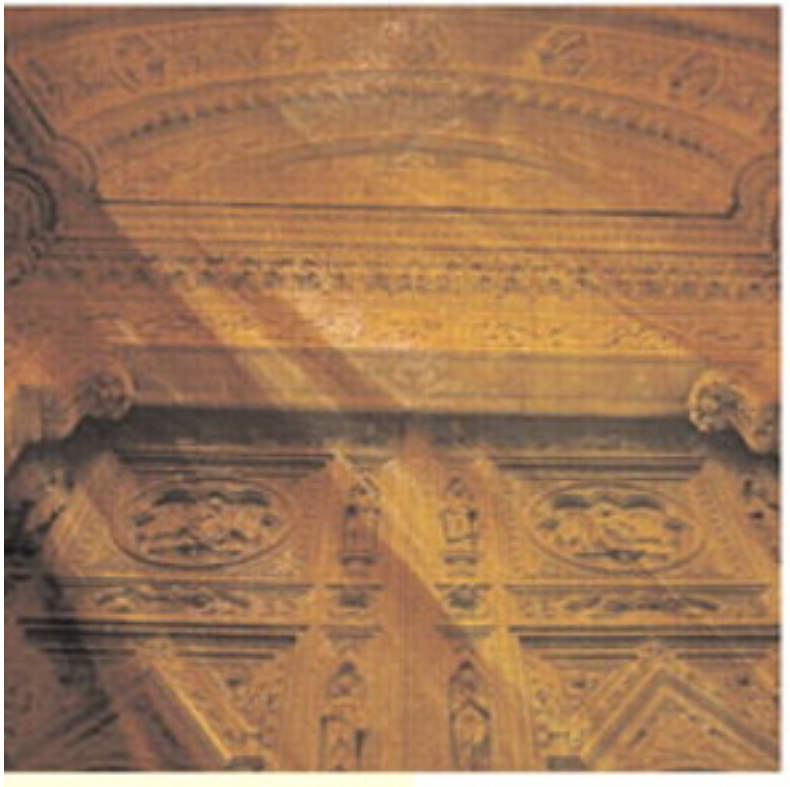
"The 'must-have' PC architecture reference set."

—*PC Magazine*'s "Read Only" column

# PCI System Architecture

Covers PCI Revision 2.2



## Fourth Edition

MindShare, Inc.

Tom Shanley / Don Anderson

PC System Architecture Series

MINDSHARE
BRINGING LIFE TO KNOWLEDGE

# world-class technical training

## Are your company's technical training needs being addressed in the most effective manner?

MindShare has over 25 years experience in conducting technical training on cutting-edge technologies. We understand the challenges companies have when searching for quality, effective training which reduces the students' time away from work and provides cost-effective alternatives. MindShare offers many flexible solutions to meet those needs. Our courses are taught by highly-skilled, enthusiastic, knowledgeable and experienced instructors. We bring life to knowledge through a wide variety of learning methods and delivery options.

## training that fits your needs

MindShare recognizes and addresses your company's technical training issues with:

- Scalable cost training
- Just-in-time training
- Training in a classroom, at your cubicle or home office
- Customizable training options
- Overview and advanced topic courses
- Reducing time away from work
- Training delivered effectively globally
- Concurrently delivered multiple-site training

## MindShare training courses expand your technical skillset

- PCI Express 2.0®
- Intel Core 2 Processor Architecture
- AMD Opteron Processor Architecture
- Intel 64 and IA-32 Software Architecture
- Intel PC and Chipset Architecture
- PC Virtualization
- USB 2.0
- Wireless USB
- Serial ATA (SATA)
- Serial Attached SCSI (SAS)
- DDR2/DDR3 DRAM Technology
- PC BIOS Firmware
- High-Speed Design
- Windows Internals and Drivers
- Linux Fundamentals

... and many more.

All courses can be customized to meet your group's needs. Detailed course outlines can be found at **www.mindshare.com**

# bringing life to knowledge.

real-world tech training put into practice worldwide

*PCI Express ® is a registered trademark of the PCISIG

## MindShare Learning Options

### MindShare Classroom

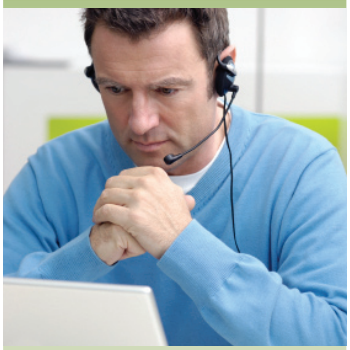
In-House Training

Public Training

#### Classroom Training

Invite MindShare to train you in-house, or sign-up to attend one of our many public classes held throughout the year and around the world. No more boring classes, the 'MindShare Experience' is sure to keep you engaged.

### MindShare Virtual Classroom



Virtual In-House Training

Virtual Public Training

#### Virtual Classroom Training

The majority of our courses live over the web in an interactive environment with WebEx and a phone bridge. We deliver training cost-effectively across multiple sites and time zones. Imagine being trained in your cubicle or home office and avoiding the hassle of travel. Contact us to attend one of our public virtual classes.

### MindShare eLearning

Intro eLearning Modules

Comprehensive eLearning Modules

#### eLearning Module Training

MindShare is also an eLearning company. Our growing list of interactive eLearning modules include:

- **Intro to Virtualization Technology**
- **Intro to IO Virtualization**
- **Intro to PCI Express 2.0 Updates**
- **PCI Express 2.0**
- **USB 2.0**
- **AMD Opteron Processor Architecture**
- **Virtualization Technology**

**...and more**

### MindShare Press


Books

eBooks

#### MindShare Press

Purchase our books and eBooks or publish your own content through us. MindShare has authored over 25 books and the list is growing. Let us help make your book project a successful one.

### Engage MindShare

Have knowledge that you want to bring to life? MindShare will work with you to "Bring Your Knowledge to Life." Engage us to transform your knowledge and design courses that can be delivered in classroom or virtual classroom settings, create online eLearning modules, or publish a book that you author.

**We are proud to be the preferred training provider at an extensive list of clients that include:**

ADAPTEC • AMD • AGILENT TECHNOLOGIES • APPLE • BROADCOM • CADENCE • CRAY • CISCO • DELL • FREESCALE
GENERAL DYNAMICS • HP • IBM • KODAK • LSI LOGIC • MOTOROLA • MICROSOFT • NASA • NATIONAL SEMICONDUCTOR
NETAPP • NOKIA • NVIDIA • PLX TECHNOLOGY • QLOGIC • SIEMENS • SUN MICROSYSTEMS SYNOPSYS • TI • UNISYS

MINDSHARE
BRINGING LIFE TO KNOWLEDGE

4285 SLASH PINE DRIVE COLORADO SPRINGS, CO 80908 USA
M 1.602.617.1123 O 1.800.633.1440 ravi@mindshare.com

www.mindshare.com

# *PCI System Architecture*

## *Fourth Edition*

*MINDSHARE, INC.*

*TOM SHANLEY*
*AND*
*DON ANDERSON*

**ADDISON-WESLEY DEVELOPER'S PRESS**

Reading, Massachusetts • Harlow, England • Menlo Park, California

New York • Don Mills, Ontario • Sydney

Bonn • Tokyo • Amsterdam • Mexico City • Seoul

San Juan • Madrid • Singapore • Paris • Taipei • Milan

Library of Congress Cataloging-in-Publication Data

ISBN: 0-201-30974-2



Sponsoring Editor:
Project Manager:
Cover Design:
Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9-MA-999897
First Printing, May 1999

Addison-Wesley books available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

Find A-W Developers Press on the World-Wide Web at:
http://www/aw.com/devpress/

## About This Book



## Chapter 1: Intro To PCI



## Chapter 2: Intro to PCI Bus Operation



## Chapter 3: Intro to Reflected-Wave Switching

# Contents



## Chapter 4: The Signal Groups



## Chapter 5: PCI Bus Arbitration

## Chapter 6: Master and Target Latency

## Chapter 7: The Commands

## Chapter 8: Read Transfers



## Chapter 9: Write Transfers



## Chapter 10: Memory and IO Addressing

# Contents



## Chapter 11: Fast Back-to-Back & Stepping



## Chapter 12: Early Transaction End

## Chapter 13: Error Detection and Handling

## Chapter 14: Interrupts

# Contents



## Chapter 15: The 64-bit PCI Extension

## Chapter 16: 66MHz PCI Implementation

## Chapter 19: Configuration Registers

## Chapter 20: Expansion ROMs

## Chapter 21: Add-in Cards and Connectors

## Chapter 22: Hot-Plug PCI

## Chapter 23: Power Management

## Chapter 24: PCI-to-PCI Bridge

## Chapter 25: Transaction Ordering & Deadlocks

## Chapter 26: The PCI BIOS



## Chapter 27: Locking

## Chapter 28: CompactPCI and PMC

# 1 Intro To PCI

## This Chapter

This chapter provides a brief history of PCI, introduces its major feature set, the concept of a PCI device versus a PCI function, and identifies the specifications that this book is based upon.

## The Next Chapter

The next chapter provides an introduction to the PCI transfer mechanism, including a definition of the following basic concepts: burst transfers, the initiator, targets, agents, single and multi-function devices, the PCI bus clock, the address phase, claiming the transaction, the data phase, transaction completion and the return of the bus to the idle state. It defines how a device must respond if the device that it is transferring data with exhibits a protocol violation. Finally, it introduces the "green" nature of PCI—power conservation is stressed in the spec.

## PCI Bus History

Intel made the decision not to back the VESA VL standard because the emerging standard did not take a sufficiently long-term approach towards the problems presented at that time and those to be faced in the coming five years. In addition, the VL bus had very limited support for burst transfers, thereby limiting the achievable throughput.

Intel defined the PCI bus to ensure that the marketplace would not become crowded with various permutations of local bus architectures peculiar to a specific processor bus. The first release of the specification, version 1.0, became available on 6/22/92. Revision 2.0 became available in April of 1993. Revision 2.1 was issued in Q1 of 1995. The latest version, 2.2, was completed on December 18, 1998, and became available in February of 1999.

## PCI Bus Features

PCI stands for ***Peripheral Component Interconnect***. The PCI bus can be populated with adapters requiring fast accesses to each other and/or system memory and that can be accessed by the processor at speeds approaching that of the processor's full native bus speed. It is very important to note that all read and write transfers over the PCI bus can be performed as burst transfers. The length of the burst is determined by the bus master. The target is given the start address and the transaction type at the start of the transaction, but is not told the transfer length. As the master becomes ready to transfer each data item, it informs the target whether or not it's the last one. The transaction completes when the final data item has been transferred.

Figure 1-1 on page 11 illustrates the basic relationship of the PCI, expansion, processor and memory buses.

- The host/PCI bridge, frequently referred to as the North Bridge, connects the host processor bus to the root PCI bus.
- The PCI-to-ISA bridge, frequently referred to as the South Bridge, connects the root PCI bus to the ISA (or EISA) bus. The South Bridge also typically incorporates the Interrupt Controller, IDE Controller, USB Host Controller, and the DMA Controller. The North and South Bridges comprise the chipset.
- One or more PCI-to-PCI bridges (not shown) may be embedded on the root PCI bus, or may reside on a PCI add-in card.
- In addition, a chipset may support more than one North Bridge (not shown).

*Table 1-1:* Major PCI Features

| Feature | Description |
|---|---|
| Processor Independence | Components designed for the PCI bus are PCI-specific, not processor-specific, thereby isolating device design from processor upgrade treadmill. |
| Support for up to approximately 80 PCI functions per PCI bus | A typical PCI bus implementation supports approximately ten electrical loads, and each device presents a load to the bus. Each device, in turn, may contain up to eight PCI functions. |

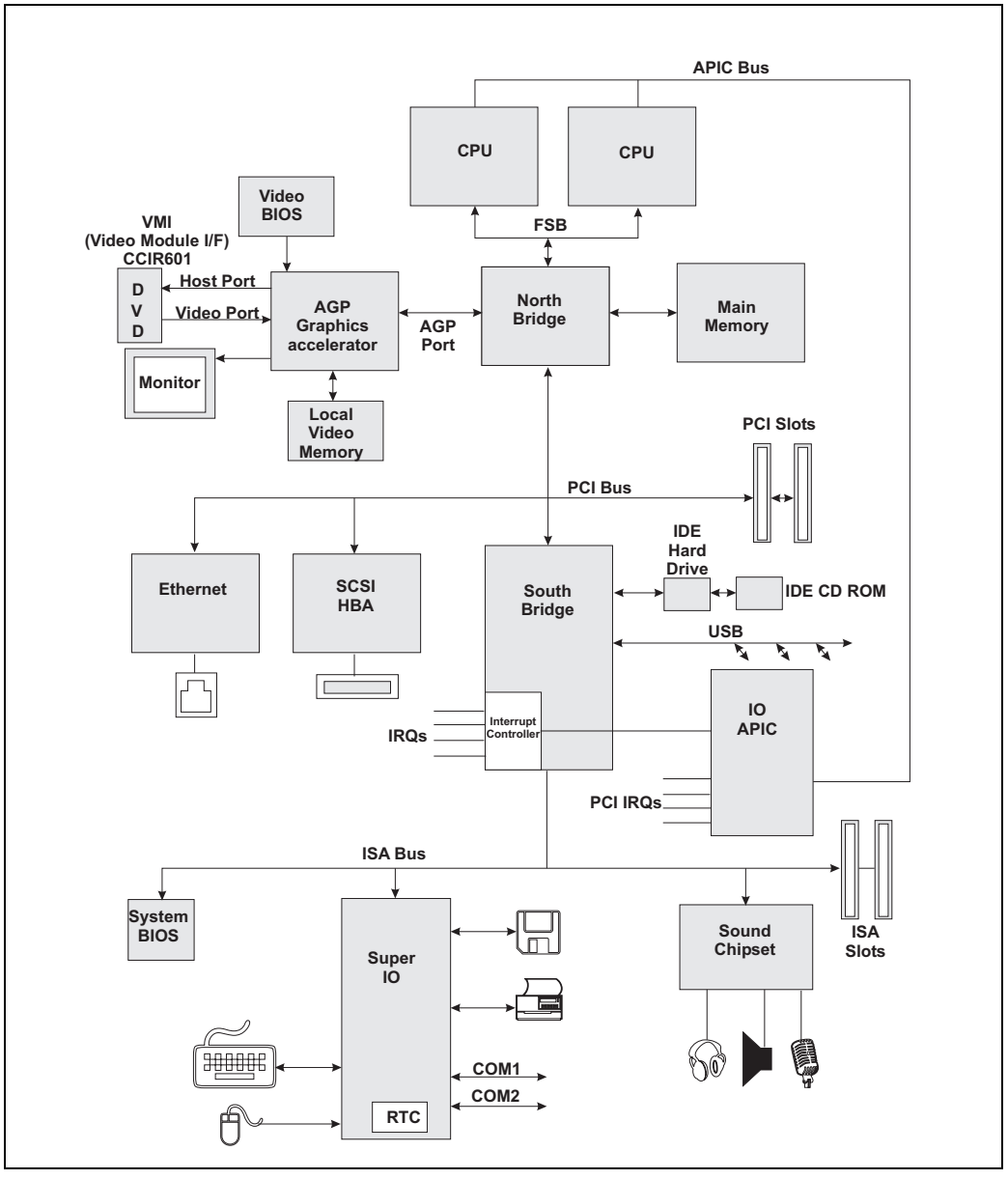
*Table 1-1:* Major PCI Features (Continued)

| Feature | Description |
|---|---|
| Support for up to 256 PCI buses | The specification provides support for up to 256 PCI buses. |
| Low-power consumption | A major design goal of the PCI specification is the creation of a system design that draws as little current as possible. |
| Bursts can be performed on all read and write transfers | A 32-bit PCI bus supports a 132Mbytes per second peak transfer rate for both read and write transfers, and a 264Mbytes per second peak transfer rate for 64-bit PCI transfers. Transfer rates of up to 528Mbytes per second are achievable on a 64-bit, 66MHz PCI bus. |
| Bus speed | Revision 2.0 spec supported PCI bus speeds up to 33MHz. Revision 2.1 adds support for 66MHz bus operation. |
| 64-bit bus width | Full definition of a 64-bit extension. |
| Access time | As fast as 60ns (at a bus speed of 33MHz when an initiator parked on the PCI bus is writing to a PCI target). |
| Concurrent bus operation | Bridges support full bus concurrency with processor bus, PCI bus (or buses), and the expansion bus simultaneously in use. |
| Bus master support | Full support of PCI bus masters allows peer-to-peer PCI bus access, as well as access to main memory and expansion bus devices through PCI-to-PCI and expansion bus bridges. In addition, a PCI master can access a target that resides on another PCI bus lower in the bus hierarchy. |
| Hidden bus arbitration | Arbitration for the PCI bus can take place while another bus master is performing a transfer on the PCI bus. |
| Low-pin count | Economical use of bus signals allows implementation of a functional PCI target with 47 pins and an initiator with 49 pins. |
| Transaction integrity check | Parity checking on the address, command and data. |
| Three address spaces | Memory, I/O and configuration address space. |

*Table 1-1:* Major PCI Features (Continued)

| Feature | Description |
|---|---|
| Auto-Configuration | Full bit-level specification of the configuration registers necessary to support automatic device detection and configuration. |
| Software Transparency | Software drivers utilize same command set and status definition when communicating with PCI device or its expansion bus-oriented cousin. |
| Add-In Cards | The specification includes a definition of PCI connectors and add-in cards. |
| Add-In Card Size | The specification defines three card sizes: long, short and variable-height short cards. |

*Figure 1-1: The PCI System*



# 2 Intro to PCI Bus Operation

## The Previous Chapter

The previous chapter provided a brief history of PCI, introduced it's major feature set, the concept of a PCI device versus a PCI function, and identified the specifications that this book is based upon. It also provided information on contacting the PCI SIG.

## In This Chapter

This chapter provides an introduction to the PCI transfer mechanism, including a definition of the following basic concepts: burst transfers, the initiator, targets, agents, single and multi-function devices, the PCI bus clock, the address phase, claiming the transaction, the data phase, transaction completion and the return of the bus to the idle state. It defines how a device must respond if the device that it is transferring data with exhibits a protocol violation. Finally, it introduces the "green" nature of PCI—power conservation is stressed in the spec.

## The Next Chapter

Unlike many buses, the PCI bus does not incorporate termination resistors at the physical end of the bus to absorb voltage changes and prevent the wavefront caused by a voltage change from being reflected back down the bus. Rather, PCI uses reflections to advantage. The next chapter provides an introduction to reflected-wave switching.

## Burst Transfer

Refer to Figure 2-1 on page 16. A burst transfer is one consisting of a single address phase followed by two or more data phases. The bus master only has to arbitrate for bus ownership one time. The start address and transaction type are issued during the address phase. All devices on the bus latch the address and

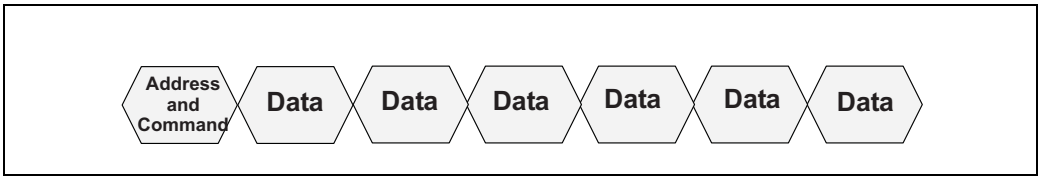
transaction type and decode them to determine which is the target device. The target device latches the start address into an address counter (assuming it supports burst mode—more on this later) and is responsible for incrementing the address from data phase to data phase.

PCI data transfers can be accomplished using burst transfers. Many PCI bus masters and target devices are designed to support burst mode. It should be noted that a PCI target may be designed such that it can only handle single data phase transactions. When a bus master attempts to perform a burst transaction, the target forces the master to terminate the transaction at the completion of the first data phase. The master must re-arbitrate for the bus to attempt resumption of the burst with the next data item. The target terminates each burst transfer attempt when the first data phase completes. This would yield very poor performance, but may be the correct approach for a device that doesn't require high throughput. Each burst transfer consists of the following basic components:

- The address and transfer type are output during the **address phase**.
- A data object (up to 32-bits in a 32-bit implementation or 64-bits in a 64-bit implementation) may then be transferred during each subsequent **data phase**.

Assuming that neither the initiator (i.e., the master) nor the target device inserts wait states in each data phase, a data object (a dword or a quadword) may be transferred on the rising-edge of each PCI clock cycle. At a PCI bus clock frequency of 33MHz, a transfer rate of 132Mbytes/second may be achieved. A transfer rate of 264Mbytes/second may be achieved in a 64-bit implementation when performing 64-bit transfers during each data phase. A 66MHz PCI bus implementation can achieve 264 or 528Mbytes/second transfer rates using 32- or 64-bit transfers. This chapter introduces the burst mechanism used to performing block transfers over the PCI bus.

*Figure 2-1: Example Burst Data Transfer*



## Initiator, Target and Agents

There are two participants in every PCI burst transfer: the initiator and the target. The initiator, or bus master, is the device that initiates a transfer. The terms bus master and initiator can be used interchangeably and frequently are in the PCI specification. The target is the device currently addressed by the initiator for the purpose of performing a data transfer. PCI initiator and target devices are commonly referred to as PCI-compliant agents in the spec.

## Single- Vs. Multi-Function PCI Devices

A PCI physical device package may take the form of a component integrated onto the system board or may be implemented on a PCI add-in card. Each PCI package (referred to in the spec as a **device**) may incorporate from one to eight separate functions. A **function** is a logical device. This is analogous to a multi-function card found in any ISA, EISA or Micro Channel machine.

- A package containing one function is referred to as a **single-function** PCI **device**,
- while a package containing two or more PCI functions is referred to as a **multi-function PCI device**.

Each function contains its own, individually-addressable configuration space, 64 dwords in size. Its configuration registers are implemented in this space. Using these registers, the configuration software can automatically detect the function's presence, determine its resource requirements (memory space, IO space, interrupt line, etc.), and can then assign resources to the function that are guaranteed not to conflict with the resources assigned to other devices.

## PCI Bus Clock

Refer to the CLK signal in Figure 2-2 on page 19. All actions on the PCI bus are synchronized to the PCI CLK signal. The frequency of the CLK signal may be anywhere from 0MHz to 33MHz. The revision 1.0 specification stated that all devices must support operation from 16 to 33MHz, while recommending support for operation down to 0MHz (in other words, when the clock has been stopped as a power conservation strategy). The revision 2.x (x = 1 or 2) PCI specification indicates that **all** PCI **devices must support** PCI **operation within the 0MHz to 33MHz range**. Support for operation down to 0MHz provides

low-power and static debug capability. On a bus with a clock running at 33MHz or slower, the PCI CLK frequency may be changed at any time and may be stopped (but only in the low state). Components integrated onto the system board may be designed to only operate at a single frequency and may require a policy of no frequency change (and the system board designer would ensure that the clock frequency remains unchanged). Devices on add-in cards must support operation from 0 through 33MHz (because the card must operate in any platform that it may be installed in).

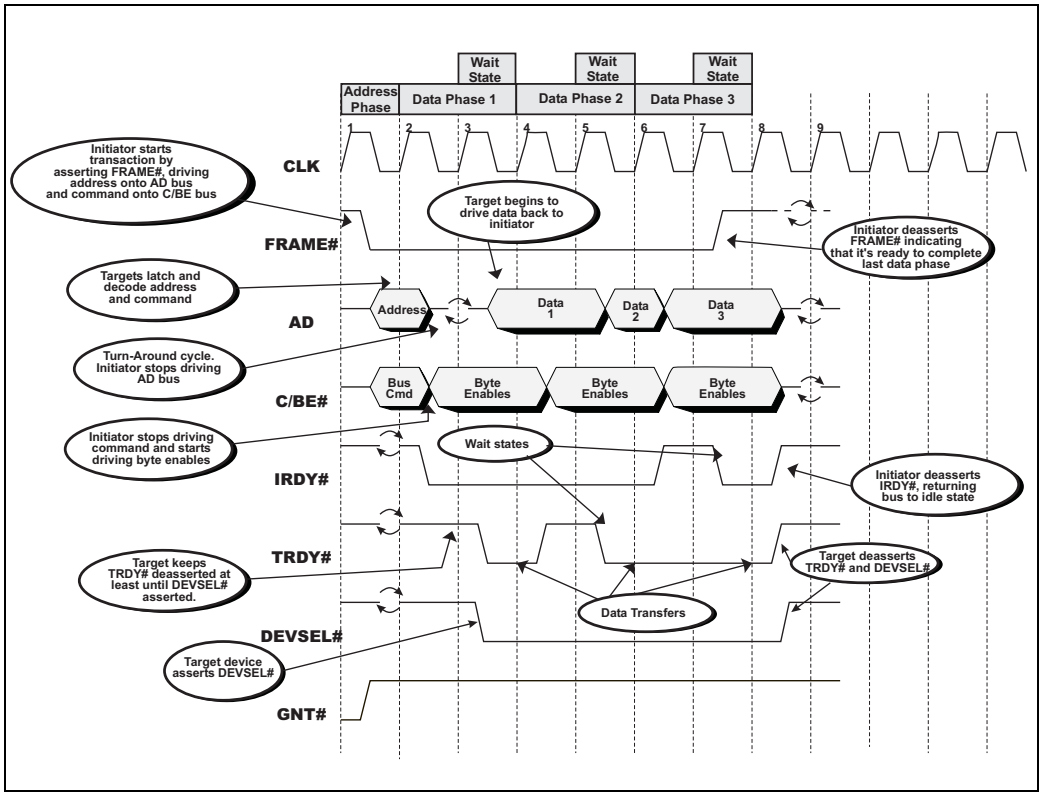
The revision 2.1 specification also defined PCI bus operation at speeds of up to 66MHz. The chapter entitled "66MHz PCI Implementation" describes the operational characteristics of the 66MHz PCI bus, embedded devices and add-in cards.

## Address Phase

Refer to Figure 2-2 on page 19. Every PCI transaction (with the exception of a transaction using 64-bit addressing) starts off with an address phase one PCI clock period in duration (the only exception is a transaction wherein the initiator uses 64-bit addressing delivered in two address phases and consuming two PCI clock periods—this topic is covered in “64-bit Addressing” on page 287). During the address phase, the initiator identifies the target device (via the address) and the type of transaction (also referred to as the command type). The target device is identified by driving a start address within its assigned range onto the PCI address/data bus. At the same time, the initiator identifies the type of transaction by driving the command type onto the 4-bit wide PCI Command/Byte Enable bus. The initiator also asserts the FRAME# signal to indicate the presence of a valid start address and transaction type on the bus. Since the initiator only presents the start address and command for one PCI clock cycle, it is the responsibility of every PCI target device to latch the address and command on the next rising-edge of the clock so that it may subsequently be decoded.

By decoding the address latched from the address bus and the command type latched from the Command/Byte Enable bus, a target device can determine if it is being addressed and the type of transaction in progress. It's important to note that the initiator only supplies a start address to the target (during the address phase). Upon completion of the address phase, the address/data bus becomes the data bus for the duration of the transaction and is used to transfer data in each of the data phases. It is the responsibility of the target to latch the start address and to auto-increment it (assuming that the target supports burst transfers) to point to the next group of locations (a dword or a quadword) during each subsequent data transfer.

*Figure 2-2: Typical PCI Transaction*




## Claiming the Transaction

Refer to Figure 2-2 on page 19. When a PCI target determines that it is the target of a transaction, it must claim the transaction by asserting DEVSEL# (Device Select). If the initiator doesn't sample DEVSEL# asserted within a predetermined amount of time, it aborts the transaction.

## Data Phase(s)

Refer to Figure 2-2 on page 19. The data phase of a transaction is the period during which a data object is transferred between the initiator and the target. The number of data bytes to be transferred during a data phase is determined by the number of Command / Byte Enable signals that are asserted by the initiator during the data phase.

# 3 Intro to Reflected-Wave Switching

## Prior To This Chapter

The previous chapter provided an introduction to the PCI transfer mechanism, including a definition of the following basic concepts: burst transfers, the initiator, targets, agents, single and multi-function devices, the PCI bus clock, the address phase, claiming the transaction, the data phase, transaction completion and the return of the bus to the idle state. It defined how a device must respond if the device that it is transferring data with exhibits a protocol violation. Finally, it introduced the "green" nature of PCI—power conservation is stressed in the spec.

## In This Chapter

Unlike many buses, the PCI bus does not incorporate termination resistors at the physical end of the bus to absorb voltage changes and prevent the wavefront caused by a voltage change from being reflected back down the bus. Rather, PCI uses reflections to advantage. This chapter provides an introduction to reflected-wave switching.

## The Next Chapter

The next chapter provides an introduction to the signal groups that comprise the PCI bus.

## Each Trace Is a Transmission Line

Refer to Figure 3-1 on page 25. Consider the case where a signal trace is fed by a driver and is attached to a number of device inputs distributed along the signal trace. In the past, in order to specify the strength of the driver to be used, the system designer would ignore the electrical characteristics of the trace itself and only factor in the electrical characteristics of the devices connected to the trace. This approach was acceptable when the system clock rate was down in the 1MHz range. The designer would add up the capacitance of each input con-

nected to the trace and treat it as a lumped capacitance. This value would be used to select the drive current capability of the driver. In high-frequency environments such as PCI, traces must switch state at rates from 25MHz on up. At these bus speeds, traces act as transmission lines and the electrical characteristics of the trace must also be factored into the equation used to select the characteristics of the output driver.

A transmission line presents impedance to the driver attempting to drive a voltage change onto the trace and also imposes a time delay in the transmission of the voltage change along the trace. The typical trace's impedance ranges from 50 to 110 Ohms. The width of the trace and the distance of the trace from a ground plane are the major factors that influence its impedance. A wide trace located close to a ground plane is more capacitive in nature and its impedance is close to 50 Ohms. A narrow trace located far from a ground plane is more inductive in nature and its impedance is in the area of 110 Ohms Each device input attached to the trace is largely capacitive in nature. This has the effect of decreasing the overall impedance that the trace offers to a driver.

## Old Method: Incident-Wave Switching

Consider the case where the driver at position one in Figure 3-1 on page 25 must drive the signal line from a logic high to a logic low. Assume that the designer has selected a strong output driver that is capable of driving the signal line from a high to a low at the point of incidence (the point at which it starts to drive). This is referred to as incident-wave switching. As the wavefront propagates down the trace (toward device 10), each device it passes detects a logic low. The amount of time it takes to switch all of the inputs along the trace to a low would be the time it takes the signal to propagate the length of the trace. This would appear to be the best approach because all device inputs are switched in the quickest possible time (one traversal of the trace).

There are negative effects associated with this approach, however. As mentioned earlier, the capacitance of each input along the trace adds capacitance and thus lowers the overall impedance of the trace. The typical overall impedance of the trace would typically be around 30 Ohms. When a 5V device begins to drive a trace, a voltage divider is created between the driver's internal impedance and the impedance of the trace that it is attempting to drive. Assuming that a 20 Ohm driver is attempting to drive a 30 Ohm trace, two of the five volts is dropped within the driver and a three volt incident voltage is propagated onto the trace. Since current = voltage divided by resistance, the current that must be sourced by the driver = 2 volts / 20 Ohms, or 100ma.

When considered by itself, this doesn't appear to present a problem. Assume, however, that a device driver must simultaneously drive 32 address traces, four command traces and four other signals. This is not atypical in a 32-bit bus architecture. Assuming that all of the drivers are encapsulated in one driver package, the package must source four Amps of current, virtually instantaneously (in as short a period as one nanosecond). This current surge presents a number of problems:

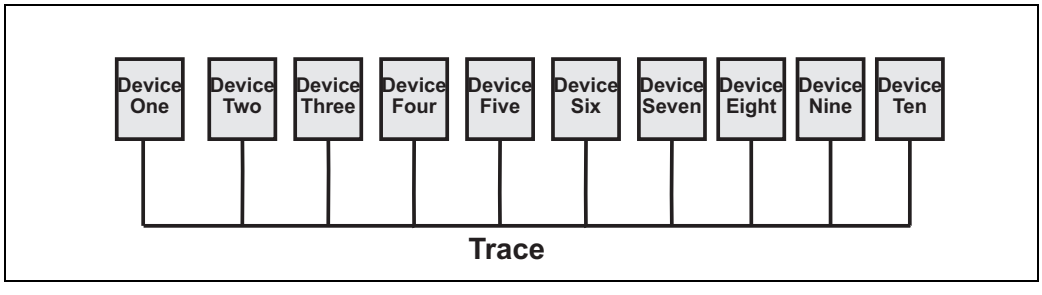
- extremely difficult to decouple.
- causes spikes on internal bond wires.
- increases EMI.
- causes crosstalk inside and outside of the package.

This is the reason that most strong drivers are available in packages that encapsulate only eight drivers. In addition, 20 Ohm output drivers consume quite a bit of silicon real-estate and become quite warm at high frequencies.

Another side-effect occurs when the signal wavefront arrives at the physical end of the trace (at device 10 in Figure 3-1 on page 25). If the designer does not incorporate a terminating resistor at the end of the trace (and the PCI bus is not terminated), the trace stub presents a very high impedance to the signal. Since the signal cannot proceed, it turns around and is reflected back down the bus. During the return passage of the wavefront, this effectively doubles the voltage change seen on the trace at each device's input and at the driver that originated the wavefront. When an incident-wave driver is used (as in this case), the already high voltage it drives onto the trace is doubled. In order to absorb the signal at the physical end of the trace, the system designer frequently includes a terminating resistor.

The usage of incident-wave switching consumes a significant amount of power and violates the green nature of the PCI bus.

*Figure 3-1: Device Loads Distributed Along a Trace*



## PCI Method: Reflected-Wave Switching

Refer to Figure 3-1 on page 25 and Figure 3-2 on page 27. The PCI bus is unterminated and uses wavefront reflection to advantage. A carefully selected, relatively weak output driver is used to drive the signal line partially towards the desired logic state (as illustrated at point **A** in Figure 3-2 on page 27). The driver only has to drive the signal line partially towards its final state, rather than completely (as a strong incident-wave driver would). No inputs along the trace will sample the signal until the next rising-edge of the clock.

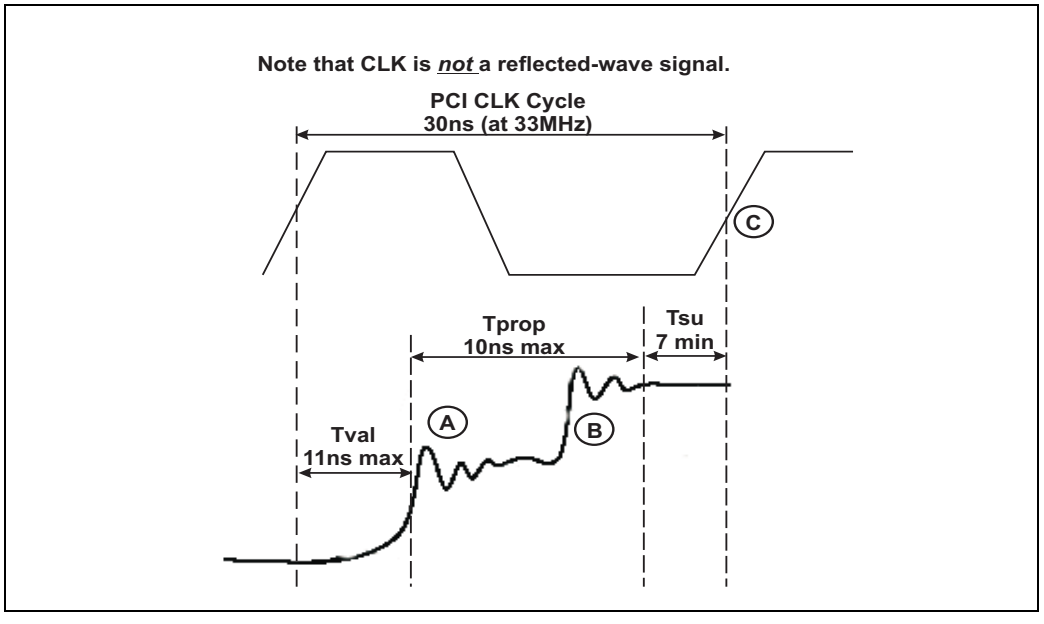
When the wavefront arrives at the unterminated end of the bus, it is reflected back and doubled (see point **B** in Figure 3-2 on page 27). Upon passing each device input again during the wavefront's return trip down the trace, a valid logic level registers at the input on each device. The signal is not sampled, however, until the next rising-edge of the PCI clock (point **C** in the figure). Finally, the wavefront is absorbed by the low-impedance within the driver. This method cuts driver size and surge current in half. There are three timing parameters associated with PCI signal timing:

- **Tval**. PCI devices always start driving a signal on the rising-edge of the PCI clock. Tval is the amount of time it takes the output driver to drive the signal a single-step towards its final logic state. The driver must ensure that its output voltage reaches a specified level (Vtest for 5vdc or Vstep for 3.3vdc switching) to ensure that a valid logic level is detected by the receivers on the next rising edge of the clock.
- **Tprop** (propagation delay). This is the amount of time that it takes the wavefront to travel to the other end of the trace, reflect (thereby doubling the voltage swing), and travel back down the trace.
- **Tsu** (setup time). The signal must have settled in its final state at all inputs at least this amount of time prior to the next rising-edge of the clock (when all receiving devices sample their inputs). The setup times for the REQ# and GNT# signals (these are point-to-point signals; all others are bussed between all devices) deviate from the value illustrated: REQ# setup time is 12ns, while GNT# has a setup time of 10ns. The setup time for all other input signals is 7ns.
- **Th** (hold time). This is the amount of time that signals must be held in their current logic state after the sample point (i.e., the rising-edge of the clock) and is specified as 0ns for PCI signals. This parameter is not illustrated in Figure 3-2 on page 27 and Figure 3-3 on page 28.

In many systems, correct operation of the PCI bus relies on diodes embedded within devices to limit reflections and to successfully meet the specified propagation delay. If a system has long trace runs without connection to a PCI component (e.g., a series of unpopulated add-in connectors), it may be necessary to add diode terminators at that end of the bus to ensure signal quality.

The PCI specification states that devices must only sample their inputs on the rising-edge of the PCI clock signal. The physical layout of the PCI bus traces are very important to ensure that signal propagation is within assigned limits. When a driver asserts or deasserts a signal, the wavefront must propagate to the physical end of the bus, reflect back and make the full passage back down the bus before the signal(s) is sampled on the next rising-edge of the PCI clock. At 33MHz, the propagation delay is specified as 10ns, but may be increased to 11ns by lowering the clock skew from component to component. The specification contains a complete description of trace length and electrical characteristics.

*Figure 3-2: High-Going Signal Reflects and Is Doubled*



# 4 The Signal Groups

## The Previous Chapter

The previous chapter provided an introduction to reflected-wave switching.

## This Chapter

This chapter divides the PCI bus signals into functional groups and describes the function of each signal.

## The Next Chapter

When a PCI bus master requires the use of the PCI bus to perform a data transfer, it must request the use of the bus from the PCI bus arbiter. The next chapter provides a detailed discussion of the PCI bus arbitration timing. The PCI specification defines the timing of the request and grant handshaking, but not the procedure used to determine the winner of a competition. The algorithm used by a system's PCI bus arbiter to decide which of the requesting bus masters will be granted use of the PCI bus is system-specific and outside the scope of the specification.

## Introduction

This chapter introduces the signals utilized to interface a PCI-compliant device to the PCI bus. Figures 2-1 and 3-2 illustrate the required and optional signals for master and target PCI devices, respectively. A PCI device that can act as the initiator or target of a transaction would obviously have to incorporate both initiator and target-related signals. In actuality, there is no such thing as a device that is purely a bus master and never a target. At a minimum, a device must act as the target of configuration reads and writes.

Each of the signal groupings are described in the following sections. It should be noted that some of the optional signals are not optional for certain types of PCI agents. The sections that follow identify the circumstances where signals must be implemented.

*Figure 4-1: PCI-Compliant Master Device Signals*

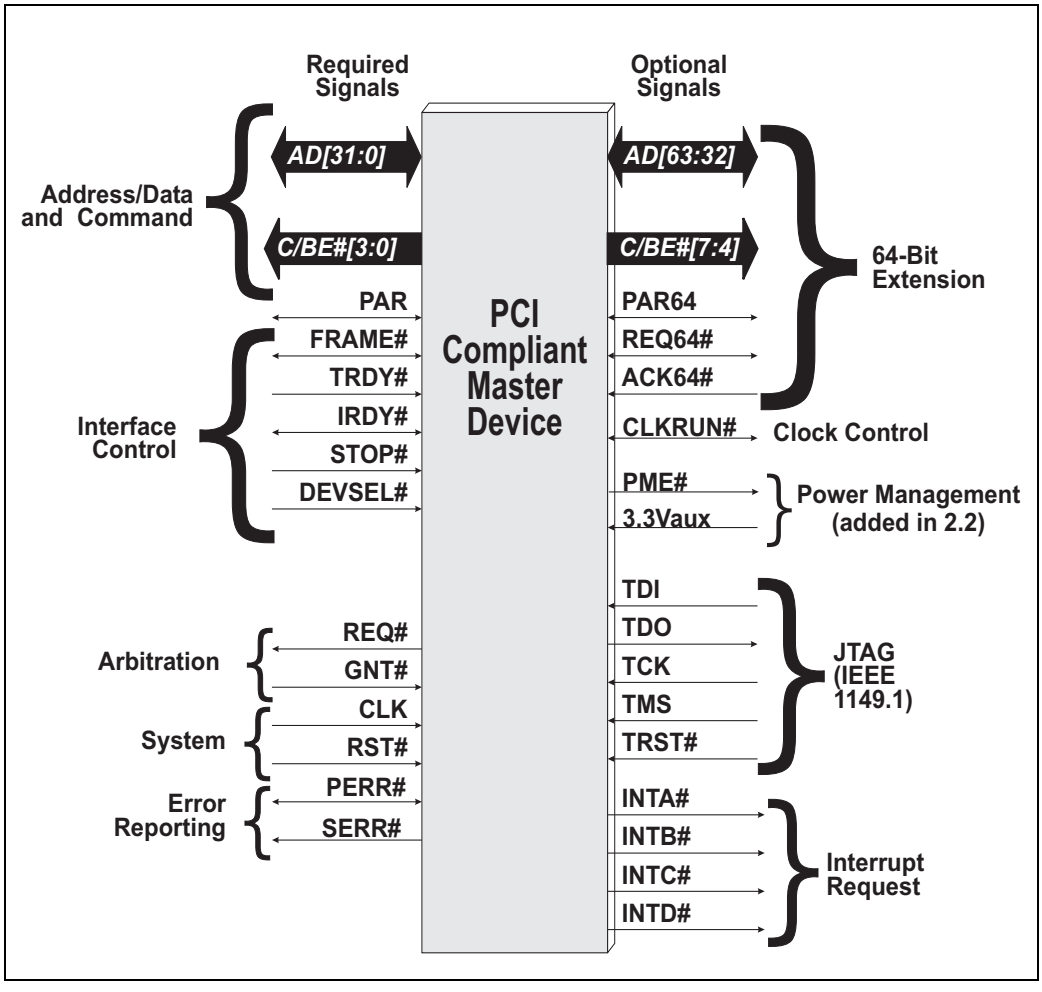
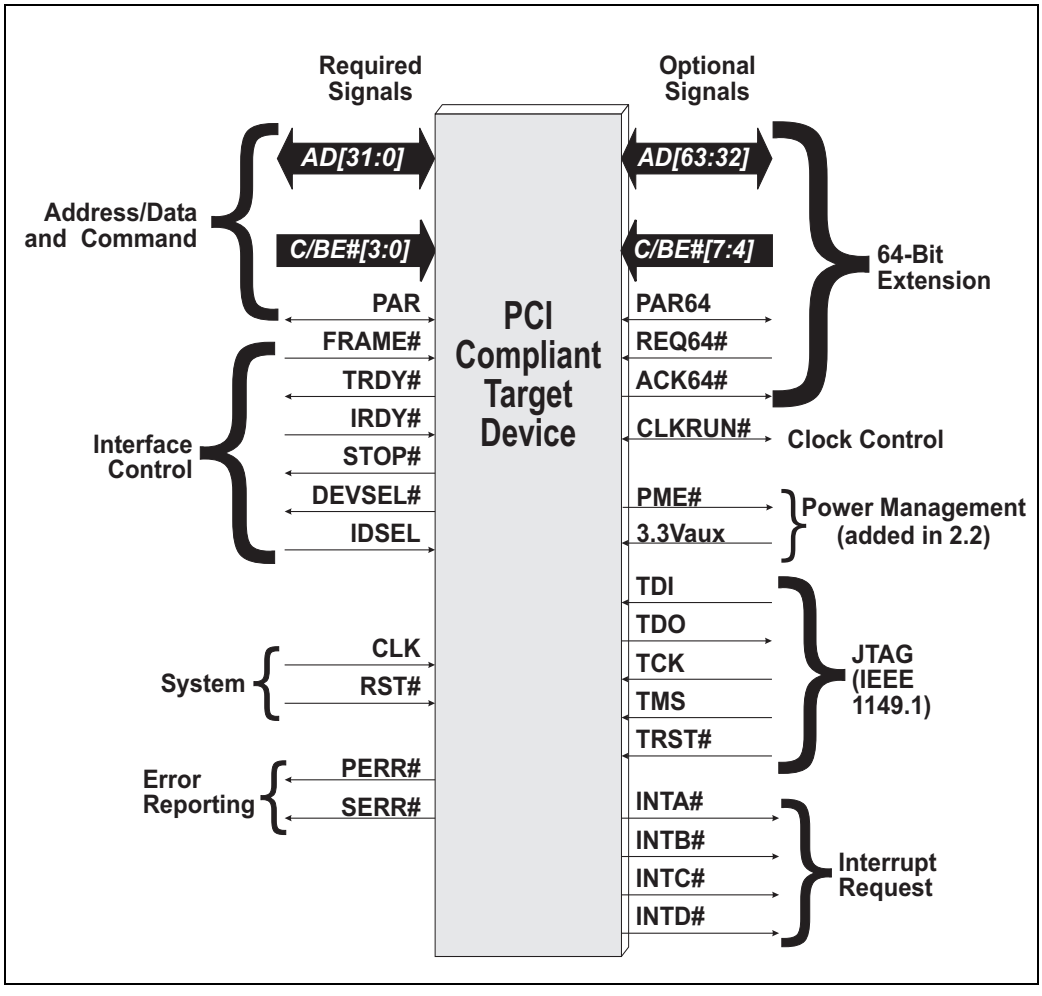


*Figure 4-2: PCI-Compliant Target Device Signals*



## System Signals

### PCI Clock Signal (CLK)

The CLK signal is an input to all devices residing on the PCI bus. It is ***not*** a reflected-wave signal. It provides timing for all transactions, including bus arbitration. All inputs to PCI devices are sampled on the rising edge of the CLK signal. The state of all input signals are don't-care at all other times. All PCI timing parameters are specified with respect to the rising-edge of the CLK signal.

All actions on the PCI bus are synchronized to the PCI CLK signal. The frequency of the CLK signal may be anywhere from 0MHz to 33MHz. The revision 1.0 PCI specification stated that all devices must support operation from 16 to 33MHz and it strongly recommended support for operation down to 0MHz for static debug and low power operation. The revision 2.x PCI specification indicates that **ALL** PCI devices (with one exception noted below) **MUST** support PCI operation within the 0MHz to 33MHz range.

The clock frequency may be changed at any time as long as:

- The clock edges remain clean.
- The minimum clock high and low times are not violated.
- There are no bus requests outstanding.
- LOCK# is not asserted.

The clock may only be stopped in a low state (to conserve power).

As an exception, components designed to be integrated onto the system board may be designed to operate at a fixed frequency (of up to 33MHz) and may only operate at that frequency.

For a discussion of 66MHz bus operation, refer to the chapter entitled "66MHz PCI Implementation" on page 299

## CLKRUN# Signal

### Description

Refer to Figure 4-3 on page 36. The CLKRUN# signal is optional and is defined for the mobile (i.e., portable) environment. It is not available on the PCI add-in connector. This section provides an introduction to this subject. A more detailed description of the mobile environment and the CLKRUN# signal's role can be found in the document entitled *PCI Mobile Design Guide* (available from the SIG). It should be noted that the CLKRUN# signal is required on a Small PCI card connector. This subject is covered in the *Small PCI Specification* and is outside the scope of this book.

Although the PCI specification states that the clock may be stopped or its frequency changed, it does not define a method for determining when to stop (or slow down) the clock, or a method for determining when to restart the clock.

A portable system includes a central resource that includes the PCI clock generation logic. With respect to the clock generation logic (typically part of the chipset), the CLKRUN# signal is a sustained tri-state input/output signal. The clock generation logic keeps CLKRUN# asserted when the clock is running normally. During periods when the clock has been stopped (or slowed), the clock generation logic monitors CLKRUN# to recognize requests from master and target devices for the PCI clock signal to be restored to full speed. The clock cannot be stopped if the bus is not idle. Before it stops (or slows down) the clock frequency, the clock generation logic deasserts CLKRUN# for one clock to inform PCI devices that the clock is about to be stopped (or slowed). After driving CLKRUN# high (deasserted) for one clock, the clock generation logic tri-states its CLKRUN# output driver. The keeper resistor on CLKRUN# then assumes responsibility for maintaining the deasserted state of CLKRUN# during the period in which the clock is stopped (or slowed).

The clock continues to run unchanged for a minimum of four clocks after the clock generation logic deasserts CLKRUN#. After deassertion of CLKRUN#, the clock generation logic must monitor CLKRUN# for two possible cases:

**CASE 1.** After the clock has been stopped (or slowed), a master (or multiple masters) may require clock restart in order to request use of the bus. Prior to issuing the bus request, the master(s) must first request clock restart. This is accomplished by assertion of CLKRUN#. When the clock generation logic detects the assertion of CLKRUN# by another party, it turns on (or speeds up) the clock and turns on its CLKRUN# output driver to assert CLKRUN#.

# 5 PCI Bus Arbitration

## The Previous Chapter

The previous chapter provided a detailed description of the PCI functional signal groups.

## This Chapter

When a PCI bus master requires the use of the PCI bus to perform a data transfer, it must request the use of the bus from the PCI bus arbiter. This chapter provides a detailed discussion of the PCI bus arbitration timing. The PCI specification defines the timing of the request and grant handshaking, but not the procedure used to determine the winner of a competition. The algorithm used by a system's PCI bus arbiter to decide which of the requesting bus masters will be granted use of the PCI bus is system-specific and outside the scope of the specification.

## The Next Chapter

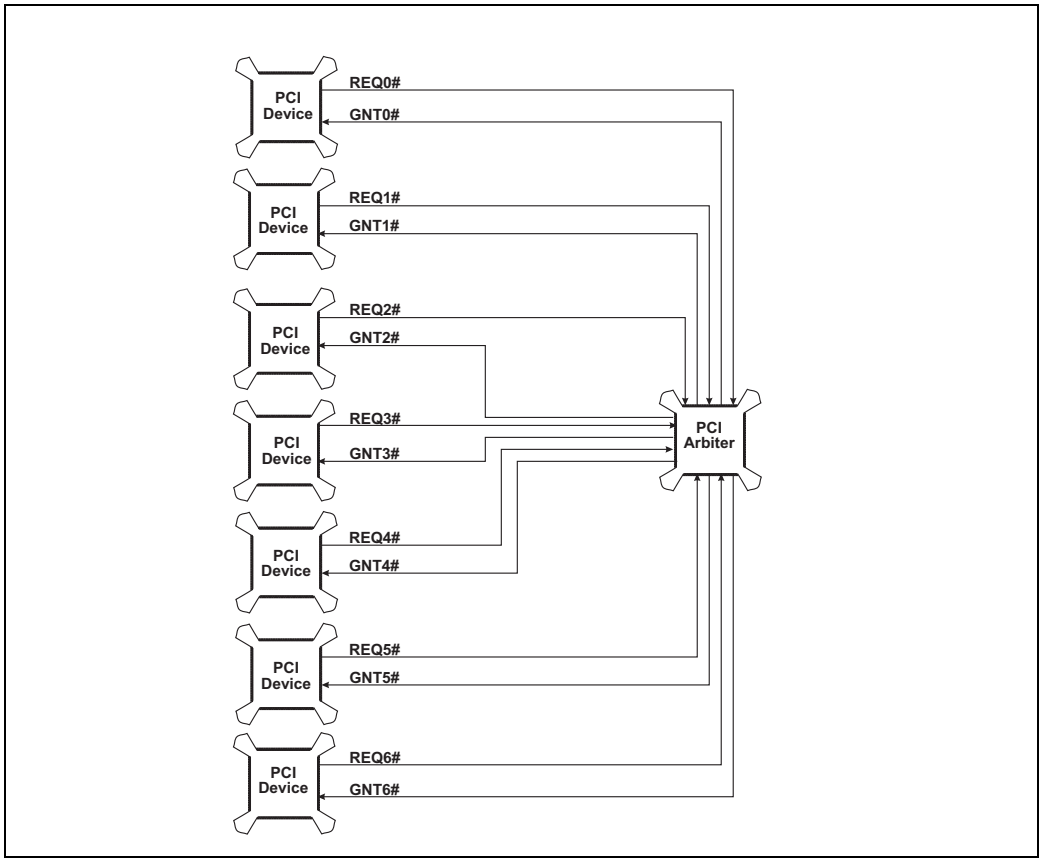
The next chapter describes the rules governing how much time a device may hold the bus in wait states during any given data phase. It describes how soon after reset is removed the first transaction may be initiated and how soon after reset is removed a target device must be prepared to transfer data. The mechanisms that a target may use to meet the latency rules are described: Delayed Transactions, as well as the posting of memory writes.

## Arbiter

At a given instant in time, one or more PCI bus master devices may require use of the PCI bus to perform a data transfer with another PCI device. Each requesting master asserts its REQ# output to inform the bus arbiter of its pending request for the use of the bus. Figure 5-1 on page 60 illustrates the relationship

of the PCI masters to the central PCI resource known as the bus arbiter. In this example, there are seven possible masters connected to the PCI bus arbiter in the illustration. Each master is connected to the arbiter via a separate pair of REQ#/GNT# signals. Although the arbiter is shown as a separate component, it usually is integrated into the PCI chip set; specifically, it is typically integrated into the host/PCI or the PCI/expansion bus bridge chip.

*Figure 5-1: The PCI Bus Arbiter*




## Arbitration Algorithm

As stated at the beginning of this chapter, the PCI specification does not define the scheme used by the PCI bus arbiter to decide the winner of the competition when multiple masters simultaneously request bus ownership. The arbiter may

utilize any scheme, such as one based on fixed or rotational priority or a combination of the two (rotational among one group of masters and fixed within another group). The 2.1 specification states that the arbiter is required to implement a fairness algorithm to avoid deadlocks. The exact verbiage that is used is:

> "The central arbiter is required to implement a fairness algorithm to avoid deadlocks. Fairness means that each potential bus master must be granted access to the bus independent of other requests. Fairness is defined as a policy that ensures that high-priority masters will not dominate the bus to the exclusion of lower-priority masters when they are continually requesting the bus. However, this does not mean that all agents are required to have equal access to the bus. By requiring a fairness algorithm there are no special conditions to handle when LOCK# is active (assuming a resource lock) or when cacheable memory is located on PCI. A system that uses a fairness algorithm is still considered fair if it implements a complete bus lock instead of a resource lock. However, the arbiter must advance to a new agent if the initial transaction attempting to establish a lock is terminated with retry."

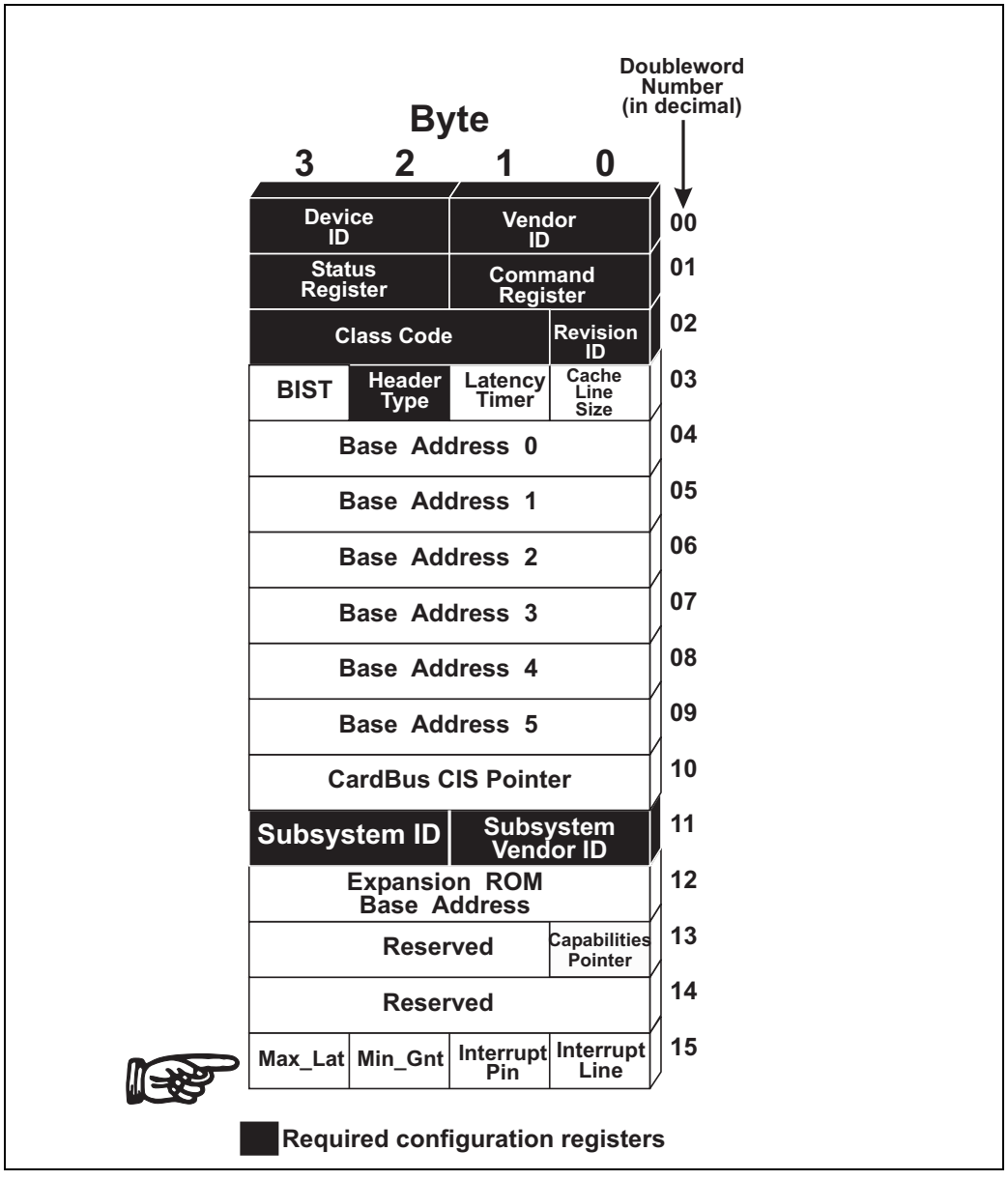
The specification contains an example arbiter implementation that does clarify the intent of the specification. The example can be found in the next section.

Ideally, the bus arbiter should be programmable by the system. If it is, the startup configuration software can determine the priority to be assigned to each member of the bus master community by reading from the Maximum Latency (Max_Lat) configuration register associated with each bus master (see Figure 5-2 on page 62). The bus master designer hardwires this register to indicate, in increments of 250ns, how quickly the master requires access to the bus in order to achieve adequate performance.

In order to grant the PCI bus to a bus master, the arbiter asserts the device's respective GNT# signal. This grants the bus to the master for one transaction (consisting of one or more data phases).

If a master generates a request, is subsequently granted the bus and does not initiate a transaction (assert FRAME#) within 16 PCI clocks after the bus goes idle, the arbiter may assume that the master is malfunctioning. In this case, the action taken by the arbiter would be system design-dependent.

*Figure 5-2: Maximum Latency Configuration Register*



## Example Arbiter with Fairness

A system may divide the overall community of bus masters on a PCI bus into two categories:

1. Bus masters that require fast access to the bus or high throughput in order to achieve good performance. Examples might be the video adapter, an ATM network interface, or an FDDI network interface.
2. Bus masters that don't require very fast access to the bus or high throughput in order to achieve good performance. Examples might be a SCSI host bus adapter or a standard expansion bus master.

The arbiter would segregate the REQ#/GNT# signals into two groups with greater precedence given to those in one group. Assume that bus masters A and B are in the group that requires fast access, while masters X, Y and Z are in the other group. The arbiter can be programmed or designed to treat each group as rotational priority within the group and rotational priority between the two groups. This is pictured in Figure 5-3 on page 64.

Assume the following conditions:

- Master A is the next to receive the bus in the first group.
- Master X is the next to receive it in the second group.
- A master in the first group is the next to receive the bus.
- All masters are asserting REQ# and wish to perform multiple transactions (i.e., they keep their respective REQ# asserted after starting a transaction).

The order in which the masters would receive access to the bus is:

1. Master A.
2. Master B.
3. Master X.
4. Master A.
5. Master B.
6. Master Y.
7. Master A.
8. Master B.
9. Master Z.
10. Master A.
11. Master B.
12. Master X, etc.

The masters in the first group are permitted to access the bus more frequently than those that reside in the second group.

# 6 Master and Target Latency

## The Previous Chapter

The previous chapter provided a detailed description of the mechanism used to arbitrate for PCI bus ownership.

## This Chapter

This chapter describes the rules governing how much time a device may hold the bus in wait states during any given data phase. It describes how soon after reset is removed the first transaction may be initiated and how soon after reset is removed a target device must be prepared to transfer data. The mechanisms that a target may use to meet the latency rules are described: Delayed Transactions, and posting of memory writes.

## The Next Chapter

The next chapter describes the transaction types, or commands, that the initiator may utilize when it has successfully acquired PCI bus ownership.

## Mandatory Delay Before First Transaction Initiated

The 2.2 spec mandates that the system (i.e., the arbiter within the chipset) must guarantee that the first transaction will not be initiated on the PCI bus for at least five PCI clock cycles after RST# is deasserted. This value is referred to as **Trhff** in the spec (Time from Reset High-to-First-FRAME# assertion).

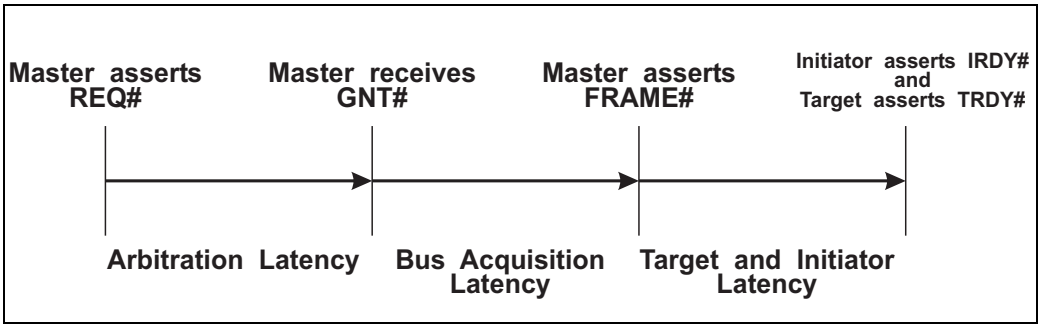
## Bus Access Latency

When a bus master wishes to transfer a block of one or more data items between itself and a target PCI device, it must request the use of the bus from the bus arbiter. Bus access latency is defined as the amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction. Figure 6-1 on page 75 illustrates the different components of the access latency experienced by a PCI bus master. Table 6-1 on page 74 describes each latency component.

*Table 6-1: Access Latency Components*

| Component | Description |
|---|---|
| Bus Access Latency | Defined as the amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction. In other words, it is the sum of arbitration, bus acquisition and target latency. |
| Arbitration Latency | Defined as the period of time from the bus master's assertion of REQ# until the bus arbiter asserts the bus master's GNT#. This period is a function of the arbitration algorithm, the master's priority and whether any other masters are requesting access to the bus. |
| Bus Acquisition Latency | Defined as the period time from the reception of GNT# by the requesting bus master until the current bus master surrenders the bus. The requesting bus master can then initiate its transaction by asserting FRAME#. The duration of this period is a function of how long the current bus master's transaction-in-progress takes to complete. This parameter is the larger of either the current master's LT value (in other words, its timeslice) or the longest latency to first data phase completion in the system (which is limited to a maximum of 16 clocks). |
| Initiator and Target Latency | Defined as the period of time from the start of a transaction until the master and the currently-addressed target are ready to complete the first data transfer of the transaction. This period is a function how fast the master is able to transfer the first data item, as well as the access time for the currently-addressed target device (and is limited to a maximum of 8 clocks for the master and 16 clocks for the target). |

*Figure 6-1: Access Latency Components*




PCI bus masters should always use burst transfers to transfer blocks of data between themselves and a target PCI device (some poorly-designed masters use a series of single-data phase transactions to transfer a block of data). The transfer may consist of anywhere from one to an unlimited number of bytes. A bus master that has requested and has been granted the use of the bus (its GNT# is asserted by the arbiter) cannot begin a transaction until the current bus master completes its transaction-in-progress. If the current master were permitted to own the bus until its entire transfer were completed, it would be possible for the current bus master to starve other bus masters from using the bus for extended periods of time. The extensive delay incurred could cause other bus masters (and/or the application programs they serve) to experience poor performance or even to malfunction (buffer overflows or starvation may be experienced).

As an example, a bus master could have a buffer full condition and is requesting the use of the bus in order to off-load its buffer contents to system memory. If it experiences an extended delay (latency) in acquiring the bus to begin the transfer, it may experience a data overrun condition as it receives more data from its associated device (such as a network) to be placed into its buffer.

In order to insure that the designers of bus masters are dealing with a predictable and manageable amount of bus latency, the PCI specification defines two mechanisms:

- Master Latency Timer (MLT).
- Target-Initiated Termination.

## Pre-2.1 Devices Can Be Bad Boys

Prior to the 2.1 spec, there were some rules regarding how quickly a master or target had to transfer data, but they were not complete, nor were they clearly stated. The following list describes the behavior permitted by the pre-2.1 versions of the spec:

- In any data phase, the master could take any amount of time before asserting IRDY# to transfer a data item.
- At the end of the final data phase, the spec didn't say how quickly the master had to return IRDY# to the deasserted state, thereby returning the bus to the idle state so another master could use it.
- There was no 16 clock first data phase completion rule for the target. It could keep TRDY# deasserted forever if it wanted to.
- There was a subsequent data phase completion rule for the target, but it was not written clearly and provided a huge loop hole that permitted the target to insert any number of wait states in a data phase other than the first one. Basically, it said that the target *should* (there's a fuzzy word that should be banned from every spec) be ready to transfer a data item within eight clocks after entering a data phase. If it couldn't meet this eight clock recommendation, however, then whenever it did become ready to transfer the data item (*could be a gazillion clocks later*), it must assert STOP# along with TRDY#.

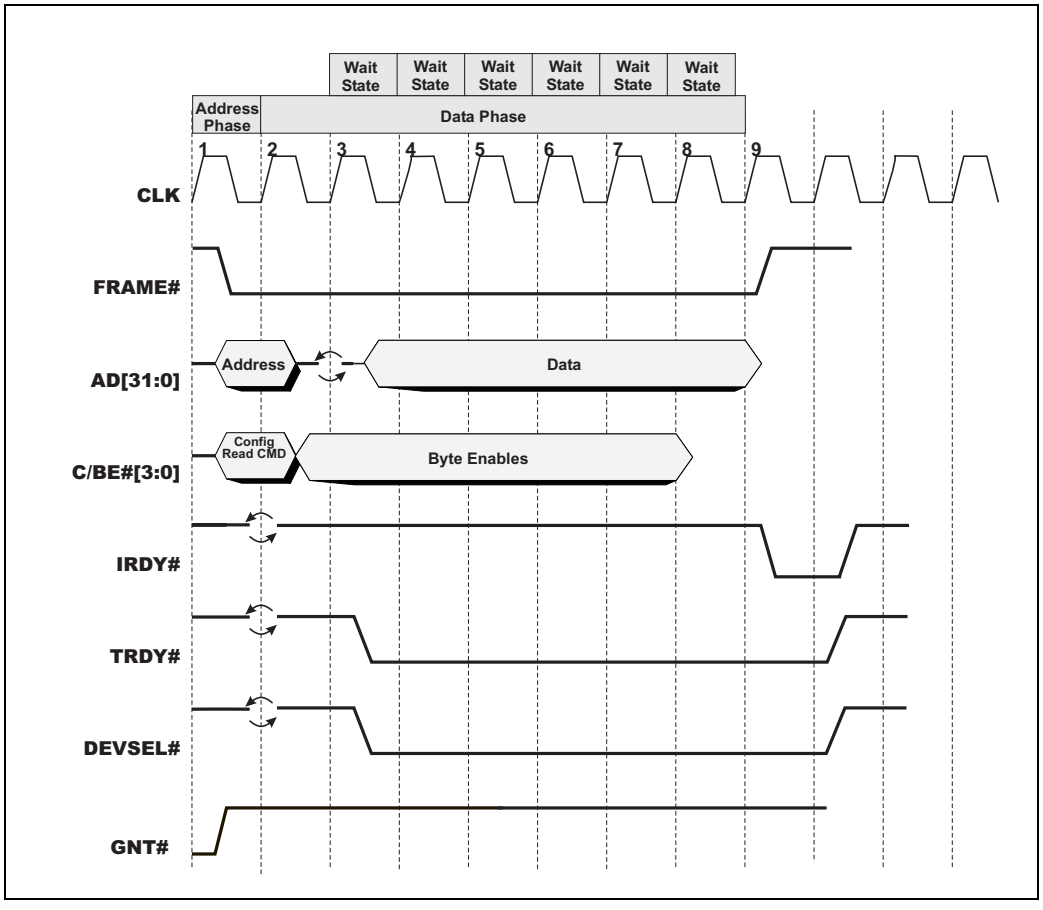
The bottom line is that pre-2.1 targets and masters can exhibit *very* poor behavior that ties up the PCI bus for awful amounts of time. The 2.1 spec closed these loop holes.

## Preventing Master from Monopolizing the Bus

### Master Must Transfer Data Within 8 CLKs

Refer to Figure 6-2 on page 77. It is a rule that the initiator must not keep IRDY# deasserted for more than seven PCI clocks during any data phase. In other words, it must be prepared to transfer a data item within eight clocks after entry into any data phase. If the initiator has no buffer space available to store read data, it must delay requesting the bus until is has room for the data. On a write transaction, the initiator must have the data available before it asks for the bus.

*Figure 6-2: Longest Legal Deassertion of IRDY# In Any Data Phase Is 8 Clocks*




## IRDY# Deasserted In Clock After Last Data Transfer 2.2

***REFER TO THE END OF CLOCK 9 IN FIGURE 6-2 ON PAGE 77. UPON COMPLETION OF THE FINAL DATA TRANSFER (IRDY# AND TRDY# ASSERTED AND FRAME# DEASSERTED), THE MASTER MUST RELEASE THE IRDY# SIGNAL DURING THE NEXT CLOCK PERIOD. THIS RULE WAS ADDED IN THE 2.2 SPEC.***

# 7 The Commands

## The Previous Chapter

The previous chapter described the rules governing how much time a device may hold the bus in wait states during any given data phase. It described how soon after reset is removed the first transaction may be initiated and how soon after reset is removed a target device must be prepared to transfer data. The mechanisms that a target may use to meet the latency rules were described: Delayed Transactions, and posting of memory writes.

## In This Chapter

This chapter defines the types of commands (i.e., transaction types) that a bus master may initiate when it has acquired ownership of the PCI bus.

## The Next Chapter

Using timing diagrams, the next chapter provides a detailed description of PCI read transactions. It also describes the treatment of the Byte Enable signals during both reads and writes.

## Introduction

When a bus master acquires ownership of the PCI bus, it may initiate one of the types of transactions listed in Table 7-1 on page 100. During the address phase of a transaction, the Command / Byte Enable bus, C / BE#[3:0], is used to indicate the command, or transaction, type. Table 7-1 on page 100 provides the setting that the initiator places on the Command / Byte Enable lines to indicate the type of transaction in progress. The sections that follow provide a description of each of the command types.

*Table 7-1: PCI Command Types*

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

# Interrupt Acknowledge Command

## Introduction

In a PC-compatible system, interrupts can be delivered to the processor in one of three ways:

**METHOD 1.** In a single processor system (see Figure 7-1 on page 103), the interrupt controller asserts **INTR** to the x86 processor. In this case, the processor responds with an Interrupt Acknowledge transaction. This section describes that transaction.

**METHOD 2.** In a multi-processor system, interrupts can be delivered to the array of processors over the **APIC** (Advanced Programmable Interrupt Controller) bus in the form of message packets. For more information, refer to the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).

**METHOD 3.** In a system that supports **Message Signaled Interrupts**, interrupts can be delivered to the host/PCI bridge in the form of memory writes. For more information, refer to "Message Signaled Interrupts (MSI)" on page 252.

In response to an interrupt request delivered over the INTR signal line, an Intel x86 processor issues two Interrupt Acknowledge transactions (note that the P6 family processors only issues one) to read the interrupt vector from the interrupt controller. The interrupt vector tells the processor which interrupt service routine to execute.

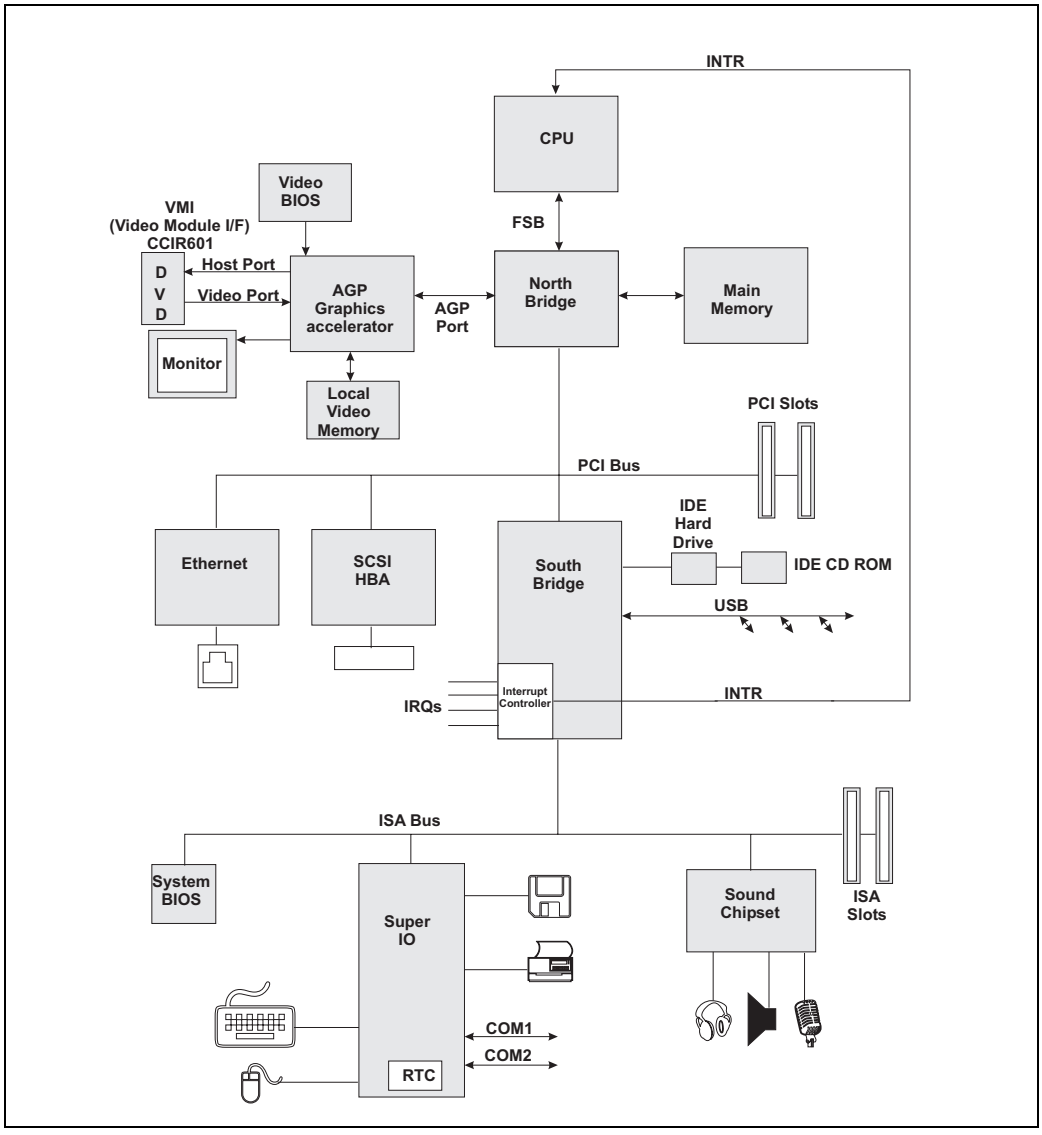
## Background

In an Intel x86-based system, the processor is usually the device that services interrupt requests received from subsystems that require servicing. In a PC-compatible system, the subsystem requiring service issues a request by asserting one of the system interrupt request signals, IRQ0 through IRQ15. When the IRQ is detected by the interrupt controller within the South Bridge (see Figure 7-1 on page 103), it asserts INTR to the host processor. Assuming that the host processor is enabled to recognize interrupt requests (the Interrupt Flag bit in the EFLAGS register is set to one), the processor responds by requesting the interrupt vector from the interrupt controller. This is accomplished by the processor performing the following sequence:

1. **The processor generates an Interrupt Acknowledge bus cycle**. *Please note that a P6 family processor does not generate this first Interrupt Acknowledge bus cycle.* No address is output by the processor because the address of the target device, the interrupt controller, is implicit in the bus cycle type. The purpose of this bus cycle is to **command the interrupt controller to prioritize** its **currently-pending requests** and select the request to be processed. The processor doesn't expect any data to be returned by the interrupt controller during this bus cycle.

2. **The processor generates a second Interrupt Acknowledge bus cycle** to **request the interrupt vector** from the interrupt controller. If this is a P6 family processor, this is the only Interrupt Acknowledge transaction it generates. BE0# is asserted by the processor, indicating that an 8-bit vector is expected to be returned on the lower data path, D[7:0]. To state this more precisely, the processor requests that the interrupt controller return the index into the interrupt table in memory. This tells the processor which table entry to read. The table entry contains the start address of the device-specific interrupt service routine in memory. In response to the second Interrupt Acknowledge bus cycle, the interrupt controller must drive the interrupt table index, or vector, associated with the highest-priority request currently pending back to the processor over the lower data path, D[7:0]. The processor reads the vector from the bus and uses it to determine the start address of the interrupt service routine that it must execute.

*Figure 7-1: Typical PC Block Diagram—Single Processor*



# 8 Read Transfers

## The Previous Chapter

The previous chapter defined the types of commands (i.e., transaction types) that a bus master may initiate when it has acquired ownership of the PCI bus.

## This Chapter

Using timing diagrams, this chapter provides a detailed description of PCI read transactions. It also describes the treatment of the Byte Enable signals during both reads and writes.

## The Next Chapter

The next chapter describes write transactions using example timing diagrams.

## Some Basic Rules For Both Reads and Writes

The ready signal (IRDY# or TRDY#) from the device sourcing the data must be asserted when it starts driving valid data onto the data bus, while the device receiving the data keeps its ready line deasserted until it is ready to receive the data. Once a device's ready signal is asserted, it must remain so until the end of the current data phase (i.e., until the data is transferred).

A device must not alter its control line settings once it has indicated that it is ready to complete the current data phase. Once the initiator has asserted IRDY# to indicate that it's ready to transfer the current data item, it may not change the state of IRDY# or FRAME# regardless of the state of TRDY#. Once a target has asserted TRDY# or STOP#, it may not change TRDY#, STOP# or DEVSEL# until the current data phase completes.

## Parity

Parity generation, checking, error reporting and timing is not discussed in this chapter. This subject is covered in detail in the chapter entitled “Error Detection and Handling” on page 199.

## Example Single Data Phase Read

Refer to Figure 8-1 on page 126. Each clock cycle is numbered for easy reference and begins on its rising-edge. It is assumed that the bus master has already arbitrated for and been granted access to the bus. The bus master then must wait for the bus to become idle. This is accomplished by sampling the state of FRAME# and IRDY# on the rising-edge of each clock (along with GNT#). When both are sampled deasserted (along with GNT# still asserted), the bus is idle and a transaction may be initiated by the bus master.

**CLOCK 1.** On detecting bus idle (FRAME# and IRDY# both deasserted), the initiator starts the transaction on the rising-edge of clock one.

**THE** initiator drives out the address on AD[31:0] and the command on C/BE#[3:0].

**THE** initiator asserts FRAME# to indicate that the transaction has started and that there is a valid address and command on the bus.

**CLOCK 2.** All targets on the bus sample the address, command and FRAME# on the rising-edge of clock two, completing the address phase.

**THE** targets begin the decode to determine which of them is the target of the transaction.

**THE** initiator asserts IRDY# to indicate that is ready to accept the first read data item from the target.

**THE** initiator also deasserts FRAME# when it asserts IRDY#, thereby indicating that it is ready to complete the final data phase of the transaction.

**THE** initiator stops driving the command onto C/BE#[3:0] and starts driving the byte enables to indicate which locations it wished to read from the first dword.

**NO** target asserts DEVSEL# in clock two to claim the transaction.

**CLOCK 3.** On the rising-edge of clock three, the initiator samples DEVSEL# deasserted indicating that the transaction has not yet been claimed by a target. The first (and only) data phase therefore cannot complete yet. It is extended by one clock (a wait state) in clock three.

**DURING** the wait state, the initiator must continue to drive the byte enables and to assert IRDY#. It must continue to drive them until the data phase completes.

**A** target asserts DEVSEL# to claim the transaction.

**THE** target also asserts TRDY# to indicate that it is driving the first dword onto the AD bus.

**CLOCK 4.** Both the initiator and the target sample IRDY# and TRDY# asserted on the rising-edge of clock four. The initiator also latches the data and the assertion of TRDY# indicates that the data is good. The first (and only) data item has been successfully read.

**IF** the target needed to sample the byte enables, it would sample them at this point. In this example, however, the target already supplied the data to the master without consulting the byte enables. This behavior is permitted if it's a well-behaved memory target (one wherein a read from a location doesn't change the content of the location). This is referred to as Prefetchable memory and is described in "What Is Prefetchable Memory?" on page 93.

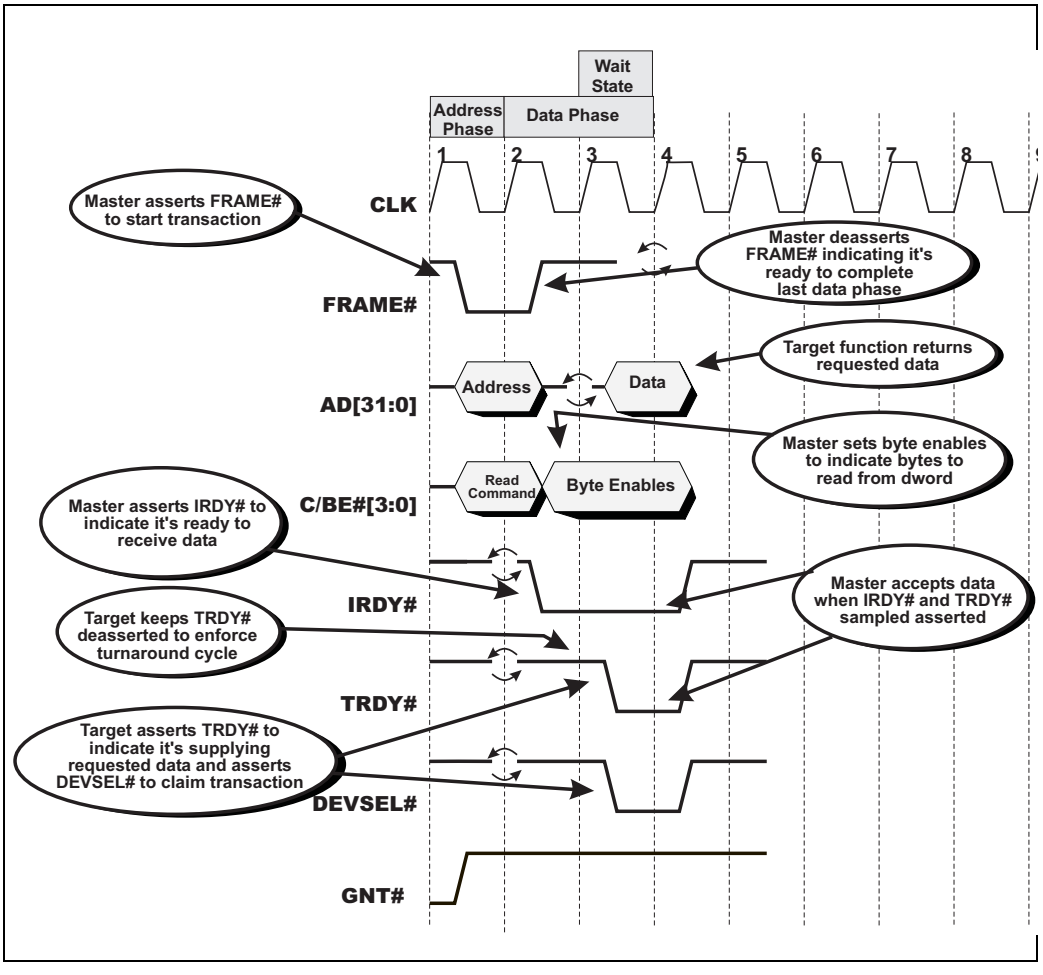
**THE** target samples FRAME# deasserted, indicating that this is the final data phase.

**BECAUSE** the transaction has been completed, the initiator deasserts IRDY# and ceases to drive the byte enables.

**THE** target deasserts TRDY# and DEVSEL# and stops driving the data during clock four.

**CLOCK 5.** The bus returns to the idle state on the rising-edge of clock five.

*Figure 8-1: Example Single Data Phase Read*




## Example Burst Read

During the following description of an example burst read transaction, refer to Figure 8-2 on page 130.

**CLOCK 1.** At the start of clock one, the initiator asserts FRAME#, indicating that the transaction has begun and that a valid start address and command are on the bus. FRAME# must remain asserted until the initiator is ready to complete the last data phase.

**AT** the same time that the initiator asserts FRAME#, it drives the start address onto the AD bus and the transaction type onto the Command/Byte Enable lines, C/BE[3:0]#. The address and transaction type are driven onto the bus for the duration of clock one.

**DURING** clock one, IRDY#, TRDY# and DEVSEL# are not driven (in preparation for takeover by the new initiator and target). They are kept in the deasserted state by keeper resistors on the system board (required system board resource).

**CLOCK 2.** At the start of clock two, the initiator ceases driving the AD bus. A turn-around cycle (i.e., a dead cycle) is required on all signals that may be driven by more than one PCI bus agent. This period is required to avoid a collision when one agent is in the process of turning off its output drivers and another agent begins driving the same signal(s). The target will take control of the AD bus to drive the first requested data item (between one and four bytes) back to the initiator. During a read, clock two is defined as the turn-around cycle because ownership of the AD bus is changing from the initiator to the addressed target. It is the responsibility of the addressed target to keep TRDY# deasserted to enforce this period.

**ALSO** at the start of clock two, the initiator ceases to drive the command onto the Command/Byte Enable lines and uses them to indicate the bytes to be transferred in the currently-addressed dword (as well as the data paths to be used during the data transfer). Typically, the initiator will assert all of the byte enables during a read.

**THE** initiator also asserts IRDY# to indicate that it is ready to receive the first data item from the target.

**UPON** asserting IRDY#, the initiator does not deassert FRAME#, thereby indicating that this is not the final data phase of the example transaction. If this were the final data phase, the initiator would assert IRDY# and deassert FRAME# simultaneously to indicate that it is ready to complete the final data phase.

**IT** should be noted that the initiator does not have to assert IRDY# immediately upon entering a data phase. It may require some time before it's ready to receive the first data item (e.g., it has a buffer full condition). However, the initiator may not keep IRDY# deasserted for more than seven PCI clocks during any data phase. This rule was added in version 2.1 of the specification.

**CLOCK 3.** During clock cycle three, the target asserts DEVSEL# to indicate that it has recognized its address and will participate in the transaction.

**THE** target also begins to drive the first data item (between one and four bytes, as requested by the setting of the C/BE lines) onto the AD bus and asserts TRDY# to indicate the presence of the requested data.

# 9 Write Transfers

## The Previous Chapter

Using timing diagrams, the previous chapter provided a detailed description of PCI read transactions. It also described the treatment of the Byte Enable signals during both reads and writes.

## In This Chapter

This chapter describes write transactions using example timing diagrams.

## The Next Chapter

The next chapter describes the differences between the memory and IO addresses issued during the address phase.

## Example Single Data Phase Write Transaction

Refer to Figure 9-1 on page 137. Each clock cycle is numbered for easy reference and begins and ends on the rising-edge. It is assumed that the bus master has already arbitrated for and been granted access to the bus. The bus master then must wait for the bus to become idle. This is accomplished by sampling the state of FRAME# and IRDY# on the rising-edge of each clock (along with GNT#). When both are sampled deasserted (clock edge one), the bus is idle and a transaction may be initiated by the bus master.

**CLOCK 1.** On detecting bus idle (FRAME# and IRDY# both deasserted), the initiator starts the transaction on the rising-edge of clock one.

**THE** initiator drives out the address on AD[31:0] and the command on C/BE#[3:0].

**THE** initiator asserts FRAME# to indicate that the transaction has started and that there is a valid address and command on the bus.

**CLOCK 2** All targets on the bus sample the address, command and FRAME# on the rising-edge of clock two, completing the address phase.

**THE** targets begin the decode to determine which of them is the target of the transaction.

**THE** initiator asserts IRDY# to indicate that is driving the first write data item to the target over the AD bus. As long as the initiator asserts IRDY# within seven clocks after entering a data phase, it is within spec.

**THE** initiator also deasserts FRAME# when it asserts IRDY#, thereby indicating that it is ready to complete the final data phase of the transaction.

**THE** initiator stops driving the command onto C/BE#[3:0] and starts driving the byte enables to indicate which locations it wished to write to in the first dword.

**THE** target asserts DEVSEL# in clock two to claim the transaction.

**THE** target also asserts TRDY# to indicate its readiness to accept the first write data item.

**CLOCK 3** The master samples DEVSEL# asserted, indicating that the target has claimed the transaction.

**THE** target samples IRDY# and the data on the AD bus. The asserted state of IRDY# indicates that it has just latched the first valid write data item.

**THE** asserted state of TRDY# indicates to the initiator that the target was ready to accept it, so both parties now know that the first data item has been transferred.

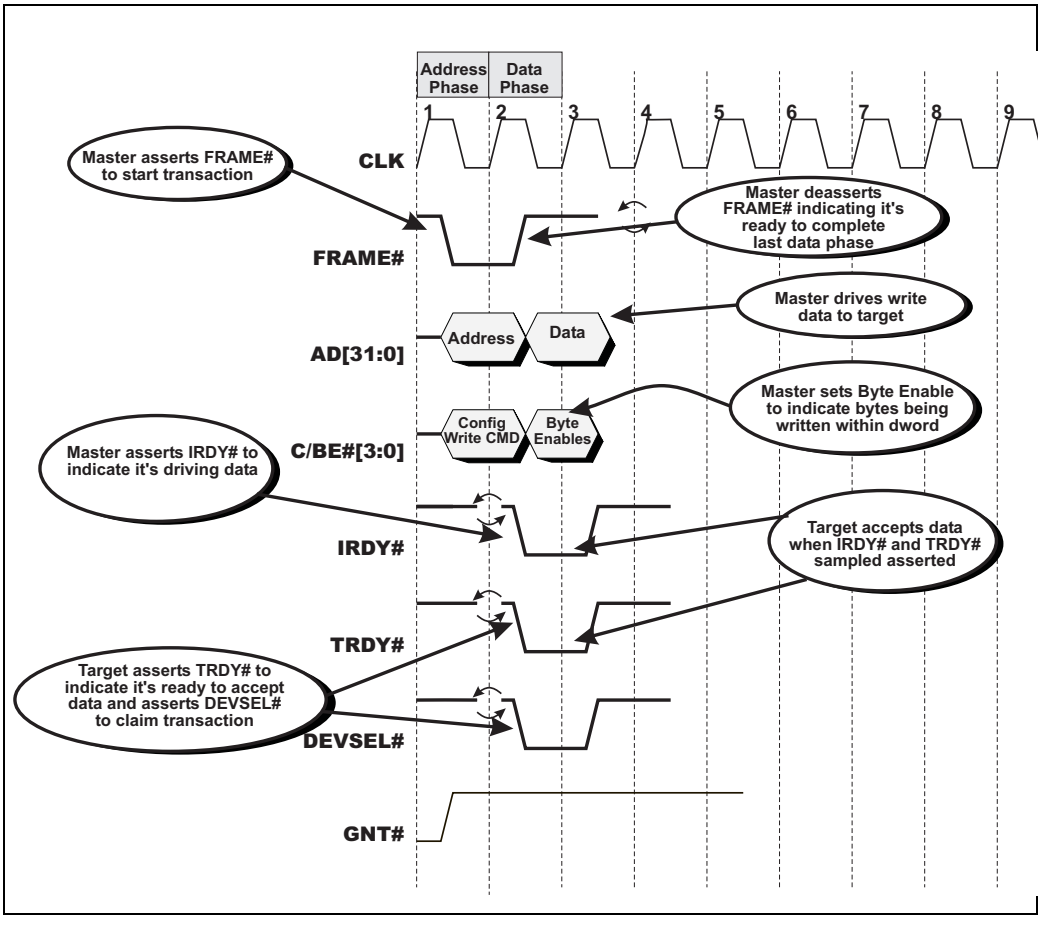
**THE** target also sampled FRAME# deasserted, indicating that this is the final data phase of the transaction.

**BECAUSE** the transaction has been completed, the initiator deasserts IRDY# and ceases to drive the byte enables and the data.

**THE** target deasserts TRDY# and DEVSEL# during clock four.

**CLOCK 4** The bus returns to the idle state on the rising-edge of clock five.

*Figure 9-1: Example Single Data Phase Write Transaction*




## Example Burst Write Transaction

During the following description of the write transaction, refer to Figure 9-2 on page 141.

**CLOCK 1.** When both IRDY# and FRAME# are sampled deasserted (on the rising-edge of clock one), the bus is idle and a transaction may be initiated by the bus master whose GNT# signal is currently asserted by the bus arbiter.

**AT** the start of clock cycle one, the initiator asserts FRAME# to indicate that the transaction has begun and that a valid start address and command are present on the bus. FRAME# remains asserted until the initiator is ready

(has asserted IRDY#) to complete the last data phase. At the same time that the initiator asserts FRAME#, it drives the start address onto the AD bus and the transaction type onto C/BE#[3:0]. The address and transaction type are driven onto the bus for the duration of clock one.

**DURING** clock cycle one, IRDY#, TRDY# and DEVSEL# are not driven (in preparation for takeover by the new initiator and target). They are maintained in the deasserted state by the pullups on the system board.

**CLOCK 2.** At the start of clock cycle two, the initiator stops driving the address onto the AD bus and begins driving the first write data item. Since it doesn't have to hand off control of the AD bus to the target (as it does during a read), a turn-around cycle is unnecessary. The initiator may begin to drive the first data item onto the AD bus immediately upon entry to clock cycle two. Remember, though, that the initiator is still in spec as long as it presents the data within eight clocks after entering a data phase.

**DURING** clock cycle two, the initiator stops driving the command and starts driving the byte enables to indicate the bytes to be written to the currently-addressed dword.

**THE** initiator drives the write data onto the AD bus and asserts IRDY# to indicate the presence of the data on the bus. The initiator doesn't deassert FRAME# when it asserts IRDY# (because this is not the final data phase). Once again, it should be noted that the initiator does not have to assert IRDY# immediately upon entering a data phase. It may require some time before it's ready to source the first data item (e.g., it has a buffer empty condition). However, the initiator may not keep IRDY# deasserted for more than seven clocks during any data phase. This rule was added in version 2.1 of the specification.

**DURING** clock cycle two, the target decodes the address and command and asserts DEVSEL# to claim the transaction.

**IN** addition, it asserts TRDY#, indicating its readiness to accept the first data item.

**CLOCK 3.** At the rising-edge of clock three, the initiator and the currently-addressed target sample both TRDY# and IRDY# asserted, indicating that they are both ready to complete the first data phase. This is a zero wait state transfer (i.e., a one clock data phase). The target accepts the first data item from the bus on the rising-edge of clock three (and samples the byte enables in order to determine which bytes are being written), completing the first data phase.

**THE** target increments its address counter by four to point to the next dword.

**DURING** clock cycle three, the initiator drives the second data item onto the AD bus and sets the byte enables to indicate the bytes being written into the next dword and the data paths to be used during the second data phase.

**THE** initiator also keeps IRDY# asserted and does not deassert FRAME#, thereby indicating that it is ready to complete the second data phase and that this is not the final data phase. Assertion of IRDY# indicates that the write data is present on the bus.

**CLOCK 4.** At the rising-edge of clock four, the initiator and the currently-addressed target sample both TRDY# and IRDY# asserted, indicating that they are both ready to complete the second data phase. This is a zero wait state data phase. The target accepts the second data item from the bus on the rising-edge of clock four (and samples the byte enables to determine which data lanes contain valid data), completing the second data phase.

**THE** initiator requires more time before beginning to drive the next data item onto the AD bus (it has a buffer-empty condition). It therefore inserts a wait state into the third data phase by deasserting IRDY# at the start of clock cycle four. This allows the initiator to delay presentation of the new data, but it must set the byte enables to the proper setting for the third data phase immediately upon entering the data phase.

**IN** this example, the target also requires more time before it will be ready to accept the third data item. To indicate the requirement for more time, the target deasserts TRDY# during clock cycle four.

**THE** target once again increments its address counter to point to the next dword.

**DURING** clock cycle four, although the initiator does not yet have the third data item available to drive, it must drive a stable pattern onto the data paths rather than let the AD bus float (required power conservation measure). The specification doesn't dictate the pattern to be driven during this period. It is usually accomplished by continuing to drive the previous data item. The target will not accept the data being presented to it for two reasons:

- By deasserting TRDY#, it has indicated that it isn't ready to accept data.
- By deasserting IRDY#, the initiator has indicated that it is not yet presenting the next data item to the target.

**CLOCK 5.** When the initiator and target sample IRDY# and TRDY# deasserted at the rising-edge of clock five, they insert a wait state (clock cycle five) into the third data phase.

**DURING** clock cycle five, the initiator asserts IRDY# and drives the final data item onto the AD bus.

**THE** initiator also deasserts FRAME# to indicate that this is the last data phase.

**THE** initiator must continue to drive the byte enables for the third data phase until it completes.

**THE** target keeps TRDY# deasserted, indicating that it is not yet ready to accept the third data item.

# 10 Memory and IO Addressing

## The Previous Chapter

The previous chapter described write transactions using example timing diagrams.

## In This Chapter

This chapter describes the differences between the memory and IO addresses issued during the address phase.

## The Next Chapter

The next chapter provides a detailed description of Fast Back-to-Back transactions and address/data Stepping.

# Memory Addressing

## The Start Address

The start address issued during any form of memory transaction is a dword-aligned address presented on AD[31:2] during the address phase. It is a quad-word-aligned address if the master is starting a 64-bit transfer, but this subject is covered in “The 64-bit PCI Extension” on page 265.

## Addressing Sequence During Memory Burst

The following discussion assumes that the memory target supports bursting. The memory target latches the start address into an address counter and uses it for the first data phase. Upon completion of the first data phase and assuming that it’s not a single data phase transaction, the memory target must update its address counter to point to the next dword to be transferred.

On a memory access, the memory target must check the state of address bits one and zero (AD[1:0]) to determine the policy to use when updating its address counter at the conclusion of each data phase. Table 10-1 on page 144 defines the addressing sequences defined in the revision 2.2 specification and encoded in the first two address bits. Only two addressing sequences are currently defined:

- Linear (sequential) Mode.
- Cache Line Wrap Mode.

*Table 10-1: Memory Burst Address Sequence*

| AD1 | AD0 | Addressing Sequence |
|---|---|---|
| 0 | 0 | **Linear**, or sequential, addressing sequence during the burst. |
| 0 | 1 | **Reserved**. Prior to revision 2.1, this indicated Intel Toggle Mode addressing. When detected, the memory target should signal a Disconnect with data transfer during the first data phase or a disconnect without data transfer in the second data phase. |
| 1 | 0 | **Cache Line Wrap** mode. First defined in revision 2.1. |
| 1 | 1 | **Reserved**. When detected, the memory target should signal a Disconnect with data transfer during the first data phase or a disconnect without data transfer in the second data phase. |

## Linear (Sequential) Mode

All memory devices that support multiple data phase transfers must support linear addressing. When the Memory Write-and-Invalidate command is used, the start address must be aligned on a cache line boundary and it must indicate (on AD[1:0]) linear addressing. At the completion of each data phase, (even if the initiator isn't asserting any Byte Enables in the next data phase) the memory target increments its address counter by four to point to the next sequential dword for the next data phase (or the next sequential quadword if performing 64-bit transfers)

## Cache Line Wrap Mode

Support for Cache Line Wrap Mode is optional and is only used for memory reads. A memory target that supports this mode must implement the Cache Line Size configuration register (so it knows when the end of a line has been

reached). The start address can be any dword within a line. At the start of each data phase of the burst read, the memory target increments the dword address in its address counter. When the end of the current cache line is encountered and assuming that the master continues the burst, the target starts the transfer of the next cache line at the same relative start position (i.e., dword) as that used in the first cache line.

Implementation of Cache Line Wrap Mode is optional for memory and meaningless for IO and configuration targets.

The author is not aware (some people, mostly my friends, would agree with that) of any bus masters that use Cache Line Wrap Mode when performing memory reads. It would mainly be useful if the processor were permitted to cache from memory targets residing on the PCI bus. This is no longer supported, but was in the 2.1 spec. The following explanation is only presented as background to explain the inclusion of Cache Line Wrap Mode in the earlier spec.

When a processor has a cache miss, it initiates a cache line fill transaction on its external bus to fetch the line from memory. If the target memory is not main memory, the host/PCI bridge starts a burst memory read from the PCI memory target using Cache Line Wrap Mode. Most processors (other than x86 processors) use Wrap addressing when performing a cache line fill. The processor expects the memory controller to understand that it wants the critical quadword first (because it contains the bytes that caused the cache miss), followed by the remaining quadwords that comprise the cache line. The transfer sequence of the remaining quadwords is typically circular (which is what Wrap Mode is all about).

The Intel x86 processors do not use wrap addressing (they use Toggle Mode addressing). The PowerPC 601, 603 and 604 processors use Wrap addressing. For a detailed description of the 486 cache line fill addressing sequence, refer to the Addison-Wesley publication entitled *80486 System Architecture*. For that used by the Pentium processor, refer to the Addison-Wesley publication entitled *Pentium Processor System Architecture*. For that used by the P6-family processors, refer to the Addison-Wesley publication entitled *Pentium Pro and Pentium II System Architecture*. For that used by the PowerPC 60x processors, refer to the Addison-Wesley publication entitled *PowerPC System Architecture*.

## When Target Doesn't Support Setting on AD[1:0]

Although a memory target may support burst mode, it may not implement the addressing sequence indicated by the bus master during the address phase. When the master uses a pattern on AD[1:0] that the target doesn't support (Reserved or Cache Line Wrap), the target must respond as follows:

- The target must either issue a **Disconnect with data transfer** on the transfer of the first data item,
- or a **Disconnect without data transfer** during the second data phase.

This is necessary because the initiator is indicating an addressing sequence the target is unfamiliar with and the target therefore doesn't know what to do with its address counter.

There are two scenarios where the master may use a bit pattern not supported by the target:

- The master was built to a different rev of the spec. and is using a pattern that was reserved when the target was designed.
- The master indicates Cache Line Wrap addressing on a burst, but the target only supports Linear addressing.

# PCI IO Addressing

## Do Not Merge Processor IO Writes

To ensure that IO devices function correctly, bridges must never merge sequential IO accesses into a single data phase (merging byte accesses performed by the processor into a single-dword transfer) or a multiple data phase transaction. Each individual IO transaction generated by the processor must be performed on the PCI bus as it appears on the host bus. This rule includes accesses to both IO space and memory-mapped IO space (non-Prefetchable memory). Bridges are not permitted to perform byte merging in the their posted memory write buffers when writes are performed to non-Prefetchable memory (see "Byte Merging" on page 95 and "What Is Prefetchable Memory?" on page 93.

## General

During an IO transaction, the start IO address placed on the AD bus during the address phase has the following format:

- AD[31:2] identify the target dword of IO space.
- AD[1:0] identify the least-significant byte (i.e., the start byte) within the target dword that the initiator wishes to perform a transfer with (00b = byte 0, 01b = byte 1, etc.).

At the end of the address phase, all IO targets latch the start address and the IO read or write command and begin the address decode.

## Decode By Device That Owns Entire IO Dword

An IO device that only implements 32-bit IO ports can ignore AD[1:0] when performing address decode. In other words, it decodes AD[31:2] plus the command in deciding whether or not to claim the transaction by asserting DEVSEL#. It then examines the byte enables in the first data phase to determine which of the four locations in the addressed IO dword are being read or written.

## Decode by Device With 8-Bit or 16-Bit Ports

An IO device may not implement all four locations within each dword of IO space assigned to it. As an example, within a given dword of IO space it may implement locations 0 and 1, but not 2 and 3. This type of IO target claims the transaction based on the *byte-specific* start address that it latched at the end of the address phase. In other words, it must decode the full 32-bit IO address consisting of AD[31:0]. If that 8-bit IO port is implemented in the target, the target asserts DEVSEL# and claims the transaction.

The byte enables asserted during the data phase identify the least-significant byte within the dword (the same one indicated by the setting of AD[1:0]) as well as any additional bytes (within the addressed dword) that the initiator wishes to transfer. It is illegal (and makes no sense) for the initiator to assert any byte enables of lesser significance than the one indicated by the AD[1:0] setting. If the initiator does assert any of the illegal byte enable patterns, the target must terminate the transaction with a Target Abort. Table 10-2 on page 147 contains some examples of valid IO addresses.

*Table 10-2: Examples of IO Addressing*

| AD[31:0] | C/BE3# | C/BE2# | C/BE1# | C/BE0# | Description |
|---|---|---|---|---|---|
| 00001000h | 1 | 1 | 1 | 0 | just location 1000h |
| 000095A2h | 0 | 0 | 1 | 1 | 95A2 and 95A3h |
| 00001510h | 0 | 0 | 0 | 0 | 1510h-1513h |
| 1267AE21h | 0 | 0 | 0 | 1 | 1267AE21h-1267AE23h |

# 11 Fast Back-to-Back & Stepping

## The Previous Chapter

The previous chapter described the differences between the memory and IO addresses issued during the address phase.

## This Chapter

This chapter provides a detailed description of Fast Back-to-Back transactions and address / data Stepping.

## The Next Chapter

The next chapter describes the early termination of a transaction before all of the intended data has been transferred between the master and the target. This includes descriptions of Master Abort, the preemption of a master, Target Retry, Target Disconnect, and Target Abort.

## Fast Back-to-Back Transactions

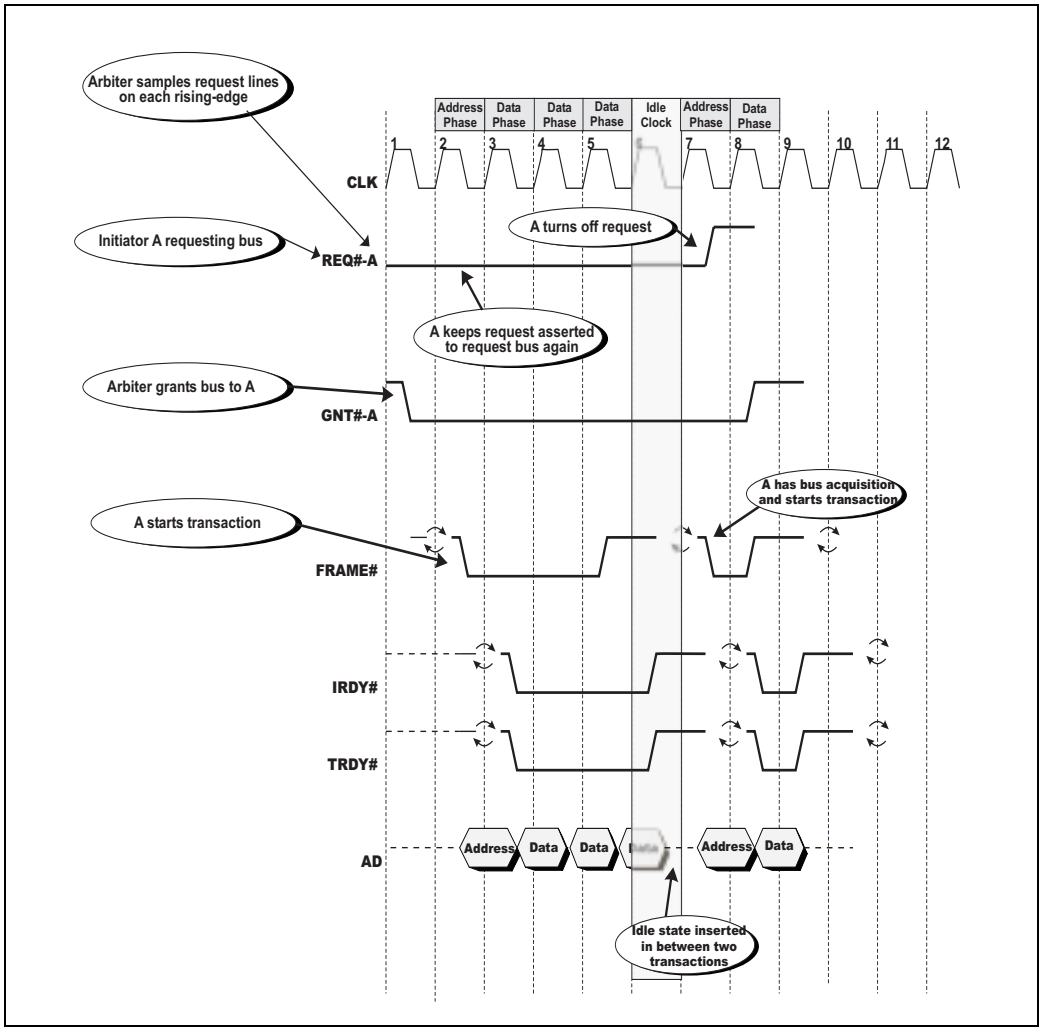
Assertion of its grant by the PCI bus arbiter gives a PCI bus master access to the bus for a single transaction consisting of one or more data phases. If a bus master desires another access, it should continue to assert its REQ# after it has asserted FRAME# for the first transaction. If the arbiter continues to assert GNT# to the master at the end of the first transaction, the master may then immediately initiate a second transaction. However, a bus master attempting to perform two, back-to-back transactions usually must insert an idle cycle between the two transactions. This is illustrated in clock 6 of Figure 11-1 on page 154. If all of the required criteria are met, the master can eliminate the idle cycle between the two bus transactions. These are referred to as Fast Back-to-Back transactions. This can only occur if there is a guarantee that there will not be contention (on any signal lines) between the masters and / or targets involved in the two transactions. There are two scenarios where this is the case.

**CASE 1.** In the first case, the master guarantees that there will be no contention. It's optional whether or not a master implements this capability, but it's mandatory that targets support it.

**CASE 2.** In the second case, the master and the community of PCI targets collectively provide the guarantee.

The sections that follow describe these two scenarios.

*Figure 11-1: Back-to-Back Transactions With an Idle State In-Between*



## Decision to Implement Fast Back-to-Back Capability

The subsequent two sections describe the rules that permit deletion of the idle state between two transactions. Since they represent a fairly constraining set of rules, the designer of a bus master should make an informed decision as to whether or not it's worth the additional logic it would take to implement it.

Assume that the nature of a particular bus master is such that it typically performs long burst transfers whenever it acquires bus ownership. A SCSI Host Bus Adapter would be a good example. In this case, including the extra logic to support Fast Back-to-Back transactions would not be worth the effort. Percentage-wise, you're only saving one clock tick of latency in between each pair of long transfers.

Assume that the nature of another master is such that it typically performs lots of small data transfers. In this case, inclusion of the extra logic may result in a measurable increase in performance. Since each of the small transactions typically only consists of a few clock ticks and the master performs lots of these small transactions in rapid succession, the savings of one clock tick in between each transaction pair can amount to the removal of a fair percentage of overhead normally spent in bus idle time.

## Scenario 1: Master Guarantees Lack of Contention

In this scenario (defined in revision 1.0 of the specification and still true in revision 2.x), the master must ensure that, when it performs a pair of back-to-back transactions with no idle state in between the two, there is no contention on any of the signals driven by the bus master or on those driven by the target. An idle cycle is required whenever AD[31:0], C/BE#[3:0], FRAME#, PAR and IRDY# are driven by different masters from one clock cycle to the next. The idle cycle allows one cycle for the master currently driving these signals to surrender control (cease driving) before the next bus master begins to drive these signals. This prevents bus contention on the bus master-related signals.

### 1st Must Be Write, 2nd Is Read or Write, But Same Target

The master must ensure that the same set of output drivers are driving the master-related signals at the end of the first transaction and the start of the second. This means that the master must ensure that it is driving the bus at the end of the first transaction and at the start of the second.

To meet this criteria, the first transaction must be a write transaction so the master will be driving the AD bus with the final write data item at the end of the transaction. The second transaction can be either a read or a write but must be initiated by the same master. This means that the master must keep its REQ# asserted to the arbiter when it starts the first transaction and must check to make sure that the arbiter leaves its GNT# asserted at the end of the first transaction. Whether or not the arbiter leaves the GNT# on the master is dependent on whether or not the arbiter receives any bus requests from other masters during this master's first transaction. Refer to Figure 11-2 on page 158.

**CLOCK 1.** When the master acquires bus ownership and starts the first transaction (on the rising-edge of clock one), it drives out the address and command and asserts FRAME#. The transaction must be a write.

**THE** initiator continues to assert its REQ# line to try and get the arbiter to leave the GNT# on it so it can immediately start the second transaction of the Fast Back-to-Back pair after the final data phase of the first transaction completes.

**CLOCK 2.** When the address phase is completed (on the rising-edge of clock two), the master drives the first data item onto the AD bus and sets the byte enables to indicate which data paths contain valid data bytes.

**THE** master asserts IRDY# to indicate the presence of the write data on the AD bus.

**THE** master simultaneously deasserts FRAME#, indicating that this is the final data phase of the write transaction.

**THE** target asserts DEVSEL# to claim the transaction.

**THE** target also asserts TRDY# to indicate its readiness to accept the first write data item.

**CLOCK 3.** At the conclusion of the first data phase (on the rising-edge of clock three) and each subsequent data phase (there aren't any in this example), the bus master is driving write data onto the AD bus and is also driving the byte enables.

**THE** master samples DEVSEL# asserted on the rising-edge of clock three indicating that the target has claimed the transaction.

**THE** target samples IRDY# asserted and FRAME# deasserted on the rising-edge of clock three, indicating that the master is ready to complete the final data phase.

**BOTH** parties sample IRDY# and TRDY# asserted on the rising-edge of clock three, indicating that the target has latched the final data item.

**THE** master samples its GNT# still asserted by the arbiter, indicating that it has retained bus ownership for the next transaction. *If GNT# were sampled deasserted at this point, the master has lost ownership and cannot proceed with the second transaction. It would have to wait until ownership passes back to it and then perform the second transaction.*

**CLOCK** three would normally be the idle clock during which the master and target would ceasing driving their respective signals. However, the master has retained bus ownership and will immediately start a new transaction (either a write or a read) during clock three. It *must ensure that it is addressing the same target*, however, so that there isn't any contention on the target-related signals (see Clock Four).

**THE** master stops driving the final write data item and the byte enables and immediately starts driving the start address and command for the second transaction.

**THE** master immediately reasserts FRAME# to indicate that it is starting a new transaction. It should be noted that *the bus does not return to the idle state* (FRAME# and IRDY# both deasserted). *It is a rule that targets must recognize a change from IRDY# asserted and FRAME# deasserted on one clock edge to IRDY# deasserted and FRAME# asserted on the next clock edge as the end of one transaction and the beginning of a new one.*

**THE** master deasserts its REQ# to the arbiter (unless it wishes to perform another transaction immediately after the new one it's just beginning).

**CLOCK 4.** When the targets on the bus detect FRAME# reasserted, this qualifies the address and command just latched as valid for a new transaction. They begin the decode.

**THE** target of the previous transaction had actively-driven TRDY# and DEVSEL# back high for one clock starting on the rising-edge of clock three and continuing until the rising-edge of clock four. That target is just beginning to back its DEVSEL# and TRDY# output drivers off those two signals starting on the rising-edge of clock four. If a different target were addressed in the second transaction and it had a fast decoder, it would turn on its DEVSEL# output driver starting on the rising-edge of clock four to assert DEVSEL#. In addition, if the second transaction were a write, the target of the second transaction might also turn on its TRDY# output driver to indicate its readiness to accept the first data item. In summary, if the master were to address different targets in the first and second transactions, there might be a collision on the target-related signals. *This is why it's a rule that the master must address the same target in both transactions.* In this way, the same target output drivers that just drove both lines high can now drive them low.

Since the configuration PCI software dynamically assigns address ranges to a device's programmable PCI memory and IO decoders each time the machine is powered up, it can be a real challenge for the master to "know" that it is addressing the same target in the first and second transactions.

# 12 Early Transaction End

## The Previous Chapter

The previous chapter provided a detailed description of Fast Back-to-Back transactions and address / data Stepping.

## In This Chapter

This chapter describes the early termination of a transaction before all of the intended data has been transferred between the master and the target. This includes descriptions of Master Abort, the preemption of a master, Target Retry, Target Disconnect, and Target Abort.

## The Next Chapter

The next chapter describes error detection, reporting and handling.

## Introduction

In certain circumstances, a transaction must be prematurely terminated before all of the data has been transferred. Either the initiator or the target makes the determination to prematurely terminate a transaction. The following sections define the circumstances requiring termination and the mechanisms used to accomplish it. The first half of this chapter discusses situations wherein the master makes the decision to prematurely terminate a transaction. The second half discusses situations wherein the target makes the decision.

## Master-Initiated Termination

The initiator terminates a transaction for one of four reasons:

1. The **transaction has completed normally**. All of the data that the master intended to transfer to or from the target has been transferred. This is a normal, rather than a premature transaction termination.
2 The **initiator has already used up its time slice and has been living on borrowed time and is then preempted** by the arbiter (because one or more other bus masters are requesting the bus) before it completes its burst transfer. In other words, the initiator's Latency Timer expired some time ago and the arbiter has now removed the initiator's bus grant signal (GNT#).
3 The initiator is **preempted during** its **time slice and** then **uses up** its allotted **time slice** before completing its overall transfer.
4 The initiator has aborted the transaction because **no target has responded** to the address. This is referred to as a Master Abort.

Normal transaction termination is described in the chapters entitled "Read Transfers" on page 123 and "Write Transfers" on page 135. The second and third scenarios are described in this chapter.

## Master Preempted

### Introduction

Figure 12-1 on page 175 illustrates two cases of preemption. In the first case (the upper part of the diagram), the arbiter removes GNT# from the initiator, but the initiator's LT (Latency Timer; see "Latency Timer Keeps Master From Monopolizing Bus" on page 78) has not yet expired, indicating that its timeslice has not yet been exhausted. It may therefore continue to use the bus either until it has completed its transfer, or until its timeslice is exhausted, whichever comes first.

In the second case, the initiator has already used up its timeslice but has not yet lost its GNT# (referred to as preemption). It may therefore continue its transaction until it has completed its transfer, or until its GNT# is removed, whichever comes first. The following two sections provide a detailed description of the two scenarios illustrated.

## Preemption During Timeslice

In the upper example in Figure 12-1 on page 175, the current initiator initiated a transaction at some earlier point in time.

**CLOCK 1.** The master is preempted on the rising-edge of clock one (GNT# has been removed by the arbiter), indicating that the arbiter has detected a request from another master and is instructing the current master to surrender the bus.

**AT** the point of preemption, however, the master's LT has not yet expired (i.e., its timeslice has not been exhausted). The master may therefore retain bus ownership until it has either completed its overall transfer or until its timeslice is exhausted, which ever comes first.

**CLOCK 2.** Data transfers occur on the rising-edge of clocks two through five (because IRDY# and TRDY# are both sampled asserted) and the master also decrements its LT on the rising-edge of each clock (the LT register is of course not visible in the timing diagram).

**CLOCK 3.** See Clock Two.

**CLOCK 4.** See Clock Two.

**CLOCK 5.** On the rising-edge of clock five, a data item is transferred (IRDY# and TRDY# sampled asserted) and the master decrements its LT again and it's exhausted (transferring data can be very tiring). The rule is that if the master has used up its timeslice (i.e., LT value) and has lost its grant, the initiator can perform one final data phase and must then surrender ownership of the bus.

**IN** what it now knows is the final data phase, the initiator keeps IRDY# asserted and deasserts FRAME#, indicating that it's ready to complete the final data phase.

**CLOCK 6.** The target realizes that this is the final data phase because it samples FRAME# deasserted and IRDY# asserted on clock six.

**THE** final data item is transferred on the rising-edge of clock six.

**THE** initiator then deasserts IRDY#, returning the bus to the idle state.

**CLOCK 7.** The bus is idle (FRAME# and IRDY# deasserted).

**THE** master that has its GNT# and has been testing for bus idle on each clock can now assume bus ownership on the rising-edge of clock seven.

## Timeslice Expiration Followed by Preemption

The lower half of Figure 12-1 on page 175 illustrates the case where the master started a transaction at some earlier point in time, used up its timeslice, and is then preempted at some later point in time.

**CLOCK 1.** The initiator determines that its LT has expired at the rising-edge of clock one, and a data transfer occurs at the same time (IRDY# and TRDY# sampled asserted). The initiator doesn't have to yield the bus yet because the arbiter hasn't removed its GNT#. It may therefore retain bus ownership until it either completes its overall data transfer or until its GNT# is removed by the arbiter, which ever occurs first.

**CLOCK 2.** A data transfer takes place on the rising-edge of clock two (because IRDY# and TRDY# are sampled asserted).

**CLOCK 3.** No data transfer takes place on the rising-edge of clock three (because the initiator wasn't ready (IRDY# deasserted).

**CLOCK 4.** A data transfer takes place on the rising-edge of clock four.

**CLOCK 5.** A data transfer takes place on the rising-edge of clock five (because IRDY# and TRDY# are sampled asserted).

**ON** the rising-edge of clock five, the initiator detects that its GNT# has been removed by the arbiter, indicating that it must surrender bus ownership after performing one more data phase.

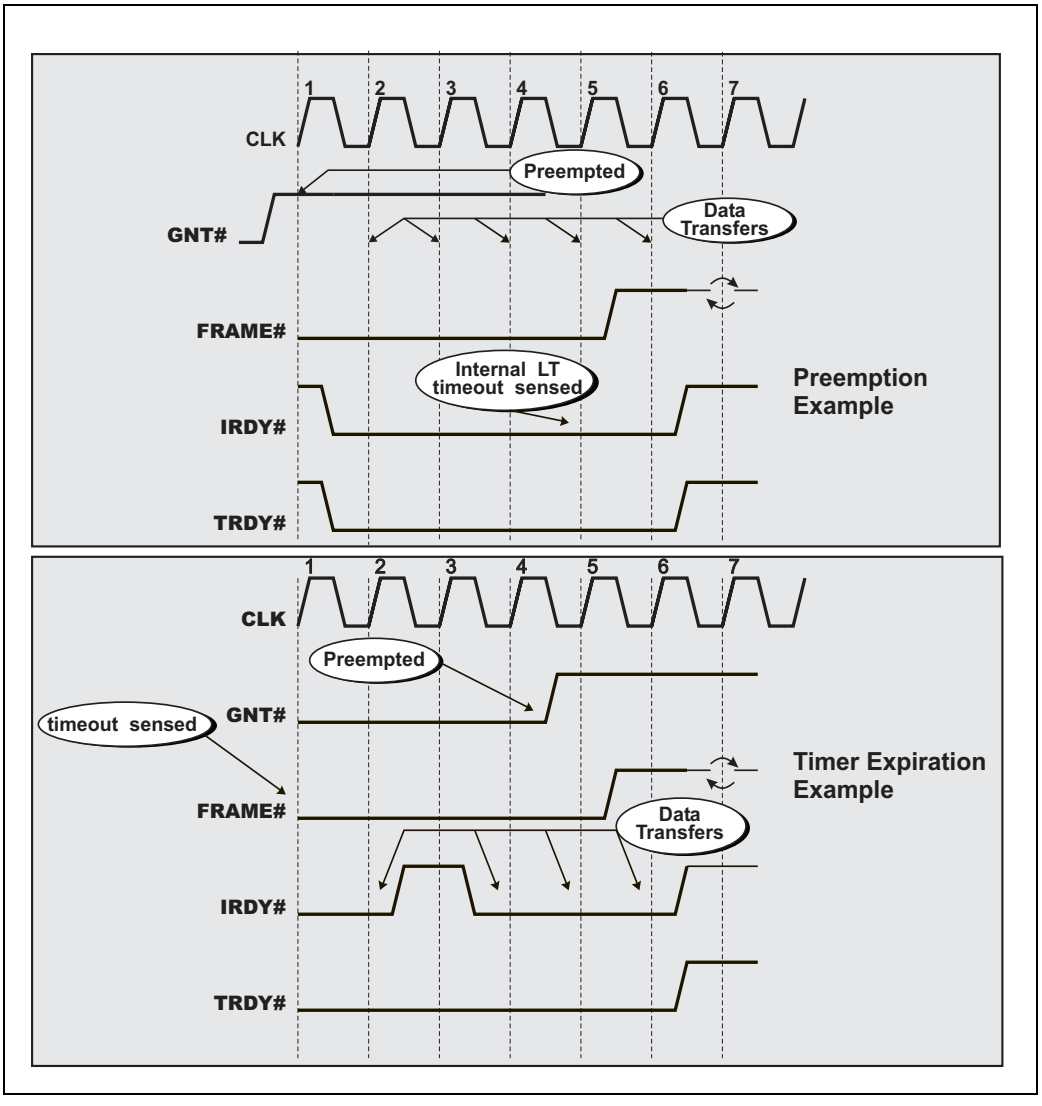
**IN** clock five, the initiator keeps IRDY# asserted and removes FRAME#, indicating that the final data phase is in progress.

**CLOCK 6.** The final data item is transferred on the rising-edge of clock six. The initiator then deasserts IRDY# in clock six, returning the bus to the idle state.

**CLOCK 7.** The bus has returned to the idle state (FRAME# and IRDY# sampled deasserted on the rising-edge of clock seven.

**THE** master that has its GNT# and has been testing for bus idle on each clock can now assume bus ownership on clock seven.

*Figure 12-1: Master-Initiated Termination Due to Preemption and Master Latency Timer Expiration*



# 13 Error Detection and Handling

## Prior To This Chapter

The previous chapter described the early termination of a transaction before all of the intended data has been transferred between the master and the target. This included descriptions of Master Abort, the preemption of a master, Target Retry, Target Disconnect, and Target Abort.

## In This Chapter

The PCI bus architecture provides two error reporting mechanisms: one for reporting data parity errors and the other for reporting more serious system errors. This chapter provides a discussion of error detection, reporting and handling using these two mechanisms.

## The Next Chapter

The next chapter provides a discussion of interrupt-related issues.

## Status Bit Name Change

Please note that the **Master Data Parity Error bit** in the Status register (see Figure 13-4 on page 211) was named Data Parity Reported in the 1.0 and 2.0 specs. Its name changed to the Data Parity Error Detected bit the 2.1 spec. Its name has changed yet again in the 2.2 spec to Master Data Parity Error. Although its name has changed over time, its meaning has remained the same.

## Introduction to PCI Parity

The PCI bus is parity-protected during both the address and data phases of a transaction. A single parity bit, PAR, protects AD[31:0] and C/BE#[3:0]. If a 64-bit data transfer is in progress, an additional parity bit, PAR64, protects

AD[63:32] and C/BE#[7:4]. 64-bit parity has the same timing as 32-bit parity and is discussed in “64-bit Parity” on page 297.

The PCI device driving the AD bus during the address phase or any data phase of a transaction must always drive a full 32-bit pattern onto the AD bus (because parity is always based on the full content of the AD bus and the C/BE bus). This includes:

- Special Cycle and Interrupt Acknowledge transactions where the address bus doesn't contain a valid address, and during Type Zero Configuration transactions where AD[31:11] do not contain valid information.
- Data phases where the device supplying data isn't supplying all four bytes (all four byte enables are not asserted).

The PCI device driving the AD bus during the address phase or any data phase of a transaction is responsible for calculating and supplying the parity bit for the phase. The parity bit must be driven one clock after the address or data is first driven onto the bus (when TRDY# is asserted during a read data phase to indicate the presence of the read data on the bus, or when IRDY# is asserted during a write data phase to indicate the presence of the write data on the bus) and must continue to be driven until one clock after the data phase completes. Even parity is used (i.e., there must be an even number of ones in the overall 37-bit pattern). The computed parity bit supplied on PAR must be set (or cleared) so that the 37-bit field consisting of AD[31:0], C/BE#[3:0] and PAR contains an even number of one bits.

During the clock cycle immediately following the conclusion of the address phase or any data phase of a transaction, the PCI agent receiving the address or data computes expected parity based on the information latched from AD[31:0] and C/BE#[3:0]. The agent supplying the parity bit must present it:

- on the rising-edge of the clock that immediately follows the conclusion of the address phase.
- one clock after presentation of data (IRDY# assertion on a write or TRDY# assertion on a read) during a data phase.

The device(s) receiving the address or data expect the parity to be present and stable at that point. The computed parity bit is then compared to the parity bit actually received on PAR to determine if address or data corruption has occurred. If the parity is correct, no action is taken. If the parity is incorrect, the error must be reported. This subject is covered in the sections that follow.

During a read transaction where the initiator is inserting wait states (by delaying assertion of IRDY#) because it has a buffer full condition, the initiator can optionally be designed with a parity checker that:

- samples the target's data (when TRDY# is sampled asserted),
- calculates expected parity,
- samples actual parity (on the clock edge after TRDY# sampled asserted)
- and asserts PERR# (if the parity is incorrect).

In this case, the initiator must keep PERR# asserted until one clock after the completion of the data phase.

The same is true in a write transaction. If the target is inserting wait states by delaying assertion of TRDY#, the target's parity checker can optionally be designed to:

- sample the initiator's data (when IRDY# is sampled asserted),
- calculate expected parity,
- sample actual parity (on the clock edge after IRDY# sampled asserted)
- and assert PERR# (if the parity is incorrect).

In this case, the target must keep PERR# asserted until two clocks after the completion of the data phase.

# PERR# Signal

PERR# is a sustained tri-state signal used to signal the detection of a parity error related to a data phase. There is one exception to this rule: a parity error detected on a data phase during a Special Cycle is reported using SERR# rather than PERR#. This subject is covered later in this chapter in "Special Case: Data Parity Error During Special Cycle" on page 213.

PERR# is implemented as an output on targets and as an input/output on masters. Although PERR# is bussed to all PCI devices, it is guaranteed to be driven by only one device at a time (the initiator on a read, or the target on a write).

# Data Parity

## Data Parity Generation and Checking on Read

### Introduction

During each data phase of a read transaction, the target drives data onto the AD bus. It is therefore the target's responsibility to supply correct parity to the initiator on the PAR signal starting one clock after the assertion of TRDY#. At the

conclusion of each data phase, it is the initiator's responsibility to latch the contents of AD[31:0] and C/BE#[3:0] and to calculate the expected parity during the clock cycle immediately following the conclusion of the data phase. The initiator then latches the parity bit supplied by the target from the PAR signal on the next rising-edge of the clock and compares computed vs. actual parity. If a miscompare occurs, the initiator then asserts PERR# during the next clock (if it's enabled to do so by a one in the Parity Error Response bit in its Command register). The assertion of PERR# lags the conclusion of each data phase by two PCI clock cycles.

The platform design (in other words, the chipset) may or may not include logic that monitors PERR# during a read and takes some system-specific action (such as asserting NMI to the processor in an Intel x86-based system; for more information, refer to "Important Note Regarding Chipsets That Monitor PERR#" on page 210) when it is asserted by a PCI master or a target that has received corrupted data. The master may also take other actions in addition to the assertion of PERR#. "Data Parity Reporting" on page 209 provides a detailed discussion of the actions taken by a bus master upon receipt of bad data.

## Example Burst Read

Refer to the read burst transaction illustrated in Figure 13-1 on page 204 during this discussion.

**CLOCK 1.** The initiator drives the address and command onto the bus during the address phase (clock one).

**CLOCK 2.** The targets latch the address and command on clock two and begin address decode. In this example, the target has a fast address decoder and asserts DEVSEL# during clock two.

**ALL** targets that latched the address and command compute the expected parity based on the information latched from AD[31:0] and C/BE#[3:0].

**THE** initiator sets the parity signal, PAR, to the appropriate value to force even parity.

**THE** target keeps TRDY# deasserted to insert the wait state necessary for the turnaround cycle on the AD bus.

**CLOCK 3.** On clock three, the targets latch the PAR bit and compare it to the expected parity computed during clock two. If any of the targets have a miscompare, they assert SERR# (if the Parity Error Response and SERR# Enable bits in their respective configuration Command registers are set to one) within two clocks (recommended) after the error was detected. This subject is covered in "Address Phase Parity" on page 215.

**THE** target begins to drive the first data item onto the AD bus and asserts TRDY# to indicate its presence.

**CLOCK 4.** The initiator latches the data on clock four (IRDY# and TRDY# are sampled asserted).

**THE** second data phase begins during clock four. The target drives the second data item onto the bus and keeps TRDY# asserted to indicate its presence.

**DURING** clock four, the target drives the PAR signal to the appropriate state for first data phase parity. The initiator computes the expected parity.

**CLOCK 5.** The initiator latches PAR on clock five and compares it to the expected parity. If the parity is incorrect, the initiator asserts PERR# during clock five.

**THE** initiator latches the data on the rising-edge of clock five (IRDY# and TRDY# sampled asserted).

**DURING** clock five, the initiator computes the expected parity.

The third data phase begins on clock five. The target drives the third data item onto the AD bus and keeps TRDY# asserted to indicate its presence.

**THE** initiator deasserts IRDY# to indicate that it is not yet ready to accept the third data item (e.g., it has a buffer full condition).

**CLOCK 6.** The actual parity is latched from PAR on clock six and checked against the expected parity. If an error is detected, the initiator asserts PERR# during clock six.

**WHEN** the initiator samples TRDY# asserted on clock six, this qualifies the presence of the third data item on the bus. The initiator's parity checker can be designed to sample the data (along with the byte enables) at this point.

**CLOCK 7.** Assuming that it is designed this way, the initiator can latch the PAR bit from the bus on clock seven (it's a rule that the target must present PAR one clock after presenting the data).

**AT** the earliest, then, the initiator could detect a parity miscompare during clock seven and assert PERR#. It must keep PERR# asserted until two clocks after completion of the data phase. ***IN THE EVENT THAT THE INITIATOR ASSERTS PERR# EARLY IN THIS MANNER, THE 2.2 SPEC HAS ADDED A RULE THAT THE INITIATOR MUST EVENTUALLY ASSERT IRDY# TO COMPLETE THE DATA PHASE. THE TARGET IS NOT PERMITTED TO END THE DATA PHASE WITH A RETRY, DISCONNECT WITHOUT DATA, OR A TARGET ABORT.*** 2.2

**DURING** the third data phase, the initiator re-asserts IRDY# during clock seven and deasserts FRAME# to indicate that it is ready to complete the final data phase.

**CLOCK 8.** The final data phase completes on clock eight when IRDY# and TRDY# are sampled asserted. The initiator reads the final data item from the bus at that point.

**CLOCK 9.** At the latest (if early parity check wasn't performed), the initiator must sample PAR one clock afterwards, on clock nine, and, in the event of an error, must assert PERR# during clock nine.

# 14 Interrupts

## Prior To This Chapter

The PCI bus architecture provides two error reporting mechanisms: one for reporting data parity errors and the other for reporting more serious system errors. The previous chapter provided a discussion of error detection, reporting and handling using these two mechanisms.

## In This Chapter

This chapter provides a discussion of issues related to interrupt routing, generation and servicing.

## The Next Chapter

The next chapter describes the 64-bit extension that permits PCI agents to perform eight byte transfers in each data phase. It also describes 64-bit addressing used to address memory targets that reside above the 4GB boundary.

## Three Ways To Deliver Interrupts To Processor

There are three ways in which interrupt requests may be issued to a processor by a hardware device:

**METHOD 1. The Legacy method** delivers interrupt requests to the processor by asserting its INTR input pin. This method is frequently used in single processor Intel x86-based systems (e.g., Celeron-based systems).

**METHOD 2.** In **multiprocessor systems**, the interrupt request lines are tied to inputs on the IO **APIC** (Advanced Programmable Interrupt Controller) and the IO APIC delivers interrupt message packets to the array of processors via the APIC bus. This method is described in the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).

**METHOD 3.** *THE 2.2 PCI SPECIFICATION DEFINES A NEW METHOD FOR DELIVERING INTERRUPT REQUESTS TO THE PROCESSORS BY PERFORMING MEMORY WRITE TRANSACTIONS. THIS METHOD ELIMINATES THE NEED FOR INTERRUPT REQUEST PINS AND SIGNAL TRACES AND IS DESCRIBED IN "MESSAGE SIGNALED INTERRUPTS (MSI)" ON PAGE 252.* 2.2

## Using Pins vs. Using MSI Capability

If a PCI function generates interrupt requests to request servicing by its device driver, the designer has **two choices**:

1. As described in the sections that follow, the device designer can use a **pin** on the device to signal an interrupt request to the processor.
2. ***Alternatively, the designer can implement Message Signaled Interrupt (MSI) capability and use it to signal an interrupt request to the processor. This method eliminates the need for an interrupt pin and trace and is described in the section entitled "Message Signaled Interrupts (MSI)" on page 252.***

2.2

***The spec recommends that a device that implements MSI capability also implement an interrupt pin to allow usage of the device in a system that doesn't support MSI capability. System configuration software must not assume, however, that an MSI-capable device has an interrupt pin.***

*With exclusion of the section entitled "Message Signaled Interrupts (MSI)" on page 252, the remainder of this chapter assumes that MSI is not being used by a device.*

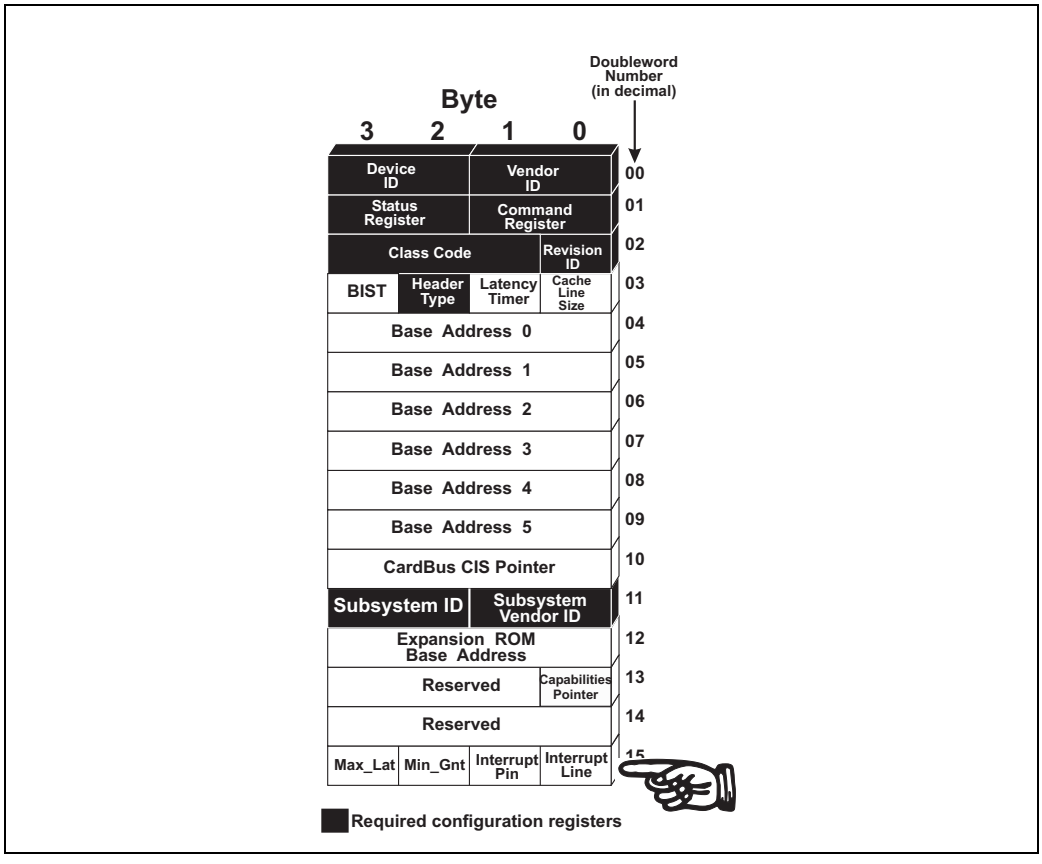
## Single-Function PCI Device

A single-function PCI device is a physical package (add-in board or a component embedded on the PCI bus) that embodies one and only one function (i.e., logical device). If a single-function device generates interrupt requests to request servicing by its device driver, the designer must bond the device's interrupt request signal to the INTA# pin on the package (component or add-in board). A single-function PCI device must only use INTA# (never INTB#, INTC# or INTD#) to generate interrupt requests. In addition, the designer must hardwire this bonding information into the device's read-only, Interrupt Pin configuration register. Table 14-1 on page 223 indicates the value to be hardwired into this register (01h for INTA# for a single-function PCI device). The Interrupt Pin register resides in the second byte of configuration dword number 15d in the device's configuration Header space. The configuration Header space (the first 16d dwords of its configuration space) is illustrated in Figure 14-1 on page 223. It should be stressed that the format illustrated is for that used for PCI device's other than PCI-to-PCI or CardBus bridge devices and is referred to as Header Type Zero. The layout of the Header space for a PCI-to-PCI bridge can be found in "Configuration Registers" on page 552, while the Header layout for a CardBus bridge can be found in the CardBus spec.

*Table 14-1: Value To Be Hardwired Into Interrupt Pin Register*

| Interrupt Signal Bonded To | Value Hardwired In Pin Register |
|---|---|
| Device doesn't generate interrupts. | 00h |
| INTA# pin | 01h |
| INTB# pin | 02h |
| INTC# pin | 03h |
| INTD# pin | 04h |

*Figure 14-1: PCI Logical Device's Configuration Header Space Format*



## Multi-Function PCI Device

A multi-function PCI device is a physical package (add-in board or a component embedded on the PCI bus) that embodies between two and eight PCI functions. An example might be a card with a high-speed communications port and a parallel port implemented in the same package. There is no conceptual difference between a multi-function PCI device and a multi-function ISA, EISA or Micro Channel card.

The device designer may implement up to four interrupt pins on a multi-function device: INTA#, INTB#, INTC# and INTD#. Each function within the package is only permitted to use one of these interrupt pins to generate requests. Each function's Interrupt Pin register indicates which of the package's interrupt pins the device's internal interrupt request signal is bonded to (refer to Table 14-1 on page 223).

If a package implements one pin, it must be called INTA#. If it implements two pins, they must be called INTA# and INTB#, etc. All functions embodied within a package may be bonded to the same pin, INTA#, or each may be bonded to a dedicated pin (this would be true for a package with up to four functions embodied within it).

Groups of functions within the package may share the same pin. As some examples, a package embodying eight functions could bond their interrupt request signals in any of the following combinations:

- all eight bonded to the INTA# pin.
- four bonded to INTA# and four to INTB#.
- two bonded to INTA#, two to INTB#, two to INTC# and two to INTD#.
- seven to INTA# and one to INTB#.
- etc.

## Connection of INTx# Pins To System Board Traces

The temptation is great to imagine that the INTA# pin on every PCI package is connected to a trace on the system board called INTA#, and that the INTB# pin on every PCI package is connected to a trace on the system board called INTB#, etc. While this may be true in a particular system design, it's only one of many different scenarios permitted by the specification.

The system board designer may route the PCI interrupt pins on the various PCI packages to the system board interrupt controller in any fashion. As examples:

- They may all be tied to one trace on the system board that is hardwired to one input on the system interrupt controller.
- They may each be connected to a separate trace on the system board and each of these traces may be hardwired to a separate input on the system interrupt controller.
- They may each be connected to a separate trace on the motherboard. Each of these traces may be connected to a separate input of a programmable interrupt routing device. This device can be programmed at startup time to route each individual PCI interrupt trace to a selected input of the system interrupt controller.
- All of the INTA# pins can be tied together on one trace. All of the INTB# pins can be tied together on another trace, etc. Each of these traces can, in turn, be hardwired to a separate input on the system interrupt controller or may be hardwired to separate inputs on a programmable routing device.
- Etc.

The exact verbiage used in this section of the specification is:

> "The system vendor is free to combine the various INTx# signals from PCI connector(s) in any way to connect them to the interrupt controller. They may be wire-ORed or electronically switched under program control, or any combination thereof."

# Interrupt Routing

## General

Ideally, the system configuration software should have maximum flexibility in choosing how to distribute the interrupt requests issued by various devices to inputs on the interrupt controller. The best scenario is pictured in Figure 14-2 on page 227. In this example, each of the individual PCI interrupt lines is provided to the programmable router as a separate input. In addition, the ISA interrupt request lines are completely segregated from the PCI lines. The ISA lines are connected to the master and slave 8259A interrupt controllers. In turn, the interrupt request output of the master interrupt controller is connected to one of the inputs on the programmable interrupt routing device. The router could be implemented using an Intel IO APIC module. The APIC I/O module can be programmed to assign a separate interrupt vector (interrupt table entry num-

# 15 The 64-bit PCI Extension

## The Previous Chapter

The previous chapter provided a discussion of issues related to interrupt routing, generation and servicing.

## In This Chapter

This chapter describes the 64-bit extension that permits masters and targets to perform eight byte transfers during each data phase. It also describes 64-bit addressing used to address memory targets that reside above the 4GB boundary.

## The Next Chapter

The next chapter describes the implementation of a 66MHz bus and components.

## 64-bit Data Transfers and 64-bit Addressing: Separate Capabilities

The PCI specification provides a mechanism that permits a 64-bit bus master to perform 64-bit data transfers with a 64-bit target. At the beginning of a transaction, the 64-bit bus master automatically senses if the responding target is a 64-bit or a 32-bit device. If it's a 64-bit device, up to eight bytes (a quadword) may be transferred during each data phase. Assuming a series of 0-wait state data phases, throughput of 264Mbytes/second can be achieved at a bus speed of 33MHz (8 bytes/transfer x 33 million transfers/second) and 528Mbytes/second at 66MHz. If the responding target is a 32-bit device, the bus master automatically senses this and steers all data to or from the target over the lower four data paths (AD[31:0]).

The specification also defines 64-bit memory addressing capability. This capability is only used to address memory targets that reside above the 4GB address boundary. Both 32- and 64-bit bus masters can perform 64-bit addressing. In addition, memory targets (that reside over the 4GB address boundary) that respond to 64-bit addressing can be implemented as either 32- or 64-bit targets.

It is important to note that 64-bit addressing and 64-bit data transfer capability are two features, separate and distinct from each other. A device may support one, the other, both, or neither.

## 64-Bit Extension Signals

In order to support the 64-bit data transfer capability, the PCI bus implements an additional thirty-nine pins:

- **REQ64#** is asserted by a 64-bit bus master to indicate that it would like to perform 64-bit data transfers. REQ64# has the same timing and duration as the FRAME# signal. The REQ64# signal line must be supplied with a pullup resistor on the system board. REQ64# cannot be permitted to float when a 32-bit bus master is performing a transaction.
- **ACK64#** is asserted by a target in response to REQ64# assertion by the master (if the target supports 64-bit data transfers). ACK64# has the same timing and duration as DEVSEL# (but ACK64# must not be asserted unless REQ64# is asserted by the initiator). Like REQ64#, the ACK64# signal line must also be supplied with a pullup resistor on the system board. ACK64# cannot be permitted to float when a 32-bit device is the target of a transaction.
- **AD[63:32]** comprise the upper four address/data paths.
- **C/BE#[7:4]** comprise the upper four command/byte enable signals.
- **PAR64** is the parity bit that provides even parity for the upper four AD paths and the upper four C/BE signal lines.

The following sections provide a detailed discussion of 64-bit data transfer and addressing capability.

## 64-bit Cards in 32-bit Add-in Connectors

A 64-bit card installed in a 32-bit expansion slot automatically only uses the lower half of the bus to perform transfers. This is true because the system board designer connects the REQ64# output pin and the ACK64# input pin on the connector to individual pullups on the system board and to nothing else.

When a 64-bit bus master is installed in a 32-bit card slot and it initiates a transaction, its assertion of REQ64# is not visible to any of the targets. In addition, its ACK64# input is always sampled deasserted (because it's pulled up on the system board). This forces the bus master to use only the lower part of the bus during the transfer. Furthermore, if the target addressed in the transaction is a 64-bit target, it samples REQ64# deasserted (because it's pulled up on the system board), forcing it to only utilize the lower half of the bus during the transaction and to disable its ACK64# output.

The 64-bit extension signal lines on the card itself cannot be permitted to float when they are not in use. The CMOS input receivers on the card would oscillate and draw excessive current, thus violating the "green" aspect of the specification. When the card is installed in a 32-bit slot, it cannot use the upper half of the bus. The manner in which the card detects the type of slot (REQ64# sampled deasserted at startup time) is described in the next section.

## Pullups Prevent 64-bit Extension from Floating When Not in Use

If the 64-bit extension signals (AD[63:32], C/BE#[7:4] and PAR64) are permitted to float when not in use, the CMOS input buffers on the card will oscillate and draw excessive current. In order to prevent the extension from floating when not in use, the system board designer is required to include pullup resistors on the extension signals to keep them from floating. Because these pullups are guaranteed to keep the extension from floating when not in use, 64-bit devices that are embedded on the system board and 64-bit cards installed in 64-bit PCI add-in connectors don't need to take any special action to keep the extension from floating when they are not using it.

The 64-bit extension is not in use under the following circumstances:

1. The PCI bus is idle.
2. A 32-bit bus master is performing a transaction with a 32-bit target.
3. A 32-bit bus master is performing a transaction with a 64-bit target. Upon detecting REQ64# deasserted at the start of the transaction, the target will not use the upper half of the bus.
4. A 64-bit bus master addresses a target to perform 32-bit data transfers (REQ64# deasserted) and the target resides below the 4GB address boundary (the upper half of the bus is not used during the address phase and is also not used in the data phases). Whether the target is a 32-bit or a 64-bit target, the upper half of the bus isn't used during the data phases (because REQ64# is deasserted).

5. A 64-bit bus master attempts a 64-bit data transfer (REQ64# asserted) with a 32-bit memory target that resides below the 4GB boundary. In this case, the initiator only uses the lower half of the bus during the address phase (because it's only generating a 32-bit address). When it discovers that the currently-addressed target is a 32-bit target (ACK64# not asserted when DEVSEL# asserted), the initiator ceases to use the upper half of the bus during the data phases.

## Problem: a 64-bit Card in a 32-bit PCI Connector

Refer to Figure 15-1 on page 269. Installation of a 64-bit card in a 32-bit card connector is permitted. The main (32-bit) portion of the connector contains all of the 32-bit PCI signals, while an extension to the connector contains the 64-bit extension signals (with the exception of REQ64# and ACK64# which are located on the 32-bit portion of the connector).

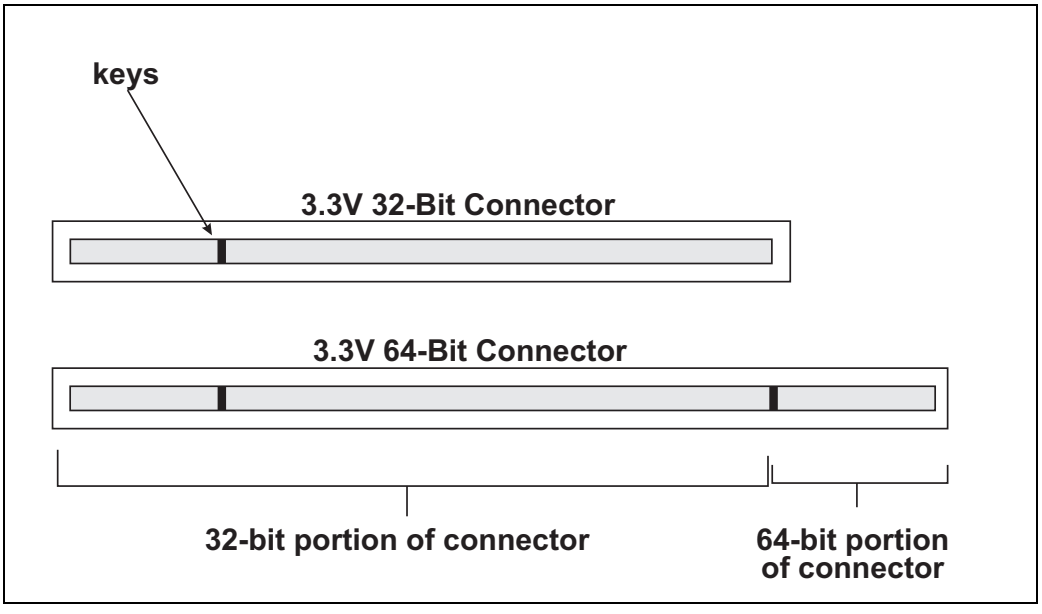
When a 64-bit device is installed in a 32-bit PCI expansion slot, the system board pullups on AD[63:32], C/BE#[7:4] and PAR64 are not available to the add-in card. This means that the add-in card's input buffers that are connected to the extension signal pins will float, oscillate, and draw excessive current.

The specification states that the add-in card designer must ***not*** solve this problem by supplying pullup resistors on the extension lines on the add-in card. Using this approach would cause problems when the card is installed in a 64-bit expansion slot. There would then be two sets of pullup resistors on these signal lines (the ones on the card plus the ones on the system board). If all designers solved the problem in this manner, a machine with multiple 64-bit cards inserted in 64-bit card connectors would have multiple pullups on the extension signals, resulting in pullup current overload.

The specification provides a method for a 64-bit card to determine at startup time whether it's installed in a 32-bit or a 64-bit connector. If the card detects that it is plugged into a 64-bit connector, the pullups on the system board will keep the input receivers on the card from floating when the extension is not in use. On the other hand, if a 64-bit card detects that it is installed in a 32-bit card connector, the logic on the card must keep the input receivers from switching. The specification states that an approach similar to one of the following should be used:

- Biasing the input buffer to turn it off.
- Actively driving the outputs continually (since they aren't connected to anything).

*Figure 15-1: 64- and 32- Bit Connectors*




## How 64-bit Card Determines Type of Slot Installed In

Refer to Figure 15-2 on page 270. When the system is powered up, the reset signal is automatically asserted. During this period of time, the logic on the system board must assert the REQ64# signal as well as RST#. REQ64# has a single pullup resistor on it and is connected to the REQ64# pin on all 64-bit devices integrated onto the system board and on all 64-bit PCI expansion slots. The specification states that the REQ64# signal line on each 32-bit PCI expansion slot (REQ64# and ACK64# are located on the 32-bit portion of the connector), however, each has its own independent pullup resistor.

During reset time, the system board reset logic initially asserts the PCI RST# signal while the POWERGOOD signal from the power supply is deasserted. During the assertion of RST#, the system board logic asserts REQ64# and keeps it asserted until after it removes the RST# signal. When POWERGOOD is asserted by the power supply logic, the system board reset logic deasserts the PCI RST# signal. On the trailing-edge of RST# assertion, all 64-bit devices are required to sample the state of the REQ64# signal.

# 16 66MHz PCI Implementation

## Prior To This Chapter

The previous chapter described the 64-bit extension that permits masters and targets to perform eight byte transfers during each data phase. It also described 64-bit addressing used to address memory targets that reside above the 4GB boundary.

## In This Chapter

This chapter describes the implementation of a 66MHz bus and components.

## The Next Chapter

The next chapter provides an introduction to PCI configuration address space. The concept of single- and multi-function devices is described. The configuration space available to the designer of a PCI device is introduced, including the device's configuration Header space and its device-specific configuration register area.

## Introduction

The PCI specification defines support for the implementation of buses and components that operate at speeds of up to 66MHz. This chapter covers the issues related to this topic. Note that all references to 66MHz in this chapter indicate a frequency within the range from 33.33MHz to 66.66MHz.

## 66MHz Uses 3.3V Signaling Environment

66MHz components only operate correctly in a 3.3V signaling environment (see "3.3V, 5V and Universal Cards" on page 449). The 5V environment is not supported. This means that 66MHz add-in cards are keyed to install only in 3.3V connectors and cannot be installed in 5V card connectors.

## How Components Indicate 66MHz Support

The 66MHz PCI component or add-in card indicates its support in two fashions: programmatically and electrically.

### 66MHz-Capable Status Bit

The 66MHz-Capable bit has been added to the Status register (see Figure 16-1 on page 302). The designer hardwires a one into this bit if the device supports operation from 0 through 66.66MHz. A 66MHz-capable device hardwires this bit to one. For all 33MHz devices, this bit is reserved and is hardwired to zero. Software can determine the speed capability of a PCI bus by checking the state of this bit in the Status register of the bridge to the bus in question (host / PCI or PCI-to-PCI bridge). Software can also check this bit in the Status register of each additional device discovered on the bus in question to determine if all of the devices on the bus are 66MHz-capable. If just one device returns a zero from this bit, the bus runs at 33MHz (or slower), not 66MHz. Table 16-1 on page 300 defines the combinations of bus and device capability that may be detected.

*Table 16-1: Combinations of 66MHz-Capable Bit Settings*

| Bridge's 66MHz-Capable Bit | Device's 66MHz-Capable Bit | Description |
|---|---|---|
| 0 | 0 | Bus is a 33MHz bus, so all devices operate at 33MHz. |
| 0 | 1 | 66MHz-capable device located on 33MHz bus. Bus and all devices operate at 33MHz. If the device is an add-in device and requires the throughput available on a 66MHz bus, the configuration software may prompt the user to install the card in an add-in connector on a different bus. |
| 1 | 0 | 33MHz device located on 66MHz-capable bus. Bus and all devices operate at 33MHz. The configuration software should prompt the user to install the card in an add-in connector on a different bus. |

*Table 16-1: Combinations of 66MHz-Capable Bit Settings (Continued)*

| Bridge's 66MHz-Capable Bit | Device's 66MHz-Capable Bit | Description |
|---|---|---|
| 1 | 1 | 66MHz-capable device located on 66MHz-capable bus. If status check of all other devices on the bus indicates that all of the devices are 66MHz-capable, the bus and all devices operate at 66MHz. |

## M66EN Signal

Refer to Figure 16-2 on page 302. A 66MHz PCI bus includes a newly-defined signal, M66EN. This signal must be bussed to the M66EN pin on all 66MHz-capable devices embedded on the system board and to a redefined pin (referred to as M66EN) on any 3.3V connectors that reside on the bus. The system board designer must supply a single pullup on this trace. The redefined pin on the 3.3V connector is B49 and is attached to the ground plane on 33MHz PCI cards. Unless grounded by insertion of a PCI device, the natural state of the M66EN signal is asserted (due to the pullup). 66MHz embedded devices and cards either use M66EN as an input or don't use it at all (this is discussed later in this chapter).

The designer must include a 0.01uF capacitor located within .25" of the M66EN pin on each add-in connector in order to provide an AC return path and to decouple the M66EN signal to ground.

It is advisable to attach M66EN to a bit in a machine-readable port to allow software to determine if the bus is currently operating at high or low speed. Refer to "66MHz-Related Issues" on page 476.

## How Clock Generator Sets Its Frequency

PCI devices embedded on a 66MHz PCI bus are all 66MHz devices. A card installed in a connector on the bus may be either a 66MHz or a 33MHz card. If the card connector(s) isn't populated, M66EN stays asserted (by virtue of the pullup) and the Clock Generator produces a high-speed clock. If any 33MHz component is installed in a connector, however, the ground plane on the 33MHz card is connected to the M66EN signal, deasserting it. This causes the Clock Generator to drop the clock frequency to 33MHz (or lower). A typical implementation would divide the high-speed clock by two to yield the new, lower clock frequency.

*Figure 16-1: Configuration Status Register*

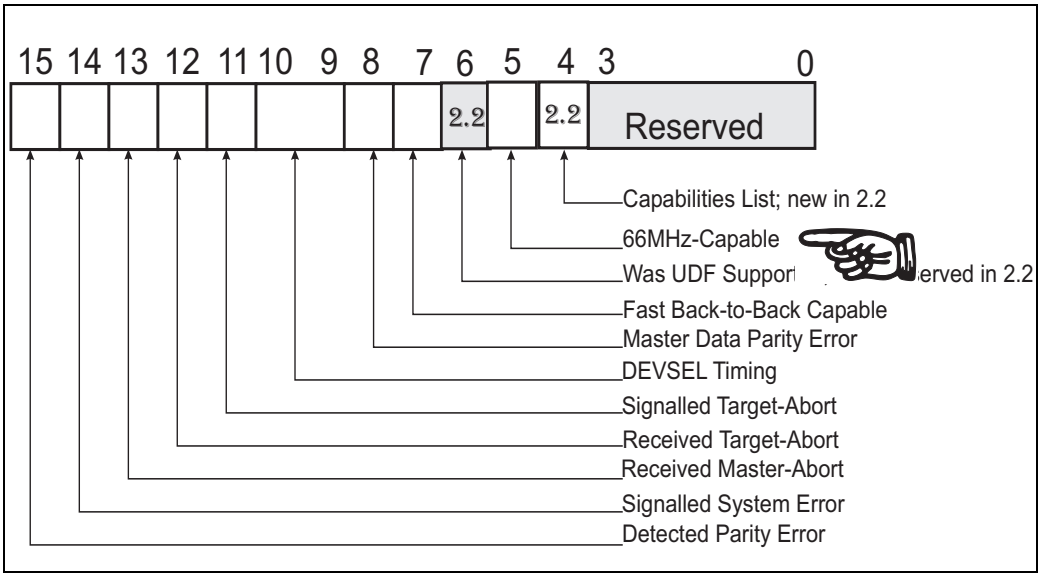
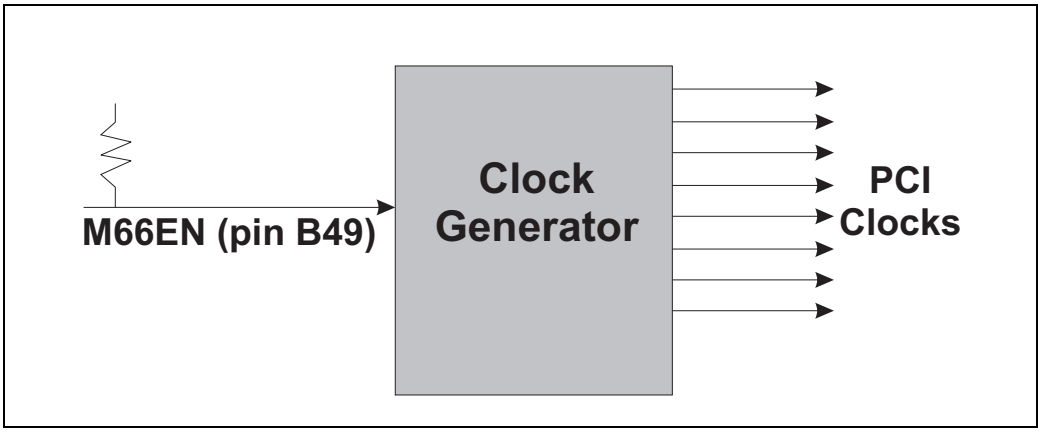



*Figure 16-2: Relationship of M66EN Signal and the PCI Clock Generator*



## Does Clock Have to be 66MHz?

As defined in revision 1.0 and 2.0 of the specification, the PCI bus does not have to be implemented at its top rated speed of 33MHz. Lower speeds are acceptable. The same is true of the 66MHz PCI bus description found in revision 2.x of the specification. All 66MHz-rated components are required to support operation from 0 through 66.66MHz. The system designer may choose to implement a 50MHz PCI bus, a 60MHz PCI bus, etc.

## Clock Signal Source and Routing

The specification recommends that the PCI clock be individually-sourced to each PCI component as a point-to-point signal from separate, low-skew clock drivers. This diminishes signal reflection effects and improves signal integrity. In addition, the add-in card designer must adhere to the clock signal trace length (defined in revision 2.0) of 2.5 inches.

## Stopping Clock and Changing Clock Frequency

The 66MHz specification states that the clock frequency may be changed at any time as long as the clock edges remain clean and the minimum high and low times are not violated. Unlike the 33MHz specification, however, the clock frequency may not be changed except in conjunction with assertion of the PCI RST# signal. Components designed to be integrated onto the system board may be designed to operate at a fixed frequency (up to 66MHz) and may require that no clock frequency changes occur.

The clock may be stopped (to conserve power), but only in the low state.

## How 66MHz Components Determine Bus Speed

When a 66MHz-capable device senses M66EN deasserted (at reset time), this automatically disables the device's ability to perform operations at speeds above 33MHz. If M66EN is sensed asserted, this indicates that no 33MHz devices are installed on the bus and the clock circuit is supplying a high-speed PCI clock.

A 66MHz device uses the M66EN signal in one of two fashions:

- The device is not connected to M66EN at all (because the device has no need to determine the bus speed in order to operate correctly).
- The device implements M66EN as an input (because the device requires knowledge of the bus speed in order to operate correctly).

# 17 Intro to Configuration Address Space

## The Previous Chapter

The previous chapter described the implementation of a 66MHz bus and components.

## This Chapter

As the chapter title states, this chapter provides an introduction to PCI configuration address space. The concept of single- and multi-function devices is described. The configuration space available to the designer of a PCI device is introduced, including the device's configuration Header space and its device-specific configuration register area.

## The Next Chapter

The next chapter provides a detailed discussion of the methods utilized to access PCI configuration registers. The methods described include the standard configuration mechanism, the legacy method that is no longer permitted, and the usage of memory-mapped configuration registers (as implemented in the PowerPC PREP platforms). The Type Zero and Type One configuration read and write transactions that are used to access configuration registers are described in detail.

# Introduction

When the machine is first powered on, the configuration software must scan the various buses in the system (PCI and others) to determine what devices exist and what configuration requirements they have. This process is commonly referred to as:

- scanning the bus
- walking the bus

- probing the bus
- the discovery process
- bus enumeration.

The program that performs the PCI bus scan is frequently referred to as the PCI bus enumerator.

In order to facilitate this process, each PCI function must implement a base set of configuration registers defined by the PCI specification. Depending on its operational characteristics, a function may also implement other required or optional configuration registers defined by the specification. In addition, the specification sets aside a number of additional configuration locations for the implementation of function-specific configuration registers.

The configuration software reads a subset of a device's configuration registers in order to determine the presence of the function and its type. Having determined the presence of the device, the software then accesses the function's other configuration registers to determine how many blocks of memory and/or IO space the device requires (in other words, how many programmable memory and/or IO decoders it implements). It then programs the device's memory and/or IO address decoders to respond to memory and/or IO address ranges that are guaranteed to be mutually-exclusive from those assigned to other system devices.

If the function indicates usage of a PCI interrupt request pin (via one of its configuration registers), the configuration software programs it with routing information indicating what system interrupt request (IRQ) line the function's PCI interrupt request pin is routed to by the system.

If the device has bus mastering capability, the configuration software can read two of its configuration registers to determine how often it requires access to the PCI bus (an indication of what arbitration priority it would like to have) and how long it would like to maintain ownership in order to achieve adequate throughput. The system configuration software can utilize this information to program the bus master's Latency Timer (or timeslice) register and the PCI bus arbiter (if it's programmable) to provide the optimal PCI bus utilization.

## PCI Device vs. PCI Function

A physical PCI device (e.g., a PCI component embedded on the system board or a PCI expansion board) may contain one or more (up to eight) separate PCI functions (i.e., logical devices). This is not a new concept. Multi-function boards were used in PCs for years prior to the advent of PCI.

In order to access a function's configuration registers, a PCI device (i.e., package) needs to know that it is the target of a configuration read or write, the identity of the target function within it, the dword of its configuration space (1-of-64), the bytes within that dword, and whether it's a read or a write. The method that the host/PCI bridge uses to identify the target PCI device, PCI function, etc., is discussed in the chapter entitled "Configuration Transactions" on page 317.

A PCI device that contains only one function is referred to as a single-function device. A PCI device that contains more than one function is referred to as a multi-function device. A bit in one of a function's configuration registers defines whether the package contains one function or more than one. For the configuration process, a device's PCI functions are identified as functions zero-through-seven. From a configuration access standpoint, the function contained in a single-function device must respond as function zero when addressed in a Type 0 PCI configuration read or write transaction.

In a multi-function device, the first function must be designed to respond to configuration accesses as function zero, while additional functions may be designed to respond as any function between one and seven. There is no requirement for multiple functions to be implemented sequentially. As an example, a card may be sold with minimal functionality and the customer may purchase additional functions as upgrades at a later time. These functions could be installed into any of several daughter-card connectors on the card or may be installed as snap-in modules on the card. As an example, a card could have functions zero, three and six populated, but not the others.

## Three Address Spaces: I/O, Memory and Configuration

Intel x86 and PowerPC 60x processors possess the ability to address two distinct address spaces: IO and memory (although most system designs do not support processor-generated IO transactions in a PowerPC environment). PCI bus masters (including the host/PCI bridge) use PCI IO and memory transactions to access PCI IO and memory locations, respectively. In addition, a third access type, the configuration access, is used to access a device's configuration registers. A function's configuration registers must be initialized at startup time to configure the function to respond to memory and/or IO address ranges assigned to it by the configuration software.

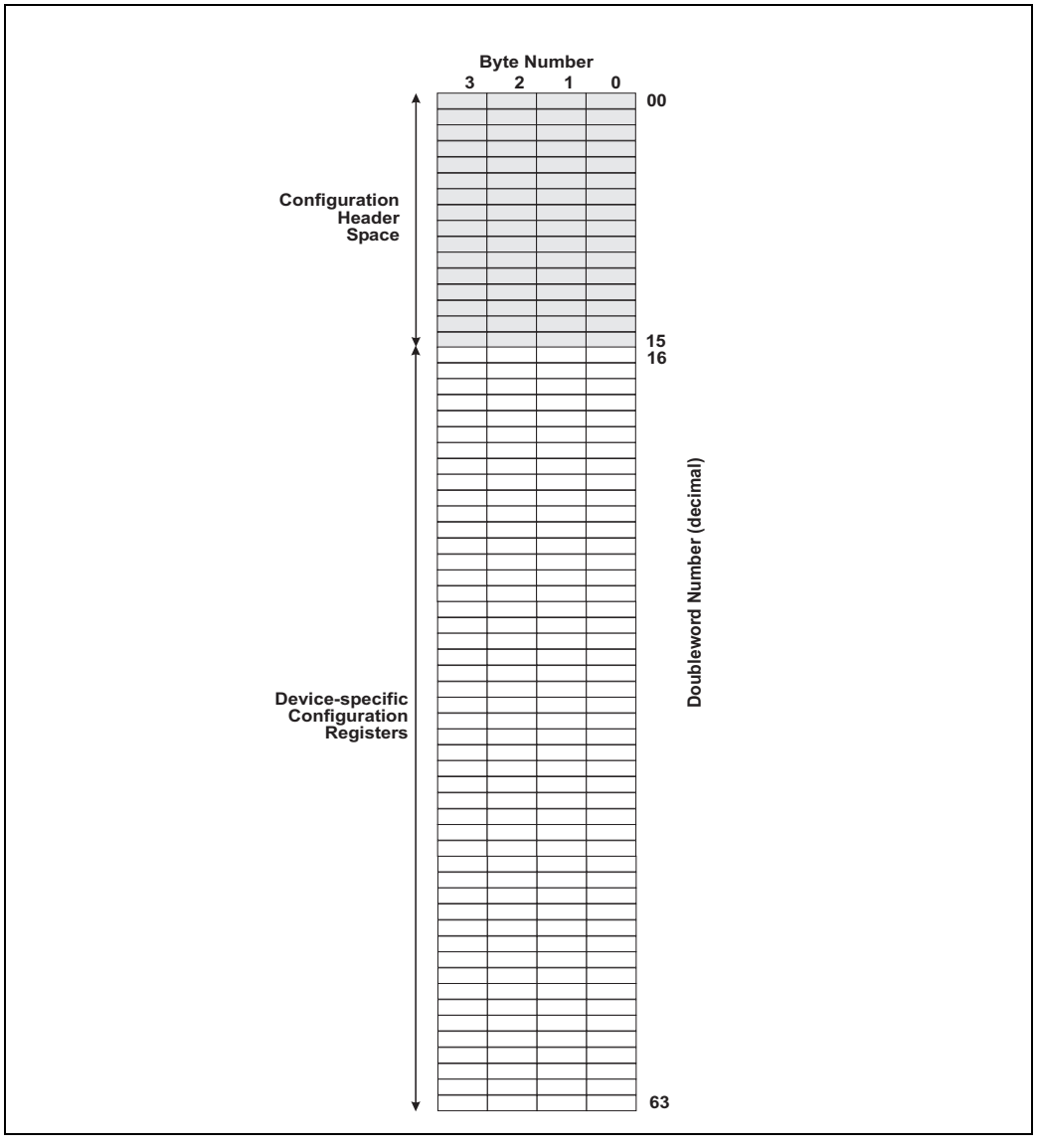
The PCI memory space is either 4GB or $2^{64}$ locations in size (if 64-bit addressing is utilized). PCI IO space is 4GB in size (although Intel x86 processors cannot generate IO addresses above the first 64KB of IO space). PCI configuration

space is divided into a separate, dedicated configuration address space for each function contained within a PCI device (i.e., in a chip or on a card). Figure 17-1 on page 313 illustrates the basic format of a PCI function's configuration space. The first 16 dwords of a function's configuration space is referred to as the function's configuration Header space. The format and usage of this area are defined by the specification. Three Header formats are currently defined:

- **Header Type Zero** (defined in "Intro to Configuration Header Region" on page 351) for all devices other than PCI-to-PCI bridges.
- **Header Type One** for PCI-to-PCI bridges (defined in "Configuration Registers" on page 552).
- **Header Type Two** for CardBus bridges (defined in the CardBus spec).

The system designer must provide a mechanism that the host/PCI bridge will use to convert processor-initiated accesses with certain pre-defined memory or IO addresses into configuration accesses on the PCI bus. The mechanism defined in the specification is described in "Intro to Configuration Mechanisms" on page 321.

*Figure 17-1: PCI Function's Basic Configuration Address Space Format*



# 18 Configuration Transactions

## The Previous Chapter

The previous chapter provided an introduction to PCI configuration address space. The concept of single- and multi-function devices was described. The configuration space available to the designer of a PCI device was introduced, including the device's configuration Header space and its device-specific configuration register area.

## This Chapter

This chapter provides a detailed discussion of the method utilized to access PCI configuration registers. The methods described include the standard configuration mechanism, the legacy method that is no longer permitted, and the usage of memory-mapped configuration registers (as implemented in the PowerPC PReP platforms). The Type Zero configuration read and write configuration transactions that are used to access configuration registers are described in detail. The Type One configuration transactions, used to access configuration registers on the secondary side of a PCI-to-PCI bridge, are also described.

## The Next Chapter

Once this chapter has described how the registers are accessed, the next chapter provides a detailed description of the configuration register format and usage for all PCI devices other than PCI-to-PCI bridges and CardBus bridges.

# Who Performs Configuration?

Initially, the BIOS code performs device configuration. Once a Plug-and-Play OS (such as Windows 98 or Windows 2000) has been booted and control is passed to it, the OS takes over device management. Additional information regarding PCI device configuration in the Windows environment can be found in "PCI Bus Driver Accesses PCI Configuration and PM Registers" on page 489.

In any case, it is a configuration program executing on the processor that performs system configuration. This means that the host processor must have some way to instruct the host/PCI bridge to perform configuration read and write transactions on the PCI bus.

# Bus Hierarchy

## Introduction

This chapter focuses on the configuration of systems with one PCI bus (see Figure 18-1 on page 320). Assuming that the system has one or more PCI expansion connectors, however, a card may be installed at any time (perhaps more than one card) that incorporates a PCI-to-PCI bridge. The card has another PCI bus on it, and one or more PCI devices reside on that PCI bus. The configuration software executing on the host processor must be able to perform configuration reads and writes with the functions on each PCI bus that lives beyond the host/PCI bridge.

This highlights the fact that the programmer must supply the following information to the host/PCI bridge when performing a configuration read or write:

- target PCI bus.
- target PCI device (i.e., package) on the bus.
- target PCI function within the device.
- target dword within the function's configuration space.
- target byte(s) within the dword.

These parameters must be supplied to the host/PCI bridge. The bridge must then determine if the target PCI bus specified is:

- the bus immediately on the other side of the host/PCI bridge (in other words, Bus 0)
- a bus further out in the bus hierarchy
- none of the buses behind the bridge.

This implies that the host/PCI bridge has some way of identifying its PCI bus and the range of PCI buses residing beyond its bus. The bus on the other side of the host/PCI bridge is always bus 0 (unless there are more than one host/PCI bridge on the processor bus; for more information, refer to "Multiple Host/PCI Bridges" on page 327). The bridge either implicitly knows this or implements a *Bus Number register* that contains zero after RST# is deasserted. The bridge also

incorporates a *Subordinate Bus Number register* that it uses to identify the bus furthest away from the host processor beyond this bridge. The bridge compares the target bus number specified by the programmer to the range of buses that exists beyond the bridge. There are three possible cases:

**CASE 1.** The target bus is bus 0.

**CASE 2.** The target bus isn't bus 0, but is less than or equal to the value in the Subordinate Bus Number register. In other words, the transaction targets a device on a subordinate bus.

**CASE 3.** The target bus doesn't fall within the range of buses that exists beyond this bridge.

In Cases 1 and 2, the bridge will initiate a PCI configuration read or write on the PCI bus in response to the processor's request. In the third case, however, the bridge doesn't respond to the processors' request to perform a PCI configuration transaction at all (because the target bus is not behind it).

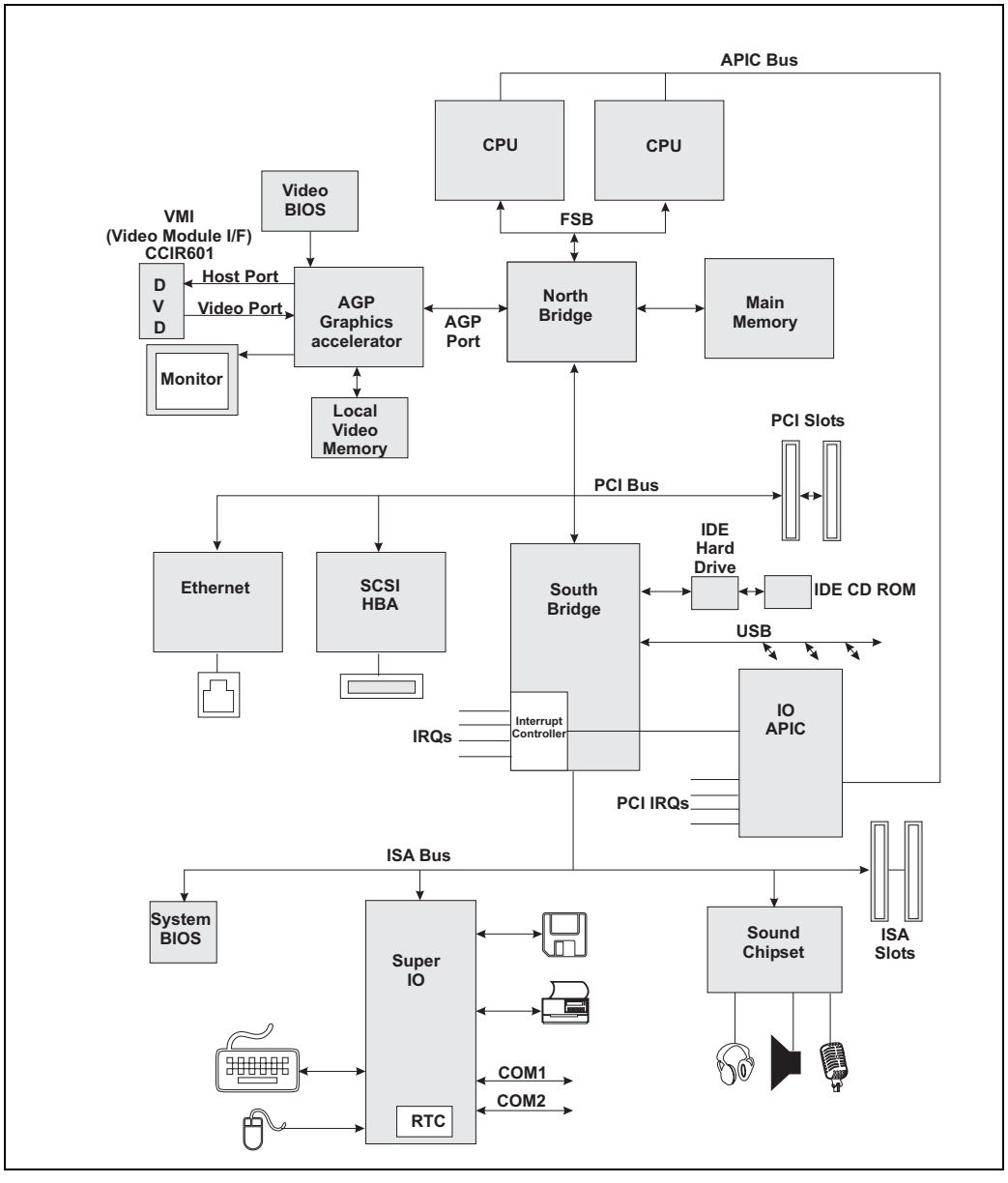
## Case 1: Target Bus Is PCI Bus 0

If the target bus is bus 0 (the first case), the bridge must initiate a PCI configuration transaction and in some way indicate to the devices on Bus 0 that one of them is the target of this configuration transaction. This is accomplished by setting AD[1:0] to 00b during the address phase of the configuration transaction. This identifies the transaction as a **Type 0** configuration transaction targeting one of the devices on this bus. This bit pattern tells the community of devices on the PCI bus that the bridge that "owns" that bus has already performed the bus number comparison and verified that the request targets a device on its bus. A detailed description of the Type 0 configuration transaction can be found in "Type 0 Configuration Transaction" on page 335.

## Case 2: Target Bus Is Subordinate To Bus 0

If, on the other hand, the target bus is a bus that is subordinate to PCI bus 0 (the second case), the bridge still must initiate the configuration transaction on bus 0, but must indicate in some manner that none of the devices on this bus is the target of the transaction. Rather, only PCI-to-PCI bridges residing on the bus should pay attention to the transaction because it targets a device on a bus further out in the hierarchy beyond a PCI-to-PCI bridge that is attached to Bus 0. This is accomplished by setting AD[1:0] to 01b during the address phase of the configuration transaction. This pattern instructs all functions other than PCI-to-PCI bridges that the transaction is not for any of them and is referred to as a **Type 1** configuration transaction. A detailed description of the Type 1 configuration access can be found in "Type 1 Configuration Transactions" on page 344.

*Figure 18-1: Typical PC System Block Diagram*



## Must Respond To Config Accesses Within $2^{25}$ Clocks After RST# 2.2

*AS DEFINED IN THE 2.2 SPEC, INITIALIZATION-TIME BEGINS WHEN RST# IS DEASSERTED AND COMPLETES $2^{25}$ PCI CLOCKS LATER. AT A BUS SPEED OF 33MHZ, THIS EQUATES TO 1.0066 SECONDS, WHILE IT EQUATES TO .5033 SECONDS AT A BUS SPEED OF 66MHZ. DURING THIS PERIOD OF TIME:*

- *THE SYSTEM'S POWER-ON SELF-TEST (POST) CODE IS EXECUTING.*
- *PCI SUBSYSTEMS (I.E., FUNCTIONS) ARE TAKING WHATEVER ACTIONS ARE NECESSARY TO PREPARE TO BE ACCESSED.*

*IF A TARGET IS ACCESSED DURING INITIALIZATION-TIME, IT IS ALLOWED TO DO ANY OF THE FOLLOWING:*

- *IGNORE THE REQUEST (UNLESS IT IS A DEVICE NECESSARY TO BOOT THE OS).*
- *CLAIM THE ACCESS AND HOLD IN WAIT STATES UNTIL IT CAN COMPLETE THE REQUEST, NOT TO EXCEED THE END OF INITIALIZATION-TIME.*
- *CLAIM THE ACCESS AND TERMINATE WITH RETRY.*

*RUN-TIME FOLLOWS INITIALIZATION-TIME. DEVICES MUST COMPLY WITH THE LATENCY RULES (SEE "PREVENTING TARGET FROM MONOPOLIZING BUS" ON PAGE 81) DURING RUN-TIME.*

## Intro to Configuration Mechanisms

This section describes the methods used to stimulate the host/PCI bridge to generate PCI configuration transactions. A subsequent section in this chapter provides a detailed description of Type 0 configuration transactions. The section entitled "Type 1 Configuration Transactions" on page 344 provides a detailed description of the Type 1 configuration transactions.

As mentioned earlier in this book, Intel x86 and PowerPC processors (as two examples processor families) do not possess the ability to perform configuration read and write transactions. They use memory and IO (IO is only in the x86 case) read and write transactions to communicate with external devices. This means that the host/PCI bridge must be designed to recognize certain IO or memory accesses initiated by the processor as requests to perform configuration accesses.

# 19 Configuration Registers

## The Previous Chapter

The previous chapter provided a detailed discussion of the mechanisms used to generate configuration transactions as well as a detailed discussion of the Type Zero and Type One configuration read and write transactions.

## This Chapter

This chapter provides a detailed description of the configuration register format and usage for all PCI devices other than PCI-to-PCI bridges and CardBus bridges.

## The Next Chapter

The next chapter provides a detailed description of device ROMs associated with PCI devices. This includes the following topics:

- device ROM detection.
- internal code / data format.
- shadowing.
- initialization code execution.
- interrupt hooking.

## Intro to Configuration Header Region

***WITH THE POSSIBLE EXCEPTION OF THE HOST/PCI BRIDGE***, every PCI function must implement PCI configuration space within which its PCI configuration registers reside. The host / PCI bridge could implement these registers, in PCI configuration space (this is most often the case), in IO space (much too crowded), or in memory space. 2.2

Each PCI function possesses a block of 64 configuration dwords reserved for the implementation of its configuration registers. The format and usage of the first 16 dwords is predefined by the PCI specification. This area is referred to as the device's Configuration Header Region (or Header Space). The specification currently defines three Header formats, referred to as Header Types Zero, One and Two.

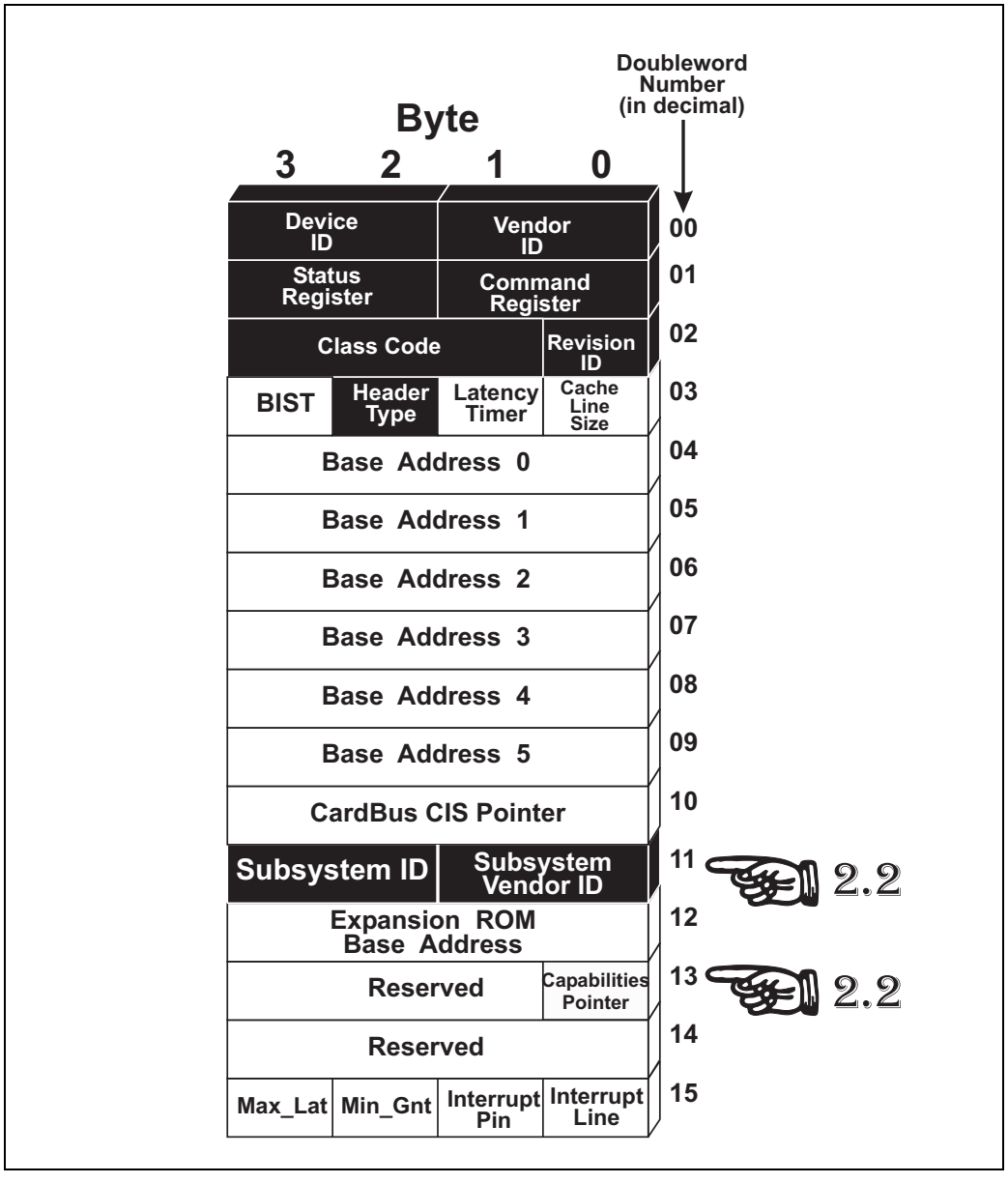
- **Header Type One** is defined for PCI-to-PCI bridges. A full description of Header Type One can be found in "Configuration Registers" on page 552.
- **Header Type Two** is defined for PCI-to-CardBus bridges and is fully-defined in the PC Card spec.
- **Header Type Zero** is used for all devices other than PCI-to-PCI and cardBus bridges. This chapter defines Header Type Zero.

Figure 19-1 on page 353 illustrates the format of a function's Header region (for functions other than PCI-to-PCI bridges and CardBus bridges). The registers marked in black are always mandatory. Note that although many of the configuration registers in the figure are not marked mandatory, a register may be mandatory for a particular type of device. The subsequent sections define each register and any circumstances where it may be mandatory.

As noted earlier, this format is defined as Header Type Zero. The registers within the Header are used to identify the device, to control its PCI functionality and to sense its PCI status in a generic manner. The usage of the device's remaining 48 dwords of configuration space is device-specific, *BUT IT IS NOW ALSO USED AS AN OVERFLOW AREA FOR SOME NEW REGISTERS DEFINED IN THE 2.2. PCI SPEC (FOR MORE INFORMATION, REFER TO "NEW CAPABILITIES" ON PAGE 390).*

2.2

*Figure 19-1: Format of a PCI Function's Configuration Header*



# Mandatory Header Registers

## Introduction

The following sections describe the mandatory configuration registers that must be implemented in every PCI device, including bridges. The registers are illustrated (in black) in Figure 19-1 on page 353.

## Registers Used to Identify Device's Driver

The OS uses some combination of the following mandatory registers to determine which driver to load for a device:

- Vendor ID.
- Device ID.
- Revision.
- Class Code.
- SubSystem Vendor ID.
- SubSystem ID.

### Vendor ID Register

*Always mandatory.* This 16-bit register identifies the manufacturer of the device. The value hardwired in this read-only register is assigned by a central authority (the PCI SIG) that controls issuance of the numbers. The value FFFFh is reserved and must be returned by the host/PCI bridge when an attempt is made to perform a configuration read from a non-existent device's configuration register. The read attempt results in a Master Abort and the bridge must respond with a Vendor ID of FFFFh. The Master Abort is not considered to be an error, but the specification says that the bridge must none the less set its Received Master Abort bit in its configuration Status register.

### Device ID Register

*Always mandatory.* This 16-bit value is assigned by the device manufacturer and identifies the type of device. In conjunction with the Vendor ID and possibly the Revision ID, the Device ID can be used to locate a device-specific (and perhaps revision-specific) driver for the device.

## Subsystem Vendor ID and Subsystem ID Registers

2.2

*Mandatory.* This register pair was added in revision **2.1** of the specification and **was optional**. *THE 2.2 SPEC HAS MADE THEM MANDATORY EXCEPT FOR THOSE THAT HAVE A BASE CLASS OF 06H (A BRIDGE) WITH A SUB CLASS OF 00H-04H (REFER TO TABLE 19-8 ON PAGE 360), OR A BASE CLASS OF 08H (BASE SYSTEM PERIPHERALS) WITH A SUB CLASS OF 00H-03H (SEE TABLE 19-10 ON PAGE 363). THIS EXCLUDES BRIDGES OF THE FOLLOWING TYPES:*

- *HOST/PCI*
- *PCI-TO-EISA*
- *PCI-TO-ISA*
- *PCI-TO-MICRO CHANNEL*
- *PCI-TO-PCI*

*IT ALSO EXCLUDES THE FOLLOWING GENERIC SYSTEM PERIPHERALS:*

- *INTERRUPT CONTROLLER*
- *DMA CONTROLLER*
- *PROGRAMMABLE TIMERS*
- *RTC CONTROLLER*

The Subsystem Vendor ID is obtained from the SIG, while the vendor supplies its own Subsystem ID (the full name of this register is really "Subsystem Device ID", but the "device" is silent). A value of zero in these registers indicates there isn't a Subsystem Vendor and Subsystem ID associated with the device.

**Purpose of This Register Pair.** A PCI function may reside on a card or within an embedded device. Two cards or subsystems that are designed around the same PCI core logic (produced by a third-party) may have the same Vendor and Device IDs (if the core logic vendor hardwired their own IDs into these registers). If this is the case, the OS would have a problem identifying the correct driver to load into memory for the device.

These two mandatory registers (Subsystem Vendor ID and Subsystem ID) are used to uniquely identify the add-in card or subsystem that the device resides within. Using these two registers, the OS can distinguish the difference between cards or subsystems manufactured by different vendors but designed around the same third-party core logic. This permits the Plug-and-Play OS to locate the correct driver to load into memory.

# 20 Expansion ROMs

## The Previous Chapter

The previous chapter provided a detailed description of the configuration register format and usage for all PCI devices other than PCI-to-PCI bridges and CardBus bridges.

## This Chapter

This chapter provides a detailed description of device ROMs associated with PCI devices. This includes the following topics:

- device ROM detection.
- internal code / data format.
- shadowing.
- initialization code execution.
- interrupt hooking.

## The Next Chapter

The next chapter provides an introduction to the PCI expansion card and connector definition. It covers card and connector types, 5V and 3.3V operability, shared slots, and pinout definition.

# ROM Purpose—Device Can Be Used In Boot Process

In order to boot the OS into memory, the system needs three devices:

- A mass storage device to load the OS from. This is sometimes referred to as the **IPL** (Initial Program Load) **device** and is typically an IDE or a SCSI hard drive.
- A display adapter to enable progress messages to be displayed during the boot process. In this context, this is typically referred to as the **output device**.
- A keyboard to allow the user to interact with the machine during the boot process. In this context, this is typically referred to as the **input device**.

The OS must locate three devices that fall into these categories and **must also locate a device driver associated with each of the devices**. Remember that the OS hasn't been booted into memory yet and therefore hasn't loaded any loadable device drivers into memory from disk! This is the main reason that device ROMs exist. It contains a device driver that permits the device to be used during the boot process.

## ROM Detection

When the configuration software is configuring a PCI function, it determines if a function-specific ROM exists by checking to see if the designer has implemented an Expansion ROM Base Address Register (refer to Figure 20-2 on page 414).

As described in "Base Address Registers (BARs)" on page 378, the programmer writes all ones (with the exception of bit zero, to prevent the enabling of the ROM address decoder; see Figure 20-1 on page 413) to the Expansion ROM Base Address Register and then reads it back. If a value of zero is returned, then the register is not implemented and there isn't an expansion ROM associated with the device.

On the other hand, the ability to set any bits to ones indicates the presence of the Expansion ROM Base Address Register. This may or may not indicate the presence of a device ROM. Although the address decoder and a socket may exist for a device ROM, the socket may not be occupied at present. The programmer determines the presence of the device ROM by:

- assigning a base address to the register's Base Address field,
- enabling its decoder (by setting bit 0 in the register to one),
- setting the Memory Space bit in the function's Command register,
- and then attempting to read the first two locations from the ROM.

If the first two locations contain the ROM signature—AA55h—then the ROM is present.

Figure 20-1 on page 413 illustrates the format of the Expansion ROM Base Address Register. Assume that the register returns a value of FFFE0000h when read back after writing all ones to it. Bit 17 is the least-significant bit that was successfully changed to a one and has a binary-weighted value of 128K. This indicates that it is a 128KB ROM decoder and bits [24:17] within the Base Address field are writable. The programmer now writes a 32-bit start address into the register and sets bit zero to one to enable its ROM address decoder. In

addition to setting this bit to one, the programmer must also set the Memory Space bit in the function's configuration Command register to a one. The function's ROM address decoder is then enabled and the ROM (if present) can be accessed. The maximum ROM decoder size permitted by the specification is 16MB, dictating that bits [31:25] must be hardwired to one.

The programmer then performs a read from the first two locations of the ROM and checks for a return value of AA55h. If this pattern is not received, the ROM is not present. The programmer disables the ROM address decoder (by clearing bit zero of the Expansion ROM Base Address Register to zero). If AA55h is received, the ROM exists and a device driver code image must be copied into main memory and its initialization code must be executed. This topic is covered in the sections that follow.

*Figure 20-1: Expansion ROM Base Address Register Bit Assignment*

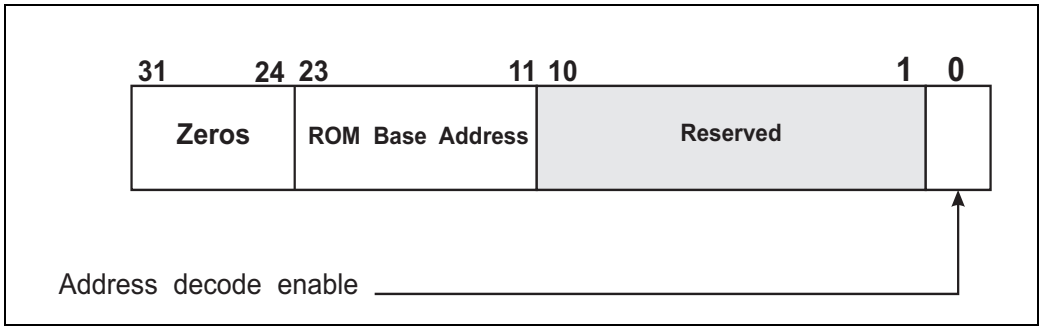
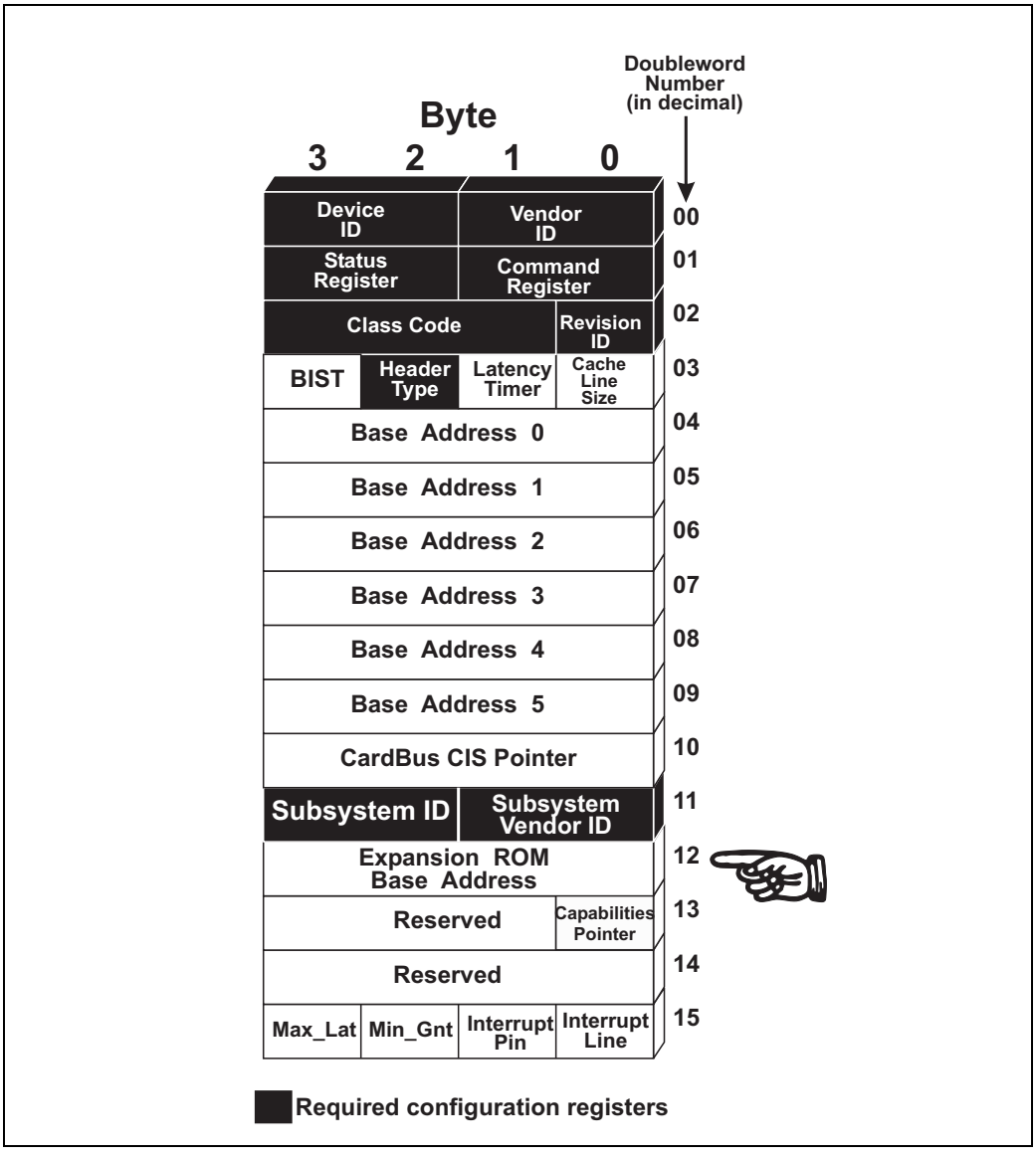


*Figure 20-2: Header Type Zero Configuration Register Format*



## ROM Shadowing Required

The PCI specification requires that device ROM code is never executed in place (i.e., from the ROM). It must be copied to main memory. This is referred to as "shadowing" the ROM code. This requirement exists for two reasons:

- ROM access time is typically quite slow, resulting in poor performance whenever the ROM code is fetched for execution.
- Once the initialization portion of the device driver in the ROM has been executed, it can be discarded and the code image in main memory can be shortened to include only the code necessary for run-time operation. The portion of main memory allocated to hold the initialization portion of the code can be freed up, allowing more efficient use of main memory.

Once the presence of the device ROM has been established (see the previous section), the configuration software must copy a code image into main memory and then disable the ROM address decoder (by clearing bit zero of the Expansion ROM Base Address Register to zero). In a non-PC environment, the area of memory the code image is copied to could be anywhere in the 4GB space. The specification for that environment may define a particular area.

In a PC environment, the ROM code image must be copied into main memory into the range of addresses historically associated with device ROMs: 000C0000h through 000DFFFFh. If the Class Code indicates that this is the VGA's device ROM, its code image must be copied into memory starting at location 000C0000h.

The next section defines the format of the information in the ROM and how the configuration software determines which code image (yes, there can be more than one device driver) to load into main memory.

## ROM Content

### Multiple Code Images

The PCI specification permits the inclusion of more than one code image in a PCI device ROM. Each code image would contain a copy of the device driver in a specific machine code, or in interpretive code (explained later). The configuration software can then scan through the images in the ROM and select the one

# 21 Add-in Cards and Connectors

## The Previous Chapter

The previous chapter provided a detailed description of device ROMs associated with PCI devices. This included the following topics:

- device ROM detection.
- internal code / data format.
- shadowing.
- initialization code execution.
- interrupt hooking.

## In This Chapter

This chapter provides an introduction to the PCI expansion card and connector definition. It covers card and connector types, 5V and 3.3V operability, shared slots, and pinout definition. For a detailed description of electrical and mechanical issues, refer to the latest version of the PCI specification (as of this printing, revision 2.2).

## The Next Chapter

The next chapter describes the Hot-Plug PCI capability defined by the revision 1.0 PCI Hot-Plug spec. A Hot-Plug capable system permits cards to be removed and installed without powering down the system.

# Add-In Connectors

## 32- and 64-bit Connectors

The PCI add-in card connector was derived from the Micro Channel connector. There are two basic types of connectors: the 32- and the 64-bit connector. A basic representation can be found in Figure 21-1 on page 438. Table 21-1 on page 439 illustrates the pinout of 32-bit and 64-bit cards (note that the 64-bit connector is a superset of the 32-bit connector). ***THE FOLLOWING PINOUT CHANGES WERE MADE IN THE 2.2 SPEC:*** 2.2

- *PIN A14 WAS RESERVED AND IS NOW DEFINED AS THE 3.3VAUX PIN (SEE "PME# AND 3.3VAUX" ON PAGE 449).*
- *PIN A19 WAS RESERVED AND IS NOW DEFINED AS THE PME# PIN (SEE "PME# AND 3.3VAUX" ON PAGE 449).*
- *PIN A40 WAS THE SDONE PIN AND IS NOW DEFINED AS A RESERVED PIN. THIS WAS THE SNOOP DONE SIGNAL, BUT ALL SUPPORT FOR CACHEABLE MEMORY AND CACHING ENTITIES ON THE PCI BUS HAS BEEN REMOVED.*
- *PIN A41 WAS THE SBO# PIN AND IS NOW DEFINED AS A RESERVED PIN. THIS WAS THE SNOOP BACKOFF SIGNAL, BUT ALL SUPPORT FOR CACHEABLE MEMORY AND CACHING ENTITIES ON THE PCI BUS HAS BEEN REMOVED.*

The table shows the card pinout for three types of cards: 5V, 3.3V, and Universal cards (the three card types are defined in "3.3V and 5V Connectors" on page 445). The system board designer must leave all reserved pins unconnected. The table illustrates the pinouts and keying for 3.3V and 5V connectors. In addition, a Universal card can be installed in either a 3.3V or a 5V connector. There is no such thing as a Universal connector, only Universal cards. Additional information regarding 3V, 5V and Universal cards can be found in this chapter in the section "3.3V and 5V Connectors" on page 445.

*Figure 21-1: 32- and 64-bit Connectors*

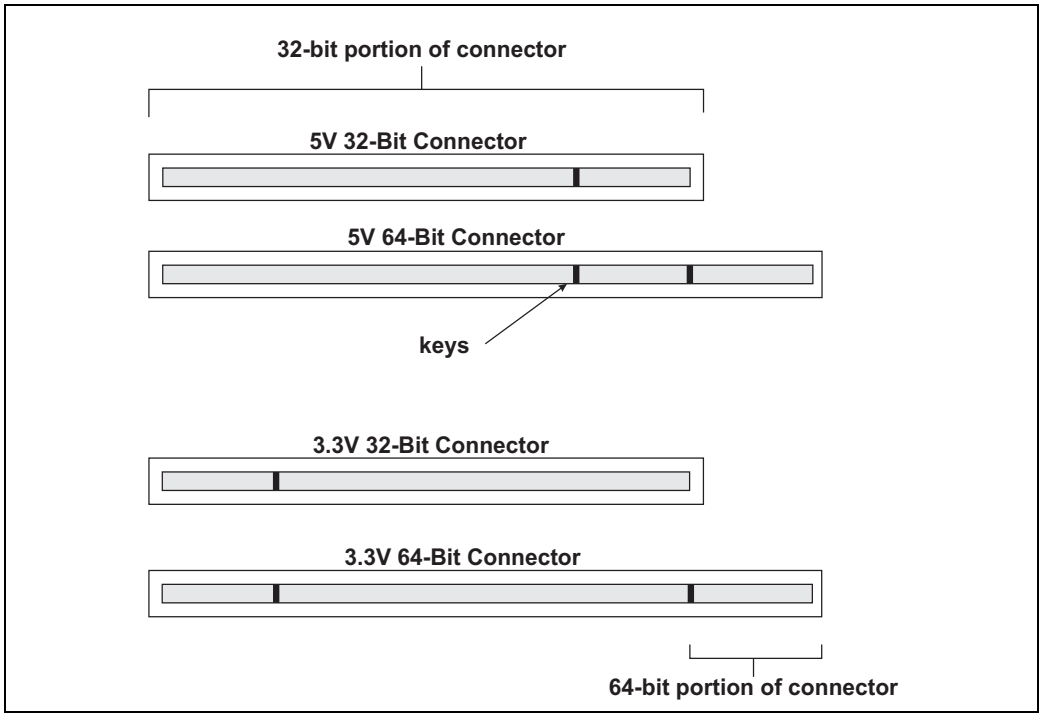


*Table 21-1: PCI Add-In Card Pinouts*

| Pin | 5V Card | | Universal Card | | 3.3V Card | | Comment | |
|---|---|---|---|---|---|---|---|---|
| | Side B | Side A | Side B | Side A | Side B | Side A | | |
| 1 | -12V | TRST# | -12V | TRST# | -12V | TRST# | 32-bit connector start | |
| 2 | TCK | +12V | TCK | +12V | TCK | +12V | | |
| 3 | Ground | TMS | Ground | TMS | Ground | TMS | | |
| 4 | TDO | TDI | TDO | TDI | TDO | TDI | | |
| 5 | +5V | +5V | +5V | +5V | +5V | +5V | | |
| 6 | +5V | INTA# | +5V | INTA# | +5V | INTA# | | |
| 7 | INTB# | INTC# | INTB# | INTC# | INTB# | INTC# | | |
| 8 | INTD# | +5V | INTD# | +5V | INTD# | +5V | | |
| 9 | PRSNT1# | Reserved | PRSNT1# | Reserved | PRSNT1# | Reserved | | |
| 10 | Reserved | +5V | Reserved | ***+Vi/o*** | Reserved | +3.3V | | |
| 11 | PRSNT2# | Reserved | PRSNT2# | Reserved | PRSNT2# | Reserved | | |
| 12 | Ground | Ground | Key | | | | 3.3V key | |
| 13 | Ground | Ground | | | | | | |
| 14 | Reserved | ***3.3VAUX*** | Reserved | ***3.3VAUX*** | Reserved | ***3.3VAUX*** | | 2.2 |
| 15 | Ground | RST# | Ground | RST# | Ground | RST# | | |
| 16 | CLK | +5V | CLK | ***+Vi/o*** | CLK | +3.3V | | |
| 17 | Ground | GNT# | Ground | GNT# | Ground | GNT# | | |
| 18 | REQ# | Ground | REQ# | Ground | REQ# | Ground | | |
| 19 | +5V | ***PME#*** | ***+Vi/o*** | ***PME#*** | +3.3V | ***PME#*** | | 2.2 |
| 20 | AD[31] | AD[30] | AD[31] | AD[30] | AD[31] | AD[30] | | |
| 21 | AD[29] | +3.3V | AD[29] | +3.3V | AD[29] | +3.3V | | |
| 22 | Ground | AD[28] | Ground | AD[28] | Ground | AD[28] | | |
| 23 | AD[27] | AD[26] | AD[27] | AD[26] | AD[27] | AD[26] | | |
| 24 | AD[25] | Ground | AD[25] | Ground | AD[25] | Ground | | |
| 25 | +3.3V | AD[24] | +3.3V | AD[24] | +3.3V | AD[24] | | |

*Table 21-1: PCI Add-In Card Pinouts (Continued)*

| Pin | 5V Card | | Universal Card | | 3.3V Card | | Comment |
|---|---|---|---|---|---|---|---|
| | Side B | Side A | Side B | Side A | Side B | Side A | |
| 26 | C/BE#[3] | IDSEL | C/BE#[3] | IDSEL | C/BE#[3] | IDSEL | |
| 27 | AD[23] | +3.3V | AD[23] | +3.3V | AD[23] | +3.3V | |
| 28 | Ground | AD[22] | Ground | AD[22] | Ground | AD[22] | |
| 29 | AD[21] | AD[20] | AD[21] | AD[20] | AD[21] | AD[20] | |
| 30 | AD[19] | Ground | AD[19] | Ground | AD[19] | Ground | |
| 31 | +3.3V | AD[18] | +3.3V | AD[18] | +3.3V | AD[18] | |
| 32 | AD[17] | AD[16] | AD[17] | AD[16] | AD[17] | AD[16] | |
| 33 | C/BE#[2] | +3.3V | C/BE#[2] | +3.3V | C/BE#[2] | +3.3V | |
| 34 | Ground | FRAME# | Ground | FRAME# | Ground | FRAME# | |
| 35 | IRDY# | Ground | IRDY# | Ground | IRDY# | Ground | |
| 36 | +3.3V | TRDY# | +3.3V | TRDY# | +3.3V | TRDY# | |
| 37 | DEVSEL# | Ground | DEVSEL# | Ground | DEVSEL# | Ground | |
| 38 | Ground | STOP# | Ground | STOP# | Ground | STOP# | |
| 39 | LOCK# | +3.3V | LOCK# | +3.3V | LOCK# | +3.3V | |
| 40 | PERR# | *RESERVED* | PERR# | *RESERVED* | PERR# | *RESERVED* | |
| 41 | +3.3V | *RESERVED* | +3.3V | *RESERVED* | +3.3V | *RESERVED* | |
| 42 | SERR# | Ground | SERR# | Ground | SERR# | Ground | |
| 43 | +3.3V | PAR | +3.3V | PAR | +3.3V | PAR | |
| 44 | C/BE[1]# | AD[15} | C/BE[1]# | AD[15} | C/BE[1]# | AD[15} | |
| 45 | AD[14] | +3.3V | AD[14] | +3.3V | AD[14] | +3.3V | |
| 46 | Ground | AD[13] | Ground | AD[13] | Ground | AD[13] | |
| 47 | AD[12] | AD[11] | AD[12] | AD[11] | AD[12] | AD[11] | |
| 48 | AD[10] | Ground | AD[10] | Ground | AD[10] | Ground | |
| 49 | Ground | AD[09] | Ground | AD[09] | Ground ** | AD[09] | ** see note |
| 50 | Keyway | | | | Ground | Ground | 5V key |
| 51 | | | | | Ground | Ground | 5V key |

2.2

*Table 21-1: PCI Add-In Card Pinouts (Continued)*

| Pin | 5V Card | | Universal Card | | 3.3V Card | | Comment |
|---|---|---|---|---|---|---|---|
| | Side B | Side A | Side B | Side A | Side B | Side A | |
| 52 | AD[08] | C/BE#[0] | AD[08] | C/BE#[0] | AD[08] | C/BE#[0] | |
| 53 | AD[07] | +3.3V | AD[07] | +3.3V | AD[07] | +3.3V | |
| 54 | +3.3V | AD[06] | +3.3V | AD[06] | +3.3V | AD[06] | |
| 55 | AD[05] | AD[04] | AD[05] | AD[04] | AD[05] | AD[04] | |
| 56 | AD[03] | Ground | AD[03] | Ground | AD[03] | Ground | |
| 57 | Ground | AD[02] | Ground | AD[02] | Ground | AD[02] | |
| 58 | AD[01] | AD[00] | AD[01] | AD[00] | AD[01] | AD[00] | |
| 59 | +5V | +5V | ***+Vi/o*** | ***+Vi/o*** | +3.3V | +3.3V | |
| 60 | ACK64# | REQ64# | ACK64# | REQ64# | ACK64# | REQ64# | |
| 61 | +5V | +5V | +5V | +5V | +5V | +5V | |
| 62 | +5V | +5V | +5V | +5V | +5V | +5V | 32-bit connector end |
| | keyway | | | | | | 64-bit spacer |
| | | | | | | | 64-bit spacer |
| 63 | Reserved | Ground | Reserved | Ground | Reserved | Ground | 64-bit start |
| 64 | Ground | C/BE#[7] | Ground | C/BE#[7] | Ground | C/BE#[7] | |
| 65 | C/BE#[6] | C/BE#[5] | C/BE#[6] | C/BE#[5] | C/BE#[6] | C/BE#[5] | |
| 66 | C/BE#[4] | +5V | C/BE#[4] | ***+Vi/o*** | C/BE#[4] | +3.3V | |
| 67 | Ground | PAR64 | Ground | PAR64 | Ground | PAR64 | |
| 68 | AD[63] | AD[62] | AD[63] | AD[62] | AD[63] | AD[62] | |
| 69 | AD[61] | Ground | AD[61] | Ground | AD[61] | Ground | |
| 70 | +5V | AD[60] | ***+Vi/o*** | AD[60] | +3.3V | AD[60] | |
| 71 | AD[59] | AD[58] | AD[59] | AD[58] | AD[59] | AD[58] | |
| 72 | AD[57] | Ground | AD[57] | Ground | AD[57] | Ground | |
| 73 | Ground | AD[56] | Ground | AD[56] | Ground | AD[56] | |
| 74 | AD[55] | AD[54] | AD[55] | AD[54] | AD[55] | AD[54] | |
| 75 | AD[53] | +5V | AD[53] | ***+Vi/o*** | AD[53] | +3.3V | |

# 22 Hot-Plug PCI

## The Previous Chapter

The previous chapter provided an introduction to the PCI expansion card and connector definition. It covered card and connector types, 5V and 3.3V operability, shared slots, and pinout definition.

## This Chapter

This chapter describes the Hot-Plug PCI capability defined by the revision 1.0 PCI Hot-Plug spec. A Hot-Plug capable system permits cards to be removed and installed without powering down the system.

## The Next Chapter

The next chapter provides a detailed description of PCI power management as defined in the revision 1.1 *PCI Bus PM Interface Specification*. In order to provide an overall context for this discussion, a description of the OnNow Initiative, ACPI (Advanced Configuration and Power Interface), and the involvement of the Windows OS is also provided.

## The Problem

In an environment where high-availability is desired, it would be a distinct advantage not to have to shut the system down before installing a new card or removing a card.

As originally designed, the PCI bus was not intended to support installation or removal of PCI cards while power is applied to the machine. This would result in probable damage to components, as well as a mightily confused OS.

## The Solution

The solution is defined in the revision 1.0 *PCI Hot-Plug spec* and gives the chipset the ability to handle the following software requests:

- Selectively **assert and deassert** the **PCI RST# signal** to an specific Hot-Plug PCI card connector. As stated in the PCI spec, the card must tri-state its output drivers within 40ns after the assertion of RST#.
- Selectively **isolate** a **card** from the logic on the system board.
- Selectively **remove or apply power** to a specific PCI card connector.
- Selectively **turn on or turn off** an **Attention Indicator** associated with a specific card connector to draw the users attention to the connector.

Hot-Plug PCI is basically a "**no surprises**" Hot-Plug methodology. In other words, the user is not permitted to install or remove a PCI card without first warning the software. The software then performs the necessary steps to prepare the card connector for the installation or removal of a card and finally indicates to the end user (via a visual indicator) when the installation or removal may be performed.

## No Changes To Adapter Cards

One of the major goals of the Hot-Plug PCI spec was that PCI add-in cards designed to the PCI spec require no changes in order to be Hot-Pluggable.

Changes are required, however, to the chipset, system board, OS and driver design.

## Software Elements

### General

Table 22-1 on page 457 describes the major software elements that must be modified to support Hot-Plug capability. Also refer to Figure 22-1 on page 459.

*Table 22-1: Introduction to Major Hot-Plug Software Elements*

| Software Element | Supplied by | Description |
|---|---|---|
| User Interface | OS vendor | An OS-supplied utility that permits the end-user to request that a card connector be turned off in order to remove a card or turned on to use a card that just been installed. |
| Hot-Plug Service | OS vendor | A service that processes requests (referred to as Hot-Plug Primitives) issued by the OS. This includes requests to:<br>• provide slot identifiers<br>• turn card On or Off<br>• turn Attention Indicator On or Off<br>• return current state of slot (On or Off)<br>The Hot-Plug Service interacts with the Hot-Plug System Driver to satisfy the requests. The interface (i.e., API) with the Hot-Plug System Driver is defined by the OS vendor, not by this spec. |
| Hot-Plug System Driver | System Board vendor | Receives requests (aka Hot-Plug Primitives) from the Hot-Plug Service within the OS. Interacts with the hardware Hot-Plug Controller to accomplish requests. The interface (i.e., API) with the Hot-Plug Service is defined by the OS vendor, not by this spec. A system board may incorporate more than one Hot-Plug Controller, each of which controls a subset of the overall slots in the machine. In this case, there would be one Hot-Plug System Driver for each Controller. |

*Table 22-1: Introduction to Major Hot-Plug Software Elements (Continued)*

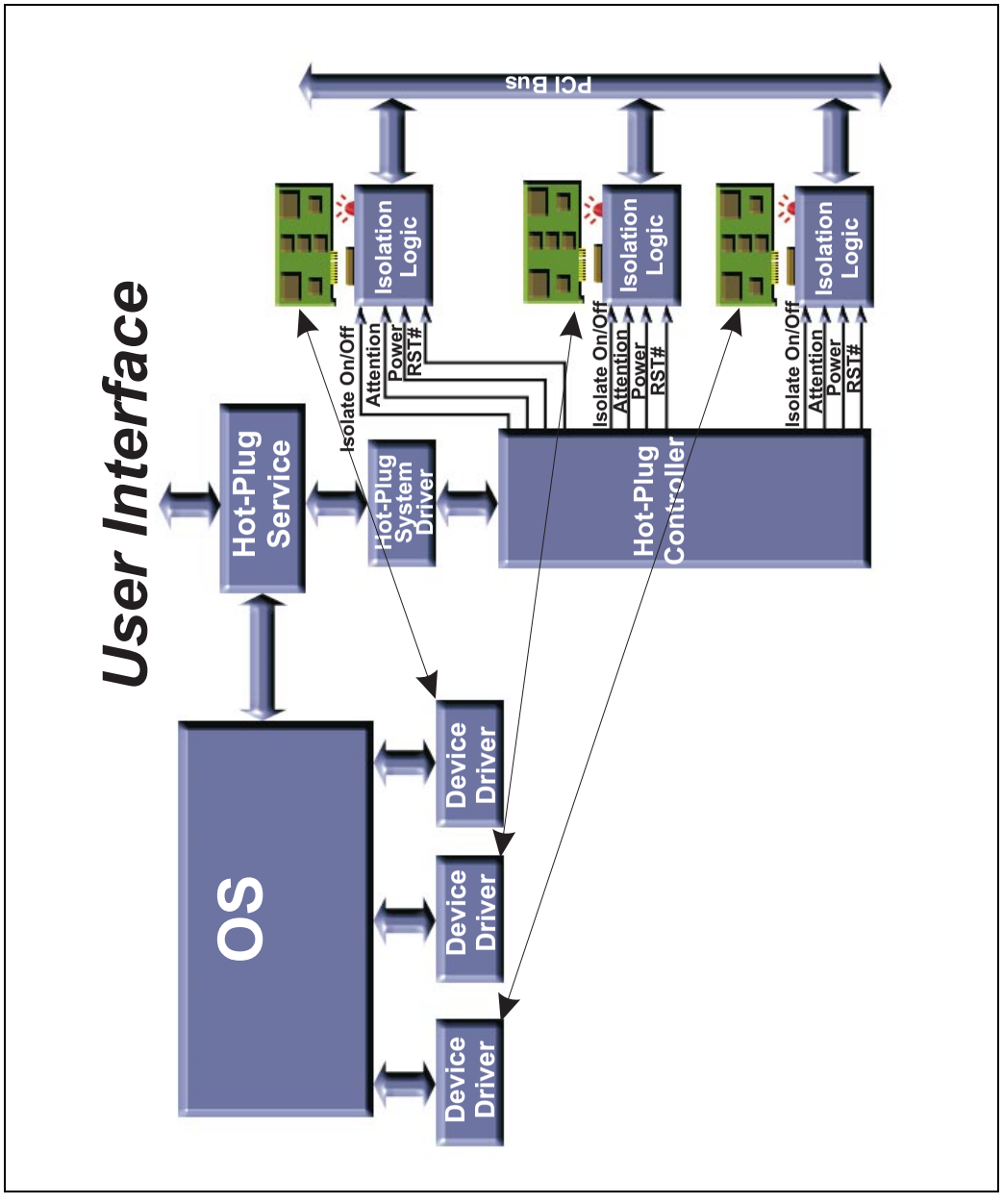
| Software Element | Supplied by | Description |
|---|---|---|
| Device Driver | Adapter card vendor | Some special, Hot-Plug-specific capabilities must be incorporated in a Hot-Plug capable device driver. This includes:<br>• support for the **Quiesce** command.<br>• optional implementation of the **Pause** command.<br>• Possible replacement for device's ROM code.<br>• Support for **Start** command or optional **Resume** command. |

## System Start Up

A Hot-Plug-capable system may be loaded with an OS that doesn't support Hot-Plug capability. In this case, although the system BIOS would contain some Hot-Plug-related software, the Hot-Plug Service and Hot-Plug System Driver would not be present. Assuming that the user doesn't attempt hot insertion or removal of a card, the system will operate as a standard, non-Hot-Plug system.

- The system startup firmware must ensure that all Attention Indicators are Off.
- The spec also states: "the Hot-Plug slots must be in a state that would be appropriate for loading non-Hot-Plug system software." The author is unclear as to what this means.

*Figure 22-1: Hot-Plug Hardware/Software Elements*



# 23 Power Management

## The Previous Chapter

The previous chapter described the Hot-Plug PCI capability defined by the revision 1.0 PCI Hot-Plug spec. A Hot-Plug capable system permits cards to be removed and installed without powering down the system.

## This Chapter

This chapter provides a detailed description of PCI power management as defined in the revision 1.1 *PCI Bus PM Interface Specification*. In order to provide an overall context for this discussion, a description of the OnNow Initiative, ACPI (Advanced Configuration and Power Interface), and the involvement of the Windows OS is also provided.

## The Next Chapter

The next chapter provides a detailed discussion of PCI-to-PCI bridge implementation. The information is drawn from the revision *1.1 PCI-to-PCI Bridge Architecture Specification*, dated December 18, 1998.

## Power Management Abbreviated "PM" In This Chapter

Throughout this chapter, the author has abbreviated Power Management as "PM" for brevity's sake.

## *PCI Bus PM Interface Specification*—But First...

The *PCI Bus PM Interface Specification* describes how to implement the optional PCI PM registers and signals. These registers and signals permit the OS to manage the power environment of PCI buses and the functions that reside on them.

Rather than immediately diving into a detailed nuts-and-bolts description of the *PCI Bus PM Interface Specification*, it's a good idea to begin by describing where it fits within the overall context of the OS and the system. Otherwise, this would just be a disconnected discussion of registers, bits, signals, etc. with no frame of reference.

# A Power Management Primer

## Basics of PC PM

The most popular OSs currently in use on PC-compatible machines are Windows 95/98/NT/2000. This section provides an overview of how the OS interacts with other major software and hardware elements to manage the power usage of individual devices and the system as a whole. Table 23-1 on page 480 introduces the major elements involved in this process and provides a very basic description of how they relate to each other. It should be noted that neither the PCI Power Management spec nor the ACPI spec (Advanced Configuration and Power Interface) dictate the policies that the OS uses to manage power. It does, however, define the registers (and some data structures) that are used to control the power usage of PCI functions.

*Table 23-1: Major Software/Hardware Elements Involved In PC PM*

| Element | Responsibility |
|---|---|
| OS | Directs the **overall system power management**.To accomplish this goal, the OS issues requests to the ACPI Driver, WDM (Windows Driver Model) device drivers, and to the PCI Bus Driver. Application programs that are power conservation-aware interact with the OS to accomplish device power management. |
| ACPI Driver | Manages configuration, power management, and thermal control of **devices embedded on the system board that do not adhere to any industry standard interface specification**. Examples would could be chipset-specific registers, system board-specific registers that control power planes and bus clocks (e.g., the PCI CLK), etc. The PM registers within PCI functions (embedded or otherwise) are defined by the PCI PM spec and are therefore not managed by the ACPI driver, but rather by the PCI Bus Driver (see entry in this table). |

*Table 23-1: Major Software/Hardware Elements Involved In PC PM (Continued)*

| Element | Responsibility |
|---|---|
| WDM Device Driver | The WDM driver is a **Class driver** that can work with any device that falls within the Class of devices that it was written to control. The fact that it's not written for a specific device from a specific vendor means that it doesn't have register and bit-level knowledge of the device's interface. When it needs to issue a command to or check the status of the device, it issues a request to the **Miniport** driver supplied by the vendor of the specific device.<br>The WDM also doesn't understand device characteristics that are peculiar to a specific bus implementation of that device type. As an example, the WDM doesn't understand a PCI device's configuration register set. It depends on the **PCI Bus Driver** to communicate with PCI configuration registers.<br>When it receives requests from the OS to control the power state of its PCI device, it passes the request to the PCI Bus Driver:<br>• When a request to power down its device is received from the OS, the WDM saves the contents of its associated PCI function's device-specific registers (in other words, it performs a context save) and then passes the request to the PCI Bus Driver to change the power state of the device.<br>• Conversely, when a request to repower the device is received from the OS, the WDM passes the request to the PCI Bus Driver to change the power state of the device. After the PCI Bus Driver has repowered the device, the WDM then restores the context to the PCI function's device-specific registers. |
| Miniport Driver | **Supplied by the vendor of a device**, it receives requests from the WDM Class driver and converts them into the proper series of accesses to the device's register set. |

*Table 23-1: Major Software/Hardware Elements Involved In PC PM (Continued)*

| Element | Responsibility |
|---|---|
| PCI Bus Driver | This driver is **generic to all PCI-compliant devices**. It **manages their power states and configuration registers**, but does not have knowledge of a PCI function's device-specific register set (that knowledge is possessed by the Miniport Driver that the WDM driver uses to communicate with the device's register set). It receives requests from the device's WDM to change the state of the device's power management logic:<br>• When a request is received to power down the device, the PCI Bus Driver is responsible for saving the context of the function's PCI configuration Header registers and any New Capability registers that the device implements. Using the device's PCI configuration Command register, it then disables the ability of the device to act as a bus master or to respond as the target of transactions. Finally, it writes to the PCI function's PM registers to change its state.<br>• Conversely, when the device must be repowered, the PCI Bus Driver writes to the PCI function's PM registers to change its state. It then restores the function's PCI configuration Header registers to their original state. |
| PCI PM registers within each PCI function's PCI configuration space. | **The location, format and usage of these registers is defined by the PCI PM spec**. The PCI Bus Driver understands this spec and therefore is the entity responsible for accessing a function's PM registers when requested to do so by the function's device driver (i.e., its WDM). |
| System Board power plane and bus clock control logic | The implementation and control of this logic is typically system board design-specific and is therefore **controlled by the ACPI Driver** (under the OS's direction). |

## OnNow Design Initiative Scheme Defines Overall PM

A whitepaper on Microsoft's website clearly defines the goals of the OnNow Design Initiative and the problems it addresses. The author has taken the liberty of reproducing the text from that paper verbatim in the two sections that follow: *Goals*, and *Current Platform Shortcomings*.

## Goals

The OnNow Design Initiative represents the overall guiding spirit behind the sought-after PC design. The following are the major goals as stated in an OnNow document:

- The PC is ready for use immediately when the user presses the On button.
- The PC is perceived to be off when not in use but is still capable of responding to wake-up events. Wake-up events might be triggered by a device receiving input such as a phone ringing, or by software that has requested the PC to wake up at some predetermined time.
- Software adjusts its behavior when the PC's power state changes. The operating system and applications work together intelligently to operate the PC to deliver effective power management in accordance with the user's current needs and expectations. For example, applications will not inadvertently keep the PC busy when it is not necessary, and instead will proactively participate in shutting down the PC to conserve energy and reduce noise.
- All devices participate in the device power management scheme, whether originally installed in the PC or added later by the user. Any new device can have its power state changed as system use dictates.

## Current Platform Shortcomings

**No Cooperation Among System Components.** Hardware, system BIOS, operating system, and applications do not cooperate, resulting in the various system components fighting for control of the hardware. This causes erratic behavior: disks spin up when they are not supposed to; screens come on unexpectedly.

**Add-on Components Do Not Participate In PM.** When a person buys a computer, the hardware in the system typically operates in an integrated power management scheme, and peripherals added by the user or reseller may not be power managed by the system. Traditional power management is no longer sufficient, because it typically does not deal outside the domain of the system board. To meet the OnNow goals, the entire system must function as an integrated power-managed environment, which requires a generalized solution in the operating system.

**Current PM Schemes Fail Purposes of OnNow Goals.** The power management schemes currently in use focus only on the system board and use only device access information to make decisions about when to power down a device. This approach causes two major problems:

# 24 PCI-to-PCI Bridge

## The Previous Chapter

The previous chapter provided a detailed description of PCI power management as defined in the revision 1.1 *PCI Bus PM Interface Specification*. In order to provide an overall context for this discussion, a description of the OnNow Initiative, ACPI (Advanced Configuration and Power Interface), and the involvement of the Windows OS was also provided.

## This Chapter

This chapter provides a detailed discussion of PCI-to-PCI bridge implementation. The information is drawn from the revision *1.1 PCI-to-PCI Bridge Architecture Specification*, dated December 18, 1998.

## The Next Chapter

The next chapter focuses on the ordering rules that govern the behavior of simple devices as well as the relationships of multiple transactions traversing a PCI-to-PCI bridge. It also describes how the rules prevent deadlocks from occurring.

## Scaleable Bus Architecture

A machine that incorporates one PCI bus has some obvious limitations. Some examples follow:

- If too many electrical loads (i.e., devices) are placed on a PCI bus, it ceases to function correctly.
- The devices that populate a particular PCI bus may not co-exist together too well. A master that requires a lot of bus time in order to achieve good performance must share the bus with other masters. Demands for bus time by these other masters may degrade the performance of this bus master subsystem.
- One PCI bus only supports a limited number of PCI expansion connectors (due to the electrical loading constraints mentioned earlier).

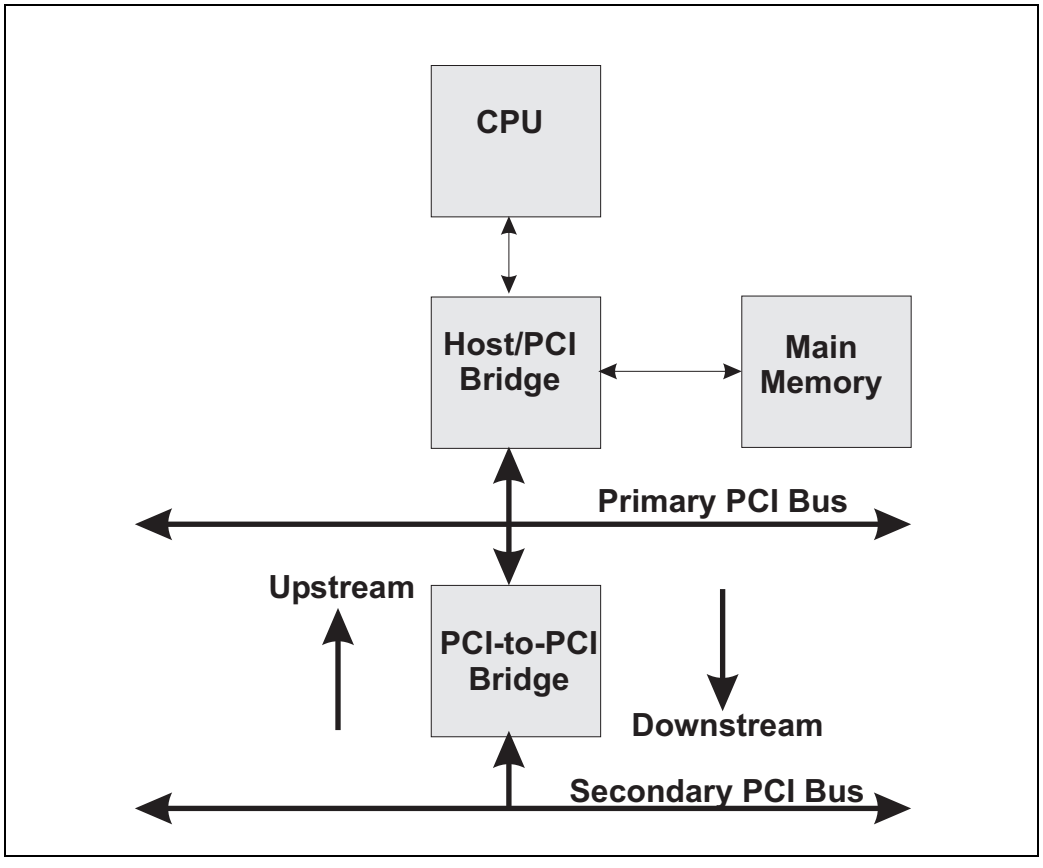
These problems could be solved by adding one or more additional PCI buses into the system and re-distributing the device population. How can a customer (or a system designer) add another PCI bus into the system? The *PCI-to-PCI Bridge Architecture Specification* provides a complete definition of a PCI-to-PCI bridge device. This device can either be embedded on a PCI bus or may be on an add-in card installed in a PCI expansion connector. The PCI-to-PCI bridge provides a bridge from one PCI bus to another, but it only places one electrical load on its host PCI bus. The new PCI bus can then support a number of additional devices and/or PCI expansion connectors. The electrical loading constraint is on a per bus basis, not a system basis. Of course, the power supply in the host system must be capable of supplying sufficient power for the load imposed by the additional devices residing on the new bus. The system designer could also include more than one host/PCI bridge.

## Terminology

Before proceeding, it's important to define some basic terms associated with PCI-to-PCI bridges. Each PCI-to-PCI bridge is connected to two PCI buses, referred to as its primary and secondary buses.

- **Downstream**. When a transaction is initiated and is passed through one or more PCI-to-PCI bridges flowing away from the host processor, it is said to be moving downstream.
- **Upstream**. When a transaction is initiated and is passed through one or more PCI-to-PCI bridges flowing towards the host processor, it is said to be moving upstream.
- **Primary bus**. PCI bus on the upstream side of a bridge.
- **Secondary bus**. PCI bus that resides on the downstream side of a PCI-to-PCI bridge.
- **Subordinate bus**. Highest-numbered PCI bus on the downstream side of the bridge.

*Figure 24-1: Basic Bridge Terminology*




## Example Systems

Figure 24-2 on page 545 and Figure 24-3 on page 546 illustrate two examples of systems with more than one PCI bus.

## Example One

The system in Figure 24-2 on page 545 has two PCI buses. Bus number one is subordinate to, or beneath, bus number zero. The PCI bus that resides directly on the other side of the host / PCI bridge is guaranteed present in every PCI sys-

tem and is always assigned a bus number of zero. Since the host/PCI bridge "knows" its bus number, the bridge designer may or may not implement a Bus Number register in the bridge. If the Bus Number register is present, the value it contains could be hardwired to zero, or reset could force it to zero.

It is a rule that each PCI-to-PCI bridge must implement three bus number registers in pre-defined locations within its configuration space. All three registers are read/writable and reset forces them to zero. They are assigned bus numbers during the configuration process. Those three registers are:

- **Primary Bus Number register**. Initialized by software with the number of the bridge's upstream PCI bus. The host/PCI bridge is only connected to one PCI bus, so it only implements a Bus Number register. The host/PCI bridge doesn't have to implement a Secondary Bus Number register (because it is irrelevant).
- **Secondary Bus Number register**. Initialized by software with the number of the bridge's downstream PCI bus.
- **Subordinate Bus Number register**. Initialized by software with the highest numbered PCI bus that exists on the secondary side. If the only bus on the bridge's downstream side is the bridge's secondary bus, then the Secondary and Subordinate Bus Number registers would be initialized with the number of the secondary bus.

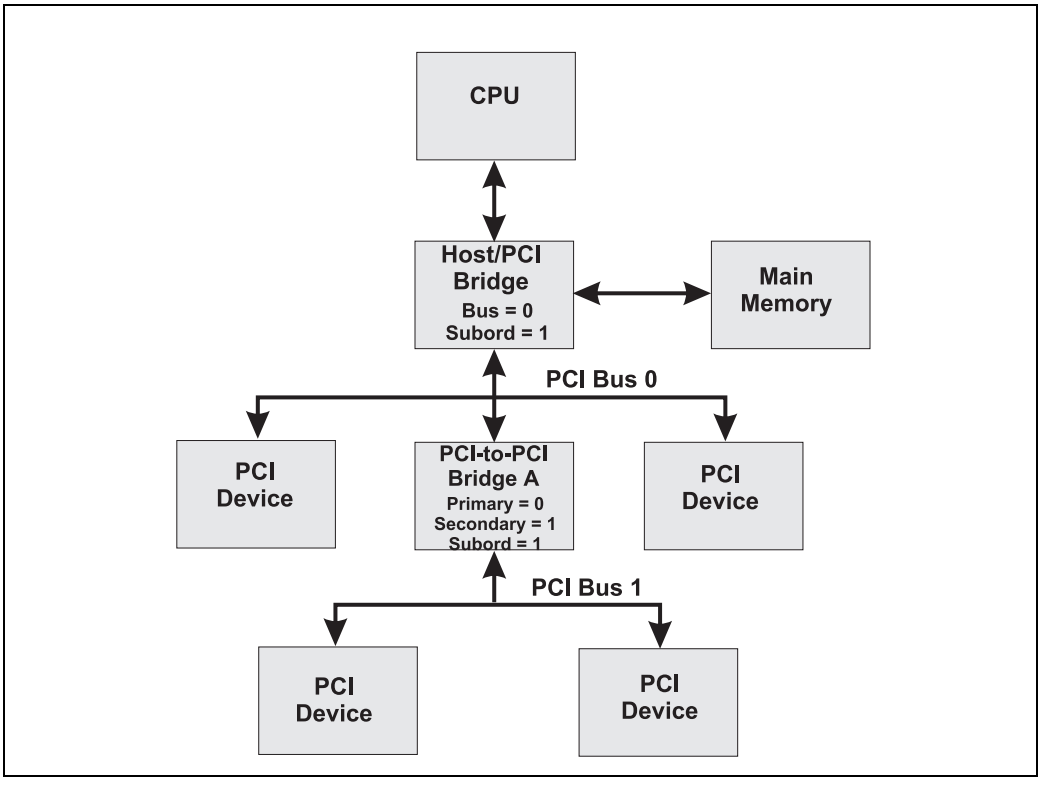
The host/PCI bridge only has to implement a Bus Number and a Subordinate Bus Number register. In Figure 24-2 on page 545, the host/PCI bridge's bus number registers are initialized (during configuration) as follows:

- **Bus Number = 0**. The host/PCI bridge's PCI bus is always numbered zero.
- **Subordinate Bus Number = 1**, the number of the highest numbered PCI bus that exists on the downstream side of the bridge. The host/PCI bridge must therefore pass through all configuration read and write transaction requests initiated by the host processor specifying a bus number in the range zero through one.

PCI-to-PCI bridge A has its bus registers initialized as follows:

- **Primary Bus = 0**. This is the number of the PCI bus closer to the host processor.
- **Secondary Bus = 1**. This is the number of the PCI bus on the downstream side of the bridge.
- **Subordinate Bus = 1**. This is the number of the highest-numbered bus that exists on the downstream side of the bridge.

*Figure 24-2: Example System One*




## Example Two

Figure 24-3 on page 546 has four PCI buses. During configuration, the host / PCI bridge's bus number registers are initialized as follows:

- **Bus Number = 0**. The host / PCI bridge's PCI bus is always numbered zero.
- **Subordinate Bus = 3**. This is the number of the highest-numbered bus that exists on the downstream side of the host / PCI bridge. The host / PCI bridge must therefore pass through all configuration read and write transaction requests initiated by the host processor specifying a target bus number in the range zero through three.

# 25 Transaction Ordering & Deadlocks

## The Previous Chapter

The previous chapter provided a detailed discussion of PCI-to-PCI bridge implementation. The information is drawn from the revision *1.1 PCI-to-PCI Bridge Architecture Specification*, dated December 18, 1998.

## This Chapter

This chapter focuses on the ordering rules that govern the behavior of simple devices as well as the relationships of multiple transactions traversing a PCI-to-PCI bridge. It also describes how the rules prevent deadlocks from occurring.

## The Next Chapter

The next chapter introduces the PCI BIOS specification, revision 2.1, dated August 26, 1994.

## Definition of Simple Device vs. a Bridge

Assume that a master incorporates an entity (e.g., a local processor) that performs writes to system memory over the PCI bus. The master can handle the internally-generated memory writes in one of two ways and the method it uses defines it (according to the 2.2 PCI spec) as a simple device or as a bridge:

## Simple Device

The 2.2 spec defines a simple device as **any device that does not require outbound write posting.** The internal logic is designed such that it would not be allowed to proceed with any other action (e.g., the update of a status register that can be read by masters external to the device) until the data has actually been written over the PCI bus to the target memory. Generally, devices that do not connect to local CPUs (in other words, devices other than Host/PCI bridges) are implemented as simple devices.

## Bridge

The internal logic is designed such that an outbound memory write is posted within a posted memory write buffer in the master's PCI interface and thus, to the internal logic, appears to have been completed. In reality, the write to the target memory location has not yet been performed on the PCI bus. Bridges that connect two buses together (e.g., a PCI-to-PCI bridge) typically exhibit this type of behavior.

# Simple Devices: Ordering Rules and Deadlocks

## Ordering Rules For Simple Devices

The **target and master state machines** in the PCI interface of a simple device **must be completely independent**. When acting as a target, it must not make the completion of any transaction that targets it (either posted or non-posted) contingent upon the prior completion of any other transaction when it is acting as a master. When acting as the target of a transaction, a simple device is only allowed to issue a retry when treating it as a Delayed Transaction, or for temporary conditions which are guaranteed to be resolved with time (i.e., a temporary in-bound posted memory write buffer full condition).

The required independence of target and master state machines in a simple device implies that a **simple device cannot internally post any outbound transactions**. Consider the following example scenario:

**STEP 1.** Assume that logic internal to the device performs a write to a memory location in another PCI device.

**STEP 2.** The write is posted in an outbound posted write buffer within the device's master state machine to be written to memory later.

**STEP 3.** Assuming (incorrectly) that the data has already been successfully written to the external memory, logic internal to the device then updates an internal status register to indicate that this is so (but it's wrong!).

**STEP 4.** Another PCI device, external to the PCI device under discussion, then performs a PCI read to read the status register in this device.

**STEP 5.** If the device, acting as the target of the read transaction, provides the status register contents to the other master, it is lying to the other guy (because it sends a status bit that says the data it thinks is in memory is fresh). In order to tell the other guy the truth, the device only has one option—issue a retry to the other guy and then perform the posted memory write.

This example highlights that the device's master and target state machines are dependent on each other, thereby defining it (according to the 2.2 specs's definitions) as a bridge and not as a simple device.

The simple device must wait until it completes the memory write transaction on the PCI bus (the target memory asserts **TRDY#**, or signals a Master Abort or a Target Abort) before proceeding internally (in the example, assuming that the write doesn't receive a Master or target Abort, updating the status register).

To increase PCI bus performance, **simple devices** are **strongly encouraged to post inbound memory write transactions** to allow memory writes targeting it to complete quickly. How the simple device orders inbound posted write data is design-dependent and outside the scope of the spec.

Simple devices do not support exclusive (i.e., locked) accesses (only bridges do) and do not use the LOCK# signal either a master or as a target. Refer to "Locking" on page 683 for a discussion of the use of LOCK# in bridge devices.

## Deadlocks Associated With Simple Devices

The following are two examples of deadlocks that could occur if devices make their target and master interfaces inter-dependent.

### Scenario One

**STEP 1.** Two devices, referred to as A and B, simultaneously start arbitrating for bus ownership to attempt IO writes to each other.

**STEP 2.** Device A is granted the bus first and initiates its IO write to device B (device B is the target of the transaction). Device B decodes the address/command and asserts DEVSEL#.

**STEP 3.** Assume that, when acting as a target, device B always terminates transactions that target it with Retry until its master state machine completes its outstanding requests (in this case, an IO write).

**STEP 4.** Device B is then granted the bus and initiates its IO write to device A.

**STEP 5.** If device A responds in the same manner that device B did (i.e., with a Retry), the system will deadlock.

### Scenario Two

As described in a later section ("Bridges: Ordering Rules and Deadlocks" on page 652), in certain cases a bridge is required to flush its posting buffer as a master before it completes a transaction as a target. As described in the following sequence, this can result in a deadlock:

**STEP 1.** A PCI-to-PCI bridge contains posted memory write data addressed to a downstream device (i.e., a device on its secondary side).

**STEP 2.** Before the bridge can acquire ownership of secondary bus to perform the write transaction, the downstream device that it intends to target initiates a read from main memory (in other words, the read has to cross the bridge from the secondary side to the primary side to get to memory).

**STEP 3.** To ensure that fresh read data is always received by any read that has to cross a bridge, the bridge ordering rules require that the bridge must flush its posted memory write buffers before the read is allowed to cross the bridge. The bridge must therefore Retry the downstream agent's read.

**STEP 4.** The bridge then performs the posted write to the downstream device on the secondary bus.

**STEP 5.** If the downstream device is designed in such a fashion that its target and master state machines are inter-dependent, it will issue a Retry in response to the bridge's posted write attempt and then re-issue its previously-attempted read.

**STEP 6.** The bus is deadlocked.

Since some PCI-to-PCI bridge devices designed to earlier versions of the PCI spec require that their posted memory write buffers be flushed before starting any non-posted transaction, the same deadlock could occur if the downstream device makes the acceptance of a posted write contingent on the prior completion of any non-posted transaction.

# Bridges: Ordering Rules and Deadlocks

## Introduction

When a bridge accepts a memory write into its posted memory write buffer, the master that initiated the write to memory considers the write completed and can initiate additional operations (i.e., PCI reads and writes) before the target memory location actually receives the write data. Any of these subsequent operations may end up completing before the memory write is finally consummated. The possible result: a read the programmer intended to occur after the write may happen before the data is actually written.

In order to prevent this from causing problems, many of the PCI ordering rules require that a bridge's posted memory write buffers be flushed before permitting subsequently-issued transactions to proceed. These same buffer flushing rules, however, can cause deadlocks. The remainder of the PCI transaction ordering rules prevent the system buses from deadlocking when posting buffers must be flushed.

## Bridge Manages Bi-Directional Traffic Flow

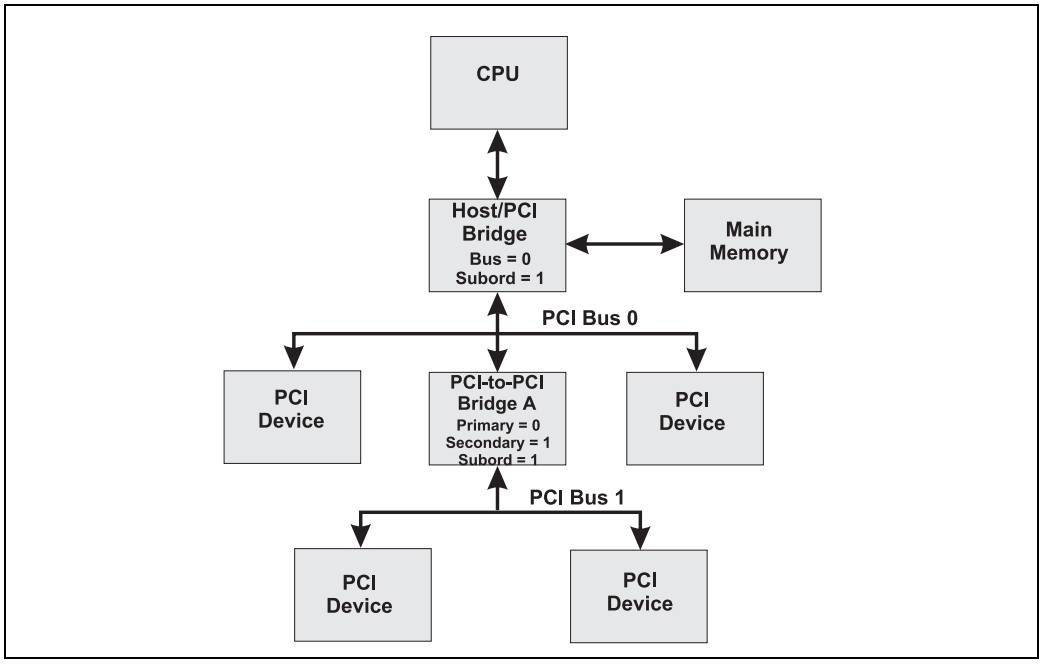
Refer to Figure 25-1 on page 653. A bridge manages traffic flow between two buses. In the figure, the processor-to-PCI bridge manages traffic flow between the processor bus and PCI Bus 0, while the PCI-to-PCI bridge manages traffic flow between PCI Bus 0 and PCI bus 1. Although the bridge ordering rules hold true for both types of bridges, this chapter focuses on the ordering of transactions that cross a PCI-to-PCI bridge. The typical PCI-to-PCI bridge incorporates two sets of posted memory write buffers:

- a posted memory write buffer that absorbs memory writes initiated on its primary side that target memory devices residing on the secondary side.
- a posted memory write buffer that absorbs memory writes initiated on its secondary side that target memory devices residing on the primary side.

In addition, the bridge handles transactions other than memory writes initiated on both sides as delayed transactions (see “Delayed Transactions” on page 86).

The PCI 2.2 spec incorporates a set of rules to govern the behavior of the bridge to ensure that operations appear to occur in the correct order (from the programmer’s perspective) and to prevent deadlocks from occurring.

*Figure 25-1: System With PCI-to-PCI Bridge*



# 26 The PCI BIOS

## The Previous Chapter

The previous chapter focused on the ordering rules that govern the behavior of simple devices as well as the relationships of multiple transactions traversing a PCI-to-PCI bridge. It also described how the rules prevent deadlocks from occurring.

## This Chapter

This chapter introduces the PCI BIOS specification, revision 2.1, dated August 26, 1994.

## The Next Chapter

The next chapter provides a detailed description of the PCI locking mechanism that permits an EISA bridge to lock main memory or the host/PCI bridge to lock an EISA memory target.

## Purpose of PCI BIOS

The OS (except for the platform-specific micro-kernel), applications programs and device drivers must not directly access the PCI configuration registers, interrupt routing logic (see "Interrupt Routing" on page 225), or the Special Cycle generation logic (see "Software Generation of Special Cycles" on page 329). The hardware methods utilized to implement these capabilities are platform-specific. Any software that directly accesses these mechanisms is therefore, by definition, platform-specific. This can lead to compatibility problems (i.e., the software works on some platforms but not on others).

Instead, the request should be issued to the PCI BIOS. The BIOS is platform-specific. It is implemented in firmware and possibly in the OS's Hardware Abstraction Layer (HAL). The PCI BIOS supplies the following services:

- Permits determination of configuration mechanism(s) supported by the PCI chipset (refer to “Intro to Configuration Mechanisms” on page 321).
- Permits determination of the chipset’s ability to generate the PCI Special Cycle transaction under software control and the mechanism(s) used to do so. For more information, refer to “Software Generation of Special Cycles” on page 329.
- Permits determination of the range of PCI buses present in system.
- Searches for all instances of a specific PCI device or a device that falls within a Class.
- Permits generation of the PCI Special Cycle transaction (if the chipset supports its generation under software control).
- Allows caller to get PCI interrupt routing options and then to assign an interrupt line to a device.
- Permits read and write of a device’s configuration registers.

# OS Environments Supported

## General

Different OSs have different operational characteristics (such as the method for defining the usage of system memory and the method utilized to call BIOS services). In systems based on the x86 processor family, the OS executing on a particular platform falls into one of the following three categories:

- Real-mode operating system (in other words, MS-DOS).
- 286 protected mode (God forbid!).
- 386 protected mode. There are two flavors of 386 protected mode:
  - the segmented model (once again, God forbid!).
  - and the flat model.

The PCI BIOS specification defines the following rules regarding the implementation of the PCI BIOS and the software that calls it:

**RULE 1.** The PCI BIOS must support all of the above-mentioned OS environments.

**RULE 2.** The BIOS must preserve all registers and flags with the exception of those used to return parameters.

**RULE 3.** Caller will be returned to with the state of Interrupt Flag bit in the EFLAGs register the same as it was on entry.

**RULE 4.** Interrupts will not be enabled during the execution of the BIOS function call.

**RULE 5.** The BIOS routines must be reentrant (i.e., they can be called from within themselves).

**RULE 6.** The OS must define a stack memory area at least 1KB in size for the BIOS.

**RULE 7.** The stack segment and code segment defined by the OS for the BIOS must have the same size (16- or 32-bit).

**RULE 8.** Protected mode OSs that call the BIOS using INT 1Ah must set the CS register to F000h.

**RULE 9.** The OS must ensure that the privilege level defined for the BIOS permits interrupt enable / disable and performance of IO instructions.

**RULE 10.** Implementers of the BIOS must assume that the CS for the BIOS defined by the OS is execute-only and that the DS is read-only.

## Real-Mode

Real-mode OSs, such as MS-DOS, are written to be executed on the 8088 processor. That processor is only capable of addressing up to 1MB of memory (00000h through FFFFFh). Using four 16-bit segment registers (CS, DS, SS, ES), the programmer defines four segments of memory, each with a fixed length of 64KB. When a program begins execution, each of the four segment registers is initialized with the upper four hex digits of the respective segment's start address in memory.

- The **code segment** contains the currently-executing program,
- the **data segment** defines the area of memory that contains the data the program operates upon,
- the **stack segment** defines the area of memory used to temporarily save values,
- and the **extra data segment** can be used to define another data segment associated with the currently-executing program.

MS-DOS makes calls to the BIOS by loading a subset of the processor's register set with request parameters and then executing a software interrupt instruction that specifies entry 1Ah in the interrupt table as containing the entry point to the BIOS. Upon execution of the INT 1Ah instruction, the processor pushes the address of the instruction that follows the INT 1Ah onto stack memory. Having saved this return address, the processor then reads the pointer from entry 1Ah in the interrupt table and starts executing at the indicated address. This is the entry point of the BIOS.

An alternative method for calling the BIOS is to make a call directly to the BIOS entry point at physical memory location 000FFE6Eh. Use of this method ensures that the caller doesn't have to worry about the 1Ah entry in the interrupt table having been "hooked" by someone else.

## 286 Protected Mode (16:16)

The BIOS specification refers to this as 16:16 mode because the 286 processor had 16-bit segment registers and the programmer specifies the address of an object in memory by defining the 16-bit offset of the object within the segment (code, data, stack or extra). Although the maximum size of each segment is still 64KB (as it is with the 8088 processor), the OS programmer can set the segment length to any value from one to 64KB in length. When operating in Real Mode, the 286 addresses memory just like the 8088 with the same fixed segment size of 64KB and the ability to only access locations within the first megabyte of memory space.

When operating in Protected Mode, however, the 286 processor addresses memory differently. Rather than containing the upper four hex digits of the segment's physical five-digit start address in memory, the value in the segment register is referred to as a Segment Selector. It points to an entry in a Segment Descriptor Table in memory that is built and maintained by the OS. Each entry in the Segment Descriptor Table contains eight bytes of information defining:

- the 24-bit start physical address of the segment in memory. In other words, the segment start address can be specified anywhere in the first 16MB of memory space.
- the length of the segment (from one byte through 64KB).
- the manner in which the program is permitted to access the segment of memory (read-only, execute-only, read / write, or not at all).

Some OSs (such as Windows 3.1 when operating in 286 mode) use the segment capability to assign separate code, data and stack segments within the 16MB total memory space accessible to each program. Whenever the OS performs a task switch, it must load the segment registers with the set of values defining the segments of memory "belonging" to the current application.

As in the Real Mode OS environment, the BIOS is called via execution of INT 1Ah or by directly calling the industry standard entry point of the BIOS (physical memory location 000FFE6Eh).

## 386 Protected Mode (16:32)

The 386 processor changed the maximum size of each segment from 64KB to 4GB in size. The 486, Pentium, and P6 family processors have the same maximum segment size as the 386. In addition to increasing the maximum segment

size to 4GB, the 386 also introduced a 32-bit register set, permitting the programmer to specify the 32-bit offset of an object within a segment. The segment registers are still 16-bits in size, however. Rather than containing the upper four hex digits of the segment's physical five-digit start address in memory, however, the value in the segment register is referred to as a Segment Selector (when the processor is operating in protected mode). It points to an entry in a Segment Descriptor Table in memory that is built and maintained by the OS. Each entry in the Segment Descriptor Table contains eight bytes of information defining:

- the 32-bit start physical address of the segment in memory. In other words, the base address of the segment can be specified anywhere within the overall 4GB of memory space.
- the length of the segment (from one byte through 4GB).
- the manner in which the program is permitted to access the segment of memory (read-only, execute-only, read / write, or not at all).

Some operating systems (such as Windows 3.1 when operating in 386 Enhanced Mode) use the segment capability to assign separate code, data and stack segments within the 4GB total memory space accessible to each program. Whenever the OS performs a task switch, it must load the segment registers with the set of values defining the segments of memory "belonging" to the current application. In the PCI BIOS specification, this is referred to as 16:32 mode because the 16-bit segment register defines (indirectly) the segment start address and the programmer can use a 32-bit value to specify the offset of the object anywhere in the 4GB of total memory space.

In a 32-bit OS environment, the BIOS is not called using INT 1Ah. In fact, if an applications program attempts to execute an INT instruction it results in a General Protection exception. Rather, the calling program executes a Far Call to the BIOS entry point. This implies that the entry point address is known. A subsequent section in this chapter defines how the BIOS entry point is discovered.

## Today's OSs Use Flat Mode (0:32)

A much simpler memory model is to set all of the segment registers to point to Segment Descriptors that define each segment as starting at physical memory location 00000000h each with a length of 4GB. This is referred as the Flat Memory Model. The BIOS specification refers to this as 0:32 mode because all segments start at location 00000000h and have a 32-bit length of FFFFFFFFh (4GB). Since separate segments aren't defined for each program, the OS has the responsibility of managing memory and making sure different programs don't play in each other's space. It accomplishes this using the Attribute bits in the Page Tables.

# 27 Locking

## The Previous Chapter

The previous chapter introduced the PCI BIOS specification, revision 2.1, dated August 26, 1994.

## This Chapter

This chapter provides a detailed description of the PCI locking mechanism that permits an EISA bridge to lock main memory or the host/PCI bridge to lock an EISA memory target.

## The Next Chapter

The next chapter describes the issues that differentiate CompactPCI from PCI. This includes mechanical, electrical and software-related issues. CompactPCI is presented as described in the 2.1 CompactPCI specification. PMC devices are also described.

## 2.2 Spec Redefined Lock Usage

2.2

*THIS CHAPTER HAS BEEN RENAMED AND REWRITTEN TO REFLECT A MAJOR CHANGE IN THE 2.2 SPEC. IN THE EARLIER VERSIONS OF THE SPEC IT WAS PERMISSIBLE FOR A PCI MASTER TO ISSUE A LOCKED TRANSACTION SERIES TO LOCK A PCI MEMORY TARGET. BUS MASTERS ARE NO LONGER ALLOWED TO ISSUE LOCKED TRANSACTIONS AND A PCI MEMORY TARGET MUST NO LONGER HONOR A REQUEST TO LOCK ITSELF.* These are the basic rules that define use of the locking mechanism:

**RULE 1.** **Only** the **host/PCI bridge** is now **permitted to initiate** a locked transaction series on behalf of a processor residing it.

**RULE 2.** A **PCI-to-PCI bridge** is **only** permitted to **pass** a **locked transaction from** its **primary to secondary side**. In other words, the bridge only passes through locked transactions that are moving outbound from the processor towards an expansion bus bridge further out in the hierarchy (e.g., an EISA bridge).

**RULE 3.** An **expansion bus bridge** (such as a PCI-to-EISA bridge) **acts as** the **target** of locked transactions and can optionally initiate them when targeting main memory behind the host/PCI bridge. For this reason, the host/PCI bridge must honor LOCK# as an input from the EISA bridge.

**RULE 4. LOCK#** is implemented as an sustained tri-state **input** pin **on** a PCI-to-PCI bridge's **primary** side and as a sustained tri-state **output** pin **on** its **secondary** side.

**RULE 5.** The first transaction of a locked transaction series must be a memory read (to read a memory semaphore).

## Scenarios That Require Locking

### General

The following sections describe the only circumstances under which the PCI locking mechanism may be used.

### EISA Master Initiates Locked Transaction Series Targeting Main Memory

If a PCI-to-EISA bridge is present in the system, there may be a master on the EISA bus that attempts locked transaction series with main memory. In this case, the EISA master may start a transaction on the EISA bus that targets main memory and it may assert the EISA LOCK# signal. In this case, the PCI-to-EISA bridge would have to initiate a PCI memory transaction with the PCI LOCK# signal asserted. This is permissible.

However, if the bridge is not on the PCI bus that is also connected to the host/PCI bridge, although the transaction will be successful in addressing main memory, it will not be successful in locking it. The transaction generated by the EISA bridge will make it through a PCI-to-PCI bridge, but not with LOCK# asserted. This is because PCI-to-PCI bridges only pass a lock through from the primary to the secondary side of the bridge, and not in the opposite direction.

In order to successfully lock main memory, the PCI-to-EISA bridge must be located directly on the same PCI bus that the host/PCI bridge is attached to. The EISA bridge's assertion of the PCI LOCK# signal is then directly visible to the host/PCI bridge.

## Processor Initiates Locked Transaction Series Targeting EISA Memory

It is possible that an EISA device driver uses a memory semaphore that resides in memory on an EISA card. If this is the case, when the processor executing the driver code initiates a locked Read / Modify / Write operation to read and update the semaphore, the host / PCI bridge must utilize the PCI locking mechanism (as defined later in this chapter) to lock the EISA bus when performing the accesses on the PCI and EISA buses.

## Possible Deadlock Scenario

Refer to Figure 27-1 on page 687. A deadlock can occur under the following circumstances:

**CLOCK 1.** The processor initiates an 8-byte read from the PCI memory-mapped IO target on Bus One.

**CLOCK 2.** To service the request, the host / PCI bridge initiates a PCI burst memory read transaction to perform a two data phase read from the 32-bit PCI memory-mapped IO target.

**CLOCK 3.** The memory-mapped IO target resides on the other side of a PCI-to-PCI bridge, so the PCI-to-PCI bridge acts as the target of the transaction. It initiates a burst memory read from the memory-mapped IO target on Bus One.

**CLOCK 4.** The memory-mapped IO target transfers the first dword to the PCI-to-PCI bridge, but then issues a disconnect to the bridge without transferring the second dword. The disconnect could have been because the target could not access the second dword within eight PCI clock cycles.

**CLOCK 5.** The PCI-to-PCI bridge in turn issues a disconnect to the host / PCI bridge.

**CLOCK 6.** Before the host / PCI bridge can re-initiate the memory read to get the second dword from the memory-mapped IO device, the PCI-to-PCI bridge accepts posted memory write data that must be written to main memory (which is behind the host / PCI bridge) from the bus master on its secondary side.

**CLOCK 7.** The host / PCI bridge then reinitiates its memory read transaction to fetch the second dword. The PCI-to-PCI bridge receives the request, memorizes it, issues a retry to the host / PCI bridge and treats it as a Delayed Read Request.

**CLOCK 8.** As per the transaction ordering rules, the PCI-to-PCI bridge cannot allow the memory read to cross onto the secondary side until it has flushed any previously-posted memory writes that are going towards its secondary side. This is to ensure that the read receives the correct data.

**CLOCK 9.** After any posted writes are completed on its secondary side, the bridge performs the memory read to obtain the second dword from the memory-mapped IO device on its secondary side. If the host/PCI bridge should retry the read request, it receives a retry.

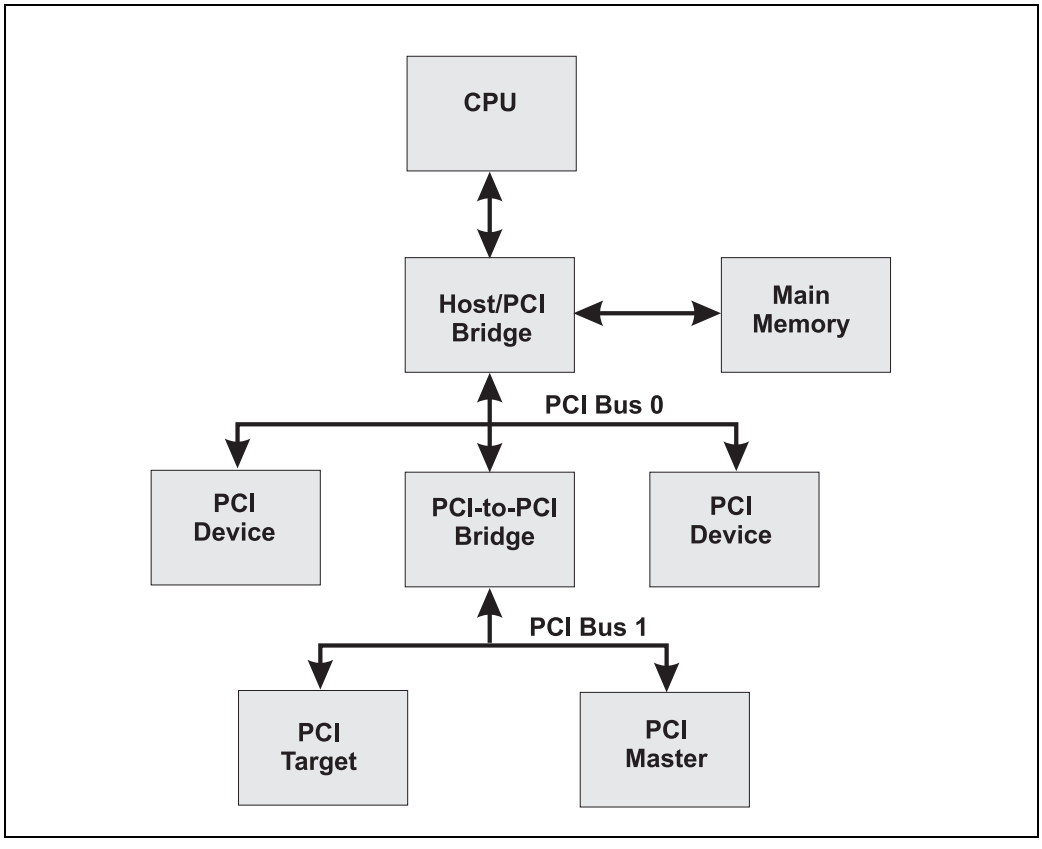
**CLOCK 10.** The PCI-to-PCI bridge now has the requested dword in a buffer. It cannot allow the host/PCI bridge's read to complete, however, until it has first performed any previously-posted memory writes to main memory. In other words, a read is not allowed to complete on its originating bus until all previously-posted writes moving in both directions have been completed.

**CLOCK 11.** When the PCI-to-PCI bridge attempts to perform the memory write to main memory, the host/PCI bridge issues a retry to it (because it will not accept any write data for main memory until the second half of its outstanding PCI memory read completes).

The result is a deadlock. Every time that the host/PCI bridge re-initiates its read request it receives a retry from the PCI-to-PCI bridge. In turn, the PCI-to-PCI bridge receives a retry each time that it attempts to dump its posted memory write buffer to memory.

This dilemma is **solved by having the host/PCI bridge start the initial memory read as a locked transaction** (in case the target resides behind one or more PCI-to-PCI bridges). *It is a rule that a PCI-to-PCI bridge must turn off write posting until a locked operation completes.* In the scenario just described, the bridge will not accept any posted write data on its secondary side until the entire read has completed and LOCK# has been deasserted.

*Figure 27-1: Possible Deadlock Scenario*




## PCI Solutions: Bus and Resource Locking

### LOCK# Signal

The PCI bus locking mechanism is implemented via the PCI LOCK# signal. There is only one LOCK# signal and only one master on a PCI bus can use it at a time (*and that master must be the host/PCI bridge or the secondary side interface of a PCI-to-PCI bridge*). This means that only one master may perform a locked transaction series during a given period of time. The LOCK# signal is a sustained tri-state signal. As with all PCI sustained tri-state signals, the system board

# 28 CompactPCI and PMC

## The Previous Chapter

The previous chapter provided a detailed description of the PCI locking mechanism that permits an EISA bridge to lock main memory or the host/PCI bridge to lock an EISA memory target.

## This Chapter

This chapter describes the issues that differentiate CompactPCI from PCI. This includes mechanical, electrical and software-related issues. CompactPCI is presented as described in the 2.1 CompactPCI specification. PMC devices are also described.

## Why CompactPCI?

The CompactPCI specification was developed by the PICMG (PCI Industrial Computer Manufacturer's Group) and defines a ruggedized version of PCI to be used in industrial and embedded applications. With regards to electrical, logical and software functionality, it is 100% compatible with the PCI standard. The cards are **rack mounted** and use standard Eurocard packaging. CompactPCI has the following features:

- Standard Eurocard dimensions (complies with the IEEE 1101.1 mechanical standard).
- HD (High-Density) 2mm pin-and-socket connectors (IEC approved and Bellcore qualified).
- Vertical card-orientation to ensure adequate cooling.
- Positive card retention mechanism.
- Optimized for maximum high shock and/or vibration environments.
- Front panel can implement front access IO connectors.
- User-defined IO pins defined on rear of the card.
- Standard chassis with multiple vendors.

- 100% compatible with standard PCI hardware and software components.
- Staged power pins to facilitate hot swap cards.
- Eight card slots per chassis (versus four in the typical PC platform).

## CompactPCI Cards are PCI-Compatible

CompactPCI cards must comply with all design rules specified for 33MHz PCI by the PCI 2.1 specification. The CompactPCI specification defines additional requirements and/or limitations that pertain to CompactPCI implementations.

## Basic PCI/CompactPCI Comparison

With respect to the PCI standard, CompactPCI exhibits the characteristics introduced in Table 28-1 on page 700.

*Table 28-1: CompactPCI versus Standard PCI*

| Item | Description |
|---|---|
| Compatibility | From both software and hardware perspectives, CompactPCI is currently 100% compatible with the 2.1 PCI spec. It has not yet been updated to reflect the 2.2 PCI spec. |
| Passive backplane environment | No active logic. |
| Connector | Shielded, 2mm-pitch, 5-row connectors defined by IEC 917 and IEC 1076-4-101. This gas-tight, high-density, pin-and-socket connector is available from multiple vendors (e.g., AMP, Framatome, Burndy, and ERNI). It exhibits low inductance and controlled impedance, crucial for PCI signaling. The connector's controlled impedance minimizes unwanted signal reflections, enabling CompactPCI backplanes to implement up to eight connectors, rather than upper limit of four imposed in normal PCI. It incorporates an external metal shield for RFI/EMI shielding purposes. Staged power and ground pins are included to facilitate hot swap implementations in the future. |

*Table 28-1: CompactPCI versus Standard PCI (Continued)*

| Item | Description |
|---|---|
| Number of cards | One system card and up to seven peripheral cards. |
| Card type | Two Eurocard form factors: small (3U); and large (6U). |
| User-defined IO signal pins | Permits passage of user-defined IO signals to / from the back plane through edge-connectors. |
| Hot swap capable | In the future. |
| Signal set is superset of standard PCI | Some signals added for non-PCI functions. |
| Responsible SIG | PICMG (PCI Industrial Computer Manufacturer's Group). |
| Modularity | Ruggedized backplane, rack-mount environment. |
| Signal termination | For increased signal integrity and more slots than standard PCI. |
| Legacy IDE support | Two edge-triggered interrupt pins defined to support primary and secondary legacy IDE controllers. |

# Basic Definitions

## Standard PCI Environment

In a standard PCI implementation, the following elements are typically all embedded on the system board:

- the host processor(s).
- main memory.
- the host / PCI bridge.
- PCI bus arbiter.
- system interrupt controller.
- embedded PCI devices (e.g., SCSI controller, Ethernet controller, video adapter).
- PCI card-edge connectors.
- the PCI / ISA bridge.

The end-user adds functionality to the system by installing additional PCI adapter cards into the card-edge connectors. These cards can be strictly targets, or may have both target and bus master capability. In order to upgrade base system characteristics such as the host processor type or the host/PCI bridge, the end-user is forced to swap out the system board. This is a major cost item. In addition, the user must also remove the PCI adapter cards from the old system board and install them in the new one.

CompactPCI is implemented in a more modular fashion, consisting of a passive-backplane with a system board slot and up to seven peripheral slots to install PCI adapter cards. The sections that follow describe the backplane, system card and peripheral cards.

## Passive Backplane

A typical passive backplane is pictured in Figure 28-1 on page 704 and contains no active logic. It provides the elements listed in Table 28-2 on page 702.

*Table 28-2: Passive Backplane Elements*

| Element | Description |
|---|---|
| System slot | One system slot to accept a system card. Implemented with between one and five male connectors numbered P1-through-P5. |
| Peripheral slots | Up to seven peripheral slots to accept cards that act strictly as targets or as both a target and bus master. Implemented with between one and five male connectors numbered P1-through-P5. |
| Staged pins | Staged pins to facilitate hot swap of CompactPCI cards. |
| Connector keying | Appropriate connector keying for either a 5Vdc or 3.3Vdc signaling environment. |
| 32-bit PCI bus | The 32-bit PCI bus interconnects the system slot and peripheral slots. |
| PCI clock distribution | PCI clock distribution from the system slot to the peripheral slots. |

*Table 28-2: Passive Backplane Elements (Continued)*

| Element | Description |
|---|---|
| Interrupt signals | Interrupt trace distribution to the system slot from the peripheral slots. |
| REQ#/GNT# signal pairs | REQ#/GNT# signal pairs between the arbiter on the system card and the peripheral slots. |
| Rear-panel IO connectors | Optionally, through-the-backplane rear-panel IO connectors that route non-PCI signals/buses through the backplane to rear-panel IO connectors. |
| Rear-panel IO transition boards | Rear-panel IO transition boards may be installed in a rear rack and connected to the non-PCI signals. |
| Modular power supply connector | Optionally, a modular power supply connector for rack installation of a modular power supply. |
| 64-bit PCI extension signals | Optionally, the 64-bit PCI extension signals (AD[63:32], C/BE[7:4]#, PAR64). |
| Geographical addressing pins | In a 64-bit implementation, a set of pins that a 64-bit card (and, optionally, a 32-bit card) may interrogate to determine which physical slot it is installed in. |

# *Appendix A*

# *Glossary of Terms*

**Access Latency**. The amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction.

**AD Bus**. The PCI address/data bus carries address information during the address phase of a transaction and data during each data phase.

**Address Ordering**. During PCI burst memory transfers, the initiator must indicate whether the addressing sequence will be sequential (also referred to as linear) or will use cacheline wrap ordering of addresses. The initiator uses the state of AD[1:0] to indicate the addressing order. During I/O accesses, there is no explicit or implicit address ordering. It is the responsibility of the programmer to understand the I/O addressing characteristic of the target device.

**Address Phase**. During the first clock period of a PCI transaction, the initiator outputs the start address and the PCI command. This period is referred to as the address phase of the transaction. When 64-bit addressing is used, there are two address phases.

**Agents**. Each PCI device, whether a bus master (initiator) or a target is referred to as a PCI agent.

**Arbiter.** The arbiter is the device that evaluates the pending requests for access to the bus and grants the bus to a bus master based on a system-specific algorithm.

**Arbitration Latency**. The period of time from the bus master's assertion of REQ# until the bus arbiter asserts the bus master's GNT#. This period is a function of the arbitration algorithm, the master's priority and system utilization.

**Atomic Operation**. A series of two or more accesses to a device by the same initiator without intervening accesses by other bus masters.

**Base Address Registers**. Device configuration registers that define the start address, length and type of memory space required by a device. The type of space required will be either memory or I/O. The value written to this register during device configuration will program its memory or I/O address decoder to detect accesses within the indicated range.

**BIST**. Some integrated devices (such as the i486 microprocessor) implement a built-in self-test that can be invoked by external logic during system start up.

**Bridge**. The device that provides the bridge between two independent buses. Examples would be the bridge between the host processor bus and the PCI bus, the bridge between the PCI bus and a standard expansion bus (such as the ISA bus) and the bridge between two PCI buses.

**Bus Access Latency**. Defined as the amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction. In other words, it is the sum of arbitration, bus acquisition and target latency.

**Bus Acquisition Latency**. Defined as the period time from the reception of GNT# by the requesting bus master until the current bus master surrenders the bus and the requesting bus master can initiate its transaction by asserting FRAME#. The duration of this period is a function of how long the current bus master's transaction-in-progress will take to complete.

**Bus Concurrency**. Separate transfers occurring simultaneously on two or more separate buses. An example would be an EISA bus master transferring data to or from another EISA device while the host processor is transferring data to or from system memory.

**Bus Idle State**. A transaction is not currently in progress on the bus. On the PCI bus, this state is signalled when FRAME# and IRDY# are both deasserted.

**Bus Lock**. Gives a bus master sole access to the bus while it performs a series of two or more transfers. This can be implemented on the PCI bus, but the preferred method is resource locking. The EISA bus implements bus locking.

**Bus Master**. A device capable of initiating a data transfer with another device.

**Bus Parking**. An arbiter may grant the buses to a bus master when the bus is idle and no bus masters are generating a request for the bus. If the bus master that the bus is parked on subsequently issues a request for the bus, it has immediate access to the bus.

**Byte Enable**. I486, Pentium™ or PCI Bus control signal that indicates that a particular data path will be used during a transfer. Indirectly, the byte enable signal also indicates what byte within an addressed doubleword (or quadword, during 64-bit transfers) is being addressed.

**Cache**. A relatively small amount of high-speed Static RAM (SRAM) that is used to keep copies of information recently read from system DRAM memory. The cache controller maintains a directory that tracks the information currently resident within the cache. If the host processor should request any of the infor-

mation currently resident in the cache, it will be returned to the processor quickly (due to the fast access time of the SRAM).

**Cache Controller**. See the definition of **Cache**.

**Cache Line Fill**. When a processor's internal cache, or its external second level cache has a miss on a read attempt by the processor, it will read a fixed amount (referred to as a line) of information from the external cache or system DRAM memory and record it in the cache. This is referred to as a cache line fill. The size of a line of information is cache controller design dependent.

**Cache Line Size**. See the definition of **Cache Line Fill**.

**CacheLine Wrap Mode**. At the start of each data phase of the burst read, the memory target increments the doubleword address in its address counter. When the end of the cache line is encountered and assuming that the transfer did not start at the first doubleword of the cache line, the target wraps to start address of the cacheline and continues incrementing the address in each data phase until the entire cache line has been transferred. If the burst continues past the point where the entire cache line has been transferred, the target starts the transfer of the next cache line at the same address that the transfer of the previous line started at.

**CAS Before RAS Refresh, or CBR Refresh**. Some DRAMs incorporate their own row counters to be used for DRAM refresh. The external DRAM refresh logic has only to activate the DRAM's CAS line and then its RAS line. The DRAM will automatically increment its internal row counter and refresh (recharge) the next row of storage.

**CBR Refresh**. See the definition of **CAS Before RAS Refresh.**

**Central Resource Functions**. Functions that are essential to operation of the PCI bus. Examples would be the PCI bus arbiter and "keeper" pullup resistors that return PCI control signals to their quiescent state or maintain them at the quiescent state once driven there by a PCI agent.

**Claiming the Transaction**. An initiator starts a PCI transaction by placing the target device's address on the AD bus and the command on the C/BE bus. All PCI targets latch the address on the next rising-edge of the PCI clock and begin to decode the address to determined if they are being addressed. The target that recognizes the address will "claim" the transaction by asserting DEVSEL#.

**Class Code**. Identifies the generic function of the device (for example, a display device) and, in some cases, a register-specific programming interface (such as the VGA register set). The upper byte defines a basic class type, the middle byte a sub-class within the basic class, and the lower byte may define the programming interface.

**Coherency**. If the information resident in a cache accurately reflects the original information in DRAM memory, the cache is said to be coherent or consistent.

**Commands**. During the address phase of a PCI transaction, the initiator broadcasts a command (such as the memory read command) on the C/BE bus.

**Compatibility Hole**. The DOS compatibility hole is defined as the memory address range from 80000h - FFFFFh. Depending on the function implemented within any of these memory address ranges, the area of memory will have to be defined in one of the following ways: Read-Only, Write-Only, Read/Writable, Inaccessible.

**Concurrent Bus Operation**. See the definition of **Bus Concurrency**.

**Configuration Access**. A PCI transaction to read or write the contents of one of a PCI device's configuration registers.

**Configuration Address Space**. x86 processors possess the ability to address two distinct address spaces: I/O and memory. The PCI bus uses I/O and memory accesses to access I/O and memory devices, respectively. In addition, a third access type, the configuration access, is used to access the configuration registers that must be implemented in all PCI devices.

**Configuration CMOS RAM**. The information used to configure devices each time an ISA, EISA or Micro Channel™ machine is powered up is stored in battery backed-up CMOS RAM.

**Configuration Header Region**. Each functional PCI device possesses a block of two hundred and fifty-six configuration addresses reserved for implementation of its configuration registers. The format, or usage, of the first sixty-four locations is predefined by the PCI specification. This area is referred to as the device's configuration header region.

**Consistency**. See the definition of **Coherency**.

**Data Packets**. In order to improve throughput, a PCI bridge may consolidate a series of single memory reads or writes into a single PCI memory burst transfer.