	4	V/ns	2, 4	

Notes for Table 14-6

1. For clock frequencies above 33 MHz, the clock frequency may not change beyond the spread-spectrum limits except while RST# is asserted.
2. This slew rate must be met across the minimum peak-to-peak portion of the clock waveform as shown in Figure 14-2.
3. The minimum clock period must not be violated for any single clock cycle, i.e., accounting for all system jitter.
4. All PCI-X 133 devices must also be capable of operating in PCI-X 66. All PCI-X devices must be capable of operating in conventional PCI 33 mode and optionally are capable of conventional PCI 66 mode.

PCI-X defines two classes of clock jitter. Class 1 is required for all Mode 1 devices. It specifies minimum and maximum clock cycle times that must not be violated on any single clock cycle taking into account all sources of clock variation tolerance and

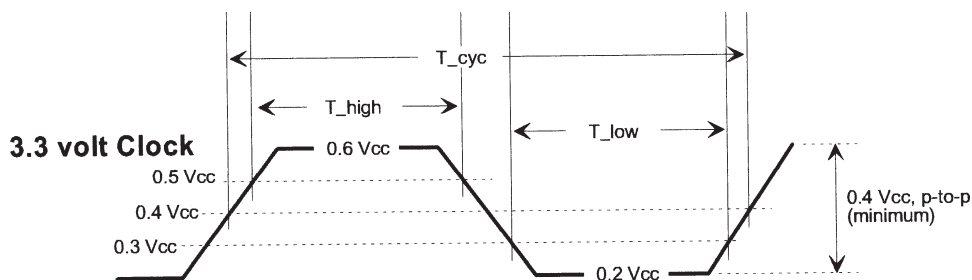


Figure 14-2: Clock waveform.

jitter. This means that even though the maximum clock frequency for PCI-X is 133 MHz, the actual operating frequency must be lower than that to allow for clock jitter.

For example, suppose a particular clock design has a maximum clock period variation of 180 ps. We can calculate the maximum nominal clock frequency as:

$$1/(7.5 \text{ ns} + 0.18 \text{ ns}) = 130.208 \text{ MHz}$$

To run the clock any faster than that would risk violating the minimum clock cycle time.

Class 2 jitter, which is optional for Mode 1 devices but required for Mode 2, specifies long-term *average* minimum and maximum cycle times that must be met over any 1 microsecond sample period. Any one clock cycle still must meet the absolute minimum. This allows the clock to run at the maximum nominal frequency provided that the overall jitter is within the absolute minimum clock cycle.

Spread-spectrum clocking is supported in PCI-X with the same specifications as shown in Table 5-10. The overall clock specifications for PCI-X 266 and PCI-X 533 are the same as described here.

Timing Parameters

Table 14-7 summarizes the timing parameters that change from conventional PCI to PCI-X along with the values for conventional PCI 66. All other timing parameters are identical to the values in Table 5-7. Note that PCI-X, as well as conventional PCI 66, makes no distinction between bussed and point-to-point signals. For reference, Figure 14-3 duplicates Figure 5-8.

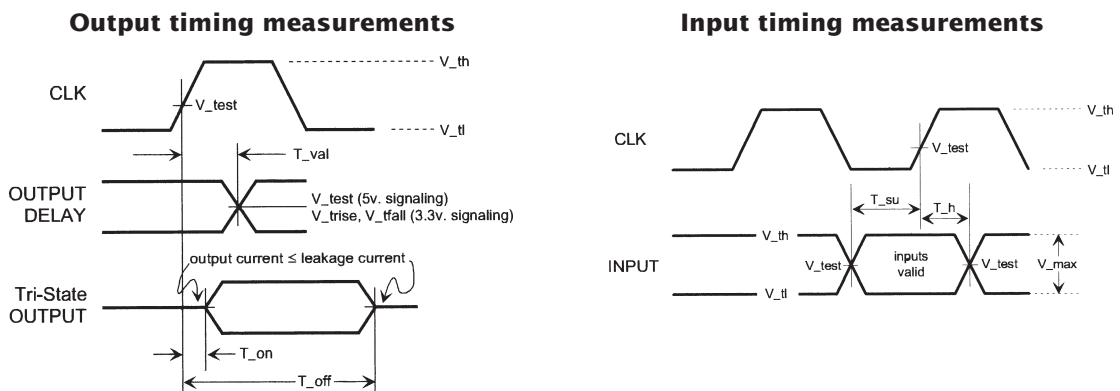


Figure 14-3: PCI-X timing measurement conditions.

Table 14-7: PCI-X Timing parameters.

Symbol	Parameter	PCI-X 133		PCI-X 66		Conv. PCI 66		Units
		Min	Max	Min	Max	Min	Max	
t_{val}	CLK to Signal Valid Delay bussed signals	0.7	3.8	0.7	3.8	2	6	ns
$T_{val} (ptp)$	CLK to Signal Valid Delay point to point signals	0.7	3.8	0.7	3.8	2	6	ns
t_{on}	Float to Active Delay	0		0		2		ns
t_{off}	Active to Float Delay		7		7		14	ns
t_{su}	Input Setup Time to CLK bussed signals	1.2		1.7		3		ns
$t_{su} (ptp)$	Input Setup Time to CLK point to point signals	1.2		1.7		5		ns
t_h	Input Hold Time from CLK	0.5		0.5		0		ns

1.5 Volt Signaling Environment

The 1.5V signaling environment applies to Mode 2, Category 1 signals only. When operating in Mode 1 Category 1 signals revert to the 3.3V signaling environment.

DC Specifications

Table 14-8 lists the DC specifications for 1.5V signaling.

Table 14-8: DC Specifications for 1.5V signaling environment.

Sym.	Parameter	Conditions	Min	Max	Units	Notes
V_{io}	I/O Supply Voltage		1.425	1.575	V	1
V_{ref}	Input Reference Voltage		$0.49V_{io}$	$0.51V_{io}$	V	7
V_{ih}	Input High Voltage		$V_{ref} + 0.1$	$V_{io} + 0.5$	V	
V_{il}	Input Low Voltage		-0.5	$V_{ref} - 0.1$	V	
I_{il}	Input Leakage Current	$0 < V_{in} < V_{io}$		+10	μA	2
V_{oh}	Output High Voltage		$V_{io} - 0.2$	$V_{io} - 0.1$	V	
V_{ol}	Output Low Voltage		0.1	0.2	V	
C_{pad}	Input Pad Capacitance			3.5	pF	3
Z_{term}	Equivalent Input Impedance	$V_{ol(AC)MAX} < V_{in} \leq V_{oh(AC)MAX}$	51	63	Ω	4, 5
		$0 < V_{in} \leq V_{io}$	45	68		
V_{term}	Equivalent Input Termination Voltage	Open circuit voltage at input pin.	$0.45V_{io}$	$0.55V_{io}$	V	6

Notes for Table 14-8

1. Total DC $V_{I/O}$ supply variation includes any system-introduced ripple or noise and is measured while the bus is idle.
2. Input leakage currents include hi-Z output leakage for all bidirectional buffers with tri-state outputs. Applies when both output driver and input terminator are disabled.
3. This is the equivalent lumped capacitance at the die pad. This capacitance specification is guaranteed by design.
4. This is a measurement of the small-signal impedance (or reciprocal of the derivative of the I/O I-V curve) and must be maintained within the specified impedance range over the specified operating bias conditions.
5. Devices designed exclusively for system-board applications (not for use on add-in cards) are permitted to use a different value, provided that the system-board vendor guarantees sufficient noise and timing margins.
6. V_{term} is optionally generated internally from $V_{I/O}$ or is implemented as a separate power supply input to the device.
7. V_{ref} is optionally generated either inside or outside the device. If V_{ref} is generated outside the device, this parameter is measured at the package pin. If V_{ref} is generated inside the device, all other specifications that are dependent on V_{ref} calculate the value of V_{ref} as a function of $V_{I/O}$ (measured at the package pin) using the formulas provided here for minimum and maximum V_{ref} .

1.5V signaling uses differential input receivers where one side is tied to a reference voltage, V_{ref} . The inputs are terminated with a Thevenin equivalent impedance and voltage source, V_{term} . Note by the way that 1.5V drivers/receivers must also be able to operate in the 3.3V environment because, in general, a Mode 2 device must also be capable of operating in Mode 1. One exception to this is the 16-bit bus, which operates exclusively in Mode 2. Drivers on a 16-bit bus are not required to support the 3.3V environment.

AC Specifications

At 1.5 volts, the noise margins are pretty small. As a result, PCI-X sets limits for power supply noise. V_{io} and V_{ref} are limited to ± 75 mV around their respective DC levels. V_{term} is limited to ± 30 mV around its DC level.

Table 14-9 summarizes the AC switching specifications for 1.5V signaling.

Table 14-9: AC Specifications for 1.5V signaling.

Symbol	Parameter	Condition	Min	Max	Units
$V_{ih(AC)}$	Input High Voltage Switching		$V_{ref} + 0.2$	$V_{ref} + 0.5$	V
$V_{il(AC)}$	Input Low Voltage Switching		-0.5	$V_{ref} - 0.2$	V
$V_{oh(AC)}$	Output High Voltage Switching		$V_{io}/2 + 0.45$		V
$V_{ol(AC)}$	Output Low Voltage Switching			$V_{io}/2 - 0.45$	V
I_{cl}	Low Clamp Current	$-1.56V < V_{in} \leq -0.8875V$	$-40 + (V_{in} + 1)/0.005$		mA
		$-0.8875V < V_{in} \leq -0.625V$	$-25 + (V_{in} + 1)/0.015$		
I_{ch}	High Clamp Current	$0.8875V \leq V_{in} - V_{io} < 4V$	$40 + (V_{in} - V_{io} - 1)/0.005$		mA
		$0.625V \leq V_{in} - V_{io} < 0.8875V$	$25 + (V_{in} - V_{io} - 1)/0.015$		
T_{ir}	Input Rise Time	From $V_{il(AC)}$ to $V_{ih(AC)}$	133	570	pS
T_{if}	Input Fall Time	From $V_{ih(AC)}$ to $V_{il(AC)}$	133	570	pS
ΔT_{if}	Input Rise & Fall Time Matching Deviation			63	pS
T_{or}	Output Rise Time	From $V_{ol(AC)}$ to $V_{oh(AC)}$	300	900	pS
T_{of}	Output Fall Time	From $V_{oh(AC)}$ to $V_{ol(AC)}$	300	900	pS
ΔT_{orf}	Output Rise & Fall Time Matching Deviation			100	pS

Source Synchronous Timing Specifications

Figure 14-4 illustrates how Common Clock Mode (CCM) and Source Synchronous Mode (SSM) relate to each other. Both the CCM clock and the SSM clock are derived from a phase locked loop driven by the CLK signal. The dashed boxes in the initiator and target represent the logic synchronized to each of these clock sources. In Source Synchronous mode, data is latched into the target directly by the FSTROBEs and SSTROBEs.

The Mode 2 protocol allows three clock periods for source synchronous data to be clocked out of the initiator, propagate across the system, be captured by the target

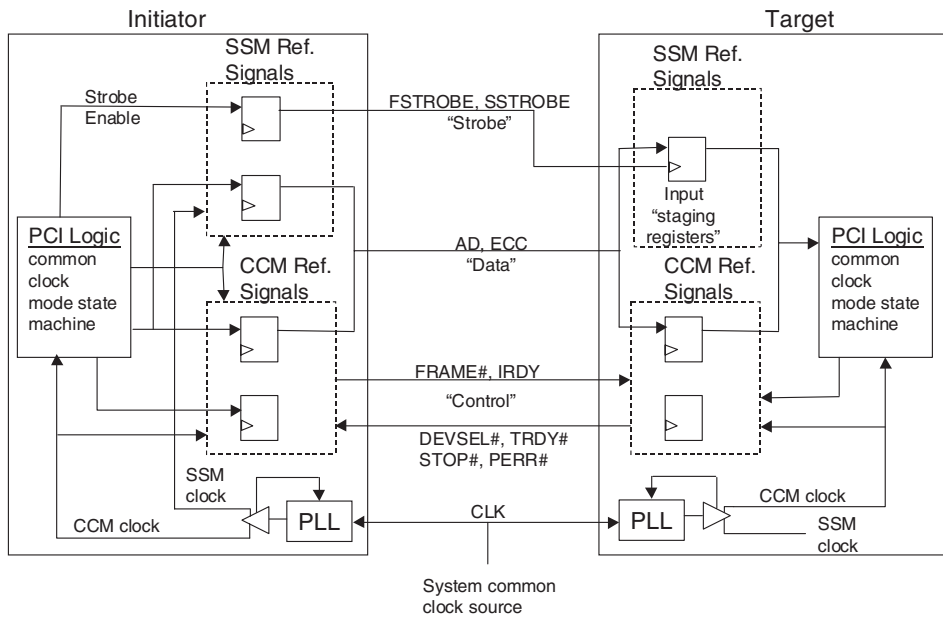


Figure 14-4: Source-synchronous and common clock domain partitioning.

and resynchronized to the common clock domain in the target (the solid box). An additional clock period is required to check ECC, and **PERR#** is asserted or deasserted one clock period after that.

Figure 14-5 shows the effect of signal propagation time on common-clock transfers. Signals are clocked out of the driving device with a maximum delay from the rising edge of **CLK**. The signals arrive at the receiving device delayed by the signal

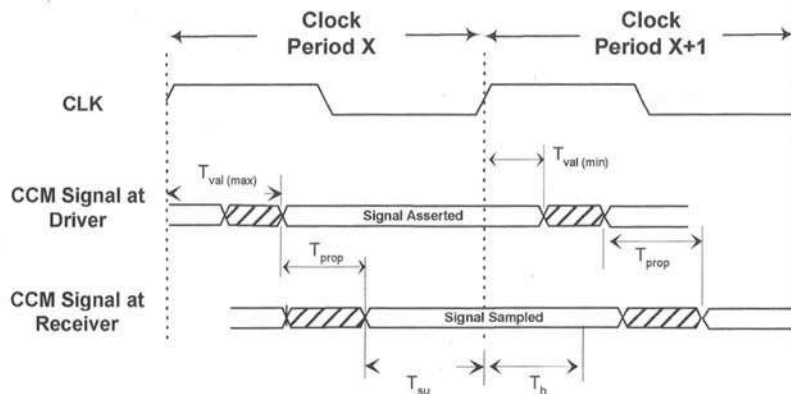


Figure 14-5: Signal propagation effects in common clock mode.

propagation time. Common-clock signals are required to be set up to and to be held after the next rising edge of CLK. The system is required to keep the signal propagation time short enough that the signals are stable and set up to CLK at all the receiving devices.

Figure 14-6 shows that source-synchronous data strobes compensate for most of the effects of signal propagation delay. Since the data strobes are driven out of the same device as the data and follow the same physical path to the receiver, they arrive at the receiver with approximately the same timing relative to the data. In source-synchronous transfers, it is the difference between the data and the strobe paths that affects data capture timing, more so than the absolute length of the path. The absolute length of the path is significant after the data is captured and demultiplexed and resynchronized to the common clock.

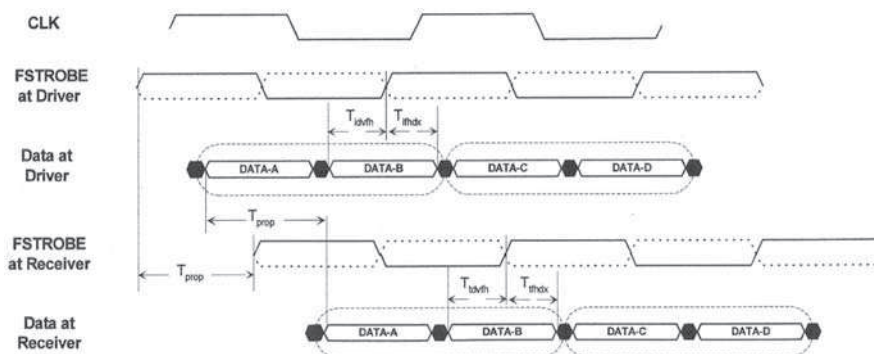


Figure 14-6: Signal propagation effects in source synchronous mode.

PCI-X uses a somewhat cumbersome convention for naming timing parameters for source synchronous transfers. Each parameter gets a five-letter subscript formatted as follows:

1. The first letter is either i or t indicating whether the parameter is an initiator output delay requirement or a target input guarantee.
2. The next two letters designate the signal and the state change that is the beginning of the parameter. The signals are:
 - c – CLK
 - f – FSTROBE
 - s – SSTROBE
 - d – data (AD or ECC)

The state changes are:

- h – signal going high
- l – signal going low
- v – data becoming valid
- x – data no longer valid

3. The final two letters designate the signal and the state change that is the end of the parameter using the same protocol as above.

For example, the parameter T_{ichfh} is the output delay measured at the initiator from CLK going high to FSTROBE going high. Similarly, T_{tshdx} is the delay measured at the target from SSTROBE going high to when data is no longer valid (data hold time). Table 14-10 shows the timing parameters for Initiator output while Table 14-11 shows the parameters for Target input. All values are in nanoseconds. T_{cyc} is the cycle time of CLK. T_{val} is the maximum value from Table 14-7. Note in particular how FSTROBE and SSTROBE are staggered from CLK.

Table 14-10: Source Synchronous Initiator Output Timing.

Sym	Parameter		PCI-X 533		PCI-X 266	
			Min	Max	Min	Max
T_{ifhfh} T_{ishsh}	STROBE cycle time		$0.5T_{cyc} - 0.125$	$0.5T_{cyc} + 0.125$	$T_{cyc} - 0.125$	$T_{cyc} + 0.125$
T_{ifhl} T_{ishl}	STROBE pulse width high		$0.25T_{cyc} - 0.375$	$0.25T_{cyc} + 0.375$	$0.5T_{cyc} - 0.75$	$0.5T_{cyc} + 0.75$
T_{ifllh} T_{isllh}	STROBE pulse width low		$0.25T_{cyc} - 0.375$	$0.25T_{cyc} + 0.375$	$0.5T_{cyc} - 0.75$	$0.5T_{cyc} + 0.75$
T_{ichfh}	Delay from CLK to FSTROBE	1 st subphase		$0.125T_{cyc} + T_{val}$		$0.25T_{cyc} + T_{val}$
		3 rd subphase		$0.625T_{cyc} + T_{val}$		
T_{ichsh}	Delay from CLK to SSTROBE	2 nd subphase		$0.375T_{cyc} + T_{val}$		$0.75T_{cyc} + T_{val}$
		4 th subphase		$0.875T_{cyc} + T_{val}$		
T_{idvth} T_{idshx} T_{idvsh} T_{idshx}	Data valid before and after STROBE		$0.125T_{cyc} - 0.355$		$0.25T_{cyc} - 0.9$	

Table 14-11: Source Synchronous Target Input Timing.

Sym	Parameter	PCI-X 533	PCI-X 266
		Min	Max
$T_{tthf\ h}$ T_{tshsh}	STROBE cycle time	3.625	7.375
$T_{tthf\ l}$ T_{tshsl}	STROBE pulse width high	1.415	2.915
T_{tflfh} T_{tshsh}	STROBE pulse width low	1.415	2.915
T_{tdvfh} T_{tdvsh}	Data setup to STROBE	0.38	0.7
T_{tfhdx} T_{tshdx}	Data hold after STROBE	0.38	0.7
T_{tfhch}	FSTROBE next to last subphase setup to CLK	11.635	14.455
T_{tshch}	SSTROBE last subphase setup to CLK	9.765	10.705

Figures 14-7 and 14-8 graphically illustrate the timing parameters for PCI-X 533 and PCI-X 266, respectively.

Note that the common clock mode (CCM) timing parameters for Mode 2 are identical to Mode 1, as shown in Table 14-7 for PCI-X 133.

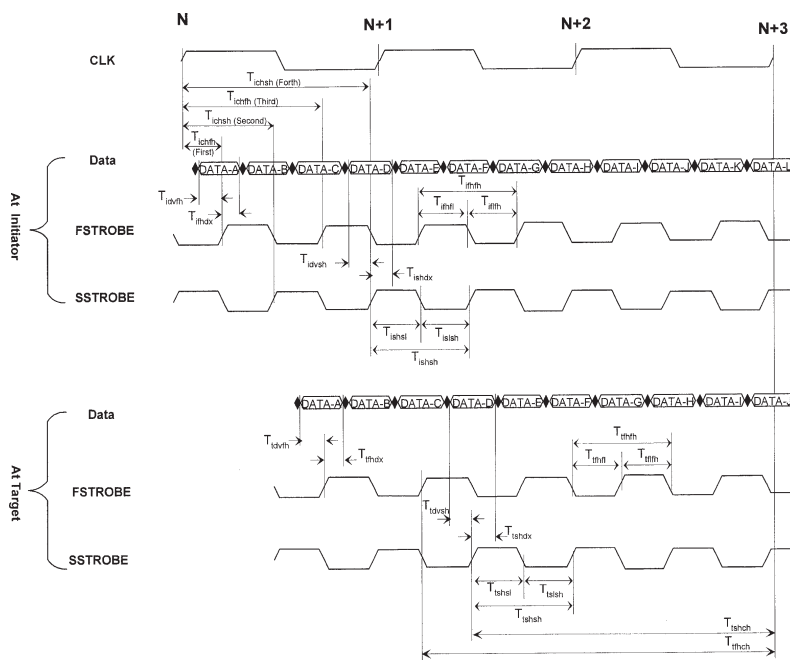


Figure 14-7: PCI-X 533 timing parameters.

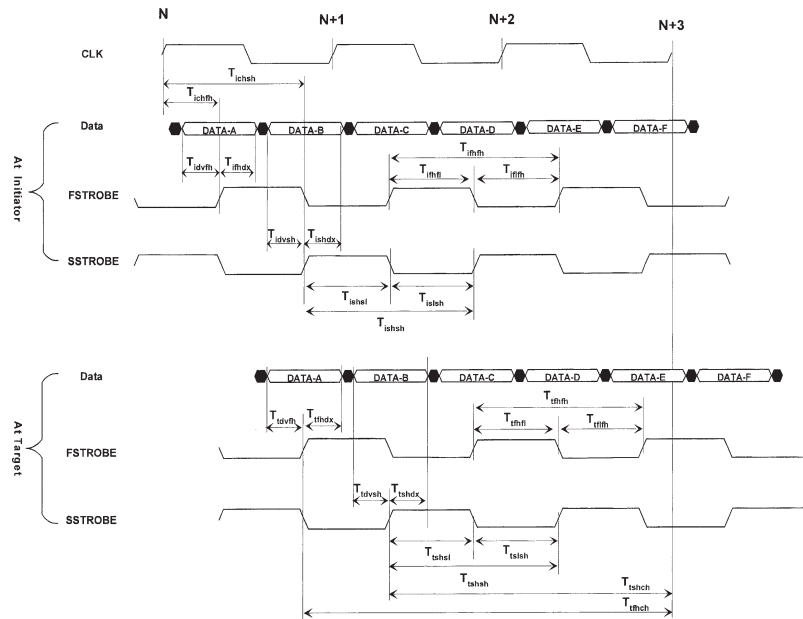


Figure 14-8: PCI-X 266 timing parameters.

Input Termination

As mentioned above, Mode 2 Category 1 inputs are terminated by a Thevenin equivalent impedance of nominally $57\ \Omega$ and an open-circuit voltage source of nominally $0.5V_{io}$. This termination must be switchable because there are several circumstances when the terminator must be switched out:

- When the device is operating in Mode 1 or conventional PCI
- When RST# is asserted
- When the corresponding output driver is enabled
- On the upper 32 bits of a 64-bit add-in card when inserted in a 32-bit slot.

Figure 14-9 shows a conceptual implementation of a Category 1 buffer block to meet these requirements. The Control Logic block requires three inputs to determine which driver to enable and whether to switch in the termination resistors. These are:

- Mode: Mode 1 or Mode 2
- OEn: Output Enable
- RST#: The PCI bus reset signal

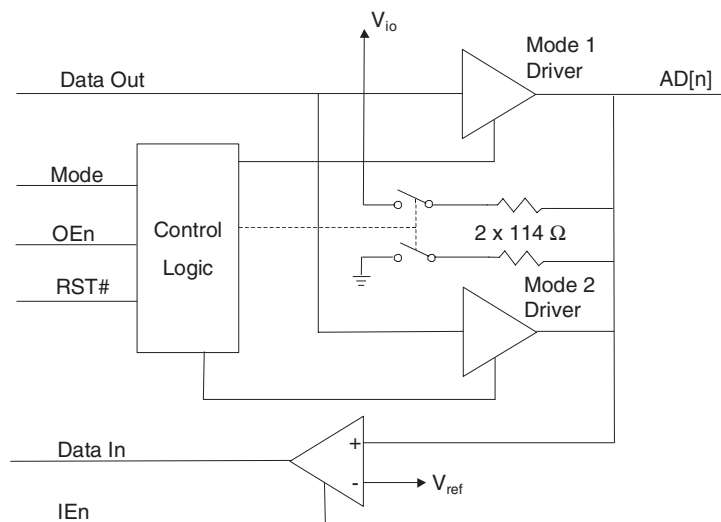


Figure 14-9: Mode 2 Category 1 buffer block.

The same differential receiver is used in both modes. V_{ref} is set to $0.4V_{io}$ in Mode 1 and $0.5V_{io}$ in Mode 2. The receiver is disabled when not needed to minimize operating current.

PCIXCAP and MODE2

Figure 14-10 shows how PCIXCAP and MODE2 are implemented on various add-in cards. The system board pulls PCIXCAP to 3.3 volts through a resistor in the range of 5 to 10 k Ω . The result is that PCIXCAP is:

- 0 volts if the bus segment includes at least one conventional PCI card
- 3.3 volts if all cards can operate at 133 MHz
- Somewhere in between if one or more cards are limited to 66 MHz

For example, if PCIXCAP is pulled up on the system board with 10 k Ω , with one PCI-X 66 card plugged in, PCIXCAP will go to about 1.65 volts. Two PCI-X 66 cards will pull it down to around 1.1 volts. A pair of comparators with thresholds set appropriately is sufficient to make this distinction.

Distinguishing Mode 2 is a little harder. A bus segment that only supports Mode 1 connects MODE2 to ground at every add-in connector. This causes the FET on a Mode 2 add-in card to switch off so that the card will be correctly identified as PCI-X 133 rather than mistaken as a PCI-X 66 card. A Mode 2 bus segment pulls MODE2

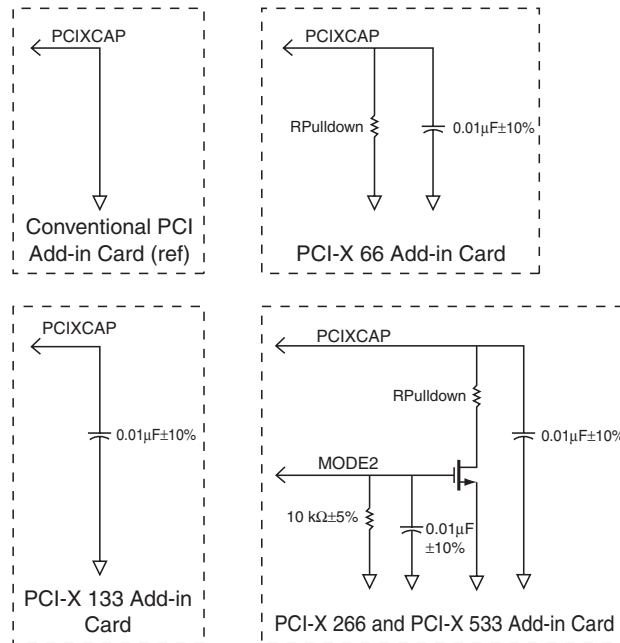


Figure 14-10: PCIXCAP and MODE2 on add-in cards.

to 3.3 volts through a 1 k Ω resistor. This turns on the FET to connect the pull-down resistor to ground.

Now there are three intermediate values to distinguish requiring four comparators. Mode 2 bus segments generally use a smaller pull-up value (3.3 k Ω works) to get extra margin in the intermediate values. Note, incidentally, that three PCI-X 266 cards plugged into a bussed PCIXCAP line will look like a PCI-X 533 card. Thus, if a Mode 2 bus segment has more than two slots, PCIXCAP should be detected independently at each slot.

Mechanical

PCI-X uses the same basic mechanics as conventional PCI. The only additional requirements relate to clearly identifying system boards and add-in cards capable of PCI-X operation. A system board capable of Mode 1 PCI-X operation must indicate which of the two maximum clock frequencies (66 MHz or 133 MHz) each slot is optimized for. A slot capable of Mode 2 operation must indicate which of the two source synchronous data rates (266 MHz or 533 MHz) the slot is optimized for.

Add-in cards must be clearly marked with the PCI-X logo and maximum data rate.

Summary

PCI-X Mode 1 uses a 3.3 volt signaling environment similar to PCI 66. The timing requirements are naturally tighter for PCI-X 133. Mode 2 uses a new 1.5 volt signaling environment for Category 1 signals. 1.5V receivers require termination that must be switchable.

In source synchronous mode, the Initiator generates the strobe signals that clock data into the Target. This minimizes the problems of propagation delay because the data and strobes are following the same path and thus have the same delay.

System board and add-in card capabilities are determined by the signals **PCIXCAP** and **MODE2** where intermediate values on **PCIXCAP** represent PCI-X 66, 266 and 533.

APPENDIX A

Class Codes

<i>Class / Subclass</i>	<i>Programming Interface</i>
All numbers are expressed in Hex.	
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
05	ATA Controller
	20 Single DMA
	30 Chained DMA
06	Serial ATA Direct Port Access
Class 02	Network controllers
00	Ethernet
01	Token Ring
02	FDDI
03	ATM
04	ISDN
05	World Fip Controller
06	PICMG 2.14 Multi-computing

Notes

- 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 03	Display controllers
00	VGA/8514
00	VGA-compatible
01	8514-compatible
01	XGA
02	3-D controller
Class 04	Multimedia devices
00	Video
01	Audio
02	Computer telephony
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
09	Semi-transparent Bridge
40	Primary PCI bus side faces system host processor
80	Secondary PCI bus side faces system host processor
0A	Infiniband to PCI Host Bridge
Class 07	Simple communication controllers
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

Class/ Subclass	Programming Interface
Class 07	Simple communication controllers (continued)
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)
04	GPIO (IEEE 488.1/2) Controller
05	Smart Card
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

Notes

2. First BAR (10h) maps appropriate compatible register set. Registers can be either memory or I/O mapped.
3. 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.

<i>Class / Subclass</i>	<i>Programming Interface</i>
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
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
06	Infiniband
07	Intelligent Platform Management Interface (IPMI)
	00 SMIC Interface
	01 Kybd Controller style interface
	02 Block Transfer interface
08	SERCOS Interface Standard (IEC 61491)
09	CANbus

Notes

4. "Legacy" game port. Byte at offset 01h aliases to byte at offset 00h.

<i>Class/ Subclass</i>	<i>Programming Interface</i>
Class 0D	Wireless controllers
00	iRDA controller
01	Consumer IR controller
10	RF controller
11	Bluetooth
12	Broadband
20	Ethernet (802.11a – 5 GHz)
21	Ethernet (802.11b – 2.4 GHz)
Class 0E	Intelligent I/O controllers
00	xx I2O Architecture Specification 1.0 00 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
01	Performance Counters
10	Communication synchronization plus time and frequency test/measurement
20	Management card

Notes

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

APPENDIX B

Connector Pin Assignments

PCI Connector

Pin	Side B	Side A	Pin	Side B	Side A
1	-12V	TRST	25	+3.3V	AD[24]
2	TCK	+12V	26	C/BE[3]	IDSEL
3	Gnd	TMS	27	AD[23]	+3.3V
4	TDO	TDI	28	Gnd	AD[22]
5	+5V	+5V	29	AD[21]	AD[20]
6	+5V	INTA#	30	AD[19]	Gnd
7	INTB#	INTC#	31	+3.3V	AD[18]
8	INTD#	+5V	32	AD[17]	AD[16]
9	PRSNT1#	ECC[5]	33	C/BE[2]	+3.3V
10	ECC[4]	+3.3V (I/O) ²	34	Gnd	FRAME#
11	PRSNT2#	ECC[3]	35	IRDY#	Gnd
12	3.3V: Keyway		36	+3.3V	TRDY#
13	5V: Gnd		37	DEVSEL#	Gnd
14	ECC[2]	3.3Vaux	38	PCIXCAP	STOP#
15	Gnd	RST#	39	LOCK#	+3.3V
16	CLK	+3.3V (I/O)	40	PERR#	SMBCLK
17	Gnd	GNT#	41	+3.3V	SMBDAT
18	REQ#	Gnd	42	SERR#	Gnd
19	+3.3V (I/O)	PME#	43	+3.3V	PAR/ECC[0]
20	AD[31]	AD[30]	44	C/BE[1]	AD[15]
21	AD[29]	+3.3V	45	AD[14]	+3.3V
22	Gnd	AD[28]	46	Gnd	AD[13]
23	AD[27]	AD[26]	47	AD[12]	AD[11]
24	AD[25]	Gnd	48	AD[10]	Gnd

PCI Connector (continued)

Pin	Side B	Side A	Pin	Side B	Side A
49	M66EN	AD[09]	71	AD[59]	AD[58]
50	MODE2	Gnd	72	AD[57]	Gnd
51	Gnd	Gnd	73	Gnd	AD[56]
52	AD[08]	C/BE[0]	74	AD[55]	AD[54]
53	AD[07]	+3.3V	75	AD[53]	+3.3V (I/O)
54	+3.3V	AD[06]	76	Gnd	AD[52]
55	AD[05]	AD[04]	77	AD[51]	AD[50]
56	AD[03]	Gnd	78	AD[49]	Gnd
57	Gnd	AD[02]	79	+3.3V (I/O)	AD[48]
58	AD[01]	AD[00]	80	AD[47]	AD[46]
59	+3.3V (I/O)	+3.3V (I/O)	81	AD[45]	Gnd
60	ACK64#/ECC[1]	REQ64#/ECC[6]	82	Gnd	AD[44]
61	+5V	+5V	83	AD[43]	AD[42]
62	+5V	+5V	84	AD[41]	+3.3V (I/O)
KEYWAY, 64 Bit Spacer			85	Gnd	AD[40]
63	Reserved	Gnd	86	AD[39]	AD[38]
64	Gnd	C/BE[7]	87	AD[37]	Gnd
65	C/BE[6]	C/BE[5]	88	+3.3V (I/O)	AD[36]
66	C/BE[4]	+3.3V (I/O)	89	AD[35]	AD[34]
67	Gnd	PAR64/ECC[7]	90	AD[33]	Gnd
68	AD[63]	AD[62]	91	Gnd	AD[32]
69	AD[61]	Gnd	92	Reserved	Reserved
70	+3.3V (I/O)	AD[60]	93	Reserved	Gnd
			94	Gnd	Reserved

Notes

1. Pin labels in *italics* are for PCI-X.
2. +3.3V (I/O) is +1.5V when operating in PCI-X source-synchronous mode.



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	+Vio ⁽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	+Vio ⁽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	+Vio ⁽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	+Vio	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#	+Vio ⁽¹⁾	INTP	INTS
3	INTA#	INTB#	INTC#	+5V	INTD#
2	TCK	+5V	TMS	TDO	TDI
1	+5V	-12V	TRST#	+12V	+5V

Notes

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

Index

Numbers

- 1.5 volt signaling environment, 224
- 3.3 volt signaling environment, 58, 218
- 3.3 vaux, 12
- 5 volt signaling environment, 55
- 16-bit bus, 196
- 64-bit extensions, 46
- 66 MHz PCI, 67
- 66MHZ_CAPABLE, 67, 83

A

- AC drive point, 58
- ACK64#, 10, 46
- acquisition latency, 19
- AD bus, 25
- AD[31::0], 8
- ADB delimited quantum, 182
- address filtering, 107
- address/data stepping, 29
- agent, 13
- allowable disconnect boundaries, 181
- arbitration, 15
- arbitration latency, 18
- asynchronous notification of slot status change, 158

- attention indicator, 153

- attribute phase, 172, 177, 182

B

- bandwidth, 21
- base address registers, 88, 204
- base class, 81
- basic hot swap, 161
- BD_SEL#, 145, 166
- BDSEL#, 160
- BIOS32 Service Directory, 129
- block size, 88
- built-in self-test register, 85
- burst push transactions, 180
- burst transactions, 180
- bus:
 - arbitration, 15
 - commands, 23
 - hierarchy, 105
 - master, 82
 - parking, 18
- byte count modified, 186

C

- C/BE#, 23, 25, 27
- C/BE[3::0], 9

- cache line wrap, 27
- cache-line size (0xC), 86
- capabilities list, 83, 94
- capabilities pointer, 94
- capabilities pointer (0x34), 86
- capability glyphs, 140
- card, 72
- cardbus CIS pointer (0x28), 86
- categories:
 - categories 4 and 5, 218
 - category 1, 217, 224
 - category 2, 218
 - category 3, 218
 - category 6, 218
- central resource, 13, 53
- class 1, 222
- class 2 jitter, 223
- class code, 81
- CLK, 8
- CLKRUN#, 11
- clock jitter, 222
- command register, 203
- command register (0x4), 82
- common clock:
 - mode, 230
 - sampling, 192
- CompactPCI, 137
 - bridging, 149
- completer, 184
 - bus, 1
- CONFIG_ADDRESS, 78
- CONFIG_DATA, 78
- configuration address, 103
- configuration address space, 23, 78
- configuration attributes, 202
- configuration header—type 0, 80

- configuration header—type 1, 104
- connector, 72, 141

D

- data parity error detected, 83
- DC drive point, 58
- DEG#, 144
- delayed transaction, 34
- detected parity error, 39, 83
- device ID, 81
- device ID messages, 193
- DEVSEL#, 9, 25
 - timing, 28, 83
- disconnect, 33, 35
 - A, 35
 - B, 35
 - with data, 35
 - without data, 36
- dual address cycle (DAC), 24, 48
- DWORD, 13
 - transactions, 180

E

- early voltages, 159
- ECC:
 - address and attribute registers, 211
 - control & status register, 210
 - registers, 209
 - signature, 189
- ENUM#, 144, 165
- enumerating the bus, 123
- error correction code, 189
- error detection and reporting, 38
- Eurocard, 139
- expansion ROM (0x30), 91
 - base address, 91
- explicit vs. implicit routing, 195

extended capabilities list, 211
extended configuration space, 211
extended system configuration data, 123

F

fairness in arbitration, 17
FAL#, 144
fast back-to-back:
 capable, 32, 83
 enable, 82
 transactions, 31
find PCI device/class, 131
FireWire, 3
form factors, 72
FRAME#, 9, 25
FSTROBE, 193

G

GA[4::0], 145
generate special cycle, 132
geographic addressing, 145
get interrupt routing options, 133
GNT#, 10, 15, 25
GPIB, 2

H

halt, 45
handle switch, 164
hard metric, 141
hardware connection, 160
header type (0xE), 80
HEALTHY#, 145, 167
high availability (HA), 161
 platforms, 161
host-to-PCI bridge, 101
hot plug:
 controller, 154

primitives, 156
service, 154
system driver, 154

hot swap, 158
 control/status register, 165
 full, 161
 models, 160
 platforms, 161
 processes, 159

I

I/O base and limit, 108
I/O space, 23, 82, 88, 90
IDSEL, 9, 79, 199
IEEE 1394, 3
incident wave switching, 51
initial target latency, 19
initiator, 13
 preempted, 33
INTA, 42
INTA#, 10
INTB, 42
INTC, 42
INTD, 42
INTD#, 10
interoperability matrix, 174
interrupt:
 acknowledge, 24
 acknowledge command, 43
 disable, 82
 handling, 41
 line (0x3C), 88
 pin (0x3D), 87
 routing, 42, 148
 routing table entry, 134
 status, 83

IPMB_PWR, 145

IPMB_SCL, 145

IRDY#, 9, 25

ISA, 4

K

keyway, 72

L

latency, 13, 18

 timer, 19

 timer (0xd), 85

 timer register, 204

LOCK#, 9, 118

low profile, 72

lower byte count, 183

M

M66EN, 11, 67

mapping device number, 202

mask bits, 114

master, 13

master abort, 33

Max_Lat (Maximum Latency) (0x3F), 86

memory, 23

memory base and limit, 108

memory read line, 23, 86

memory read multiple, 23, 86

memory space, 82, 88, 90

memory write and invalidate, 24

 enable, 82

message control register, 114

message signaled interrupt (MSI), 111

message type, 44

Min_Gnt (Minimum Grant) (0x3E), 87

modes, 173

 mode 1, 173, 224

 mode 1 category 1, 224

 mode 2, 213, 173, 224, 232

 mode 2 configuration space, 211

Moore's Law, xi

MSI capability structure, 112

 MSI-X, 114

multifunction devices, 41

N

no snoop, 183

non-hot swap platforms, 161

north bridge, 101

P

pallet bridge, 150

PAR, 9, 38

PAR and PERR#, 38

PAR64, 10

parity error response, 39, 82

PCI, 5

PCI BIOS, 128

PCI BIOS present, 130

PCI data structure, 92

PCI hot plug, 153

PCI Industrial Computer Manufacturer's
 Group (PICM), 138

PCI Special Interest Group, 6

PCI-to-Legacy bus bridge, 102

PCI-to-PCI bridge, 102

PCI-X, 171, 173, 200

 PCI-X 66, 173

 PCI-X 133, 173

 PCI-X 266, 173, 192

 PCI-X 533, 173, 192

 PCI-X capabilities list item, 204

 PCI-X command register, 204

 PCI-X initialization pattern, 214

- PCI-X status register, 207
- PCIXCAP, 213, 232
- pending bits registers, 114
- PERR#, 10, 38
- phase protection, 191
- physical connection, 159
- pin staging, 159
- plug-and-play, 77
- PMB_SDA, 145
- PME#, 12
- post write data, 110
- prefetch read data, 110
- prefetchable base and limit, 109
- prefetchable memory, 90
- primary bus number, 105
- primary interface, 102
- programming interface, 81
- PRSNT[1,2]#, 11
- PRST#, 144
- pull-ups, 53
- Q**
 - qualified signals, 30
 - query hot plug system driver, 156
 - query slot status, 157
- R**
 - read configuration register (Byte, Word, Dword), 132
 - rear panel transition module, 143
 - received master abort, 83
 - received target abort, 83
 - reflected wave switching, 52
 - relaxed ordering, 184
 - REQ#, 10, 15
 - REQ64#, 10, 46
 - requester, 184
 - bus number, 182, 202
 - device number, 182
 - function number, 182
 - reserved, 83
 - resource descriptors, 97
 - resource locking, 118
 - retry, 33, 34
 - revision ID, 81
 - ROM headers, 91
 - ROM signature, 92
 - RST#, 8, 167
- S**
 - secondary bus number, 105, 202
 - secondary interface, 102
 - secondary latency timer, 105
 - secondary status register, 105
 - SERR#, 10
 - SERR# ENABLE, 40, 82
 - set PCI interrupt, 135
 - set slot status, 156
 - shared slots, 72
 - short length, 72
 - shutdown, 45
 - sideband signals, 13
 - signal categories, 217
 - signaled system error, 40
 - signaled system error, 83
 - signaled target abort, 83
 - signaling environments, 54
 - silent drop, 195
 - single-function, 41
 - snoop enable, 116
 - software connection, 160
 - source synchronous clocking, 172
 - source synchronous data transfers, 192

- source synchronous timing, 226
- special cycle, 24, 44, 82, 106
- split completion, 172, 185
 - address, 185
 - attributes, 186
 - error, 187
 - message, 187
- split cycle, 171
- split response, 184
- split transactions, 175, 184
- spread-spectrum clocking, 70
- SSTROBE, 193
- standard length, 72
- status LED, 164
- status register, 203
- status register (0x6), 83
- STOP#, 9, 33
- sub-class, 81
- subordinate bus number, 105
- subsystem device ID, 81
- subsystem vendor ID, 81
- subtractive-decode, 29
- sustained tri-state, 12
- syndrome, 190
- SYSEN, 144
- system errors—SERR#, 39
- system management bus (SMB), 49, 145

T

- tag, 183
- target, 13
 - abort, 33, 36
 - response phase, 178
- test point, 58
- timing parameters, 64
- transaction, 13

- transaction termination—initiator, 33
- transaction termination—target, 33
- TRDY#, 9, 25
- turnaround cycle, 25
- type 0, 103, 200
- type 1, 103, 200
- type 1 configuration transactions, 200

U

- universal board, 54
- upper byte count, 183
- USB, 3

V

- V/I curves, 57, 58, 61
- vendor ID, 81
- VESA local bus, 4
- VGA palette snoop, 82, 116
- V_{io}, 54
- vital product data, 96

W

- wait cycle, 25
 - control, 82
- write configuration register (Byte, Word, Dword), 133

X

- X86-specific message, 45