Extended Capabilities Port: Specifications

Revision 1.06
July 14, 1993

Cover Letter

Important Note This may not be the first time you have received the ECP Specifications. From time to time, Microsoft may make some modifications to the documentation and/or software and we want to ensure that you have the latest update to the materials. If this is not the first time you have received this kit, please read the section titled "Corrections to Previous Versions (Revisions)" very carefully for any changes that may apply to you.

Environments such as the Microsoft® Windows™ operating system are making it easier for anybody to use computers, and the decision to buy computers, peripherals, and software is easier to make now that Windows provides ease of use to the personal computer (PC) environment.

Communications with peripherals is the "Achilles heel" of the industry. Low bandwidth and lack of bidirectionality have prevented suppliers of peripherals such as printers, scanners, fax/modem cards, and network adapters from introducing innovative solutions. The serial port provides bidirectionality but does not offer enough bandwidth. The standard parallel port (Centronics®) offers a higher but limited bandwidth and no bidirectionality. Centronics also requires a lot of assistance from the CPU, making it inadequate for environments such as Windows in which multiple applications are running concurrently.

The extended capabilities port (ECP) is the answer to peripheral communications problems in the PC environment. It is a fast, bidirectional parallel interface that is backwards-compatible with the existing PC standard parallel port configuration—and it uses the existing parallel connectors and cables. ECP has been jointly developed by Microsoft and Hewlett-Packard with the hopes of making it a widely adopted standard. Several chip vendors are working on designing support for ECP into their next generation I/O chips. On the system software side, Microsoft is building support for the full capabilities of ECP (bidirectionality, enhanced bandwidth, enhanced protocols) into the next generation of its Windows operating environments. A new set of application programming interfaces (APIs) will soon be available to independent software and hardware vendors (ISVs and IHVs) to enable the writing of device drivers and applications programs that take advantage of ECP.

Enclosed you will find the ECP Specification, which includes the following documents:

- **ECP cover letter** (this document).
- **Extended Capabilities Port Protocol and ISA Interface Standard.** Covers the ECP signal protocols, IEEE P1284 issues, and the ISA implementation specifications. Includes an errata sheet.
- **ECP Driver Hardware Notes.** Covers issues software/hardware designers must be aware of.
- **ECP Compliance Test Functional Specification.** Describes the setup and use of the ECP compliance test.

The ECP Specifications document describes in detail the ECP ISA implementation requirements, so you do not actually need the ECP Adaptation Kit to develop an ECP ASIC or to make ECP part of a larger chip solution. We feel, however, that having it may shorten the time required for your ECP implementation cycle.

For additional information, please contact Doug Hogarth at 206/936-3002. Doug is a Technical Evangelist in the Windows Developer Relations Group. If you are signing a contract, please send it back to:

Doug Hogarth

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Implementing ECP accrues many obvious and substantial benefits to users. Microsoft is very excited about what we feel is the most significant event in PC/peripheral communications since the introduction of the PC, and we hope to have your support and cooperation in making this a public standard for the benefit of PC users everywhere.

Regards,

Microsoft Corporation
Windows Developer Relations

Extended Capabilities Port Protocol and ISA Interface Standard

Revision: 1.14
July 14, 1993

Document History

Revision	Date	Action	Author(s)
0.1	5-8-92	Described standard ISA interface for ECP ("Zippy").	Dave Voth (MS)
0.2	5-18-92	Wrote HP's "High Speed Parallel Port Spec (Zippy)" as an addendum to the BOISE / IEEE P1284 specs.	Rich Taylor (HP)
0.3	7-23-92	Edited and formatted.	Frank Williams
0.4	9-11-92	Edited and formatted.	Ray Styles
0.5	9-18-92	Incorporated several changes.	Ray Styles
1.0	10-13-92	Finalized the specification.	Ray Styles
1.01	10-26-92	Corrected mode switching section and switched to ECP from High Speed.	Joseph Mouhanna
1.01	10-26-92	Did same High Speed-to-ECP changes for Figures 2 and 3.	Ray Styles
1.02	11-03-92	Modified timing for IEEE, removed fast bit and added compress bit in its place.	Dave Voth
1.03	11-05-92	New timing diagrams.	Joseph Mouhanna
1.04	11-06-92	Minor changes.	Dave Voth
1.05	11-09-92	Minor changes.	Joseph Mouhanna
1.06	11-12-92	Minor changes for IEEE.	David Voth
1.07	12-2-92	Minor changes.	Joseph Mouhanna
1.08	12-18-92	Minor bug fixes (affects pages 16, 19, and 33 of rev 1.07).	Joseph Mouhanna
1.09	01-07-93	Minor fixes (affects pages 28 and 32 of rev 1.07 and 1.08).	Joseph Mouhanna
1.10	02-04-93	Fixes to Protocol Description (in section 2), paragraphs for Reverse to Forward Phase (page 19) and Valid Termination (Page 20). Fixed ECP ID in Figure 2 (ECP mode timing 1 of 2, page 21). Fixed Table 8, page 28 (ecpAFifo and cFifo). Added Appendix A.	Joseph Mouhanna
1.11	02-10-93	Minor fixes to page 23 (figure 2, events 0 and 34).	Joseph Mouhanna

1.12	4-28-93	Fixed forward/reverse wording (pg. 11), P1284 termination (pg. 20). Minor fixes to busy signal definition (pg. 26). Fixed error in compress bit in cnfgB (pg. 31); added Appendix B.	David Voth
1.13	6-05-93	Added bit to cnfgA (page 31), **ackIntEn** (page 35). Fixed ISA Level interrupt design in Appendix B.	David Voth
1.14	7-14-93	nPeriphRequest (nFault), page 10; nReverseRequest (nInit), page 11; Table 5 (Signal Timing), page 14; Operating Phases Diagram, page 15; Aborting the Forward Data Transfer Phase, page 19;Forward to Reverse Phase; page 19; Reverse Data Transfer Phase, page 19; Timing Diagrams on pages 22, 23, and 24 (new diagram on page 24); Event 26 on page 25; Events 40, 48, 72, 73, 74, 75 on page 26; Table 7: dRq, page 29; Table 11, page 31; cFIFO, page 32; Table 12, page 33; Table 14, page 36; DMA, page 39; added Appendix C.	David Voth

Section 1: Introduction

Document Purpose

Design specifications from both Microsoft and Hewlett-Packard have been combined in this document to describe both the extended capabilities port (ECP) protocol, and an industry standard architecture (ISA) implementation.

Other Documents

The reader should be familiar with the *Standard P1284* by IEEE, a parent document of this document.

Scope

This document defines an ISA standard for the implementation of all ECP parallel port ISA devices. ECP is a joint Hewlett-Packard/Microsoft design and development effort. ECP is an enhancement to the high-speed IEEE P1284 and "BOISE" parallel port specifications.

Vocabulary

The following terms are used in this document:

assert
When a signal asserts, it transitions to a "true" state. When a signal deasserts, it transitions to a "false" state.

forward
Host-to-Peripheral communication.

reverse
Peripheral-to-Host communication.

PWord
A port word, equal in size to the width of the ISA interface. Typically, this can be 8 or 16 bits.

1
A high level.

0
A low level.

Each row in the following table consists of terms that are equivalent.

PeriphClk	nAck	
HostAck	nAutoFd	
PeriphAck	Busy	
nPeriphRequest	nFault	
nReverseRequest	nInit	
nAckReverse	PError	
Xflag	Select	
ECPmode	BOISEmode	nSelectIn
HostClk	nStrobe	

Section 2: Signal Protocol

Overview

This section describes a high-performance, bidirectional signal protocol that was jointly developed by Hewlett-Packard and Microsoft. Only the protocol is described in this section; for information related to the standard PC ISA hardware implementation of the protocol, refer to Section 3.

This specification is an enhancement to the IEEE P1284 standard, which describes three basic data transfer modes:

- Compatible mode (forward channel, industry-standard parallel port interface)
- Nibble mode (reverse channel, compatible with all existing PC hosts)
- Byte mode (reverse channel, compatible with IBM® PS/2® hosts)

This document describes two very similar additional modes that may be nested within the P1284 standard:

- ECP mode (fast bidirectional; requires custom hardware on interfaces)
- ECP mode including RLE decompression

The ECP modes conform to the conventions and philosophies established in the IEEE P1284 standard and will be implemented on future generations of hosts and peripherals. To attain the highest performance, hardware is required on both the peripheral and host. ECP boosts the I/O bandwidth to meet the demands of high-performance peripherals.

Hewlett-Packard and Microsoft recommend that the ECP modes be incorporated into the IEEE P1284 standard. Hewlett-Packard and Microsoft are taking steps to ensure that the ECP becomes a de facto standard throughout the personal computer industry.

ECP provides a number of advantages, some of which are listed below. The individual features are explained in greater detail in the remainder of the document.

- High-performance half-duplex forward and reverse channel
- Interlocked handshake, for fast, reliable transfer
- Optional single-byte RLE compression for improved throughput (64:1)
- Channel addressing for low-cost peripherals
- Link and data layer separation
- Use of active output drivers
- Use of adaptive signal timing
- Peer-to-peer capability

Description

Supplementary signal definitions

The ECP modes conform to the signal line definitions established by the proposed IEEE 1284 specification (also termed "BOISE"). However, a few signal lines have alternate uses during ECP mode.

The following signals are redefined to provide additional functionality in ECP mode.

Table 1. Redefined Signals

Compatible mode	Nibble/Byte mode	ECP mode
nFault	nDataAvailable	nPeriphRequest
nSelectIn	RnW	BOISEmode
nInit	always high	nReverseRequest
PError	AckDataReq	nAckReverse
nAutoFd	HostBusy	HostAck
Busy	PeriphBusy	PeriphAck

nPeriphRequest (nFault)

During ECP mode the peripheral is permitted (but not required) to drive this pin low to request a reverse transfer. The request is merely a "hint" to the host; the host has ultimate control over the transfer direction. This signal provides a mechanism for peer-to-peer communication. This signal typically would be used to generate an interrupt to the host CPU. The signal is asserted low and kept there until the interrupt is serviced or the port exits ECP mode.

BOISEmode (nSelectIn)

This pin is driven high during all P1284 modes. It is driven low to terminate. This signal operates the same in ECP mode as in Nibble/Byte mode, but the name has been changed because "RnW" was inappropriate for a bidirectional mode.

nReverseRequest (nInit)

This pin is driven low to place the channel in the reverse direction. The peripheral is only allowed to drive the bidirectional data bus while in ECP mode, when BOISEmode is high and nReverseRequest is low.

nAckReverse (PError)

The peripheral drives this signal low to acknowledge nReverseRequest. It is an "interlocked" handshake with nReverseRequest. The host relies upon nAckReverse to determine when it is permitted to drive the data bus.

HostAck (nAutoFd)

The host drives this signal to flow control in the reverse direction. It is an "interlocked" handshake with nAck. HostAck also provides command information in the forward phase.

PeriphAck (Busy)

The peripheral uses this signal to flow control in the forward direction. It is an "interlocked" handshake with nStrobe. PeriphAck also provides command information in the reverse direction.

Negotiation into ECP modes

Negotiation into the ECP modes from Compatibility mode is accomplished according to the established P1284 methodology, with a few minor differences. The following P1284 extensibility request values are provided for ECP:

Table 2. Extensibility Request Values

Extensibility Request Value	Definition
0001 0000	ECP mode
0001 0100	ECP Device ID
0011 0000	ECP mode with RLE compression
0011 0100	ECP Device ID with RLE compression
0010 0000	Reserved
0010 0100	Reserved

During negotiation, Xflag (Select) is driven high to indicate that the peripheral supports the request transfer mode. Unlike Nibble/Byte modes, the nDataAvail (nFault) does *not* indicate the availability of data during negotiation (or thereafter). Also, nDataAvail (nFault) is redefined as nPeriphRequest during ECP mode.

Immediately following negotiation (and a short setup phase), the interface defaults to the forward direction. If no forward channel data is to be transmitted, the interface may then be reversed.

Termination from ECP mode

Termination from ECP mode is similar to the termination from Nibble/Byte modes. The host is permitted to terminate from ECP mode only in specific, well-defined states. The termination can only be executed while the bus is in the forward direction. To terminate while the channel is in the reverse direction, it must first be transitioned into the forward direction.

Device ID

Device ID for High Speed devices is handled in the conventional manner as defined in the proposed IEEE 1284 specification. That is, when the Device ID Request and ECP Mode Request bits are both asserted in the Extensibility Request Value during negotiation, the High Speed device will return the standard P1284 Device ID string. Channel Addressing is not used during Device ID. Compression may optionally be used during Device ID. Forward channel data is not sent during Device ID mode. To transfer normal data, the host must first terminate from the Device ID mode and renegotiate with the Device ID Request bit deasserted in the Extensibility Request Value.

Command/Data

ECP mode supports two advanced features to improve the effectiveness of the protocol for some applications. The features are implemented by allowing the transfer of normal 8-bit data or 8-bit commands.

When in the forward direction, normal data is transferred when (HostAck) nAutoFd is high and an 8-bit command is transferred when (HostAck) nAutoFd is low. The most significant bit of the command indicates whether it is a run-length count (for compression) or a channel address.

Table 3. Forward Channel Commands (When HostAck Is Low)

D7	D[6:0]
0	Run-Length Count (0-127) (mode 0011 0X00 only)
1	Channel Address (0-127)

When in the reverse direction, normal data is transferred when busy (PeriphAck) is high and an 8-bit command is transferred when busy (PeriphAck) is low. The most significant bit of the command is always zero. Reverse channel addresses are seldom used and may not be supported in hardware.

Table 4. Reverse Channel Commands (When PeriphAck Is Low)

D7	D[6:0]
0	Run-Length Count (0-127) (mode 0011 0X00 only)
1	Channel Address (0-127)

Optional support of compression/decompression

Devices may or may not choose to support decompression via the negotiation sequence. Devices that negotiate into ECP RLE mode (mode 0011 0X00) must support decompression of RLE data and may optionally compress it. Devices using the non-RLE ECP mode *must not* transfer compressed data.

Data Compression

To provide the potential for increased performance, a simple data compression (64:1max, 4:1 typical) technique is built into the protocol specification. Single-byte, run-length encoding is supported, which compresses strings of identical bytes while guaranteeing that incompressible data will not be expanded. The compression is particularly useful on raster imaging devices. The decompression/compression is handled very simply and economically in hardware. This simple compression does not preclude the use of other data compression schemes at a higher (data stream or packet) level.

When a run-length count is received, the subsequent data byte is replicated the specified number of times. A run-length count of zero specifies that only one byte of data is represented by the next data byte, whereas a run-length count of 127 indicates that the next byte should be expanded to 128 bytes. To prevent data expansion, however, run-length counts of zero should be avoided.

Channel Addressing

To support simple, low-cost peripherals that do not desire to parse a data stream or packet, a channel-addressing scheme is provided in ECP mode. ECP mode provides 128 channel addresses. The channel addresses may be dynamically changed while in ECP mode. The support of channel addresses does not incur any overhead for typical devices that wish to transmit or receive only data stream or packet data (stream or packet peripherals may ignore channel addresses). Specific channel address definitions are device-specific. Although the use of channel addresses seemingly violates the P1284 specification's data and link layer separation philosophy, it is permissible because the use of addresses is optional for the peripheral. The channel address defaults to zero after each negotiation. After a channel address command is issued, the address remains in effect indefinitely until another channel address command is issued, or until termination.

Output Drivers

To facilitate higher performance data transfer, the use of balanced CMOS active drivers for critical signals (Data, HostAck, HostClk, PeriphAck, PeriphClk) is permitted and encouraged during ECP mode. Because the use of active drivers can present compatibility problems in Compatible mode (the control signals, by tradition, are specified as open-collector), the drivers may be dynamically changed from open-collector to totem-pole. The timing for the dynamic driver change is specified in this document. The dynamic driver change must be implemented properly to prevent glitching the outputs.

Signal Timing

The signal timing for ECP mode is designed for performance and reliability over cabled systems.

Table 5. Signal Timing

Time	Minimum	Maximum
T^H	0	1.0 sec.
T^T	0	infinite
T^L	0	35 ms
T^S	35ms	
T^P	500 ns	
T^D	0 ns	
TR	Host may enter Data Transfer Recovery after TS (Software application dependent)	

The use of interlocked handshaking provides reliable data transfers. Interlocked handshaking

also provides the ability to make trade-offs concerning performance and reliability at the host or peripheral (analog or digital filtering may be implemented to improve noise immunity or support long cables, at the cost of reduced performance).

The timing is completely self-adapting, allowing more time for systems using longer cables.

Peer-to-Peer Capability

Unlike other P1284 modes, the ECP mode may reverse the channel direction at will without having to renegotiate. In addition, a rudimentary "peer-to-peer" capability is provided that allows the peripheral to indicate to the host when it would like to reverse the channel (the host is always in ultimate control of the channel direction, however).

Operation

Operating Phases

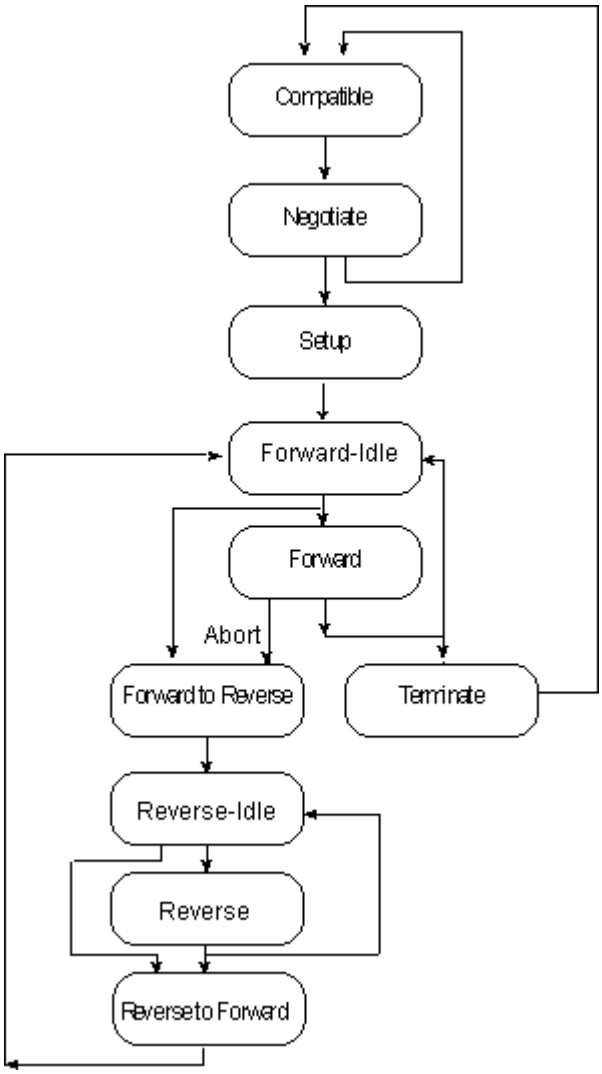

Figure 1. Phase transition diagram

The P1284 interface operates in phases. Additional phases are defined for use in the ECP mode as described below. These phases represent the state of the interface as data is moved between the host and the printer. The transitions are based on the present state of the interface, signal transitions from the host and the printer, and time-outs.

Compatible Data Transfer phase

Interface is in compatible mode, performing host-to-printer data transfer.

Compatible Idle phase

Interface is in compatible mode, no data transfer. Printer status lines indicate current parallel port status.

Reverse Data Transfer phase

Interface is in ECP mode, performing printer-to-host data transfer.

Negotiation phase

Signal handshaking to change interface from compatible mode to P1284 mode.

Setup phase

This phase immediately follows the Negotiation phase (it is actually part of the negotiation, but it differs from Nibble/Byte modes, so it is denoted separately) and is necessary to set up the interface signals to the correct state for the Forward Data Transfer phase. The interface may optionally switch from open-collector to active-drive outputs during this phase.

Forward-Idle

When the host has no data to send, it keeps HostClk (nStrobe) high and the peripheral will leave PeriphAck (Busy) low.

Forward Data Transfer phase

The interface transfers data and commands from the host to the peripheral using an interlocked PeriphAck and HostClk. The peripheral may indicate its desire to send data to the host by asserting nPeriphRequest.

Forward to Reverse phase

The interface is changing from the forward direction to the reverse direction.

Reverse-Idle phase

The peripheral has no data to send and keeps PeriphClk high. The host is idle and keeps HostAck low.

Reverse Data Transfer phase

The interface transfers data and commands from the peripheral to the host using an interlocked HostAck and PeriphClk.

Reverse to Forward phase

The interface is changing from the reverse direction to the forward direction.

Termination phase

Signal handshaking to change from P1284 mode to Compatible mode. Also, the output drivers return to open-collector in this phase. Termination may only be accomplished from the Forward Data Transfer phase.

ECP Mode Interface Errors

Errors can occur during interface transfers due to time-outs, noise, incorrect protocol implementation, device being powered off, and so on. Many errors can be detected by both the host computer and the printer. When the host or peripheral detects an error, it should immediately abort (with no termination phase) and resume Compatibility mode operation.

In particular, to protect against bus fight conditions on the bidirectional data pins, the peripheral must immediately (within 1µS) stop driving the data bus in the event of a protocol exception.

The peripheral should carefully monitor both BOISEmode (nSelectIn) and nReverseRequest (nInit) to detect when the host has aborted to Compatibility mode.

When a High Speed device detects an error, it should terminate the current transfer and assume that the current byte was not successfully transferred. The interface provides no error detection on the data itself. Any desired error detection and recovery should be handled at a higher level.

Protocol Description

The High Speed protocol is described in the following section. Timing diagrams are provided to detail the handshake sequences. Along the bottom of each diagram are numbers corresponding to signal transition events. These events are listed in the "Event List" section. The event numbers are also shown in parentheses in the textual descriptions below to improve readability of the diagrams.

High Speed Negotiation Phase

To begin the Negotiation phase, the host places the High Speed extensibility request value on the data bus (event 0), then sets BOISEmode (nSelectIn) high and HostAck (nAutoFd) low (event 1). The peripheral responds by setting PeriphClk (nAck) low, nPeriphRequest (nFault) high, Xflag(Select) high, and nAckReverse (PError) high (event 2). The host then sets HostClk (nStrobe) low (event 3). The host then sets HostClk (nStrobe) and HostAck (nAutoFd) high (event 4), acknowledging that it has recognized a High Speed compatible peripheral. The peripheral then sets nAckReverse (PError) low, PeriphAck (Busy) low, and Xflag (Select) high if it supports ECP mode (event 5). The printer then sets PeriphClk (nAck) high (event 6), indicating that the other status lines may be read. The interface now enters the Setup phase. Figure 2 demonstrates a successful negotiation.

Extended Capabilities Port (ECP) mode: Negotiation, Setup, Forward, Termination

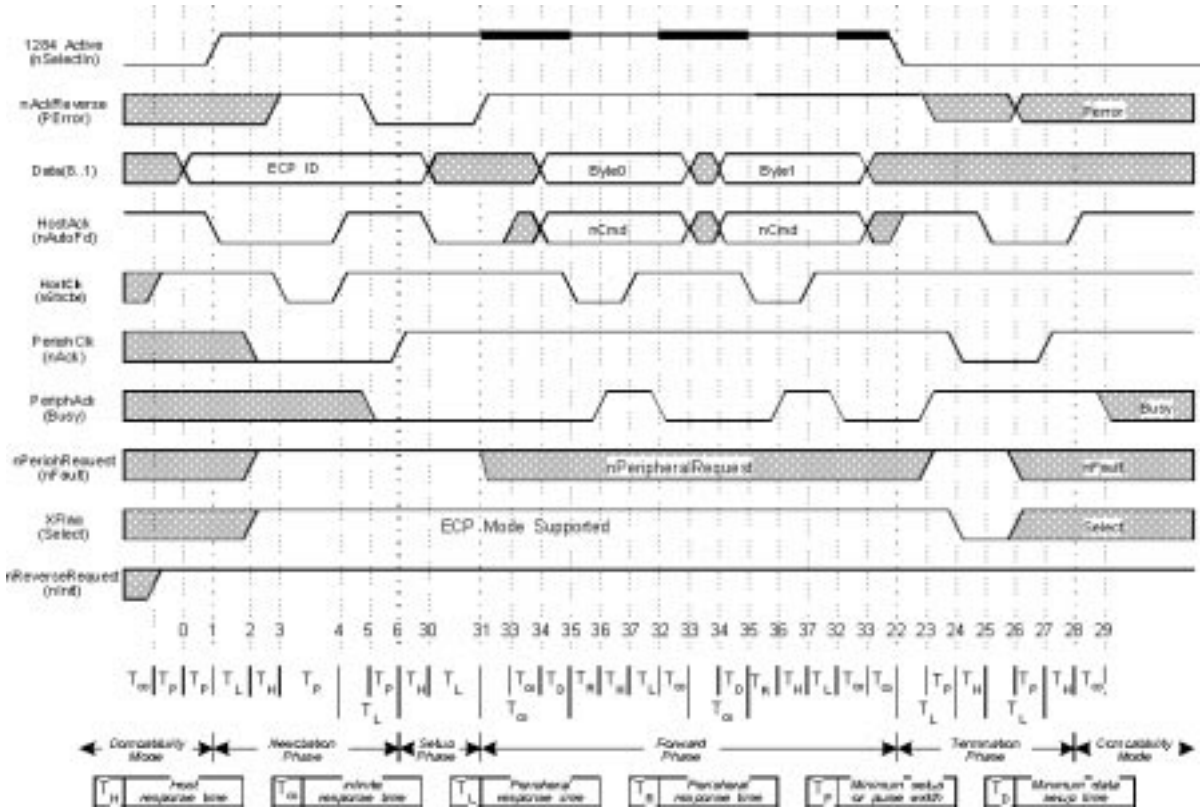

Figure 2. ECP mode timing (1 of 3 charts)

Failed Negotiation

If the peripheral does not support ECP mode, it will set Xflag(Select) low during negotiation.

When this occurs, the host must terminate the session and renegotiate for another transfer mode. This sequence is fully documented in the P1284 specification.

Setup Phase

The Setup phase is entered immediately following a successful negotiation. After seeing PeriphClk (nAck) go high, the host optionally changes to active drivers and sets HostAck (nAutoFd) low (event 30). The peripheral responds by setting nAckReverse (PError) high and optionally changing to active drivers (event 31). The interface now enters the Forward phase. The setup phase transitions are shown in Figure 2.

Forward Data Transfer Phase

The Forward Phase may be entered from the Forward-Idle Phase. When the peripheral is not busy it sets PeriphAck (Busy) low (event 32). The host then sets HostClk (nStrobe) low when it is prepared to send data (event 35). The data must be stable for the specified setup time prior to the falling edge of HostClk. The peripheral then sets PeriphAck (Busy) high to acknowledge the handshake (event 36). The host then sets HostClk (nStrobe) high (event 37). The peripheral then accepts the data and sets PeriphAck (Busy) low, completing the transfer. This sequence is shown in Figure 2.

The timing is designed to provide three cable round-trip times for data setup if Data is driven simultaneously with HostClk (nStrobe).

Aborting the Forward Data Transfer phase

There is a possibility of the forward channel becoming stalled. The stall condition will exist if the peripheral is unable to accept the data byte being transferred by the host at event 35. In this condition the peripheral will not acknowledge the handshake (event 36). A mechanism has been provided to recover from this condition. If the host, following event 35, determines that a stall condition may exist, the host may abort the transfer of the current byte by setting nReverseRequest (nInit) low (event 72). The peripheral, regardless of whether it has accepted the byte from the host (event 36 happened), shall discard the byte (if applicable) and acknowledge the host by setting nAckReverse (PError) low. The host then returns nReverseRequest (nInit) high (event 74) and the peripheral follows by returning nAckReverse (PError) high (event 75). This sequence, shown in Figure 4, will return the interface to the state that existed prior to host event 35.

Forward to Reverse Phase

The Forward to Reverse phase is entered from the Forward phase. The host tri-states the data bus and sets HostAck (nAutoFd) low (event 38). After waiting for the minimum setup time, the host then sets nReverseRequest (nInit) low (event 39). The peripheral then acknowledges the reversal by setting nAckReverse (PError) low (event 40). The peripheral is now permitted to drive the data bus after setting nStrobe high. The interface now enters the Reverse phase. This sequence is shown in Figure 3.

Extended Capabilities Port (ECP) Mode: Forward to Reverse (Fwd2Rev), Reverse, Reverse to Forward (Rev2Fwd)

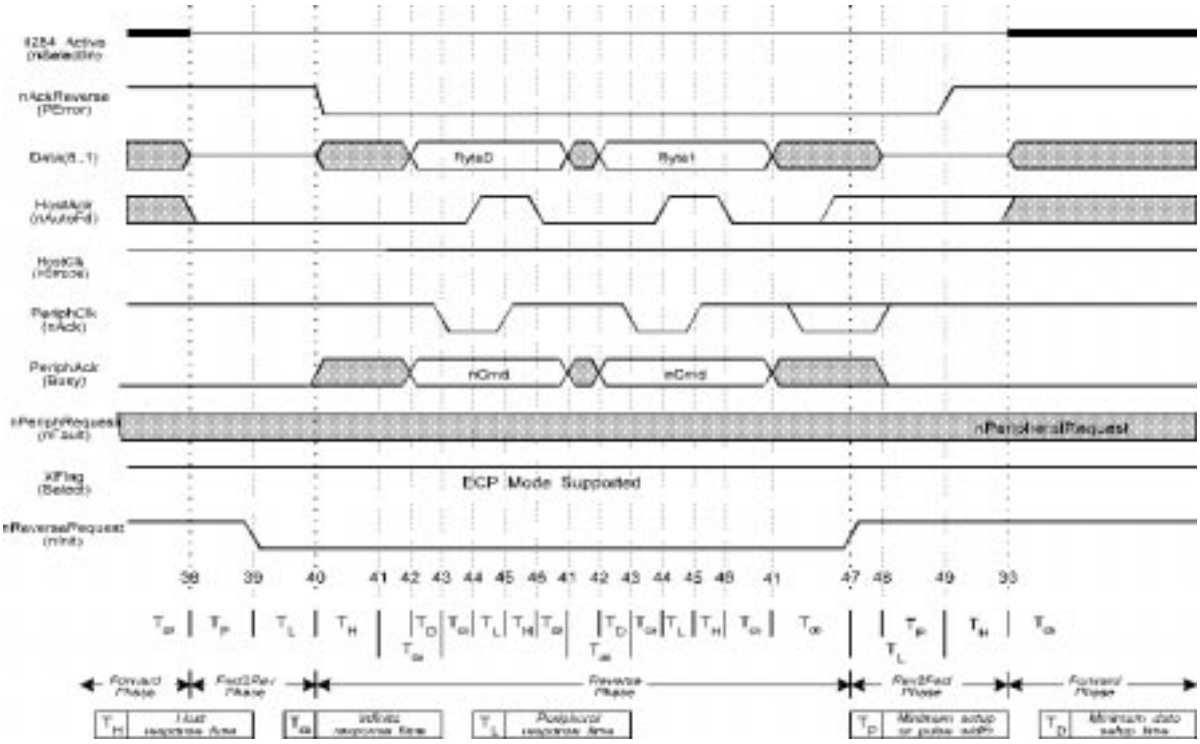

Figure 3. ECP mode timing (2 of 3)

Extended Capabilities Port (ECP) Mode: Host Transfer Recovery

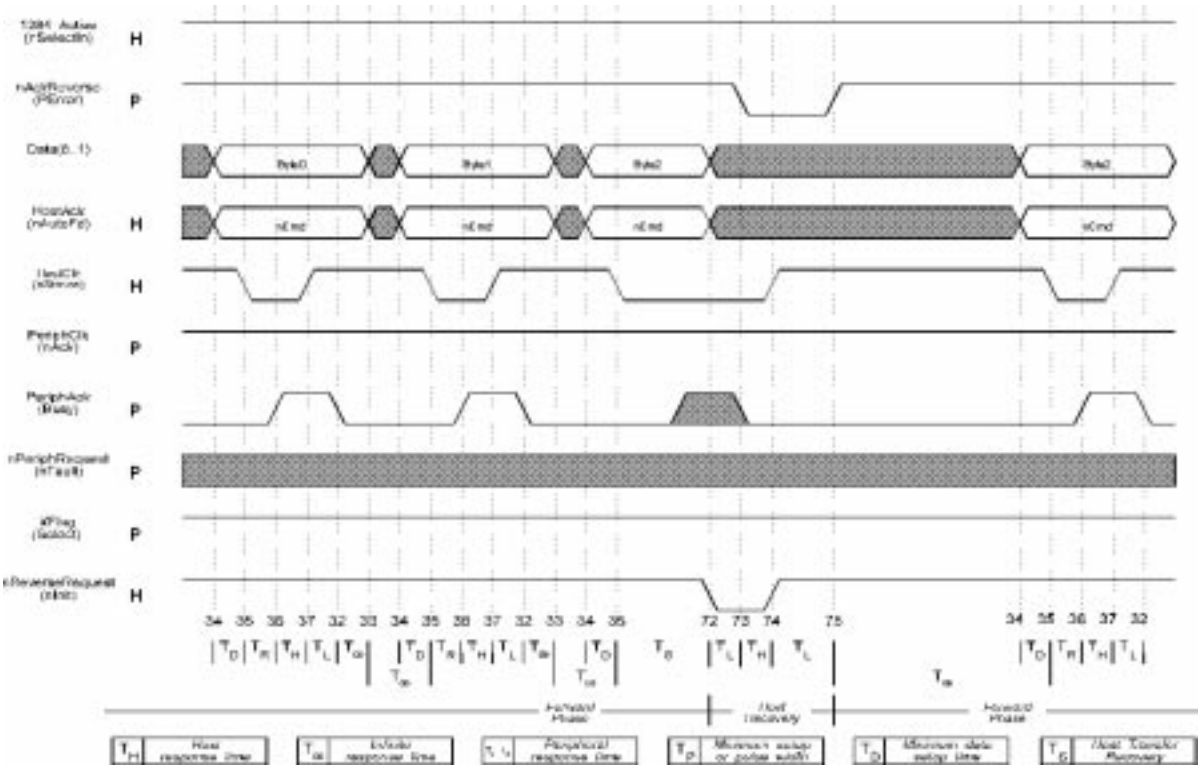

Figure 4. ECP mode timing (3 of 3)

Reverse Data Transfer Phase

The Reverse phase may be entered from the Reverse-Idle phase. After the previous byte has been accepted, the host sets HostAck (nAutoFd) low (event 46). The peripheral then sets

PeriphClk (nAck) low when it has data to send (event 43). The data must be stable for the specified setup time prior to the falling edge of PeriphClk. When the host is ready to accept a byte, it sets HostAck (nAutoFd) high to acknowledge the handshake (event 44). The peripheral then sets PeriphClk (nAck) high, causing the host to accept the data (event 45). After the host has accepted the data, it sets HostAck (nAutoFd) low (event 46), completing the transfer. This sequence is shown in Figure 3.

Reverse to Forward Phase

The Reverse to Forward phase is entered from the Reverse phase. HostAck (nAutoFd) may be high or low when the Reverse to Forward phase is entered. The host sets nReverseRequest (nInit) high (event 47). The peripheral then tri-states the data bus, sets PeriphAck (Busy) low to indicate the proper forward channel status, and sets PeriphClk (nAck) high (event 48). If the peripheral was in the middle of a data transfer (PeriphClk low), it assumes that the data byte will be discarded by the host and suspends the transfer. After waiting the minimum setup time, the peripheral then sets nAckReverse (PError) high to acknowledge the change of direction (event 49). The host is now permitted to drive the data bus. The interface now enters the Forward phase. This sequence is shown in Figure 3.

Valid Termination

To terminate from P1284 mode, the host sets BOISEmode (nSelectIn) low (event 22), which will initiate one of two types of termination. The first type is a handshake that allows the printer to tell the host when it has returned to compatible mode. The second is an immediate abort, with no guarantee of interface integrity. If the interface was in a "valid" state, which is any state where a reverse data transfer is not in progress, the printer will perform the handshake. If the interface was not in a "valid" state, the printer will abort immediately. Valid states are indicated in the data transfer diagrams by BOISEmode (nSelectIn) being shown as a heavy line.

To terminate from a valid state, the printer will respond to BOISEmode (nSelectIn) being set low by setting nAckReverse(PError) to low and PeriphAck (Busy) and nPeriphRequest (nFault) high (event 23). The printer will then set Xflag (Select) to its opposite sense and PeriphClk (nAck) low (event 24). The host then sets HostAck (nAutoFd) low (event 25). The printer then sets the compatible mode printer status on nPeriphRequest (nFault), Xflag (Select) and nAckReverse (PError) while the host sets the compatible mode status on nReverseRequest (nInit) (event 26). The printer then sets PeriphClk (nAck) high (event 27). The host ends the termination handshake by setting HostAck (nAutoFd) high (event 29), which returns the interface to the compatible mode idle phase. The printer may then change PeriphAck (Busy) (event 30) to accept host-to-printer data. This sequence is shown following a data transfer in Figure 2.

Aborting a Transfer

When BOISEmode (nSelectIn) is set low in an invalid state, the printer aborts immediately. This is to protect both the printer and the host. The unexpected transition of BOISEmode (nSelectIn) and possibly other signals could be caused by a user switching a switch box at the wrong time, or a cable that has worked loose. If a reverse channel data transfer is aborted, the current byte in transit is lost, but the printer will hold that byte in its output register. The next time the host performs a reverse channel transfer, that byte will be the first one sent.

Event List

The timing charts that constitute Figures 2, 3, and 4 contain numbers corresponding to the events that cause the transitions. Following is a list of those numbers and descriptions of the corresponding events.

Figure 2 Numbers and Events

0 Host sets extensibility request value on data bus.

1 Host requests a ECP mode transfer by setting BOISEmode (nSelectIn) high and HostAck (nAutoFd) low.

2 Peripheral indicates ECP mode support by setting nAckReverse (PError), Xflag (Select), and nPeriphRequest (nFault) high, and PeriphClk (nAck) low.

3 Host sets HostClk (nStrobe) low to latch extensibility request value into printer.

4 After waiting the minimum HostClk (nStrobe) pulse width, the host sets HostClk (nStrobe) and HostAck (nAutoFd) high to acknowledge the peripheral's support of the High Speed protocol.

5 Peripheral sets nAckReverse (PError) low. Xflag (Select) is set to reflect the peripheral's support of the requested extension. PeriphAck (Busy) is set to indicate whether the printer can accept data from the host.

6 Peripheral sets PeriphClk (nAck) high, informing the host that the four interface status signals are valid.

Note: *Events 7 through 21 are defined in the IEEE P1284 specification document; these events deal with Byte and Nibble modes and are not discussed here.*

22 Host sets BOISEmode (nSelectIn) low and HostAck (nAutoFd) high to request termination of ECP mode.

23 Peripheral sets PeriphAck (Busy) and nPeriphRequest (nFault) high, and nAckReverse (PError) low.

24 Peripheral acknowledges host's request by setting PeriphClk (nAck) low and Xflag (Select) to its opposite sense.

25 Host handshakes with peripheral by setting HostAck (nAutoFd) low.

26 Peripheral sets nAckReverse (PError), nPeriphRequest (nFault), and Xflag (Select) to their current compatible mode values. PeriphAck (Busy) is still high to block incoming data.

27 Peripheral completes handshake by setting PeriphClk (nAck) high.

28 Host completes handshake by setting HostAck (nAutoFd) high.

29 Peripheral updates PeriphAck (Busy) to current compatible mode status.

30 Host sets HostAck (nAutoFd) low to acknowledge successful negotiation.

31 Peripheral acknowledges that it is now operating in ECP mode by raising nAckReverse (PError). PeriphAck (Busy) and nPeriphRequest (nFault) are now active.

32 The peripheral drives HostAck (nAutoFd) low, indicating that it has accepted the data. This signals the end of the transfer.

33 The host is idle.

34 Host places Data on the bus. The command bit (nCmd/HostAck) is driven to the appropriate level.

35 The host sets HostClk (nStrobe) low to indicate valid data is on the bus.

36 The peripheral handshakes, setting PeriphAck (Busy) high.

37 The host raises nStrobe to continue the handshake. The peripheral will use this edge of nStrobe to latch the data.

Figures 3 and 4, Numbers and Events

38 The host tri-states the Data bus and sets HostAck (nAutoFd) low to prepare for a bus reversal.

39 The Host sets nReverseRequest (nInit) low to initiate a bus reversal.

40 The peripheral sets nAckReverse (PError) low to acknowledge the bus reversal. (nAutoFd) is now active.

41 The peripheral is now idle.

42 The peripheral drives Data and nCmd (Busy) onto the bus.

43 The peripheral sets PeriphClk (nAck) low to indicate that data is available on the bus.

44 The host acknowledges the assertion of PeriphClk (nAck) by setting HostAck (nAutoFd) high.

45 The peripheral continues the handshake by setting PeriphClk (nAck) high.

46 The host completes the transfer, accepting the byte by setting HostAck (nAutoFd) low.

47 The host sets nReverseRequest (nInit) high to initiate a bus reversal (back to the forward direction). The host may continue to handshake, receiving "don't care" data.

48 The peripheral terminates any ongoing transfer, tri-states the data bus, sets PeriphClk (nAck) high, and places valid status on the PeriphAck (Busy) line.

49 The peripheral acknowledges that the bus has been relinquished by setting nAckReverse

(PError) high.

50 The host drives the data bus and continues with forward data transfer.

72 After waiting for the minimum required time (Ts), the host may abort the host to peripheral transfer in progress by setting nReverseRequest (nInit) low.

73 The peripheral handshakes, setting nAckReverse (PError) low, and if not already PeriphAck (Busy) low, indicating that the peripheral-to-host data transfer in progress has been aborted and the data byte has been discarded.

74 The host raises nReverseRequest to continue the handshake.

75 The peripheral completes the handshake by raising nAckReverse (PError) high, returning the link to a host-idle condition.

Section 3: ISA Implementation Standard

Overview

This specification describes the standard ISA interface to the extended capabilities port (ECP). All ISA devices supporting ECP must meet the requirements contained in this section or the port will not be supported by Microsoft.

Description

The port is software- and hardware-compatible with existing parallel ports so that it may be used as a standard LPT port if ECP is not required. The port is designed to be simple and requires a small number of gates to implement. It does not do any "protocol" negotiation, rather it provides an automatic high-burst bandwidth channel that supports DMA for ECP in both the forward and reverse directions.

Small FIFOs are employed in both forward and reverse directions to smooth data flow and improve the maximum bandwidth requirement. The size of the FIFO must be at least 16 bytes deep.

The port supports an automatic handshake for the standard parallel port to improve compatibility mode transfer speed.

The port also supports run-length encoded (RLE) decompression (required) in hardware. Compression is accomplished by counting identical bytes and transmitting an RLE byte that indicates how many times the next byte is to be repeated. Decompression simply intercepts the RLE byte and repeats the following byte the specified number of times. Hardware support for compression is optional. Please consult Section 2 for signal protocol details on ECP.

Description of Pins

Table 6. ECP Parallel Port Signal List

Name	Qty	Dir	Host Pin #	Slave Pin #	ECP Function
nStrobe	1	O	1	1	During write operations nStrobe registers data or address into the slave on the asserting edge (handshakes with Busy).
data<7:0>	8	I/O	9-2	9-2	Contains address or data or RLE data.
nAck	1	I	10	10	Indicates valid data driven by the peripheral when asserted. This signal handshakes with nAutoFd in reverse.
Busy	1	I	11	11	This signal deasserts to indicate that the peripheral can accept data. This signal handshakes with nStrobe in the forward direction. In the reverse direction this signal, when low, indicates the data is RLE.

PError	1	I	12	12	Used to acknowledge a change in the direction the transfer (asserted = forward).
Select	1	I	13	13	Indicates printer on line.
nAutoFd	1	O	14	14	Requests a byte of data from the peripheral when asserted, handshaking with nAck in the reverse direction. In the forward direction this signal indicates whether the data lines contain ECP address or data.
nFault	1	I	15	32	Generates an error interrupt when asserted.
nInit	1	O	16	31	Sets the transfer direction (asserted = reverse, deasserted = forward).
nSelectIn	1	O	17	36	Always deasserted in ECP mode.
gnd	1	—	18	33	Aux-Out
gnd	1	—	19	19	Return ground for strobe.
gnd	1	—	20	21	Return ground for data<1>.
gnd	1	—	21	23	Return ground for data<3>.
gnd	1	—	22	25	Return ground for data<5>.
gnd	1	—	23	27	Return ground for data<7>.
gnd	1	—	24	29	Return ground for Busy.
gnd	1	—	25	30	Return logic ground.
gnd			NC	20	Grounded on Connector.
gnd			NC	22	Grounded on Connector.
gnd			NC	24	Grounded on Connector.
gnd			NC	26	Grounded on Connector.
gnd			NC	28	Grounded on Connector.
			NC	15-17 34-35	Unused.

ISA Connections

The interface can never stall, causing the host to hang. The width of data transfers is strictly controlled on an I/O address basis per this specification. All FIFO-DMA transfers are PWord wide, PWord aligned, and end on a PWord boundary. (The PWord value can be obtained by reading Configuration Register A, **cnfgA**, described in the next section.) Single-byte-wide transfers are always possible with standard or PS/2 mode using program control of the control signals.

Table 7. ISA Interface

Name	Qty	Dir	Function
da<10:0>	11	I/O	System Address bus
sd<15-7:0>	16-8	I/O	System Data Bus
~ioR	1	I	I/O Read Command
~ioW	1	I	I/O Write Command
~ioCs16	1	O	I/O is 16 bit
dRq	1	O	DRQ DMA Request (Note: Use a 1K pulldown here to prevent requests.)
~dAck	1	I	DACK DMA Grant
iRq	1	O	IRQ
sysClk	1	I	System Clock
resetDrv	1	I	Reset

Register Definitions

The register definitions are based on the standard IBM addresses for LPT. All of the standard printer ports are supported. The additional registers attach to an upper bit decode of the standard LPT port definition to avoid conflict with standard ISA devices.

The port is equivalent to a generic parallel port interface and may be operated in that **mode**. The port registers vary depending on the **mode** field in the **ecr**. The table below lists these dependencies. Operation of the devices in modes other that those specified is undefined.

Note that all addresses shown in Table 8 are added to the base of 03bch, 0278h, 0378h.

Table 8. Register Definitions

Name	Address		Size	Mode	Function
data	0x000	R/W	byte	000-001	Data Register
ecpAFifo	0x000	W-R/W	byte	011	ECP FIFO (Address)
dsr	0x001	R	byte	All	Status Register
dcr	0x002	R/W	byte	All	Control Register
cFifo	0x400	W-R/W	PWord	010	Parallel Port Data FIFO
ecpDFifo	0x400	R/W	PWord	011	ECP FIFO (Data)
tFifo	0x400	R/W	PWord	110	Test FIFO
cnfgA	0x400	R-R/W	byte	111	Configuration Register A
cnfgB	0x401	R-R/W	byte	111	Configuration Register B
ecr	0x402	R/W	byte	All	Extended Control Reg.

data

0x000 modes 000,001 (Parallel Port Data Register)

This is the standard parallel port data register. Writing to this register in mode 000 will drive data to the parallel port data lines. In all other modes the drivers may be tri-stated by setting the **direction** bit in the **dcr**. Reads to this register return the value on the data lines.

ecpAFifo

0x000 mode 011 (ECP FIFO: Address/RLE)

A data byte written to this address is placed in the FIFO and tagged as a ECP Address/RLE. The hardware at the ECP port will transmit this byte to the peripheral automatically. The operation of this register is defined only for the forward direction (**direction** is 0).

Table 9. ECP Address FIFO

<7>	W	Indicates data Type
	1:	Bits <6:0> are a ECP Address
	0:	Bit field <6:0> is a run length, indicating how many times the next data byte is to appear (0 = 1 time, 1 = 2 times, 2 = 3 times, and so on).
<6:0>	W	Address or RLE field described above.

dsr

0x001 (Device Status Register)

This read-only register reflects the inputs on the parallel port interface.

Table 10. Device Status Register

<7>	R	**nBusy**	inverted version of parallel port **Busy** signal
<6>	R	**nAck**	version of parallel port **nAck** signal
<5>	R	**PError**	version of parallel port **PError** signal
<4>	R	**Select**	version of the parallel port **Select** signal
<3>	R	**nFault**	version of the parallel port **nFault** signal
<2:0>	R	**reserved**	returns undefined when read

dcr

0x002 (Device Control Register)

This register directly controls several output signals as well as enabling some functions. The drivers for nStrobe, nAutoFd, nInit, and nSelectIn are open-collector in mode 000, and are push-pull in all other modes.

In all modes the **dcr** *shall* be able to override any hardware state machine and force the signal active. For example, writing 1s to bits<1:0> *shall* force **nStrobe** and **nAutoFd** low, even in ECP mode. Software will make sure that **dcr** bits <1:0> are set to 0 prior to entering ECP mode.

Table 11. Device Control Register

<7:6>	R	**Reserved**, returns undefined when read.
<5>	R/W	**Direction**.
	1:	If **mode** = 000 or **mode** = 010, we are standard parallel port and this bit has no effect (drivers are enabled). Otherwise, this bit tri-states the drivers and sets the direction so that data will be read from the peripheral. Note: some designs actually force this bit to a 0 when in modes 000 or 010. Software must be in PS2 mode 001 in order to reliably write this bit to a 1.
	0:	Drivers are enabled. DMA, data are written to the peripheral.
<4>	R/W	**ackIntEn**.
	1:	Enables an interrupt on the rising edge of **nAck**.
	0:	Disables the **nAck** interrupt.
<3>	R/W	**SelectIn** is inverted and then driven as parallel port **nSelectIn**.
<2>	R/W	**nInit** is driven as parallel port **nInit**.
<1>	R/W	**autofd** is inverted and then driven as parallel port **nAutoFd**.
<0>	R/W	**strobe** is inverted and then driven as parallel port **nStrobe**.

cFifo

0x400, mode = 010 (Parallel Port Data FIFO)

PWords written or DMAed from the system to this FIFO are transmitted by a hardware handshake to the peripheral using the standard parallel port protocol. Transfers to the FIFO are PWord aligned. If partial PWords need to be transferred then the operation must be handled in mode 000. This mode is only defined for the forward direction.

ecpDFifo

0x400, mode = 011 (ECP Data FIFO)

PWords written or DMAed from the system to this FIFO when **direction** is 0 are transmitted to the peripheral by hardware handshake using the ECP parallel port protocol. Transfers to the FIFO are PWord-aligned. If odd bytes need to be transferred, the operation must be handled in mode 000.

Data PWords from the peripheral are read under automatic hardware handshake from ECP into this FIFO when **direction** is 1. Reads or DMAs from the FIFO will return PWords of ECP data to the system.

tFifo

0x400, mode = 110 (Test mode)

Data PWords may be read, written, or DMAed to or from the system to this FIFO in any **direction**.

Data in the **tFifo** will not be transmitted to the parallel port lines using a hardware protocol handshake. However, data in the **tFifo** may be displayed on the parallel port data lines.

The **tFifo** will not stall when overwritten or underrun. Data will simply be rewritten or overrun. The **full** and **empty** bits must always keep track of the correct FIFO state. The **tFifo** will transfer data at the maximum ISA rate so that software can generate performance metrics.

The **writeIntrThreshold** can be determined by starting with a **full tFifo**, and emptying it one PWord at a time until **serviceIntr** is set. This may generate a spurious interrupt, but will indicate

that the threshold has been reached. Likewise, **readIntrThreshold** can be determined by setting the direction bit to 1, and filling the **empty tFifo** one PWord at a time until **serviceIntr** is set.

Data PWords are always read from the head of **tFifo**, regardless of the value of the direction bit. For example, if 0x4433, 0x2211, 0x00ff is written to the FIFO, then reading the **tFifo** will return 0x4433, 0x2211, 0x0ff (in the same order it was written).

The FIFO size and interrupt threshold can be determined by writing PWords and checking the **full** and **serviceIntr** bits.

cnfgA

0x400, mode = 111 (Configuration Register A)

This register allows software to obtain implementation-specific information. All ISA ports shall implement the read-only **implD** as a minimum.

Table 12. Configuration Register A

<7>	R	Indicates if interrupts are pulsed or ISA-Level.
	1:	Interrupts are ISA-Level (See Appendix B).
	0:	Interrupts are ISA-Pulses.
<6:4>	R	**implD.** Implementation ID number; identifies the design and PWord size.
	0x00:	The design is a 16-bit implementation (PWord = 2 bytes).
	0x01:	The design is an 8-bit implementation (PWord = 1 byte).
	0x02:	The design is a 32-bit implementation (PWord = 4 bytes).
	0x03-0x07:	Reserved and not supported by Microsoft Software.
<3>	R/RW	**misc. reserved.** May be used for anything design-specific. If software, may try to write it to a 1.
<2>	R	**nByteInTransceiver.** This design-dependent, read-only parameter indicates if the design uses an extra pipeline byte when transmitting ECP in event 35. See the section on ECP Host Recovery for more information.
	0:	When transmitting (at event 35) there is 1 byte in the transceiver waiting to be transmitted that does not affect the FIFO **full** bit.
	1:	When transmitting (at event 35) the state of the **full** bit includes the byte being transmitted. There are no extra bytes to be accounted for at abort time.
<1:0>	R/RW	This field is a "don't care" for a PWord size of 1 byte. For Host Recovery situations these bits indicate what fraction of a PWord was not transmitted so that software can retransmit the unsent bytes. If the PWord size is 2 or 4 bytes, the value of these two bits is a snapshot of the last PWord being transmitted in mode 011 (event 35) when the FIFO was reset (port was transitioned from mode 011 to mode 000 or 001).
	00:	The PWord at the head of the FIFO contained a complete PWord.
	01:	The PWord at the head of the FIFO contained only 1 valid byte.
	10:	The PWord at the head of the FIFO contained 2 valid bytes.
	11:	The PWord at the head of the FIFO contained 3 valid bytes.

cnfgB

0x401, mode = 111 (Configuration Register B)

This register allows software to control the selecting of interrupts and DMA channels. A read-write implementation implies a "software-configurable" device. All ISA ports must implement this as a read-only register as a minimum.

Some or all of the bits may be read-only; for example, if the port is configurable but only supports 8-bit DMA transfers, then <2> will be a read-only bit set to a 0, while <1:0> may be (R/W). Likewise, any or all of the interrupt bits may be read-only.

If a value is not set to 000 (the jumper-default) then it is assumed that the value in the register is correct and software will use this interrupt and/or DMA channel.

Table 13. Configuration Register B

<7>	R/W	**compress.** This is an "optional" feature and need not be implemented. If not implemented, the bit should be read-only and always return a 0 when read. When this bit is 0 compression will not occur.
	1:	When set, this bit causes the sending state machine to compress the data before sending. All devices supporting compression must implement this bit.
	0:	(Default) The transmitter shall send only uncompressed (raw) data in this case.
<6>	R	**intrValue.** Returns the value on the ISA **iRq** line to determine possible conflicts.
<5:3>	R-R/W	**intrLine.**
	111:	Selects IRQ 5.
	110:	Selects IRQ 15.
	101:	Selects IRQ 14.
	100:	Selects IRQ 11.
	011:	Selects IRQ 10.
	010:	Selects IRQ 9.
	001:	Selects IRQ 7 (default).
	000:	If read-only, indicates that the interrupt must be selected with jumpers.
<2:0>	R-R/W	**dmaChannel.**
	111:	Selects DMA channel 7.
	110:	Selects DMA channel 6.
	101:	Selects DMA channel 5 (default, 16-bit).
	100:	Indicates jumpered 16-bit DMA if read-only.
	011:	Selects DMA channel 3 (default, 8-bit).
	010:	Selects DMA channel 2.
	001:	Selects DMA channel 1.
	000:	Indicates jumpered 8-bit DMA if read-only.

ecr

0x402 (Extended Control Register).
This register controls the extended ECP/parallel port functions.

Table 14. Extended Control Register

<7:5>	R/W	**mode**
	000:	*Standard Parallel Port mode*. In this mode the FIFO is reset and common collector drivers are used on the control lines (**nStrobe**, **nAutoFd**, **nInit**, and **nSelectIn**). Setting the **direction** bit will not tri-state the output drivers in this mode.
	001:	*PS/2 Parallel Port mode*. Same as above except that **direction** may be used to tri-state the **data** lines, and reading the **data** register returns the value on the **data** lines and not the value in the **data** register. It is always best for the hardware design to read the value of the lines and not the register. (Some old Centronics interfaces actually returned the reg value and not the wire value.) All drivers have active pull-ups (push-pull).
	010:	*Parallel Port FIFO mode*. This is the same as **000** except that PWords are written or DMAed to the FIFO. FIFO data is automatically transmitted using the standard parallel port protocol. Note that this mode is only useful when **direction** is 0. All drivers have active pull-ups (push-pull).
	011:	*ECP Parallel Port mode*. In the forward direction (**direction** is 0), PWords placed into the **ecpDFifo** and bytes written to the **ecpAFifo** are placed in a single FIFO and transmitted automatically to the peripheral using ECP

Protocol. In the reverse direction (**direction** is 1), bytes are moved from the ECP parallel port and packed into PWords in the **ecpDFifo**. All drivers have active pull-ups (push-pull).

100: *Vendor-specified function.*

101: *Vendor-specified function.*

110: *Test mode*. In this mode the FIFO may be written and read, but the data will not be transmitted on the parallel port.

111: *Configuration mode*. In this mode the **cnfgA** and **cnfgB** registers are accessible at 0x400 and 0x401.

<4> R/W **nErrIntrEn** (valid only in ECP mode)

1: Disables the interrupt generated on the asserting edge of **nFault**.

0: Enables an interrupt pulse on the high to low edge of **nFault**. Note that an interrupt will be generated if **nFault** is asserted (interrupting) and this bit is written from a 1 to a 0. This prevents interrupts from being lost in the time between the read of the **ecr** and the write of the **ecr**.

<3> R/W **dmaEn**

1: Enables DMA (DMA starts when **serviceIntr** is 0).

0: Disables DMA unconditionally.

<2> R/W **serviceIntr**

1: Disables DMA and all of the service interrupts.

0: Enables one of the following 3 cases of interrupts. Once one of the 3 service interrupts has occurred, **serviceIntr** bit shall be set to a 1 by hardware. Writing this bit to a 1 will not cause an interrupt.

case dmaEn=1: During DMA (this bit is set to a 1 when terminal count is reached).

case dmaEn=0 direction=0: This bit shall be set to 1 whenever there are **writeIntrThreshold** or more PWords free in the FIFO.

case dmaEn=0 direction=1: This bit shall be set to 1 whenever there are **readIntrThreshold** or more valid PWords to be read from the FIFO.

<1> R **full**

1: **direction** = 0. The FIFO cannot accept another PWord.

1: **direction** = 1. The FIFO is completely full.

0: **direction** = 0. The FIFO has at least 1 free PWord.

0: **direction** = 1. The FIFO has at least 1 free byte.

<0> R **empty**

1: **direction** = 0. The FIFO is completely empty.

1: **direction** = 1. The FIFO contains less than 1 PWord of data.

0: **direction** = 0. The FIFO contains at least 1 byte of data.

0: **direction** = 1. The FIFO contains at least 1 PWord of data.

Operation

Interrupts

An interrupt shall be generated in the following cases:

- When **serviceIntr** is 0, **dmaEn** is 1, and the DMA reaches a terminal count.
- When **serviceIntr** is 0, **dmaEn** is 0, **direction** is 0, and there are **writeIntrThreshold** or more free PWords in the FIFO. Note that this means an interrupt will be generated when **serviceIntr** is cleared to 0 whenever there are **writeIntrThreshold** or more free PWords in the FIFO.

- When **serviceIntr** is 0, **dmaEn** is 0, **direction** is 1, and there are **readIntrThreshold** or more full PWords in the FIFO. Note that this means an interrupt will be generated when **serviceIntr** is cleared to 0 whenever there are **readIntrThreshold** or more full PWords in the FIFO.
- When **nErrIntrEn** is 0 and **nFault** transitions from high to low, or when **nErrIntrEn** is set from 1 to 0 and **nFault** is asserted.
- When **ackIntEn** is 1 the way existing parallel ports implement this today. The interrupt generated is ISA-friendly in that it may pulse the interrupt line low; optionally it may also drive a level (see Appendix B).

Mode Switching/Software Control

Software will execute P1284 negotiation and all operation prior to a data transfer phase under programmed I/O control (mode 000 or 001). Hardware provides an automatic control line handshake, moving data between the FIFO and the ECP port only in the data transfer phase (modes 011 or 010).

Setting the mode to 011 or 010 will cause the hardware to initiate data transfer.

If the port is in mode 000 or 001 it may switch to any other mode. If the port is not in mode 000 or 001 it can only be switched into mode 000 or 001. The **direction** can only be changed in mode 001.

Once in an extended forward mode, the software should wait for the FIFO to be **empty** before switching back to mode 000 or 001. In this case all control signals will be deasserted before the mode switch. In an **ECP** reverse mode the software waits for all the data to be read from the FIFO before changing back to mode 000 or 001. Since the automatic hardware **ECP** reverse handshake only cares about the state of the FIFO, it may have acquired extra data that will be discarded. It may in fact be in the middle of a transfer when the mode is changed back to 000 or 001; in this case, the port will deassert nAutoFd independent of the state of the transfer. The design will not cause glitches on the handshake signals if the software meets the constraints above.

ECP Operation

Prior to ECP operation the Host must negotiate on the parallel port to determine if the peripheral supports the ECP protocol. Consult Section 2 for details. This is a somewhat complex negotiation carried out under program control in mode 000.

After negotiation, it is necessary to initialize some of the port bits. The following are required:

- Set **Direction** = 0, enabling the drivers.
- Set **strobe** = 0, causing the **nStrobe** signal to default to the deasserted state.
- Set **autoFd** = 0, causing the **nAutoFd** signal to default to the deasserted state.
- Set **mode** = 011 (ECP mode)

ECP address/RLE bytes or data PWords may be sent automatically by writing the **ecpAFifo** or **ecpDFifo**, respectively.

Note that all FIFO data transfers are PWord-wide and PWord-aligned. Address/RLE transfers are byte-wide and only allowed in the forward direction.

The host may switch directions by first switching to mode = 001, negotiating for the forward or reverse channel, setting **direction** to 1 or 0, then setting mode = 011. When **direction** is 1 the hardware shall handshake for each ECP read data byte and attempt to fill the FIFO. PWords may then be read from the **ecpDFifo** as long as it is not **empty**.

ECP transfers may also be accomplished (albeit slowly) by handshaking individual bytes under program control in mode = 001 or 000.

DMA

DMA uses the standard PC DMA services. The software first sets up the direction and state as

in the programmed I/O case. Then it programs the DMA controller in the PC with the desired count and memory address. Then it sets **dmaEn** to 1 and **serviceIntr** to 0. Lastly, it unmasks the DMA at the DMA controller. The DMA will empty or fill the FIFO using the appropriate **direction** and **mode**.

DMA is always to or from the FIFO located at 0x400.

When the terminal count in the DMA controller is reached, an interrupt is generated and **serviceIntr** is asserted, disabling DMA. In order to prevent possible blocking of refresh requests, **dReq** shall not be asserted for more than 32 DMA cycles in a row.

DMA may be disabled in the middle of a transfer by first disabling the host DMA controller, then setting **serviceIntr** to 1, setting **dmaEn** to 0, and waiting for the FIFO to become **empty** or **full**. Restarting the DMA is accomplished by enabling DMA in the host, setting **dmaEn** to 1, and setting **serviceIntr** to 0.

After the end of a DMA transfer in the forward direction, software must wait until the FIFO is **empty** and the state of the **busy** line (visible in the **dsr)** is low. This ensures that all data has been transmitted to the peripheral.

Interrupt-Driven Programmed I/O

The ECP or parallel port FIFOs may also be operated using interrupt-driven programmed I/O. In the forward direction an interrupt occurs when **serviceIntr** is 0 and there are **writeIntrThreshold** or more PWords free in the FIFO. At this time, if the FIFO is **empty** it can be filled with a single burst before the **empty** bit needs to be re-read. Otherwise it can be filled with **writeIntrThreshold** PWords.

In the reverse direction, an interrupt occurs when **serviceIntr** is 0 and **readIntrThreshold** PWords are available in the FIFO. If at this time the FIFO is **full**, it can be emptied completely in a single burst, otherwise **readIntrThreshold** PWords may be read from the FIFO in a single burst.

Software can determine the **writeIntrThreshold**, **readIntrThreshold**, and FIFO depth by accessing the FIFO in Test mode.

Real Time Constraints

Though we cannot constrain the PC ISA bus itself, we can constrain the ECP port interface so that it operates as fast as possible. Some PC ISA implementations can be adjusted in software to speed up the DMA or I/O transfer. An implementation must ensure, however, that whatever bandwidth is on the ISA is not lost in the interface.

Timing Diagrams

Standard Parallel Port

The standard parallel port is run at or near the peak 500 Kbytes/sec allowed in the forward direction using DMA. The state machine does not examine **nAck** and begins the next DMA based on **Busy**.

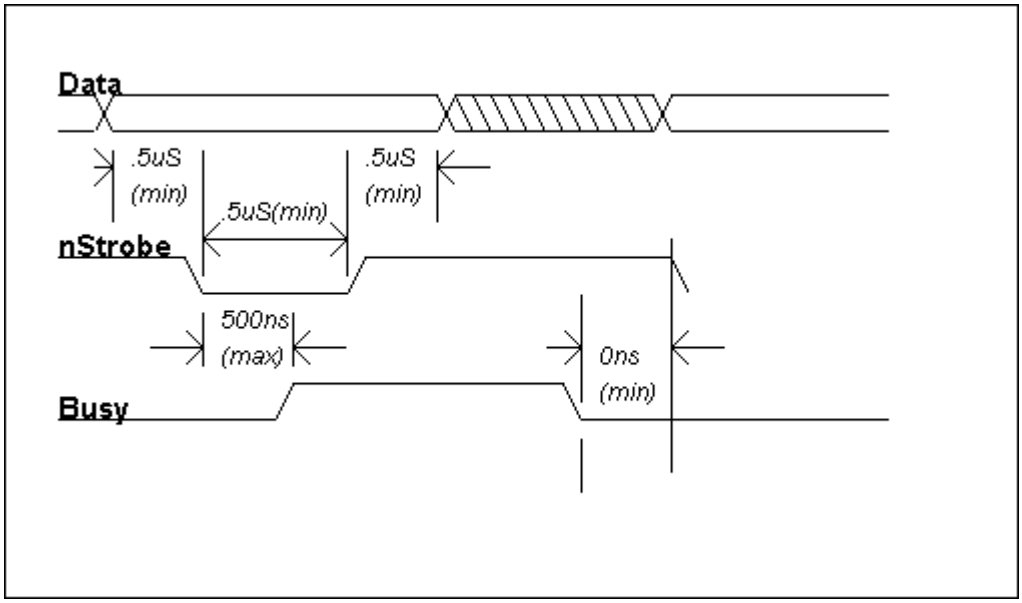

Figure 5. Standard parallel port timing diagram

ISA Interface

I/O read/write

The I/O read, write, and DMA timings on the ISA bus are well known industry standards. This document does not describe them.

ECP parallel port timing

The timing is designed to allow operation at approximately 2.0 Mbytes/sec over a 15-foot cable. If a shorter cable is used then the bandwidth will increase. The maximum timings from the host side are *required*. The timings should be as tight as possible to enhance performance.

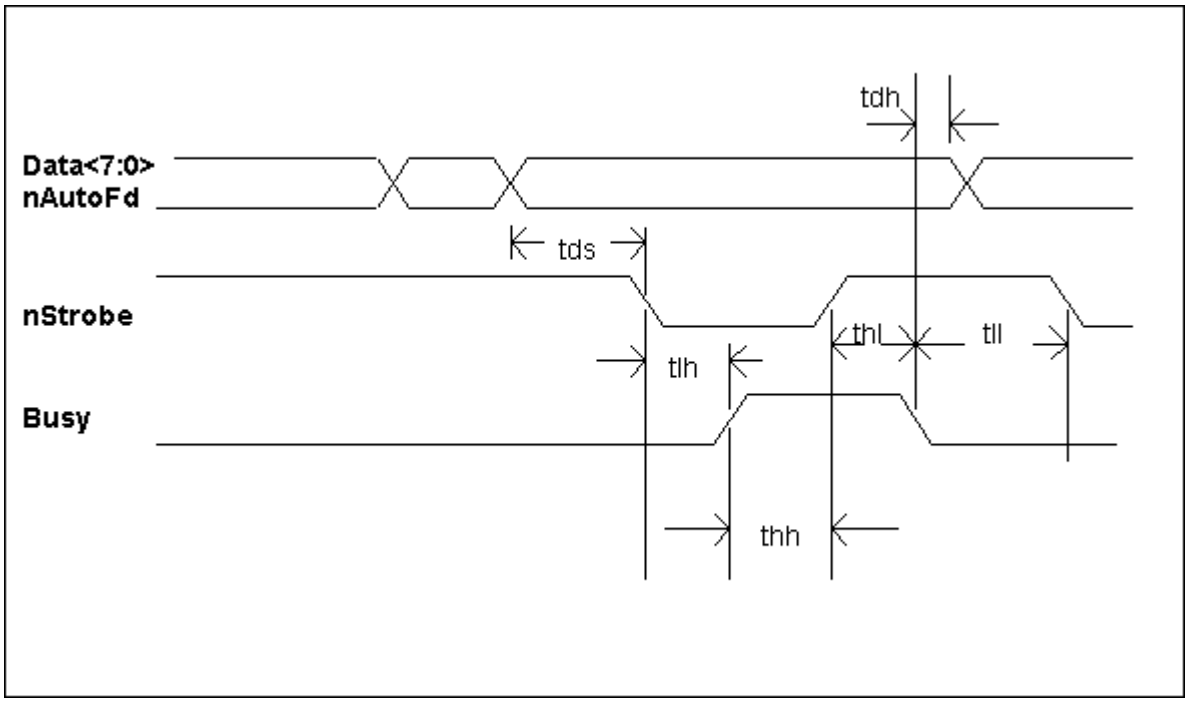

Figure 6. ECP parallel port forward timing diagram

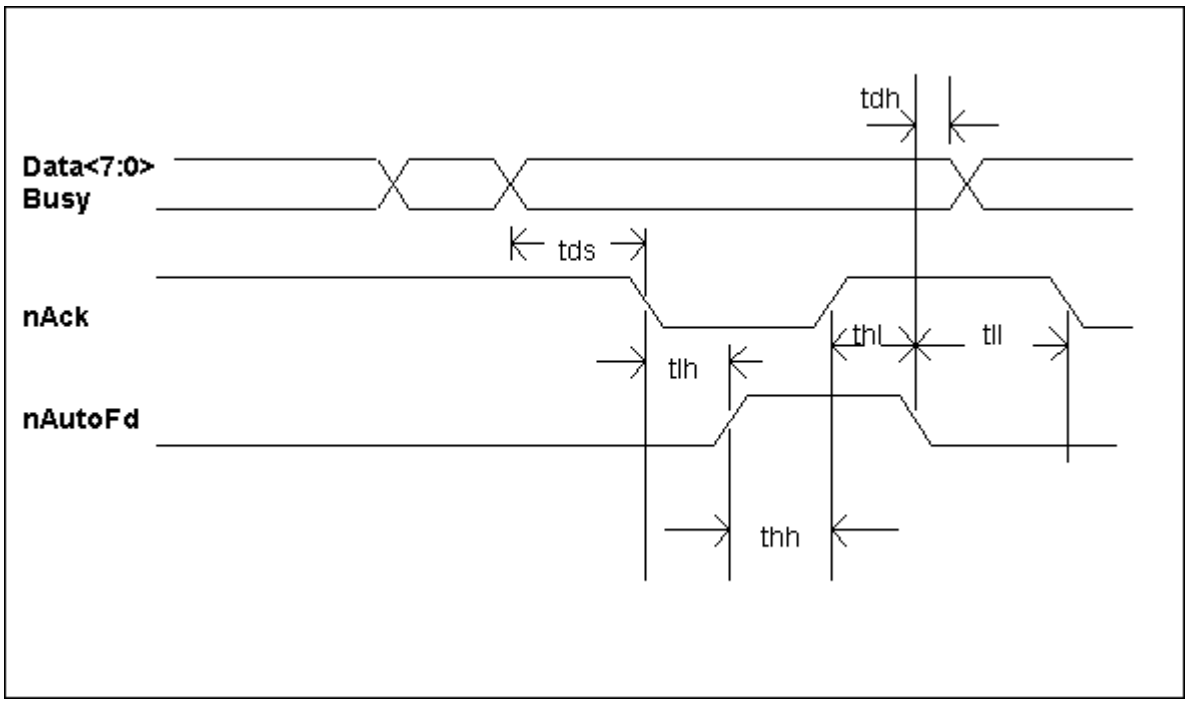

Figure 7. ECP parallel port backward timing diagram

Electrical Information

Table 15. Timing Characteristics

Parameter	Location	Minimum	Maximum
tds	driver	0	infinite
tdh	driver	0	infinite
tlh	receiver	75ns	infinite
thh	driver	0	1.0s
thl	receiver	0	35ms
tll	receiver	0	infinite

Preliminary Discussion of Cable and Driver Characteristics

There are several problems designing the cable-driver-receiver system. First, there is the problem of impedance matching. We measured a number of cables from various manufacturers in various lengths. All cables were shielded with at least a foil shield. Most cables were of the foil shield, 18-wire variety (one ground line).

While switching more than one line, the impedance of the cables varied from 49 to 100 Ω. With one-line switching, all cables presented a more uniform 49-62 Ω impedance.

Thus, in some cases we can implement incident wave switching on the handshake lines. The data lines will incident-wave-switch given a good cable, but data setup time is added to allow poor cables (six feet in length and under) to function properly.

Unfortunately, incident wave switching works only when there is no significant inductive coupling from other signal lines. This is true as long as there is only one-line switching in the system. When the data lines switch, a noise spike is generated on other signal lines (that is, **nStrobe** and **nAck**).

In order to guarantee noise immunity, the receiver shall ensure that the strobe (either **nAck** or **nStrobe**) has been asserted for at least tlh ns before responding. This serves to make the system more impervious to inductive switching noise.

Likewise, the driver port state machine will not examine the control handshake response lines until the data has had time to switch.

Some handshake signals (tll, thl, thh) may be assumed to incident-switch since there is no noise coupling and a reasonable impedance match.

The RC network is designed to provide an RF filter, and to match the cable's impedance. The capacitance has the effect of lowering the characteristic impedance, and this is matched by the lower driver impedance range. Of major importance is that the capacitance on both the driver and receiver be of the same value to attain operation without reflections.

Table 16. Cable Requirements

Cable (Shielded)	Minimum	Maximum
Z^0 (multiple switching, length <= 6 feet)	48 Ω	100 Ω
Z^0 (multiple switching, length > 6 feet)	48 Ω	65 Ω

DC Characteristics

Table 17. DC Characteristics

Rs Driver (pull-down)	30	50	Ω
VCC	4.75	5.25	V
Vil		1.0	V
Vih	2.0	VCC	V

Table 18. Capacitance

Capacitor	pf	tolerance
C^{data}	180	10%
$C^{control}$	180	10%

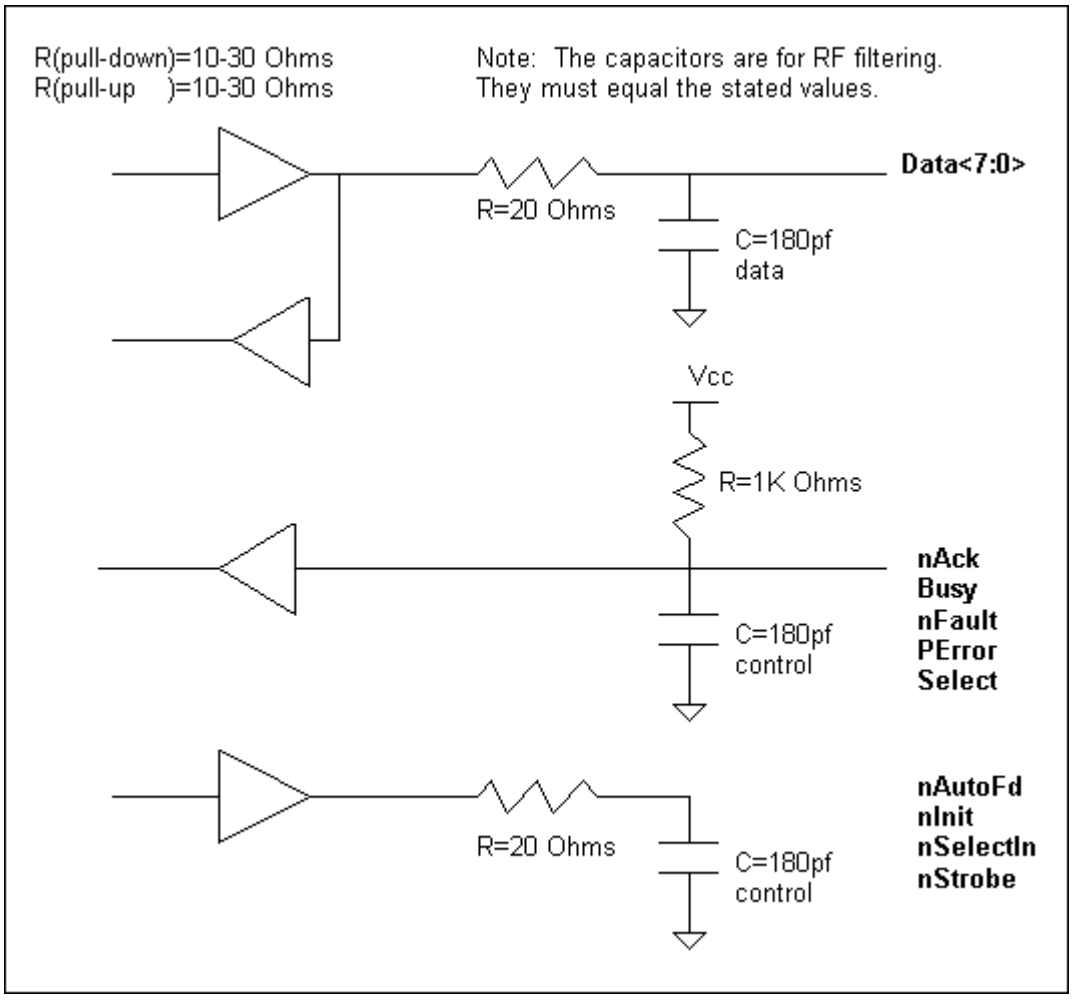

Figure 8. Driver electronics diagram: One implementation of this specification

Appendix A: Peripheral-Side Design Note

The specifications and guidelines described in this document are specific to ISA implementation on a host PC system. Peripherals tend to have bus architecture that has nothing in common with the host PC architecture.

Although this document is not meant to provide a guideline for ECP port design on a peripheral, you need to keep the following special consideration in mind.

The peripheral must always return data when a byte is requested. Peripheral devices that know the total number of bytes to be transmitted must not stop sending data after that number of bytes is reached. Additional "don't care" data bytes must be sent to "pad" the transfer. The host interface will discard any extra bytes received. The extra "don't care" bytes are useful to provide alignment to wider busses (that is, 16-bit, 32-bit, 64-bit).

Use of nFault (nPeripheralRequest)

When a peripheral wishes to interrupt the host, it asserts nFault low. It will hold nFault low until service is established or until a negotiation phase. If faster latency is desired, it is a good idea to continue to accept or transmit data while nFault is asserted. If latency is not a problem (that is, an error interrupt) then stalling the transfer and asserting nFault is acceptable.

Appendix B: Known Enhancements to This Standard

Detection of FIFO State Errors

At least one port design has implemented a feature that generates an error if the FIFO is overwritten in the forward direction or overread in the reverse direction. This error should never occur with proper hardware and software design. In the case of the error, some designs may set *both* the **full** and **empty** bits in the **ecr** and generate a service interrupt (setting the service interrupt bit).

In order to clear the FIFO error condition, software shall exit ECP mode, placing the port instead into Standard (000) or PS2 (001) mode.

This feature is noted here so that drivers may be able to interpret the event properly.

Use of Nonpulsed (Level-Triggered) Interrupts

The original design of this port generated pulses on each interrupt event. This will work fine on ISA machines, but some designers wish to make the port function in the standard level ISA fashion. In order to do this, the level interrupt is enabled (driven low) whenever the device is in ECP, Test, or Centronics FIFO mode. When an interrupt condition exists, the signal is driven high.

After receiving, the interrupt driver will read the ECR to determine the cause of the interrupt. It then writes the ECR, setting the **serviceIntr** bit to 1 and the **nErrIntrEn** bit to 1. This masks all interrupt sources and causes the **iRq** line to go low. After servicing the Interrupt, the driver will re-enable interrupts, if desired, by writing **serviceIntr** and/or **nErrIntrEn** bits to 0.

After the completion of each DMA transfer (terminal count), the driver shall first write **dmaEn** to 0 before beginning another DMA transfer.

The software driver shall ensure that the interrupt due to **nAck** is disabled whenever the port is using ECP protocol or ECP mode to transfer data.

Appendix C: Host Recovery of a Forward Transfer at (Event 35)

Background

Long after this specification was implemented, IEEE noticed a potential problem for some designs: It was possible for some peripherals to stall forever in event 35 and there was no way for the host to legally "break out" of the forward transfer without data loss and protocol violation. The specification was modified to allow for a Host Recovery phase.

Software can easily implement the recovery handshake; however, software must determine the total number of bytes that remain in the FIFO first so they can be retransmitted. The basic idea is to fill the FIFO so we know how many bytes *were* in the FIFO. Then software adjusts for any partially transmitted PWords or bytes that may be in an output transceiver stage. This byte adjustment data is placed in **cnfgA**<2:0>.

Software will perform the following steps to recover from a Forward Data Transfer at event 35:

[The ECP port is in mode 011, stuck on event 35, trying to transmit, and we want to recover.]

1. Write to the **dcr** driving the **nStrobe** signal low. This prevents further data transfers even if the peripheral starts accepting data.
2. The number of PWords in the FIFO at abort time is computed by writing PWords to the FIFO until the **full** is set to 1. For PWord sizes of 2 and 4 bytes, the PWord at the head of the FIFO may be partially transmitted.
3. The host writes the **ecr** and sets the mode to 001. This causes the port to reset the FIFO and load FIFO state information into **cnfgA**<1:0>. This will be used by software to determine how many bytes remain untransmitted in the PWord at the head of the FIFO.

   Note Steps 4 through 8 describe the Recovery Handshake.
4. The host tri-states its drivers by writing the **dcr direction** bit to 1.
5. The host writes the **dcr** setting **nInit** low and waits for **dsr PE** signal to go low.
6. The host writes the **dcr** setting **nStrobe** high.
7. The host writes the **dcr** setting **nInit** high and waits for **dsr PE** signal to go high.
8. The host has recovered and is in the Forward Idle phase. It now reads **cnfgA** and computes the actual number of bytes that need to be retransmitted. The software takes the number of PWords in the FIFO, and adjusts for a partial PWord and for a byte in the output transceiver by using the values in **cnfgA**<2:0>.

Example 1

Suppose we have a byte-wide interface, a FIFO depth of 32 PWords, and **cnfgA**<2:0> = 011b. At abort time we write 2 bytes to the FIFO to make it **full**. The value in **cnfgA**<1:0> does not matter, since this is a byte-wide interface; however, the 0 in **cnfgA**<2> indicates that there is a byte in the output transceiver we must count. Thus the total number of bytes to be re-sent is:

Bytes to be reset = (32-2) + 1 (transceiver byte)

Example 2

Suppose we have a word-wide interface, a FIFO depth of 32 PWords, and **cnfgA**<2:0> = 111b. At abort time we find that 2 more PWords make the FIFO **full**. The value in **cnfgA**<1> does not matter, since this is a word-wide interface. The 1 in **cnfgA**<2> indicates that there are no bytes in the output transceiver, and the 1 in **cnfgA**<0> indicates that the PWord at the head of the FIFO is partial, containing only 1 byte. Thus:

PWords in FIFO = 32 - 2 = 30

One of these is a partial PWord, thus:

- Complete PWords in FIFO = 29
- Partial PWords in FIFO = 1
- Bytes to be reset = (29) * 2 + 0(transceiver byte)+ 1(partial PWord) = 59 bytes

Example 3

Suppose we have a 32-bit-wide interface, a FIFO depth of 16 PWords, and **cnfgA**<2:0> = 011b. At abort time we write 4 PWords to fill the FIFO; thus we know that there are 12 PWord entries in the FIFO used. The value in **cnfgA** indicates that there is 1 byte in the transceiver and the PWord in the FIFO being transmitted contains only 3 bytes. Thus:

- PWords in FIFO = 16 - 4 = 12
- Complete PWords in FIFO = 11
- Bytes to be re-sent = 4 * 11 + 3 (partial bytes in PWord being transmitted) + 1 (transceiver byte) = 48 bytes

ECP Driver Hardware Notes

Revision: 1.1
July 20, 1993

Microsoft assumes no liability for direct, indirect, incidental, consequential, or other damages arising from the use of this specification contained herein, and reserves the right to update, revise, or change any information in this document without notice.

Document History

Revision	Date	Action	Author(s)
1.0	6-1-93	Wrote first draft of notes	Dave Voth
1.1	7-9-93	Updated Abort Section to reflect IEEE spec wording	Dave Voth

Introduction

Document Purpose

This document is a set of notes to aid the software engineer in writing drivers for the ECP/ISA interface. It attempts to impart an understanding of the hardware and answer some commonly asked questions about writing ECP drivers.

The reader of this document should be aware that the port specification, not this document, is the official source of information. If there are any conflicts between this document and the ECP ISA port specification, this document should be considered in error.

Other Documents

The reader should be familiar with the *Standard P1284* by IEEE and the Microsoft/Hewlett-Packard *ISA ECP Port Specification*.

Scope

This document is useful for all compatible ECP ISA port devices.

Overview

The ECP ISA parallel port is an enhancement to the existing port. It offers hardware-assisted data transfer in any of several modes. The hardware assist accelerates the transfer by providing handshaking on the control lines reducing the software burden.

The driver is responsible for port detection, configuration, and any data transfer and/or alignment operations. The data transfer operations may use interrupts and/or DMA.

API Interface

The actual mechanism a driver will use for receiving or transmitting data from an application through the operating system is beyond the scope of this document. What is specified here is the required range of API functionality and those functions that the interface must provide, and not exactly how they are implemented. The exact parameters and return values are not defined here. For example, many of the functions described here will be carried out by an **ioctl** procedure and may not have specific procedure calls of their own.

The drivers *shall* be able to implement the following interface functionality:

API Function and Purpose	Before Call	After Call	Method of Operation
Initialize()	Undefined	Compatibility	Not applicable
Open()–Negotiates the peripheral into P1284 ECP mode.	Compatibility mode	Forward-Idle	Not applicable
Close()–Terminates any ongoing operations and negotiates the peripheral into compatibility mode.	Any ECP phase	Compatibility mode	Not applicable
SetChannelAddress(0..0x7f)–Sets the peripherals data channel to a specified value.	Any ECP phase	Forward-Idle	Hardware assist or software emulation
ReadString(char* lpsBuffer, int byteCount)–Negotiates to reverse, reads byteCount or more bytes from the peripheral and places byteCount bytes into lpsBuffer. Then negotiates back.	Any ECP phase	Reverse	Hardware assist (DMA or Intr or polled)
WriteString(char * lpsBuffer, int byteCount)–Writes the specified number of bytes from lpsBuffer to the peripheral.	Any ECP phase	Forward-Idle	Hardware assist (DMA or interrupt or polled)
ReadByte()–reads a byte from the peripheral.	Any ECP phase	Reverse	Hardware assist
WriteByte()–Writes a byte to the peripheral.	Any ECP phase	Forward-Idle	Software emulation or hardware assist
Abort()	Any ECP phase	Forward-Idle	Software emulation
SetAbortTimeout(int seconds)–Sets the time the driver must wait before it Abort()s the transfer if no progress is being made.	Forward or Reverse-Transfer phase	Forward-Idle	Software emulation

Port Detection: Initialize()

Prior to any driver being installed, or perhaps as a check of the installed driver, software must check the ISA port for the presence of an ECP ISA port. This can be done in many ways, but a couple of key features are noteworthy:

- Generally the port is detected by the presence of the ECR register.
- This must be done after boot time and the port must be in an idle state.

1. Read the ECR and verify that the Full bit is a 0 and the Empty bit is a 1. Verify that this differs from the corresponding bit-position values in the **dcr**.
2. Now we know that the Port is probably an ECP port and we can try to do some additional testing. Write the ECR to 0x34 and read the ECR: It should contain 0x35 (Full and Empty are read-only status).

Port Information

After the port is detected, it is important to determine some of the attributes of the port. This is done by writing the ECP register with 0xf4, placing the port in Configuration mode.

Reading **cnfgA**, you determine PWord size (if the port is 1, 2, or 4 bytes wide). This knowledge is essential for setting up DMA and programmed I/O transfers.

Bit 7 of configuration A register is useful for setting up the **nFault** interrupt. If this bit is set to 1 then the driver may enable both **nFault** and service interrupt at the same time. If this bit is 0 then the driver must allow only one interrupt type at a time (more on this later). In any case, this information is usually collected once at boot time.

Next the driver should determine if the port supports hardware compression. The **compress** bit field allows software to turn off (or on) any hardware compression. Software should attempt to write **compress** to 1 and then read it back. Devices not supporting hardware compression will return a 0 while devices supporting compression will allow this bit to be written to a 1. The **compression** bit should then be initialized to 0 (until such a time as a compressed data transfer is desired).

Bit 6 of **cnfgB** is useful for detecting interrupt line conflicts. If this bit is 0 when read, we are guaranteed to have an interrupt conflict. The software should report the conflict and not enable interrupts. If bit 6 is read as a 1 then there may or may not be an interrupt conflict.

The other bit fields in **cnfgB** (**intrLine**, **dmaChannel**) were intended to allow for a software-configurable hardware implementation. Unfortunately, most (if not all) designs do not have enough pins to implement this and will return read-only 0s in this register.

Software should attempt to write **intrLine** and **dmaChannel** to the state defined in the win.ini file. If undefined, then the default state (Intr channel 7, DMA channel 3 or 5) should be tried. Software must try to detect resource conflicts and ensure that no other hardware is sharing the resources. In any case the **iRq** , **dRq** selection process must be modifiable via the windows ports dialog box, win.ini file, so the user can override the default settings.

Direction Bit and Mode 000, 001

Beware! Setting the port to mode 000 may clear the direction bit on some designs. For example, if software writes direction bit to 1 in mode 000, then sets the mode to 011 (ECP mode), the direction bit will still remain 0 and the ECP mode will be in the forward direction. Software, to avoid this potential problem, should not use mode 000, but use only mode 001 as the direction bit is writeable in mode 001. Mode 001 is identical to mode 000 except the direction bit functions properly.

FIFO Information and Test

The size of the FIFO and the interrupt thresholds can be determined in test mode. These parameters are acquired via the following sequence of events:

- The chip is placed in forward test mode with the interrupts off. The FIFO is written one PWord at a time until the **full** bit is set. The total number of PWords written equals the size of the FIFO, which is assumed to be the same in either direction.
- The **serviceIntr** bit is written first to 1 then 0, enabling the service interrupt (the FIFO is still full from the previous step). PWords are then read from the FIFO until a service interrupt occurs (**serviceIntr** becomes set to 1). The number of PWords read from the FIFO is equal to the **writeIntrThreshold**.
- Software should verify that the interrupt was in fact registered at the PIC by installing an interrupt handler and counting the interrupts. The **writeIntrThreshold** indicates the number

of bytes that may safely be written when an interrupt occurs.

- Software then initializes the FIFO by setting the mode field to 001, and places the port in the *reverse* direction by setting the **direction** bit to 1 in the **dcr**. The port is then placed back in test mode and service interrupt is written first to 1, then to 0 (enabling the interrupt). PWords are then written to the FIFO until a service interrupt occurs (**serviceIntr** becomes set to 1). The number of PWords written to the FIFO is equal to the **readIntrThreshold**.
- Software should verify that the interrupt was in fact registered at the PIC by installing an interrupt handler and counting the interrupts. The **readIntrThreshold** indicates the number of bytes that may be safely read when an interrupt occurs.

Port Initialization (Open)

When an application opens a channel through the port, software shall use the P1284 negotiation to place the peripheral in the Forward-Idle state. The **ecr** should be set to 0x34 by the time this call returns.

Forward Transfer

WriteByte()

Software will execute the proper negotiation sequence to place the peripheral into the Forward-Idle state if it needs to. Software may then, using the **dcr**/**dsr** and data register, write a byte to the peripheral. Software may also use the hardware to assist the data transfer if the PWord size is 1 byte. This is done by writing to the FIFO when the **ecr** is set to ECP mode with the byte to be transmitted.

To ensure that the byte has been completely transmitted in ECP mode, or prior to changing the mode bits, the FIFO must be **empty** and the state of **busy** must be low.

SetChannelAddress(0..0x7f)

Software will execute the proper negotiation sequence to place the peripheral into the Forward-Idle state if it needs to. Software may then, using the DCR/DSR and data register, write a byte to the peripheral. It may also write to the **aFifo** if it has already set up the port in ECP mode (011).

WriteString()-Programmed I/O

Because the channel has been established via the Open call, the job of the driver at this point is very specific. If the driver does not use interrupts or DMA it may use the FIFO and automatic hardware handshake provided by the hardware to transfer data.

1. Set Port in ECP mode.
2. Use String Repeat Operation to move the desired number of PWords to the FIFO. If there are extra bytes left over due to byte-PWord misalignment, the driver must change to STD mode after the FIFO is **empty** and then transfer the remaining bytes using programmed I/O to the **dcr**/**dsr**/**data** registers.
3. When the FIFO is empty and the state of the busy signal is low, the data has been transferred and the operation is over. Note that if for any reason both the **empty** and **full** bits are simultaneously set, an error has occurred and should be reported.
4. IF **nFault** in the DSR is asserted, a message or some signaling mechanism must be passed back to a user-defined service routine for handling.

WriteString()-Interrupt-Driven Programmed I/O Forward Data Transfer

An interrupt may be used to indicate that the port requires service (more data) for string transfers that exceed a FIFO size. Of course, prior to the transfer an interrupt service routine must be installed and the port set up properly. The interrupt service routine handles the interrupt and refills the FIFO with data. The steps below must be followed in order by the interrupt handler:

1. Read **ecr**. If **serviceIntr** is not set, then the interrupt was false and the service routine should exit. Read DSR if an **nFault** interrupt is asserting, then send a Message to signal that fact to the peripheral.
2. Write **ecr** to 0x74. This sets **serviceIntr** to 1 and **nErrIntrEn** to 1, masking the interrupts.
3. If the **empty** bit was set in the ECR, then FIFO-size PWords may be written to FIFO, otherwise **writeIntrThreshold** PWords may be written to the FIFO. A string repeat command of PWord width should be used to maximize performance.
4. Hardware will move the FIFO data to the peripheral using the ECP mode.
5. The **ecr** may be read again, if desired, to see if **empty** is 1. If so, the hardware has completely sent the previous data burst and the FIFO may be filled with a FIFO-size burst transfer. Rechecking the **empty** a few times will greatly improve the high-end performance, as the interrupt latency is usually around 100ms.
6. If more data remains to be transferred and the FIFO is not **empty**, the interrupt service routine should enable interrupts by unmasking the service interrupt (and possibly the **nFault** interrupt) just before returning from the interrupt routine.

Cleanup

After the transfer (remaining byte count < PWord) to the FIFO has been completed, the driver *must* verify that the **empty** bit is 1 in the **ecr** and the state of **busy** is low, to ensure that the last data has been transferred. If software changes the mode field in the **ecr** prior to **empty**, the data in the FIFO will be lost.

If a misalignment exists (remaining count != 0 < PWord) between the PWord size and number of bytes to be transmitted at the end of a transfer, the driver must not use the FIFO to transfer the final misaligned byte(s). It must do so in STD mode using the DSR/DCR/DATA registers.

WriteString()-DMA-Driven Forward Data Transfer

DMA may be used to move a specific amount of data. Of course, to use DMA the device must have the use of a DMA channel (**dRq**) and interrupt (**iRq**). Software should do some sanity checking sometime in the process to ensure that the device is actually connected to the channel and interrupt.

The steps below must be followed in order to set up the DMA:

1. The **ecr** is written to 0x34, clearing the FIFO. The DMA controller is programmed with buffer address and count. The DMA controller remains masked at this time.
2. An interrupt routine is installed to handle the resulting terminal count interrupt. The **ecr** is then written to 0x7C, setting **dmaEn** first to 1 and then to 78, enabling the **serviceIntr**. Lastly, the DMA is unmasked at the DMA controller and data is moved to the peripheral.
3. DMA is complete when a service interrupt occurs. The interrupt handler follows the following steps:
    a. Read **ecr** and check the interrupt sources. If no interrupt occurred then the interrupt was false and the routine should be exited.
    b. Write **ecr** to 0x74. This masks interrupts by setting **serviceIntr** and **nErrIntrEn** to 1.
    c. If the interrupt was caused by the service interrupt, the DMA is complete. The **dmaEn** bit in the **ecr** must then be written to a 0.
    d. Software must wait for the FIFO to become **empty** and the state of **busy** to be low before the transfer is considered complete.
    e. If desirable or necessary, set up another DMA and reenable interrupts as before.

Negotiating from Forward to Reverse

In order to do a reverse ECP transfer, it is necessary to change the phase of the peripheral from forward to reverse. This is done by negotiation and is carried out by the following steps. Be sure to refer to IEEE P1284 and the ECP ISA port specification.

1. Complete the current forward transfer.
2. Place the ECP port into PS2 mode (001).
3. Set the direction bit to 1 (reverse), causing the ECP port data drivers to tri-state.
4. Set the ECP port into ECP mode (011), enabling the hardware assist.
5. Write to the DCR, causing **nInit** to go low. This requests a reverse transfer from the peripheral.
6. The peripheral will drive **pe** low when it has started the reverse transfer. Hardware will automatically move data into the ECP FIFO from the ECP data lines.
7. Set up a ReadString or execute a ReadByte operation.

Reverse Data Transfer

ReadByte()

Software will set the ECP port into hardware-assisted reverse mode (see "Negotiating from Forward to Reverse") and negotiate the peripheral into the Reverse phase if it has not already done so. This will cause the hardware to read data from the peripheral and place it into the FIFO. When the FIFO is not **empty**, a PWord may be read from the FIFO and the byte requested returned.

If a subsequent **Read()** command is issued, the driver may read another PWord burst from the FIFO or use the unused part of a previously read PWord in order to return the byte. If a subsequent **ReadString()** command is issued, the driver sets up and executes the operation. However, in both of these two cases, negotiation must not occur and the mode (ECP = 011) of the port must not be changed, or data loss will result.

If a **Write**, **WriteString**, or **Close** command is issued, the phase and mode of the port will of course be changed.

ReadString()

Software will set the ECP port into hardware-assisted reverse mode (see "Negotiating from Forward to Reverse") and negotiate the peripheral into the Reverse phase if it has not already done so. This will cause the hardware to read data from the peripheral and place it into the FIFO. When the FIFO is not **empty**, a PWord may be read from the FIFO and the byte requested returned.

If a subsequent **Read()** command is issued, the driver may read another PWord burst from the FIFO or use the unused part of a previously read PWord in order to return the byte. If a subsequent **ReadString()** command is issued, the driver sets up and executes the operation. However, in both of these two cases, negotiation must not occur and the mode (ECP = 011) of the port must not be changed, or data loss will result.

If a **Write**, **WriteString**, or **Close** command is issued, the phase and mode of the port will of course be changed.

Interrupt Driven Programmed I/O

One method of moving string data from the FIFO is via interrupt-driven programmed I/O. As data in the FIFO reaches the **readIntrThereshold** it generates a service interrupt. The interrupt routine moves the data from the FIFO to memory. The following sequence should be used by

the interrupt service routine:

1. Read the **ecr** to determine the source of the interrupt. If no interrupt source is found, the interrupt is false and should be ignored.
2. Mask the interrupts by writing **serviceIntr** and **nErrIntrEn** to 1.
3. If the **full** bit in the **ecr** is set, the entire FIFO may be read; otherwise a **readIntrThreshold** number of PWords may be read from the FIFO. A string repeat command should be used to maximize performance.
4. The peripheral will attempt to keep the FIFO full, sending extra "don't care" data to fix any FIFO alignment problems. Thus, software may read more bytes than required and discard any unneeded bytes in order to generate correct alignment. For example, say the host wants to read 5 bytes from a FIFO with a PWord size of 2 bytes. Software reads 3 PWords, discarding the extra byte.
5. When the interrupt service is complete, the interrupts are unmasked if more data transfer is required.

DMA-Driven Reverse Transfer

DMA may be used to move a specific amount of data from the FIFO to memory. Of course, to use DMA, the device must have use of a DMA channel (**dRq**) and interrupt (**iRq**). Software should do some sanity checking sometime in the process to ensure that the device is actually connected to the channel and interrupt.

The steps below must be followed in order to set up the DMA:

1. Prior to negotiation, the DMA controller is programmed with buffer address and count. The DMA controller remains masked at this time.
2. An interrupt routine is installed to handle the resulting terminal count interrupt. The **ecr** is then written to 0x78; this sets (enables) the **serviceIntr** and the DMA. Lastly, the DMA is unmasked at the DMA controller—the DMA is ready to go. Some devices allow the **nFault** interrupt to be safely enabled (0x68) at the same time as **serviceInterrupt** (see discussion of **nFault** interrupt).
3. After negotiation into the reverse direction, the peripheral will fill the FIFO and the DMA will move the data into memory.
4. DMA is complete when a service interrupt occurs. At that time the number of PWords programmed for the DMA transfer have been moved to memory. The interrupt service routine should follow the following steps:
    a. Read **ecr** and check the interrupt sources. If no interrupt occurred, the interrupt was false and the routine should be exited.
    b. Write **ecr** to 0x74. This masks interrupts by setting **serviceIntr** and **nErrIntrEn** to 1.
    c. If the interrupt was caused by the service interrupt, the DMA is complete. The **dmaEn** bit in the **ecr** must then be written to 0.
    d. If the FIFO is not **empty**, software may read another PWord from the FIFO in order to fix an alignment problem. For example, suppose 4K+1 bytes were required from a 2-byte wide PWord interface. The DMA could transfer 4K bytes using DMA, then the software could read 1 additional PWord from the FIFO and extract the extra byte.
    e. If the desired number of bytes is greater than the DMA buffer size, multiple transfers are required. In this case the ECP port *must* be kept in ECP mode until all the data has been moved. Changing modes will delete valid data from the FIFO that *cannot* be recovered.

       In order to do multiple buffer transfers, the DMA controller should be set up for another transfer and enabled (the **dmaEn** bit is set to 1).
    f. In order to restart a multi-buffer transfer, the service interrupts must be unmasked and the interrupt service routine exited.

Reverse to Forward Negotiation

After the ECP port has moved data in ECP mode (011) in the reverse direction and a change of direction is required, the following steps must be taken:

1. First, negotiate the state of the ECP port (the peripheral) back into forward mode. This is done by setting **nInit** high and waiting for the state of **pe** go high. This causes the peripheral to terminate any ongoing reverse transfer.
2. The mode of the ECP port is changed to PS2 mode 001.
3. The direction bit is changed to 0. At this point, the bus and the ECP port are in the forward-idle state.

Abort()

Background

Long after this specification was implemented, IEEE noticed a potential problem for some designs: It was possible for some peripherals to stall forever in event 35 and there was no way for the host to legally "break out" of the forward transfer without data loss and protocol violation. The specification was modified to allow for a Host Recovery phase.

Software can easily implement the recovery handshake; however, software must first determine the total number of bytes that remain in the FIFO so they can be retransmitted. The basic idea is to fill the FIFO so we know how many bytes *were* in the FIFO. Then software adjusts for any partially transmitted PWords or bytes that may be in an output transceiver stage. This byte adjustment data is placed in **cnfgA**<2:0>.

Software will perform the following steps to recover from a Forward Data Transfer at event 35:

1. The ECP port is in mode 011, stuck on event 35, trying to transmit, and we want to recover.
2. Write to the **dcr**, driving the **nStrobe** signal low. This prevents further data transfers, even if the peripheral starts accepting data.
3. The number of PWords in the FIFO at abort time is computed by writing PWords to the FIFO until the **full** is set to 1. For PWord sizes of 2 and 4 bytes, the PWord at the head of the FIFO may be partially transmitted.
4. The host writes the **ecr** and sets the mode to 001. This causes the port to reset the FIFO and load FIFO state information into **cnfgA**<1:0>. This will be used by software to determine how many bytes remain untransmitted in the PWord at the head of the FIFO.

Note Steps 4 through 8 describe the Recovery Handshake.

1. The host tri-states its drivers by writing **dcr direction** bit to 1.
2. The host writes the **dcr**, setting **nInit** low, and waits for **dsr pe** signal to go low.
3. The host writes the **dcr**, setting **nStrobe** high.
4. The host writes the **dcr**, setting **nInit** high, and waits for **dsr pe** signal to go high.
5. The host has recovered and is in the Forward Idle phase. It now reads **cnfgA** and computes the actual number of bytes that need to be retransmitted. The software takes the number of PWords in the FIFO, and adjusts for a partial PWord and for a byte in the output transceiver by using the values in **cnfgA**<2:0>.

Example 1

Suppose we have a byte-wide interface, a FIFO depth of 32 PWords, and **cnfgA**<2:0> = 011b. At abort time we write 2 bytes to the FIFO to make it **full**. The value in **cnfgA**<1:0> does not matter, since this is a byte-wide interface; however, the 0 in **cnfgA**<2> indicates that there is a byte in the output transceiver we must count. Thus the total number of bytes to be re-sent is:

Bytes to be reset = (32-2) + 1 (transceiver byte)

Example 2

Suppose we have a word-wide interface, a FIFO depth of 32 PWords, and **cnfgA**<2:0> = 111b. At abort time we find that 2 more PWords make the FIFO **full**. The value in **cnfgA**<1> does not matter, since this is a word-wide interface. The 1 in **cnfgA**<2> indicates that there are no bytes in the output transceiver, and the 1 in **cnfgA**<0> indicates that the PWord at the head of the FIFO is partial, containing only 1 byte. Thus:

PWords in FIFO = 32 - 2 = 30

One of these is a partial PWord, thus:

- Complete PWords in FIFO = 29
- Partial PWords in FIFO = 1
- Bytes to be reset = (29) * 2 + 0(transceiver byte)+ 1(partial PWord) = 59 bytes

Example 3

Suppose we have a 32-bit-wide interface, a FIFO depth of 16 PWords, and **cnfgA**<2:0> = 011b. At abort time we write 4 PWords to fill the FIFO; thus we know that there are 12 PWord entries in the FIFO used. The value in **cnfgA** indicates that there is 1 byte in the transceiver and the PWord in the FIFO being transmitted contains only 3 bytes. Thus:

- PWords in FIFO = 16 - 4 = 12
- Complete PWords in FIFO = 11
- Bytes to be re-sent = 4 * 11 + 3 (partial bytes in PWord being transmitted) + 1 (transceiver byte) = 48 bytes

When a device aborts a transfer, the duty of the software is to return the port to the Forward-Idle state without data loss. If the port is in the reverse direction, this is done by simply negotiating into Forward-Idle. If the port is Forward phase (stuck in state 35) then the software needs to find out how many bytes are in the FIFO before issuing the abort sequence.

1. Write to the **dcr**, setting **nStrobe** low.
2. Fill the FIFO until it is **full**. At this point there are FIFO-size bytes in the FIFO, and software can determine how many bytes have not been transmitted.
3. Set the **ecr** mode to 001 or 000, resetting the FIFO and returning the port to standard mode.
4. Complete the abort sequence.

Time-out on Transfer

During any transfer, it is entirely possible that the peripheral will stall indefinitely and the interrupt routine may not get called. To report the time-out case (in which data has not been transmitted for a time-out period) a time-out must be provided. The time-out period value is beyond the scope of this document, but is in the range of seconds.

Discussion of nFault Interrupt

The **nFault** interrupt, referred to in the P1284 spec as **nPeriphRequest**, is essentially a level-triggered interrupt that allows the peripheral to interrupt the host. It may do this when moving data or when it is not moving data.

Some ECP designs (denoted by bit 7 = 1 in **cnfgA**) have been designed to allow an **nFault** interrupt to occur with the service interrupt. In this case, it is acceptable for the driver to have both sources enabled at the same time.

When bit 7 = 0 in **cnfgA**, the **nFault** interrupt must always be masked when service interrupt is

enabled. In this case, the software must poll the **nFault** during idle data transfer and examine **nFault** state sometime during the data transfer itself. A good way to provide this support is to add the polling case to the time-out routine so it samples the signal state every 100ms or so.

Software *must* support the case in which bit 7 = 0 in **cnfgA** and *may* support the other case as well in order to improve **nFault** interrupt latency for some designs.

Software *must never* be in a state in which it is unable to report **nFault** interrupts while the peripheral (not the **ecr** mode field) is in ECP mode.

ECP Compliance Test Functional Specification

Revision: 3.1
July 7, 1993

Document History

Revision	Date	Action	Author(s)
0.1	4-14-93	Drafted first copy of Functional Specification	Dave Voth
0.2	4-28-93	Fixed spec bugs reported by testing group	Dave Voth
3.0	4-20-93	Added Misc test, made spec same revision as comply.exe	Dave Voth
3.1	7-7-93	Added the ECPAbortTest	Dave Voth

Introduction

Document Purpose

This design specification from Microsoft describes the setup and use of the software called the ECP compliance test.

Other Documents

The reader should be familiar with the Microsoft (ECP) Standard document *Standard P1284* by IEEE; it is a parent document of this document. An understanding of it is essential.

Overview

The ECP compliance test is used to verify that a given ECP ISA port functions properly. Of course, the test cannot possibly test all possible combinations of problems. It is useful however to verify much of the known ECP hardware design functions. The ECP compliance test must pass on all 386 IBM Compatible PCs in order for an ECP port to be in compliance with the Microsoft ECP Port Specification.

There are several Items that can only be checked by direct measurement or design. These are specified later in this document.

Vocabulary

The following terms are used in this document:

assert

When a signal asserts, it transitions to a "true" state; when a signal deasserts it transitions to a "false" state.

forward
Host-to-Peripheral communication.

reverse
Peripheral-to-Host communication.

1
A high level.

0
A low level.

Hardware Test Setup

The test is set up on a single ISA PC. Two ECP cards are placed in a PC at different LPT port addresses (0x278, 0x378, 0x3bc). The cards must use interrupt and DMA channels that do not conflict with any devices existing on the system.

After the cards are installed, they are connected together with a cable in such a way that the ECP handshake lines are swapped; this allows loop back testing of ECP functions.

Port-to-Port Test Cable Connections

Name	Pin	Name	Pin
nStrobe	1	nAck	10
Data	2-9	Data	2-9
nAck	10	nStrobe	1
Busy	11	nAutoFd	14
*PError	12	*nInit	16
*Select	13	*Select, *nSelectIn	13,17
nAutoFd	14	Busy	11
*nFault	15	*nSelectIn	17
*nInit	16	*PError	12
*nSelectIn	17	*nFault	15

* Note: These connections are not tested in the current version of comply.exe, as they are not directly responsible for data movement. Thus some cable setups may not have these connections. However, testing of these connections may occur in subsequent revisions of the test, so older test cables may need modification.

The test must be run on a 386 on an ISA bus running Microsoft® Windows™ version 3.1 or later in enhanced mode.

Installing and Running the Test

This software is generally supplied by Microsoft in the form of a set of floppies and a specification known as the *ECP Adaptation Kit*. Read the instructions on the kit concerning the compliance test and copy all the files associated with the compliance test into their own directory called c:\comply.

The compliance test requires no VXDs or other software, and must run under Windows 3.1 or higher; Windows NT™ will not work.

Documentation for the individual files are recorded in the makefile. The complete set of source files, executable files, and makefiles are supplied. To recompile the test (not required), one must install the Microsoft Windows Software Development Kit (SDK) and use a suitable c compiler (c7 in this case).

Under Windows, use the File Manager and double-click on the executable, c:\comply\comply.exe. This will run the test and present you with a window.

The first thing you must do is to specify the settings (click on Settings).

Settings

- The settings menu is used to inform the software of the specific hardware configuration. After selecting the settings menu a dialog box is presented to the user.
- If a port exists (LPT[1, 2, 3]) it is labeled as either STD (standard parallel port) or ECP. Software determines a port is an ECP port by writing and reading the **ecr** of that port. If the port is a standard port, the **ecr** will not exist, usually returning the value of the DSR instead.
- The user *must* then use one ECP port for transmitter and another as the receiver.
- The user *must* also specify which interrupt and DMA channel are associated with the transmitter and receiver.
- The Cable Test will be grayed out (disabled) if the some of the settings are unspecified or if either transmitter or receiver is not an ECP port. Thus incomplete settings prevent further tests from being run.
- The settings are saved in the win.ini file under the [comply] heading in the current Windows directory.

Info

If *any* test fails, no state change will be made to any of the port registers. This is done to allow the user to examine those registers to help identify the problem.

Selecting Info will display the current state of the transmitter and receiver registers. It also displays some information that is gathered in test mode about the FIFO size and interrupt thresholds.

Info may be selected any time and is never disabled. Of course, selecting Info prior to Settings may not be very useful.

Cable Test

Assuming the settings have been entered, the Cable Test dialog box item will become visible. Clicking on Cable Test will verify the continuity of the cable. If errors exist in the cable construction, the test will fail and indicate which wires it did not find a proper connection.

If the cable test passes, it will enable both the register test and the test mode test.

Register Test

Assuming the cable test has passed, the Register Test dialog box item will become visible.

This test verifies that every register bit can be written and read according to the specifications. No attempt is made to verify the functionality of a specific register bit, just the ability to write and/or read those bits specified in the spec.

This test also reads the configuration registers, determines the PWord size of the port, and verifies, if possible, the DMA channel and interrupt line settings.

If the test fails, an appropriate error message is displayed. If the test passes, a message box is displayed.

Test Mode Test

The test mode test is used to determine the size of the FIFOs, the service interrupt thresholds,

and general test mode functionality for both the forward and reverse directions. After this information is obtained, it may be displayed by selecting Info.

The test also verifies that interrupts do indeed occur on the specified interrupt line according to the test mode specification. If the interrupt line is not connected as specified in Settings, or if the interrupt is not functioning properly, this test will fail.

Since this test generates information required in both the Centronics and ECP tests, both Cent and ECP tests are grayed out until this test is passed.

Centronics Test

The Centronics test is used to try to verify, as much as possible, the function of the hardware-assisted Centronics Port (Mode 010).

There are two software drivers used to transmit information. These may be tested individually or one after the other by setting the appropriate check boxes.

The interrupt-driven programmed I/O software driver is activated on the service interrupt and bursts data to the FIFO whenever an interrupt is generated. If the FIFO is empty, a burst equal to the FIFO SIZE is sent; otherwise a burst size equal to that allowed by the write interrupt threshold is sent. This interrupt-driven software driver continues until it has transmitted 8192 bytes.

The other software driver performs a single DMA transfer of 8192 bytes.

Test Sequence

The test first asserts **Busy** (**High**) to block the transfer. Then the transfer is enabled and the FIFO is checked to make sure it is full. **Busy** is then lowered and the transfer is in progress. Unfortunately there is no way to verify the data or timing of **nStrobe**, this must be done using a logic analyzer.

The transfer is given a specified period of time in which to complete (.5 sec).

After this time various sanity checks are made about the expected state of the port. For example, the service interrupt bit should be 1, DMA transfers should generate exactly 1 interrupt, and interrupt-driven programmed I/O transfers must generate over some small number of interrupts.

The data pattern may be selected from the menu and may be useful in debugging specific problems.

ECP Test

This is the bulk of the testing and thus the dialog box is somewhat complicated. There are three different software driver models for the transmitter and three driver models for the receiver. In all cases, 8192 bytes of data are transmitted, received, and verified for each legally selected transmitter/receiver combination.

PIO Transmitter

This software driver emulates the ECP protocol by writing to the transmitter DCR,DATA and reading from its DSR register in the standard mode (mode 000). This is how the driver checks the protocol independent of the timing.

The PIO transmitter presents valid data only during the last portion of the ECP transaction. ECP receivers that accept the data (incorrectly) on the failing edge of **nStrobe** will fail this test and receive 0xaa as data.

Selecting the compression option will cause this model to compress runs of like data. This means that random data and all 1s/all 0s data patterns will be compressed, but the ff00 and 55aa data patterns will be unaffected.

The combination of this driver and the PIO Receiver is not tested.

INTR Transmitter

This software driver is activated whenever an interrupt occurs on the transmitter. On each interrupt it bursts PWords of either FIFO size (empty transmitter FIFO) or write-interrupt threshold size (not empty transmitter FIFO). After the bursts it reenables the interrupt by writing the service interrupt to 0.

Selecting the compression option will cause this model to compress runs of like data and write RLE count bytes into the AFifo (to verify the function of the AFifo). Note that this means that random data and all 1s/all 0s data patterns will be compressed, but the ff00 and 55aa data will be unaffected.

This driver is tested against all selected receiver combinations.

DMA Transmitter

This software driver uses DMA to move the 8192 bytes of test data.

Selecting the compression option has no effect on this test, as there is no way to do DMA and compression.

This driver is not tested against the DMA receiver driver.

PIO Receiver

This software driver emulates the ECP protocol by writing to the transmitter DCR and reading from DSR and DATA in PS2 mode (mode 001). This is useful to check the protocol independent of the timing.

The PIO receiver checks valid data on both portions of the ECP transaction. ECP transmitters that remove the data (incorrectly) after the failing edge of **nStrobe** will fail this test.

This model will decompress any data that happens to be compressed.

The combination of this driver and the PIO transmitter is not tested.

INTR Receiver

This software driver is activated whenever an interrupt occurs on the receiver. On each interrupt it reads bursts of PWords of FIFO size (full receiver FIFO) or read-interrupt threshold size (not full receiver FIFO). After the bursts it reenables the interrupt by writing the service interrupt to 0.

The ECP hardware is counted on to decompress any compressed data.

This driver is tested against all selected transmitters.

DMA Receiver

This software driver uses DMA to move the 8192 bytes of test data.

The hardware is counted on to decompress any compressed data.

This driver is not tested against the DMA transmitter.

ECPAbortTest

This tests to make sure that the port can execute the abort sequence (Spec version 1.14 and later) and that no bytes are lost. The test fills the FIFO with FIFO size PWords while **Busy** is low. At this time, if bit 2 in **cnfgA** is 0 then the FIFO should not be full until one more byte is added.

Software then places **nStrobe** low (via the DCR) and **Busy** is switched high, software tests to make sure that the DCR controls **nStrobe** at this time and that **nStrobe** is still low.

If the port is a 16-bit one, the test checks that bit 1 of **cnfgA** correctly reports the possible partial byte and that the FIFO is full when expected.

As with the rest of this test suite, the 32-bit port has not been implemented.

Misc Test

This test checks several miscellaneous functions not covered in the rest of the test. The first thing tested is the **nFault** ECP interrupt function. Both the assertion of **nFault** when enabled or the enabling of the interrupt when **nFault** is low should generate an interrupt.

The **nAck** interrupt is tested to make sure that toggling **nAck** will generate an interrupt when enabled.

Lastly, a check is made to ensure the DCR and DSR registers are functional when in ECP mode.

Meeting Compliance

In order to meet compliance requirements, the following tests and measurements are required:

- Run the device as transmitter and receiver for all test modes and data patterns without failure.
- Click on Info and verify that the proper FIFO size, PWord size, and interrupt thresholds have been detected.
- Verify the Centronics mode timing and protocol with a logic analyzer.
- Run the test on as many different PC types as possible.
- Verify that all ECP mode drivers are push-pull, that they have an impedance-controlled series resistor of at least 20 Ohms, and that the typical resistance of the combination of the driver-resistor pair is in the 45-65 Ohm range.
- Examine the waveform in a typical open-ended cable and verify that the impedance match is reasonable (no horrible overshoot, undershoot, or ringing for the single line switching case).
- Run the ECP test in loop mode for a long period of time with Random data.

Known Problems

It is useful to test the new design with the Xilinx reference design. Some problems have been uncovered with this design and they are noted here:

- Most systems have a pulldown on DRQ to keep it low when not in use. However, some do not, and there is not currently a pulldown on the Xilinx board. If problems occur on the Xilinx board, pulling down DRQ5 with a 1K resistor may fix them.
- On one system the Xilinx design generated parity errors during DMA in the forward direction only. This was with an OPTI system board chip set. This problem has not been tracked down, but has not been seen with any commercial vendor designs, only the one case of the Xilinx design.

Corrections to Previous Versions (Revisions) of the Extended Capabilities Port Protocol and ISA Interface Standard

Date: 14 July 1993

Revision #	Date
1.07	2 Dec 1992
1.08	18 Dec 1992
1.09	7 Jan 1993

1.10	4 Feb 1993
1.11	10 Feb 1993
1.12	28 Apr 1993
1.14	14 July 1993

Corrections to Revision 1.07 Only (brings document to Revision 1.08 level)

Page 16

Forward-Idle

When the host has no data to send, it keeps HostClk (nStrobe) high and the peripheral will leave PeriphAck (Busy) low.

Page 19

Forward Data Transfer phase

The Forward phase may be entered from the Forward-Idle phase. When the peripheral is not busy, it sets PeriphAck (Busy) low (event 32). The host then sets HostClk (nStrobe) low when it is prepared to send data (event 35). The data must be stable for the specified setup time prior to the falling edge of HostClk. The peripheral then sets PeriphAck (Busy) high to acknowledge the handshake (event 36). The host then sets HostClk (nStrobe) high (event 37). The peripheral then accepts the data and sets PeriphAck (Busy) low, completing the transfer. This sequence is shown in Figure 2.

The timing is designed to provide three cable round-trip times for data setup if Data is driven simultaneously with HostClk (nStrobe).

Reverse Data Transfer phase

The Reverse phase may be entered from the Reverse-Idle phase. After the pervious byte has been accepted, the host sets HostAck (nAutoFd) low (event 46). The peripheral then sets PeriphClk (nAck) low when it has data to send (event 43). The data must be stable for the specified setup time prior to the falling edge of PeriphClk. When the host is ready it to accept a byte, it sets HostAck (nAutoFd) high to acknowledge the handshake (event 44). The peripheral then sets PeriphClk (nAck) high (event 45). After the host has accepted the data, it sets HostAck (nAutoFd) low (event 46), completing the transfer. This sequence is shown in Figure 3.

Page 33

Table 14. Extended Control Register

	0:	Enables an interrupt pulse on the high to low edge of **nFault**. Note that an interrupt will be generated if **nFault** is asserted (interrupting) and this bit is written from 1 to 0. This prevents interrupts from being lost in the time between the read of the **ecr** and the write of the **ecr**.
<3>	R/W	**dmaEn**
	1:	Enables DMA (DMA starts when **serviceIntr** is 0).
	0:	Disables DMA unconditionally.
<2>	R/W	**serviceIntr**
	1:	Disables DMA and all of the service interrupts.
	0:	Enables one of the following 3 cases of interrupts. Once one of the 3 service interrupts has occurred **serviceIntr** bit shall be set to a 1 by hardware. Writing this bit to a 1 will not cause an interrupt.
		case dmaEn=1: During DMA (this bit is set to a 1 when terminal count is reached).
		case dmaEn=0 direction=0: This bit shall be set to 1 whenever there are **writeIntrThreshold** or more PWords free in the FIFO.

		case dmaEn=0 direction=1:	This bit shall be set to 1 whenever there are **readIntrThreshold** or more valid PWords to be read from the FIFO.
<1>	R	**full**	
	1:	**direction** = 0 The FIFO cannot accept another PWord.	
	1:	**direction** = 1 The FIFO is completely full.	
	0:	**direction** = 0 The FIFO has at least 1 free PWord.	
	0:	**direction** = 1 The FIFO has at least 1 free byte.	
<0>	R	**empty**	
	1:	**direction** = 0 The FIFO is completely empty.	
	1:	**direction** = 1 The FIFO contains less than 1 PWord of data.	
	0:	**direction** = 0 The FIFO contains at least 1 byte of data.	
	0:	**direction** = 1 The FIFO contains at least 1 PWord of data.	

Corrections to Revision 1.07 and Revision 1.08 (brings document to Rev 1.09 level)

Page 28

Table 8, corrections to ecpAFifo and cFifo:

Table 8. Register Definitions

Name	Address		Size	Mode	Function
ecpAFifo	0x000	R-R/W	byte	011	ECP FIFO (Address)
cFifo	0x400	R-R/W	PWord	010	Parallel Port Data FIFO

Page 32

Table 14, correction to mode 001:

Table 14. Extended Control Register

<7:5>	R/W	**mode**
	001:	*PS/2 Parallel Port mode*. Same as above except that **direction** may be used to tri-state the **data** lines, and reading the **data** register returns the value on the **data** lines and not the value in the **data** register. It is always best for the hardware design to read the value of the lines and not the register (some old Centronics interfaces actually returned the reg value and not the wire value). All drivers have active pull-ups (push-pull).

Corrections for Revision 1.09 (brings document to Rev 1.10 level)

Page 19

Reverse to Forward phase

The Reverse to Forward phase is entered from the Reverse phase. HostAck (nAutoFd) may be high or low when the Reverse to Forward phase is entered. The host sets nReverseRequest (nInit) high (event 47). The peripheral then tri-states the data bus, sets PeriphAck (Busy) low to indicate the proper forward channel status, and sets PeriphClk (nAck) high (event 48). If the peripheral was in the middle of a data transfer (PeriphClk low) it assumes that the data byte will be discarded by the host and suspends the transfer. After waiting the minimum setup time, the peripheral then sets nAckReverse (PError) high to acknowledge the change of direction (event 49). The host is now permitted to drive the data bus. The interface now enters the Forward phase. This sequence is shown in Figure 3.

Page 20

Valid termination

To terminate from P1284 mode the host sets BOISEmode (nSelectIn) low (event 22), which will initiate one of two types of termination. The first type is a handshake which allows the printer to tell the host when it has returned to Compatible mode. The second is an immediate abort, with no guarantee of interface integrity. If the interface was in a "valid" state, which is any state where a reverse data transfer is not in progress, the printer will perform the handshake. If the interface was not in a "valid" state, the printer will abort immediately. Valid states are indicated in the data transfer diagrams by BOISEmode (nSelectIn) being shown as a heavy line.

To terminate from a valid state, the printer will respond to BOISEmode (nSelectIn) being set low by setting nAckReverse(PError) to low and PeriphAck (Busy) and nPeriphRequest (nFault) high (event 23). The printer will then set Xflag (Select) to its opposite sense, and PeriphClk (nAck) low (event 24). The host then sets HostAck (nAutoFd) low (event 25). The printer then sets the compatible mode printer status on nPeriphRequest (nFault), Xflag (Select), nAckReverse (PError), and nReverseRequest (nInit) (event 26). The printer then sets PeriphClk (nAck) high (event 27). The host ends the termination handshake by setting HostAck (nAutoFd) high (event 29), which returns the interface to the compatible mode idle phase. The printer may then change PeriphAck (Busy) (event 30) to accept host-to-printer data. This sequence is shown following a data transfer in figure 2.

Page 21 (NO modifications to the timing diagrams, just a simple modification to ECP ID on the Data(8..1) line)

Extended Capabilities Port (ECP) Mode: Negotiation, Setup, Forward, Termination

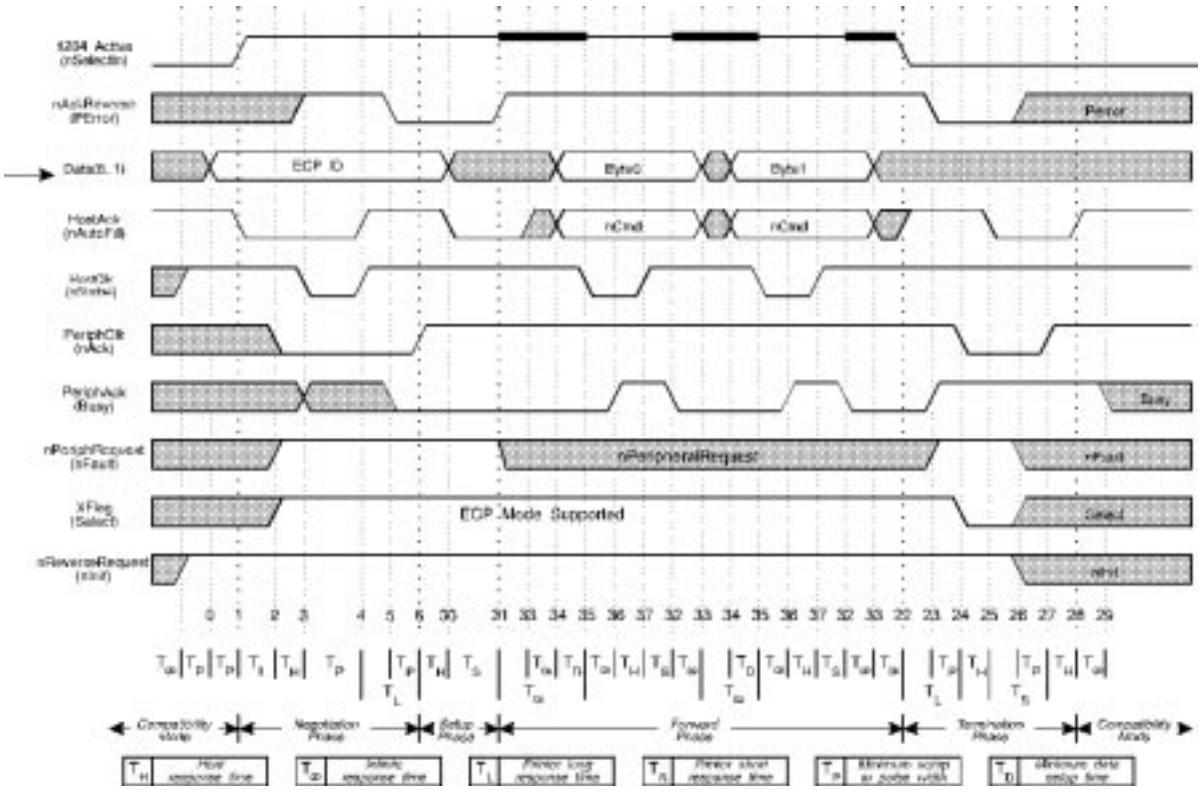

Figure 2. ECP mode timing (1 of 2)

Page 28 (incorrect in all revisions prior to 1.10)

Table 8, corrections to ecpAFifo and cFifo:

Table 8. Register Definitions

Name	Address		Size	Mode	Function
ecpAFifo	0x000	W-R/W	byte	011	ECP FIFO (Address)
cFifo	0x400	W-R/W	PWord	010	Parallel Port Data FIFO

Page 41 (new to Rev 1.10, was not part of Rev 1.09)

Appendix A: Peripheral—Side Design Note

The specifications and guidelines described in this document are specific to ISA implementation on a host PC system. Peripherals tend to have bus architecture that has nothing in common with the host PC architecture.

Although this document is not meant to provide a guideline for ECP port design on a peripheral, you need to keep the following special consideration in mind.

The peripheral must always return data when a byte is requested. Peripheral devices that know the total number of bytes to be transmitted must not stop sending data after that number of bytes is reached. Additional "don't care" data bytes must be sent to "pad" the transfer. The host interface will discard any extra bytes received. The extra "don't care" bytes are useful to provide alignment to wider busses (i.e., 16-bit, 32-bit, 64-bit).

Page 23 (was incorrect in all versions prior to 1.11)

The following events in figure 2 are corrected:

Figure 2

0	Host sets extensibility request value on data bus.
34	Host places Data on the bus. The command bit (nCmd/HostAck) is driven to the appropriate level.

Corrections to Revision 1.11 ONLY (brings it to Revision 1.12 level)

Page 11 (was incorrect in all versions prior to 1.12)

The new paragraph is listed here:

PeriphAck (Busy)

The peripheral uses this signal to flow control in the forward direction. It is an "interlocked" handshake with nStrobe. PeriphAck also provides command information in the reverse direction.

Page 20 (was incorrect in all versions prior to 1.12)

Valid termination

First paragraph is the same. The second paragraph is changed:

To terminate from a valid state, the printer will respond to BOISEmode (nSelectIn) being set low by setting nAckReverse(PError) to low and PeriphAck (Busy) and nPeriphRequest (nFault) high (event 23). The printer will then set Xflag (Select) to its opposite sense, and PeriphClk (nAck) low (event 24). The host then sets HostAck (nAutoFd) low (event 25). The printer then sets the compatible mode printer status on nPeriphRequest (nFault), Xflag (Select), and nAckReverse (PError) while the host sets the compatible mode status on nReverseRequest (nInit) (event 26). The printer then sets PeriphClk (nAck) high (event 27). The host ends the termination handshake by setting HostAck (nAutoFd) high (event 29), which returns the interface to the compatible mode idle phase. The printer may then change PeriphAck (Busy) (event 30) to accept host-to-printer data. This sequence is shown following a data transfer in figure 2.

Page 26 (was incorrect in all versions prior to 1.12)

Busy	1	I	11	11	This signal deasserts to indicate that the peripheral can accept data. This signal handshakes with nStrobe in the forward direction. In the reverse direction this signal, when low, indicates the data is RLE.

Page 31 (was incorrect in all versions prior to 1.11)

0: (Default) The transmitter shall send only uncompressed (raw) data in this case.

Appendix B (added for version 1.12)

Appendix B: Known Deviations from or Additions to the ISA Standard

Detection of FIFO state errors

At least one port design has implemented a feature that generates an error if the FIFO is overwritten in the forward direction or overread in the reverse direction. This error should never occur with proper hardware and software design. In the case of the error some designs may set *both* the **full** and **empty** bits in the **ecr** and generate a service interrupt (setting the service interrupt bit).

This feature is noted here so that drivers may be able to interpret the event properly.

Use of level-triggered interrupts/Non-ISA designs

The port, as defined, functions properly on ISA with the use of edge-triggered interrupts. In order for this port to function properly on busses that use level-triggered interrupts, a modification to the hardware must be made.

The difficulty is that the service interrupt bit serves as both an enable and an interrupt. To use level-triggered interrupts, the service interrupt shall be disabled when in standard or PS2 mode (mode 000,001). Depending on the bus, interrupt disabling will either tri-state the interrupt line or force it to a noninterrupting state. This gives the software the ability to independently enable and sense the interrupt. When enabled (not mode 001, 000), the value of the service interrupt bit may be translated directly, logically into a level-sensitive interrupt. Otherwise the function of the service interrupt bit remains as specified.

None of the other interrupt sources presents a problem for conversion of the port to a level-sensitive interrupt scheme.

Corrections for Revision 1.12 (brings document to Rev 1.13 level)

Page 31 (was incorrect in all versions prior to 1.13)

Table 12. Configuration Register A (Added Bits)

<7>	R	Indicates if interrupts are pulsed or ISA-level.
	1:	Interrupts are ISA-level (see Appendix B).
	0:	Interrupts are ISA-pulses.
<6:4>	R	**impID.** Implementation ID number; identifies the design and PWord size.
	0x00:	The design is a 16-bit implementation (PWord = 2 bytes).
	0x01:	The design is an 8-bit implementation (PWord = 1 byte).
	0x02:	The design is a 32-bit implementation (PWord = 4 bytes).
	0x03-0x07:	Reserved and not supported by Microsoft software.

Page 35 (was incorrect in all versions prior to 1.13; prior to Rev 1.13 this was on page 34)

- When **ackIntEn** is 1, the way existing parallel ports implement this today. The interrupt generated is ISA-friendly in that it may pulse the interrupt line low. Optionally it may also drive a level (see Appendix B).

Appendix B: Known Enhancements to This Standard (significantly modified in version 1.13)

Use of nonpulsed (level-triggered) interrupts

The original design of this port generated pulses on each interrupt event. This will work fine on ISA machines, but some designers wish to make the port function in the standard level-ISA fashion. In order to do this, the level interrupt is enabled in (driven low) whenever the device is in ECP, Test, or Centronics FIFO mode. When an interrupt condition exists, the signal is driven

high.

After receiving, the interrupt driver will read the ECR to determine the cause of the interrupt. It then writes the ECR, setting the **serviceIntr** bit to 1 and the **nErrIntrEn** bit to 1. This masks all interrupt sources and causes the **iRq** line to go low. After servicing the interrupt, the driver will reenable interrupts, if desired, by writing the **serviceIntr** and/or **nErrIntrEn** bits to 0.

After the completion of each DMA transfer (terminal count), the driver shall first write **dmaEn** to 0 before beginning another DMA transfer.

The software driver shall ensure that the interrupt due to **nAck** is disabled whenever the port is using ECP protocol or ECP mode to transfer data.

Corrections for Revision 1.13 (brings document to Rev 1.14 level)

Page 11

nPeriphRequest (nFault)

During ECP mode the peripheral is permitted (but not required) to drive this pin low to request a reverse transfer. The request is merely a "hint" to the host; the host has ultimate control over the transfer direction. This signal provides a mechanism for peer-to-peer communication. ~~.~~ This signal would be typically used to generate an interrupt to the host CPU. The signal is asserted low and kept there until the interrupt is serviced or the port exits ECP mode.

Page 12

nReverseRequest (nInit)

This pin is driven low to place the channel in the reverse direction. The peripheral is only allowed to drive the bidirectional data bus while in ECP mode ~~-~~when BOISEmode is high and nReverseRequest is low.

Page 15

TR specified, TL/TS modified slightly.

Table 5. Signal Timing

Time	Minimum	Maximum
T^H	0	1.0 sec.
T^T	0	infinite
T^L	0	35 ms
T^S	35ms	
T^P	500 ns	
T^D	0 ns	
TR	Host may enter Data Transfer Recovery after TS (Software application-dependent)	

Page 20

Describes Host Data Recovery.

Aborting the Forward Data Transfer phase

There is a possibility of the forward channel becoming stalled. The stall condition will exist if the peripheral is unable to accept the data byte being transferred by the host at event 35. In this condition the peripheral will not acknowledge the handshake (event 36). A mechanism has been provided to recover from this condition. If the host, following event 35, determines that a stall condition exists, the host may abort the transfer of the current byte by setting nReverseRequest (nInit) low (event 72). The peripheral, regardless of whether it has accepted the byte from the host (event 36 happened), shall discard the byte (if applicable) and acknowledge the host by

setting nAckReverse (PError) low. The host then returns nReverseRequest (nInit) high (event 74) and the peripheral follows by returning nAckReverse (PError) high (event 75). This sequence, shown in figure 4, will return the interface to the state that existed prior to host event 35.

Forward to Reverse phase

The Forward to Reverse phase is entered from the Forward phase. The host tri-states the data bus and sets HostAck (nAutoFd) low (event 38). After waiting for the minimum setup time, the host then sets nReverseRequest (nInit) low (event 39). The peripheral then acknowledges the reversal by setting nAckReverse (PError) low (event 40). The peripheral is now permitted to drive the data bus after setting nStrobe high. The interface now enters the Reverse phase. This sequence is shown in Figure 3.

Page 20

Clarifies when the host will accept the data on the reverse transfer.

Reverse Data Transfer phase

The Reverse phase may be entered from the Reverse-Idle phase. After the pervious byte has been accepted, the host sets HostAck (nAutoFd) low (event 46). The peripheral then sets PeriphClk (nAck) low when it has data to send (event 43). The data must be stable for the specified setup time prior to the falling edge of PeriphClk. When the host is ready to accept a byte, it sets HostAck (nAutoFd) high to acknowledge the handshake (event 44). The peripheral then sets PeriphClk (nAck) high, causing the host to accept the data (event 45). After the host has accepted the data, it sets HostAck (nAutoFd) low (event 46), completing the transfer. This sequence is shown in Figure 3.

Page 23

The net timing parameter Tr is added.

Init stays high at Event 26.

PError shown to toggle, not set low, at Event 23.

Page 24

nPeriph Request allowed to assert during reverse transfer.

Page 25

All-new timing diagram for Host recovery timing.

Page 27

nFault no longer needs to be deasserted in Reverse mode:

40. The peripheral sets nAckReverse (PError) low to acknowledge the bus reversal. (nAutoFd) is now active.

nFault no longer needs to be deasserted during a reverse transfer:

48. The peripheral terminates any ongoing transfer, tri-states the data bus, sets PeriphClk (nAck) high, and places valid status on the PeriphAck (Busy) line.

New events for the Host Data Recovery:

72. After waiting for the minimum required time (Ts), the host may abort the host to peripheral transfer in progress by setting nReverseRequest (nInit) low.
73. The peripheral handshakes, setting nAckReverse (PError) low, and if not already PeriphAck (Busy) low, indicating that the peripheral-to-host data transfer in progress has been aborted and the data byte has been discarded.
74. The host raises nReverseRequest to continue the handshake.
75. The peripheral completes the handshake by raising nAckReverse (PError) high, returning

the link to a host-idle condition.

Page 30

Note to designers about pulldown resistor:

dRq	1	O	DRQ DMA Request, Note: Use a 1K pulldown here to prevent requests.

Page 32

Added function, all current designs happen to support enforce it here. This is needed to support the host data recovery.

In all modes the **dcr** *shall* be able to override any hardware state machine and force the signal active. For example, writing 1s to bits<1:0> *shall* force **nStrobe** and **nAutoFd** low, even in ECP mode. Software will make sure that **dcr** bits <1:0> are set to 0 prior to entering ECP mode.

Page 32

Clarifies Direction bit function (this has caused some confusion in the past).

Table 11. Device Control Register

<7:6>	R	**Reserved**, returns undefined when read.
<5>	R/W	**Direction**
	1:	If **mode** = 000 or **mode** = 010, we are standard parallel port and this bit has no effect (drivers are enabled). Otherwise, this bit tri-states the drivers and sets the direction so that data will be read from the peripheral. Note: some designs actually force this bit to 0 when in modes 000 or 010. Software must be in PS2 mode 001 in order to reliably write this bit to 1.

Page 33

Improved wording to avoid confusion.

cFifo

0x400, Mode = 010 (Parallel Port Data FIFO)

PWords written or DMAed from the system to this FIFO are transmitted by a hardware handshake to the peripheral using the standard parallel port protocol. Transfers to the FIFO are PWord-aligned. If partial PWords need to be transferred, the operation must be handled in mode 000. This mode is only defined for the forward direction.

Page 34

Changes to **cnfgA** to add information required to support Host Data Recovery.

Table 12. Configuration Register A

<7>	R	Indicates if interrupts are pulsed or ISA-level.
	1:	Interrupts are ISA-level (see Appendix B).
	0:	Interrupts are ISA-pulses.
<6:4>	R	**impID.** Implementation ID number, identifies the design and PWord size.
	0x00:	The design is a 16-bit implementation (PWord = 2 bytes).
	0x01:	The design is an 8-bit implementation (PWord = 1 byte).
	0x02:	The design is a 32-bit implementation (PWord = 4 bytes).
	0x03-0x07:	Reserved and not supported by Microsoft software.
<3>	R/RW	**Misc. reserved.** May be used for anything design-specific. If software, may try to write it to 1.
<2>	R	**nByteInTransceiver**. This design-dependent, read-only parameter indicates if the design uses an extra pipeline byte when transmitting ECP in event 35. See the section on ECP Host Recovery for more information.

	0:	When transmitting (at event 35), there is 1 byte in the transceiver waiting to be transmitted that does not affect the FIFO **full** bit.
	1:	When transmitting (at event 35), the state of the **full** bit includes the byte being transmitted. There are no extra bytes to be accounted for at abort time.
<1:0>	R/RW	This field is a "don't care" for a PWord size of 1 byte. For Host Recovery situations these bits indicate what fraction of a PWord was not transmitted so that software can retransmit the unsent bytes. If the PWord size is 2 or 4 bytes, the value of these two bits is a snapshot of the last PWord being transmitted in mode 011 (event 35) when the FIFO was reset (port was transitioned from mode 011 to mode 000 or 001).
	00:	The PWord at the head of the FIFO contained a complete PWord.
	01:	The PWord at the head of the FIFO contained only 1 valid byte.
	10:	The PWord at the head of the FIFO contained 2 valid bytes.
	11:	The PWord at the head of the FIFO contained 3 valid bytes.

Page 37

Allow vendors to do what they want with these fields:

100:	*Vendor-specified function*
101:	*Vendor-specified function*

Page 40

After the end of a DMA transfer in the forward direction, software must wait until the FIFO is **empty** and the state of the **busy** line (visible in the **dsr)** is low. This ensures that all data has been transmitted to the peripheral.

Page 48

Examples of how software executes a host data recovery.

Appendix C: Host Recovery of a Forward Transfer at (Event 35)

Background

Long after this specification was implemented, IEEE noticed a potential problem for some designs. It was possible for some peripherals to stall forever in event 35 and there was no way for the host to legally "break out" of the forward transfer without data loss and protocol violation. The specification was modified to allow a for Host Recovery phase.

Software can easily implement the recovery handshake; however, software must determine the total number of bytes that remain in the FIFO first so they may be retransmitted. The basic idea is to fill the FIFO so we know how many bytes *were* in the FIFO. Then software adjusts for any partially transmitted PWords or bytes that may be in an output transceiver stage. This byte adjustment data is placed in **cnfgA**<2:0>.

Software will perform the following steps to recover from a Forward Data Transfer at event 35:

[The ECP port is in mode 011,stuck on event 35, trying to transmit, and we want to recover.]

1. Write to the **dcr**, driving the **nStrobe** signal low. This prevents further data transfers even if the peripheral starts accepting data.
2. The number of PWords in the FIFO at abort time is computed by writing PWords to the FIFO until the **full** is set to 1. For PWord sizes of 2 and 4 bytes, the PWord at the head of the FIFO may be partially transmitted.
3. The host writes the **ecr** and sets the mode to 001. This causes the port to reset the FIFO and load FIFO state information into **cnfgA**<1:0>. This will be used by software to determine how many bytes remain untransmitted in the PWord at the head of the FIFO.

Note Steps 4 through 8 describe the Recovery Handshake.

4. The host tri-states its drivers by writing the **dcr direction** bit to 1.
5. The host writes the **dcr** setting **nInit** low and waits for **dsr PE** signal to go low.
6. The host writes the **dcr** setting **nStrobe** high.
7. The host writes the **dcr** setting **nInit** high and waits for **dsr PE** signal to go high.
8. The host has recovered and is in the Forward Idle phase. It now reads **cnfgA** and computes the actual number of bytes that need to be retransmitted. The software takes the number of PWords in the FIFO, and adjusts for a partial PWord and for a byte in the output transceiver by using the values in **cnfgA**<2:0>.

Example 1

Suppose we have a byte-wide interface, a FIFO depth of 32 PWords, and **cnfgA**<2:0> = 011b. At abort time we write 2 bytes to the FIFO to make it **full**. The value in **cnfgA**<1:0> does not matter, since this is a byte-wide interface; however, the 0 in **cnfgA**<2> indicates that there is a byte in the output transceiver we must count. Thus the total number of bytes to be re-sent is:

Bytes to be reset = (32-2) + 1 (transceiver byte)

Example 2

Suppose we have a word-wide interface, a FIFO depth of 32 PWords, and **cnfgA**<2:0> = 111b. At abort time we find that 2 more PWords make the FIFO **full**. The value in **cnfgA**<1> does not matter, since this is a word-wide interface. The 1 in **cnfgA**<2> indicates that there are no bytes in the output transceiver, and the 1 in **cnfgA**<0> indicates that the PWord at the head of the FIFO is partial, containing only 1 byte. Thus:

PWords in FIFO = 32 - 2 = 30

One of these is a partial PWord, thus:

- Complete PWords in FIFO = 29
- Partial PWords in FIFO = 1
- Bytes to be reset = (29) * 2 + 0(transceiver byte)+ 1(partial PWord) = 59 bytes

Example 3

Suppose we have a 32-bit-wide interface, a FIFO depth of 16 PWords, and **cnfgA**<2:0> = 011b. At abort time we write 4 PWords to fill the FIFO; thus we know that there are 12 PWord entries in the FIFO used. The value in **cnfgA** indicates that there is 1 byte in the transceiver and the PWord in the FIFO being transmitted contains only 3 bytes. Thus:

- PWords in FIFO = 16 - 4 = 12
- Complete PWords in FIFO = 11
- Bytes to be re-sent = 4 * 11 + 3 (partial bytes in PWord being transmitted) + 1 (transceiver byte) = 48 bytes