IEEE Standard for Interface and Protocol Extensions to IEEE 1284-Compliant Peripherals and Host Adapters

Sponsor

Microprocessor and Microcomputer Standards Committee
of the
IEEE Computer Society

Approved 21 September 2000

IEEE-SA Standards Board

Abstract: System extensions consistent with the implementation and functionality of IEEE Std 1284-2000 are covered. Multiport expansion architectures, daisy chains, an application and device driver programming interface architecture, and data link layer services are explored.
Keywords: daisy chain, data link layer services, device driver programming interface architecture, IEEE 1284, multiport expansion architecture, port sharing

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-2617-9 SH94881
PDF: ISBN 0-7381-1618-7 SS94881

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Introduction

(This introduction is not part of IEEE Std 1284.3-2000, IEEE Standard for Interface and Protocol Extensions to IEEE 1284-Compliant Peripherals and Host Adapters.)

This standard was formally started as an IEEE effort in March 1994. Members of the IEEE P1284.3 Working Group would like to thank the Enhanced Parallel Port Committee for their efforts in establishing the foundation for this standard and allowing this committee to include their work as the basis for much of this standard.

At the time this standard was completed, key contributors to the IEEE P1284.3 Working Group were as follows:

Larry A. Stein, *Chair*
Bill Stanley, *Vice-Chair*
Robert Gross, *Secretary*
R. Randall McBride, *Editor*
Grant Deardon, *Editor*

Balaji Baktha
Brian Batchelder
Motti Beck
Eric Byer
Dale Cronau
Lee Farrell
John Fobel
Bill Hawkins
Kenneth Hilliard
Reed Hinkel
Monte Johnson
Dave Jolley
Ken Konechy
Peter Leunig
Mark Myran
Darryn McDade
Mike Moldovan
Jon Newman
Ron Norton
Rick Pennington
Ron Proesel
Walt Scheiderich
Randy Turner
John Vitek
Forrest D. Wright

The following members of the balloting committee voted on this standard:

Malcolm J. Airst
Scott Akers
Harry A. Andreas
Keith D. Anthony
Bill Baker
Charles Brill
Vivian Cancio
Robert S. Crowder
Dante Del Corso
Stephen L. Diamond
Julio Gonzalez-Sanz
Lawrence Lamers
Joseph R. Marshall
R. Randall McBride
Robert McComiskie
Edward McCreight
Mike Moldovan
Roman Orzol
Granville Ott
Maia Pindar
Jeff Rackowitz
Thomas J. Schaal
Jim Soriano
Robert K. Southard
Larry A. Stein
Robert G. Stewart
Paul Walker
James Wolffe
Forrest D. Wright
Oren Yuen
Janusz Zalewski

When the IEEE-SA Standards Board approved this standard on 21 September 2000, it had the following membership:

Donald N. Heirman, *Chair*
James T. Carlo, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal
Mark D. Bowman
Gary R. Engmann
Harold E. Epstein
H. Landis Floyd
Jay Forster*
Howard M. Frazier
Ruben D. Garzon
James H. Gurney
Richard J. Holleman
Lowell G. Johnson
Robert J. Kennelly
Joseph L. Koepfinger*
Peter H. Lips
L. Bruce McClung
Daleep C. Mohla
James W. Moore
Robert F. Munzner
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Alan Cookson, *NIST Representative*
Donald R. Volzka, *TAB Representative*

Greg Kohn
IEEE Standards Project Editor

The following information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products.

Centronics is a registered trademark of the Genicom Corporation.

Intel is a registered trademark of Intel Corporation.

Contents

IEEE Standard Interface and Protocol Extensions to IEEE 1284-Compliant Peripherals and Host Adapters

1. Overview

1.1 Scope

IEEE Std 1284.3-2000 defines system extensions consistent with the implementation and functionality of IEEE Std 1284-2000[1].

These functions include the following:

- Multiport expansion architectures
  - Multiplexor
  - Daisy chain
- Application and device driver programming interface architecture that can be supported across various operating systems.
- Data link layer services for supporting IEEE 1284 parallel ports.

1.2 Purpose

IEEE Std 1284-2000 specifies all the required mechanical, electrical, and timing characteristics for a high-speed, bidirectional parallel port for printers and other peripherals. While these parameters define the physical environment and protocol requirements for an operational IEEE 1284 port, there is a need to standardize a number of extensions and operating system interfaces to help facilitate operation across various system platforms and operating environments.

[1]Information on references can be found in Clause 2.

2. References

This standard shall be used in conjunction with the following publication. If the following publication is superseded by an approved revision, that revision shall apply.

IEEE Std 1284-2000, IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers.[2]

3. Definitions

3.1 General terminology

The following terms are used in this standard and may be used in conjunction with signaling methods defined in this standard. The definitions are not intended to be absolute, but reflect the use of the terms in this standard.

3.1.1 bidirectional operation: The peripheral and host communicate using forward and reverse data channels. As defined in this document, Nibble and Byte Modes provide reverse channel communication and are used in conjunction with Compatibility Mode to provide bidirectional operation. Extended Capabilities Port (ECP) and Enhanced Parallel Port (EPP) Modes support bidirectional communication.

3.1.2 Basic Input Output System (BIOS): A set of functions and services that drivers may use to implement system functions independent of the platform implementation.

NOTE—See Annex A.

3.1.3 Centronics®: The popular name for the parallel printer port used as the parallel interface for most printers and supported by most personal computers. The name is derived from Centronics Data Computer Corporation, the printer manufacturer that introduced this interface. This interface has never been published as a standard. Despite a basic similarity, many variations have been implemented in different peripherals and hosts.

3.1.4 control lines: Unidirectional signal lines from host to peripheral(s) used to implement handshaking and protocol. These are the nStrobe, nAutoFeed, nInit, and nSelectIn lines.

3.1.5 daisy chain (DC): An interconnection signaling protocol defined in this standard by which multiple devices are attached to a single host or multiplexor port in “chained” fashion (i.e., one input, one output). In this architecture, peripheral 1 is attached to the host, or Mux port. Peripheral 2 is attached to peripheral 1, peripheral 3 is attached to peripheral 2, etc. Up to four DC-capable devices may be chained together, with one additional non-DC-capable device added to the end.

3.1.6 data lines: Bidirectional signal lines that carry binary data during forward and reverse channel data transfer cycles. These are lines defined by IEEE Std 1284-2000 as AD1 through AD8 for forward channel data transfer and 8-bit reverse channel data transfer. In Nibble Mode, reverse data is transferred using the status lines.

3.1.7 data link layer: A group of functions that provide data transfer services to a high-layer client.

NOTE—See 4.4 and Clause 7.

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

3.1.8 Dongles: External device with pass-through connections commonly used as software security keys.

3.1.9 forward direction: Data communication whose direction is from host to peripheral.

3.1.10 host: The device that controls communications to the attached peripheral devices.

3.1.11 IEEE 1284 compatible device: A device that supports any of the common variants of the Centronics interface. Compatible devices shall interoperate with compliant devices in Compatibility Mode.

3.1.12 IEEE 1284-compliant device: A device that supports either IEEE 1284 Level 1 or Level 2 electrical interface, plus Compatibility and Nibble Mode operation, as well as the negotiation phases necessary to transition between the two modes.

NOTE—See Clause 4 of IEEE Std 1284-2000.

3.1.13 IEEE 1284 Type A connector: A plug or receptacle 25-pin sub-miniature D-shell connector as specified in IEEE Std 1284-2000. This is the type of connector used on most personal computer (PC) printer port adapters. It is commonly referred to as the DB25 connector.

3.1.14 IEEE 1284 Type B connector: A plug or receptacle 36-pin ribbon type connector as specified in IEEE Std 1284-2000. This type of connector is also known as a Centronics connector.

3.1.15 IEEE 1284 Type C connector: A miniature plug or receptacle 36-pin ribbon type connector as specified in IEEE Std 1284-2000.

3.1.16 multiport device (MP device or MPD): A device designed to operate in a multiplexor or daisy chain environment, as defined in IEEE Std 1284.3-2000.

3.1.17 multiplexor: A device that attaches to the host parallel port and provides two to four device ports that are logically connected to the host port.

NOTE—See 4.2.1.

3.1.18 n: When preceding a capitalized signal name, is used to denote a signal having negative true logic.

3.1.19 non-multiport device (non-MP device): A device that does not meet the compliance criteria of IEEE Std 1284.3-2000.

NOTE—See 4.6.1 or 4.6.2.

3.1.20 parallel port (IEEE 1284 parallel port): The physical interface as defined by IEEE Std 1284-2000.

3.1.21 peripheral: A device controlled by a host.

3.1.22 plug and play (PnP): A hardware and software framework to allow for autodetection and configuration of peripheral devices and add-in cards attached to a host.

3.1.23 reverse direction: Data communication whose direction is from peripheral to host.

3.1.24 service provider interface (SPI): The operating system interface to the IEEE 1284.3 physical layer.

NOTE—See 4.3 and Clause 6.

3.1.25 status lines: Unidirectional signals lines from peripheral(s) to host used to implement handshaking and protocol. These are the Busy, nAck, Select, PError, and nFault lines.

3.2 Acronyms

DC Daisy chain (see 3.1.13)
MP Multiport (see 3.1.14)
Mux Multiplexor (see 3.1.15)
EPP Enhanced parallel port (see 3.3)
ECP Extended capabilities port (see 3.3)

3.3 Communication modes

The Communication Modes as defined in IEEE Std 1284-2000 provide different levels of communication capabilities and allow a peripheral device to implement the most appropriate method. A mechanism is provided to synchronize the host and peripheral modes.

3.3.1 Byte Mode: An asynchronous, byte-wide reverse channel in which data is transmitted one byte at a time using the data lines.

3.3.2 Compatibility Mode: An asynchronous, byte-wide forward channel with data and status lines as used in the Centronics interface. Compatibility Mode is backward compatible with many existing devices, including the personal computer (PC) parallel port, and is a mode common to all IEEE 1284-compliant devices.

3.3.3 Extended Capabilities Port Mode (ECP Mode): An asynchronous, interlocked, byte-wide, bidirectional channel. A control line is provided to distinguish between command and data transfers. A command may optionally be used to indicate RLE count for data compression or channel address.

3.3.4 Enhanced Parallel Port Mode (EPP Mode): An asynchronous, interlocked, byte-wide, bidirectional channel controlled by the host device. This mode provides separate address and data read/write cycles over the eight data lines of the interface.

3.3.5 Negotiation: A signaling protocol used by a compliant host adapter to determine the capabilities and set the mode of an IEEE 1284-compliant peripheral. Both host and peripheral enter Compatibility Mode following reset. The host may then initiate the negotiation sequence. A compliant peripheral will respond correctly, while a noncompliant peripheral will not respond. At the end of the negotiation sequence, a compliant peripheral and host interface may enter one of the advanced modes. If a negotiation sequence fails, then both interfaces remain in Compatibility Mode.

3.3.6 Nibble Mode: An asynchronous reverse channel. Data bytes are transmitted as two sequential nibbles using the status lines.

4. Features and compliance

4.1 Overview

With the adoption of IEEE Std 1284-2000, there has been a tremendous increase in the types of peripherals that are being attached to the parallel port. This, combined with the increasing use of portable platforms, has led to the desire to share the parallel port with other peripherals. This standard defines two multiport configurations:

— Multiplexing
— Daisy chaining

The additional features of IEEE 1284 have created a requirement for a definition of the data link layer to support this interface, as well as a method for the software to manage port contention and sharing. This requirement is satisfied in the clauses titled

— Service provider interface (SPI) (Clause 6)
— Data link services (Clause 7)

Subclause 4.5 deals with electrical issues resulting from an architecture where multiple devices are sharing one port. This subclause makes changes to the interface recommendations of IEEE Std 1284-2000, but stays within the specification for compliance.

4.2 Multiport (MP)

This standard defines methods by which the host's parallel port can be connected to more than one peripheral device. Two types of multiport configurations are supported: multiplexor and daisy chain. Peripheral device ports are accessed on a time-shared basis and only one device may be selected at a time.

In both multiport configurations (multiplexor and daisy chain), interrupt requests are considered to be 'service requests'. Interrupt requests for an unselected device will not be passed through to the host interface. The peripheral shall maintain a local interrupt status. The host shall recognize this status when a 'Query Interrupt' cycle is performed. There is no guaranteed minimum latency with respect to port selection. Peripheral devices shall provide sufficient buffering to cope with real-time considerations.

If both a multiplexor and daisy chain are present, the multiplexor shall be directly attached to the host's IEEE 1284 parallel port. Daisy chains are then attached to the multiplexor's device ports. There can only be one Multiplexor attached to a host port.

4.2.1 Multiplexor

A multiplexor (Mux) is an *external device* interposed between the host port and two or more peripherals, permitting multiple parallel port devices to share a single host port. The Mux has a single port that connects to the host, and up to four device ports to which single peripherals or daisy chained devices may be attached. The Mux provides a mechanism by which the host may make a connection between its host port and any of the device ports. The Mux provides a *transparent* IEEE 1284 signaling path between the selected device port and host's port. When selected, the peripheral device appears to be directly connected to the host's port. Only one device port may be selected at any given time. Figure 1 shows a Mux with its device port numbering and attached peripherals. These peripherals can be non-MP devices or MP devices.

4.2.2 Daisy chaining

A daisy chain (DC) peripheral device has two ports: host and pass-through. The device's host port connects either to the host's parallel port, a device port of a Mux, or the pass-through port of another upstream DC device. The pass-through port is used to connect to the next downstream peripheral device in the daisy chain, if any. The last device may be a non-MP device. A daisy chain may be composed of up to 4 DC-compliant devices and an optional fifth non-MP device.

Figure 2 shows DC device numbering. The non-MP device at the end of the chain is referred to as the end-of-chain (EOC) device.

The protocol used to address, select, and query a DC device is called the command packet protocol (CPP). The CPP is a method of selection that does not interfere with any of the operational modes of IEEE Std 1284-2000 and is completely transparent to an IEEE 1284 device (see 5.2).

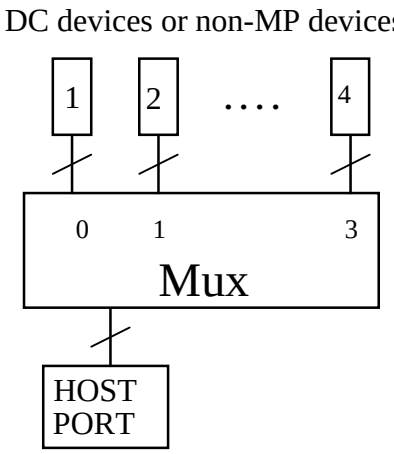


Figure 1—IEEE 1284.3 Mux

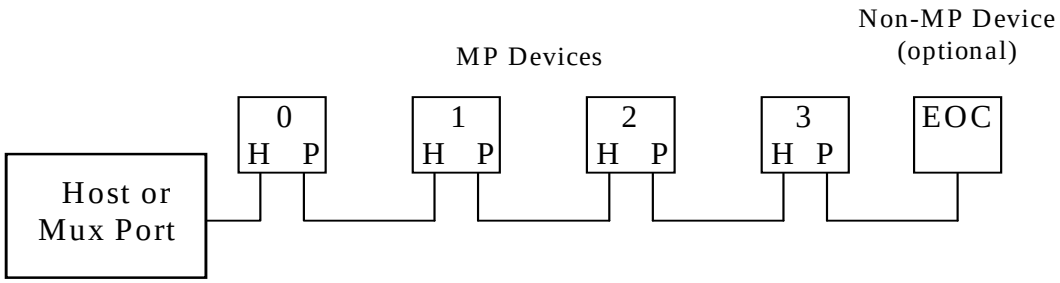


Figure 2—IEEE 1284.3 daisy chain (DC)

4.3 Service provider interface (SPI) features

The SPI layer is the software component that directly interfaces between other software components and the IEEE 1284 physical interface. This layer is responsible for providing the following:

— Data transfer
— Device enumeration of the IEEE 1284.3 topology
— Host adapter information
— Port contention

A complete description of the SPI can be found in Clause 6.

4.4 Data link layer features

The IEEE 1284.3 data link layer extends the services provided by the SPI by exchanging packets of data between one or more software components on the host and their peer components on the peripheral.

The IEEE 1284.3 data link layer provides the following:

— Unacknowledged connectionless packet data transfer services.
— Routing between peer software components.

A complete description of the data link layer services can be found in Clause 7.

4.5 Physical interface

DC devices shall meet all the electrical specifications of IEEE Std 1284-2000. However, for these devices, the potential for multiple pull-up termination resistors on non re-driven data lines requires the use of no less than 10 kΩ rather than the 1.2 kΩ values recommended in IEEE Std 1284-2000.

Figure 3 shows the electrical connections for the data lines of a system with a host, two DC devices, and a non-MP device at the end. All control and status lines shall be re-driven by each DC device. The maximum combined cable length between the host port and any peripheral shall not exceed 10 m. All cables shall be IEEE 1284-compliant.

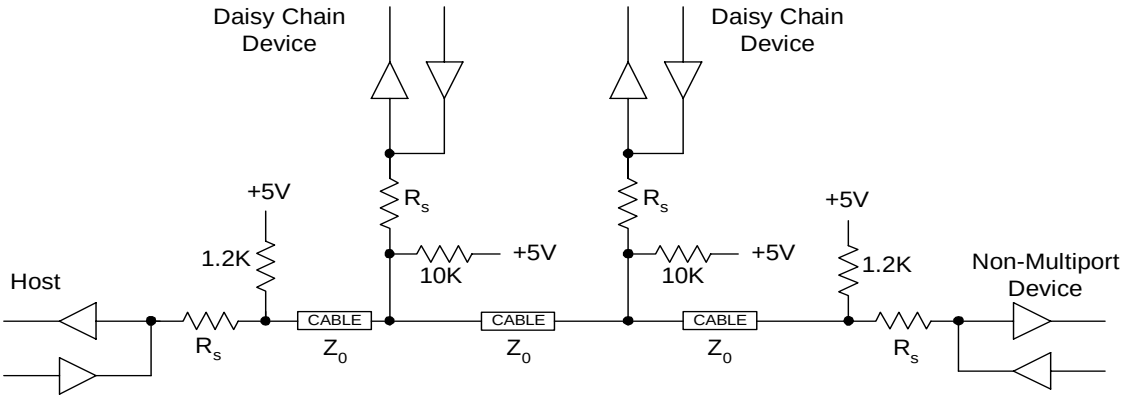


Figure 3—IEEE 1284.3-compliant data line circuitry example

4.6 Compliance

Devices that meet the mechanical interface requirements of IEEE Std 1284-2000, the electrical requirements of 4.5, and the protocol compatibility requirements of Clause 5 are referred to as “IEEE 1284.3-compliant” devices.

4.6.1 DC device

An IEEE 1284.3-compliant DC device shall meet the electrical requirements of 4.5 and the protocol requirements of 5.2 and 5.3.5. Additionally, it shall meet the compliance requirements in 4.4 of IEEE Std 1284-2000 and implement the Device ID string as specified in 7.6 of IEEE Std 1284-2000.

4.6.2 Mux device

An IEEE 1284.3-compliant Mux shall meet the electrical requirements of IEEE Std 1284-2000 and the protocol requirements of 5.2 and 5.4 of this standard.

4.6.3 SPI

Implementations of the IEEE 1284.3 SPI shall implement the functionality specified in 6.2 and 6.3.

4.6.4 Data link

Implementations of the IEEE 1284.3 data link shall comply with 7.3 and shall implement the functionality specified in 7.2.

5. Port-sharing protocols

5.1 Overview

When communicating with MP devices, the host uses a protocol that does not interfere with any IEEE 1284 protocols. This clause defines the protocol, which is transparent to non-MP Devices. It is first described in general terms, then in detail for each specific type of MP device.

5.2 Command packet protocol (CPP)

The host uses the CPP to select the peripheral device. This protocol is transparent to all IEEE 1284 communication modes. CPP defines a packet consisting of a preamble, a function ID, a command, and a terminator. The format of the initial portion of the CPP sequence is such that the data is self-qualifying; i.e., no strobes are required for the preamble or function ID. The initial portion of the CPP is transmitted as a sequence of data bytes onto the parallel port data lines without any host control line transitions. The data pattern provides synchronization and validation.

The MP device will have a means to detect, decode, and validate the CPP. Packet detection begins when stable data corresponding to the first byte of the preamble is detected. Detection aborts if any control line transitions are detected, or if stable data is found that does not match the next expected preamble byte. After the preamble and function ID, a command byte (or bytes) is sent. Depending on the type of MP device, each command byte may be qualified by nStrobe. An 0xFF on the data lines terminates the packet. Data is stable when it has remained unchanged for at least T_W. The host must maintain data on the parallel port for a minimum of T_H (see Table 4).

5.2.1 CPP format

The general format of the all CPP packets is shown in Table 1.

Table 1—General CPP format

Preamble	Function ID	Command Code	Terminator
0xAA 0x55	XX $\overline{XX}$ YY $\overline{YY}$	ZZ…	0xFF

5.2.1.1 Preamble

The first two bytes of the CPP shall be 0xAA followed by 0x55.

5.2.1.2 Function ID

Function ID’s provide a mechanism to communicate with specific MP devices without causing interference with other devices. There are two types of Function ID's: top level and second level (see Table 2 and Table 3). Top-level ID’s are associated with IEEE 1284.3 components that are required to implement the MP topology. The SPI shall support these types of function ID's. Top-level devices are as follows:

— Multiplexor
— Daisy chain

Second-level Function ID’s are defined options to provide a mechanism for other types of peripherals to communicate without causing interference. The operational details of these functions are beyond the scope of this standard. These are essentially CPP escape codes. Examples of second-level devices are as follows:

— Dongles
— Vendor unique
— Extension key (reserved for future IEEE 1284.3 extensions)

The SPI, as defined in this standard, does not support second-level codes.

Table 2—Defined top-level Function ID’s

Function ID codes	Type of device or action
0x00 0xFF 0x87 0x78	Daisy chain
0xF0 0x0F 0x52 0xAD	Multiplexor

Table 3—Defined second-level Function ID’s

Function ID codes	Type of device or action
0xFA 0x05 0xBC 0x43	Dongle (security) devices
0xFA 0x05 0x61 0x9E	Extension key...indicates two additional extension bytes follow
0xFA 0x05 0x27 0xD8	Vendor unique sequence follows

5.2.1.3 Command codes

The command code field of the CPP contains one or more bytes, which are specific to the function ID. The control lines may be used to qualify these command bytes. Valid command codes are from 0x00–0xEF. Values from 0xF0–0xFF are invalid.

5.2.1.4 Termination

The termination field follows the command sequence of the CPP. Termination is defined by a value of 0xFF on the data lines for T_W. During the preamble and function ID times packet, detection shall be terminated by any transition of a host control line.

5.2.2 General CPP timing

Figure 4 is an example of CPP timing for a DC device. At event 1, the CPP sequence begins at the IEEE 1284-compatible Idle state. At event 2, the host drives the first byte of the preamble onto the data lines. The host shall maintain the data for a minimum of T_H. The MP device waits T_W time (event 3) before recognizing the byte as stable data. Each of the preamble and function ID bytes is driven in a similar fashion according to the timing shown. In this example, the command is qualified by nStrobe falling low at event 15 and then being driven high at event 16. The terminator byte is driven at event 17 and recognized by the device at event 18.

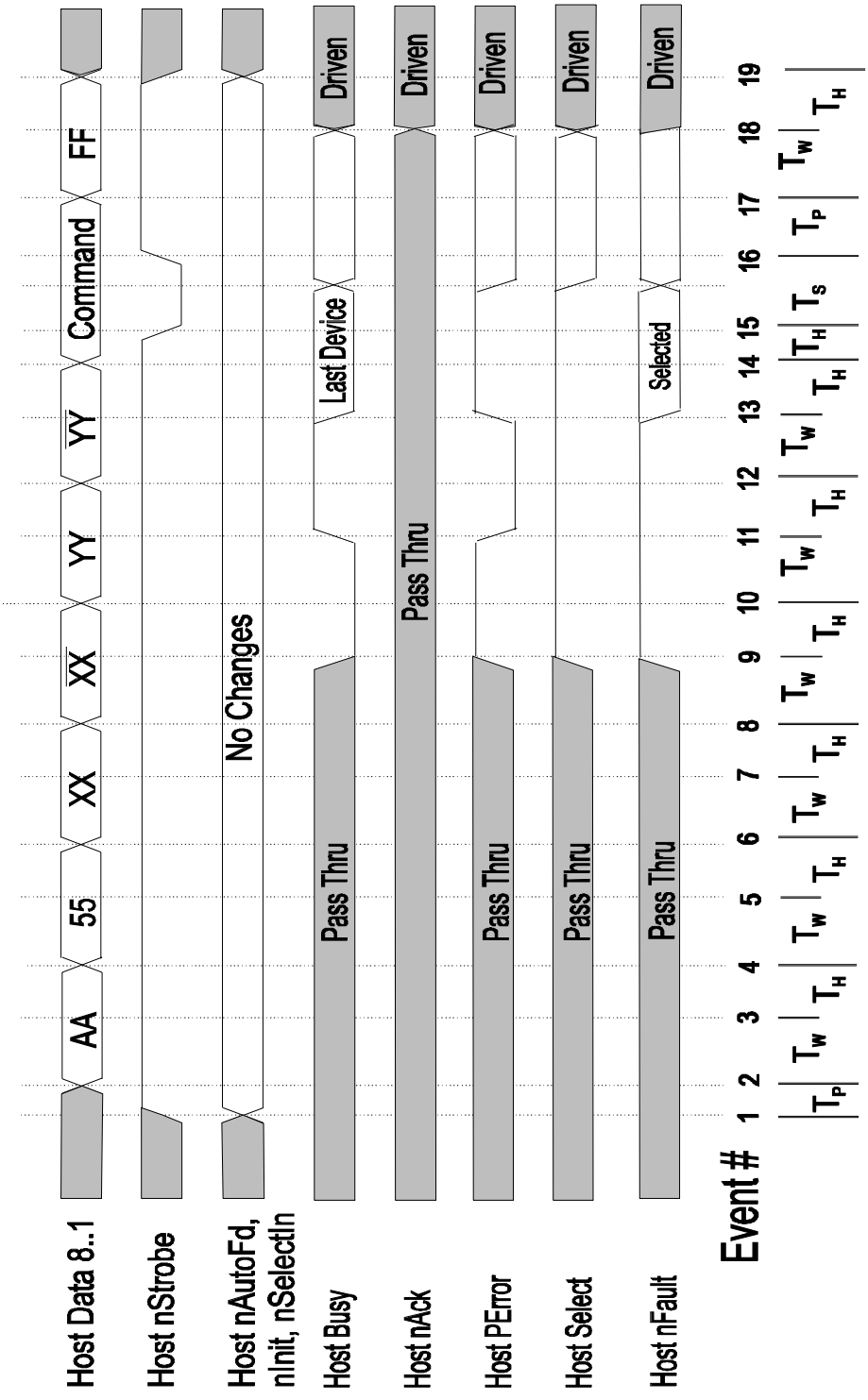


Figure 4—Daisy chain CPP timing example

Table 4 defines CPP timing values.

Table 4—CPP timing values

Time	Minimum	Maximum	Description
T_H	1800 ns	Infinite	Host data width
T_W	750 ns	1200 ns	Peripheral validation time
T_S	900 ns	2000 ns	Command strobe pulse width
T_P	900 ns	Infinite	Setup or hold time
T_A	0	100 ns	Host to pass-through data propagation delay
T_T	10 ns	250 ns	Signal transition time
T_{PR}	1000 ns	1450 ns	Peripheral response time
T_{HR}	1700 ns	Infinite	Host response time
T_{PSR1}	1700 ns	Infinite	Peripheral status response 1
T_{PSR2}	0	750 ns	Peripheral status response 2
T_{HRSEL}	2000 ns	Infinite	Selected status propagation delay
T_{CR}	0	750 ns	Peripheral command response time
T_{HSR}	1050 ns	Infinite	Host command response time

5.2.2.1 Preamble and function ID timing

T_H is the time the host shall maintain stable values on the data lines. T_W is the time in which the MP device shall detect stable data (see Figure 5).

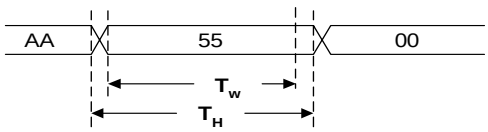


Figure 5—Preamble and function ID timing

5.2.2.2 MP device response time

This timing applies to the periods in the CPP cycle where the host is waiting for a response from the MP device (see Figure 6).

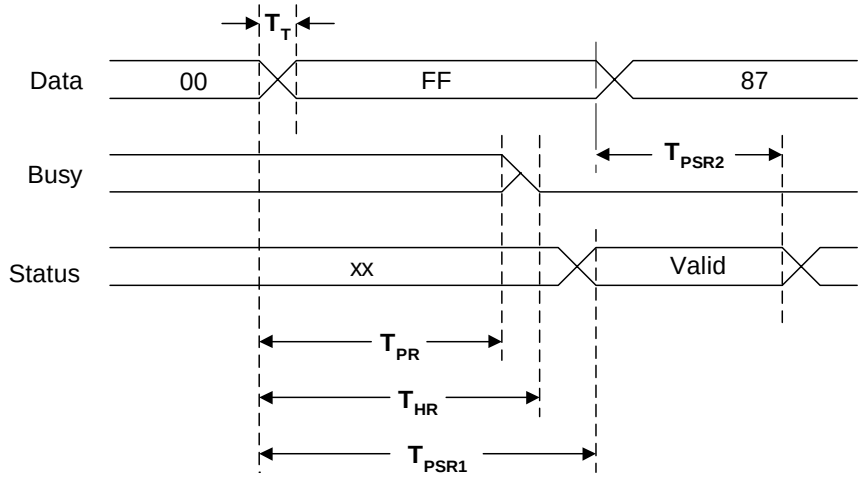


Figure 6—CPP response time example

T_T = 250 ns max. (see 8.3 of IEEE Std 1284-2000)

Maximum time for data to change and for that change to propagate down the cable.

T_{PR} (min.) = T_T + T_W (min.) = 1.0 µs

T_{PR} (max.) = T_T + T_W (max.) = 1.45 µs

Minimum time for MP device response to CPP function ID.

For example, in the DC CPP sequence, the DC device must respond with Busy and PE within T_{PR} of the transition from 0x00 to 0xFF on the data lines.

T_{HR} = 1700 ns min.

Minimum time the host shall wait to read valid response from MP device.

A special case occurs in the DC Select command. The host shall wait T_{HRSEL} after issuing the Select command and before asserting nStrobe. This is to allow the Select status to propagate from the last device on the DC to the host.

T_{HRSEL} = T_{PR} (max.) + T_T + 3T_A = 1.45 µs + 0.250 µs + 3(0.100) = 2.0 µs

T_{PSR1} = T_W (max.) + 2T_T = 1.7 µs

Minimum time the host must wait until reading the pass through status lines.

T_{PSR2} = T_W (min.) = 750 ns

Maximum time the host has to read the pass through status after changing the CPP symbol.

5.2.2.3 Command strobe

Figure 7 is used to indicate timing constraints of MP device response when command strobe is utilized to validate commands.

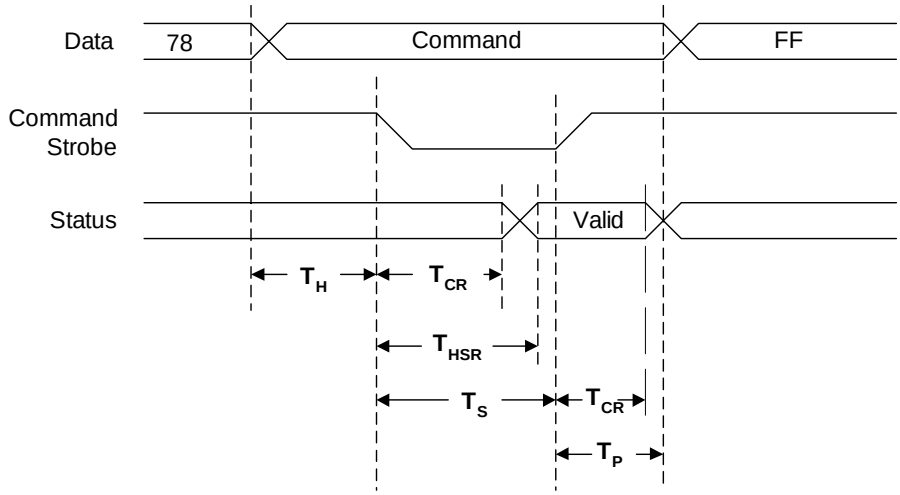


Figure 7—CPP common strobe timing

T_{CR} (min.) = 0

T_{CR} (max.) = 750 ns

Time for MP device to respond with status in response to the host assertion of the command strobe signal.

T_{HSR} = 1050 ns min.

Time the host shall wait for a valid response from the MP device in response to a command strobe.

T_S = 900 ns min.

Command strobe pulse width.

T_P = 900 ns min.

Various setup and hold times.

5.3 DC architecture

5.3.1 Interconnection

This subclause describes the physical configuration of the daisy chain. It consists of one or more devices having both host and pass-through connectors for the parallel port as defined in IEEE Std 1284-2000 (Type A, B, or C). These devices may be used together by connecting the host connector on one device to the computer's parallel port and the pass-through connector to the host connector of the next device. Up to four devices may be connected in this fashion, as shown in Figure 2. Any parallel port device, which is a non-MP device (i.e., a standard printer), may be connected to the pass-through connector of the last device for a total of five devices on the chain.

The daisy chain device host port connector shall be one of the following:

— IEEE 1284 Type A plug
— IEEE 1284 Type B receptacle
— IEEE 1284 Type C receptacle

The DC device pass-through port connector shall be one of the following:

— IEEE 1284 Type A receptacle
— IEEE 1284 Type C receptacle

5.3.2 Implementation

The parallel port contains three groups of signals: Status, Control, and Data.

The status lines (Busy, nAck, PError, Select, and nFault) are inputs to the host. The DC device shall be able to read these inputs from the pass-through port and re-drive them to the host port. It also shall be able to block the pass-through status to the host port.

The control lines (nSelectIn, nInit, nAutoFd, and nStrobe) are outputs from the host. The DC device shall be able to receive host port control signals and re-drive them to the pass-through port. The daisy chain device shall be able to latch the state of the host port control signals driven to the pass-through port.

The Data lines (Data8–Data1) are bidirectional, and shall not interfere with the bidirectional data flow between the host and pass-through ports.

With the DC implementation the Status and Control lines are point to point and shall be governed by the driver and termination requirements as outlined in IEEE Std 1284-2000. The interface to the data lines shall be governed by this document (see 4.5).

5.3.3 Operation

DC devices default to transparent mode until addressed and selected. Upon selection, the peripheral and host port are in the Compatibility Mode. Once selected, the peripheral may be negotiated into another mode. The interface shall always terminate to Compatibility Mode prior to de-selection. IEEE Std 1284-2000 details the method which the host and peripheral may use to negotiate a communications mode. See Table 5 for an example of DC device state changes. In this example, DC device 1 is attached to the host and DC device 2 is attached to the pass-through port of DC device 1.

Once a device is addressed, that address remains in effect until a new Assign Address command is completed.

In the un-selected state, the DC device is considered transparent and shall pass through and not alter Data, Control or Status lines. In selected state, the selected DC peripheral shall be the only device actively communicating with the parallel port host. The state of the output control lines is latched at the time of selection. While selected, the pass-through port of the DC device shall maintain the value of the output control lines in this latched state.

When selected, the host port status lines of the DC device reflect the state of the DC device, not that of the pass-through port.

When not selected, the host status lines reflect the status of the DC device pass-through port.

Table 5—Daisy chain device state table

Action	DC device 1 state	DC device 2 state	Host: IEEE 1284 Mode
Power Up	Un-Addressed, Un-selected	Un-Addressed, Un-selected	Compatibility Mode
Assign Address	Addressed 0, Un-selected	Addressed 1, Un-selected	Compatibility Mode
Select Device 1	Selected	Un-selected	Compatibility Mode
Negotiate	Selected	Un-selected	New Mode
Data Transfer	Selected	Un-selected	New Mode
Terminate	Selected	Un-selected	Compatibility Mode
De-Select Device	Addressed, Un-selected	Addressed, Un-selected	Compatibility Mode
Select Device 2	Un-selected	Selected	Compatibility Mode
Negotiate	Un-selected	Selected	New Mode
Data Transfer	Un-selected	Selected	New Mode
Terminate	Un-selected	Selected	Compatibility Mode
De-Select Device	Addressed, Un-selected	Addressed, Un-selected	Compatibility Mode

When a CPP preamble has been detected the host status lines reflect the appropriate status response for the DC CPP protocol (see Figure 8).

5.3.4 General DC CPP

This subclause details the CPP for managing the DC devices. See Figure 9 for logic flow information.

Figure 10 is a timing diagram showing a DC CPP transaction. At event 1 the CPP sequence is initiated. The host control signals nAutoFd, nInit, and nSelectIn are not allowed to transition until event 27. Because the CPP is initiated from IEEE 1284 Compatible Idle state, the signal nStrobe is high at event 1 and not allowed to transition until event 22. Pass-through control signals shall be stable at event 2. DC devices are allowed T_A maximum time for propagation of the control signals from the host port to the pass-through port. Similarly, they are allowed T_A time to propagate status lines from the pass-through port to the host port. The pass-through port status lines are driven to the host status lines.

At event 3 the host places the first preamble value on the data lines. The host shall hold this data for a minimum of T_H and shall be accepted by the DC device as stable CPP sequence data after time T_W, event 4. The host follows the first preamble value with the remaining values as shown at events 5, 7, and 9. After the DC device detects a stable 0xFF in the function ID, it responds by taking over the host status lines and setting Busy low, and PError, Select, and nFault high (event 10). Status lines are received by the DC device on the pass-through port at events 11–13 (as early as event 11 and as late as event 13), events 15–17, and events 19–21.

The mechanism used by a DC device to determine if it is the last device in the chain is the status information. After a stable 0x87 is recognized, the device will set PError low, and Busy, Select and nFault

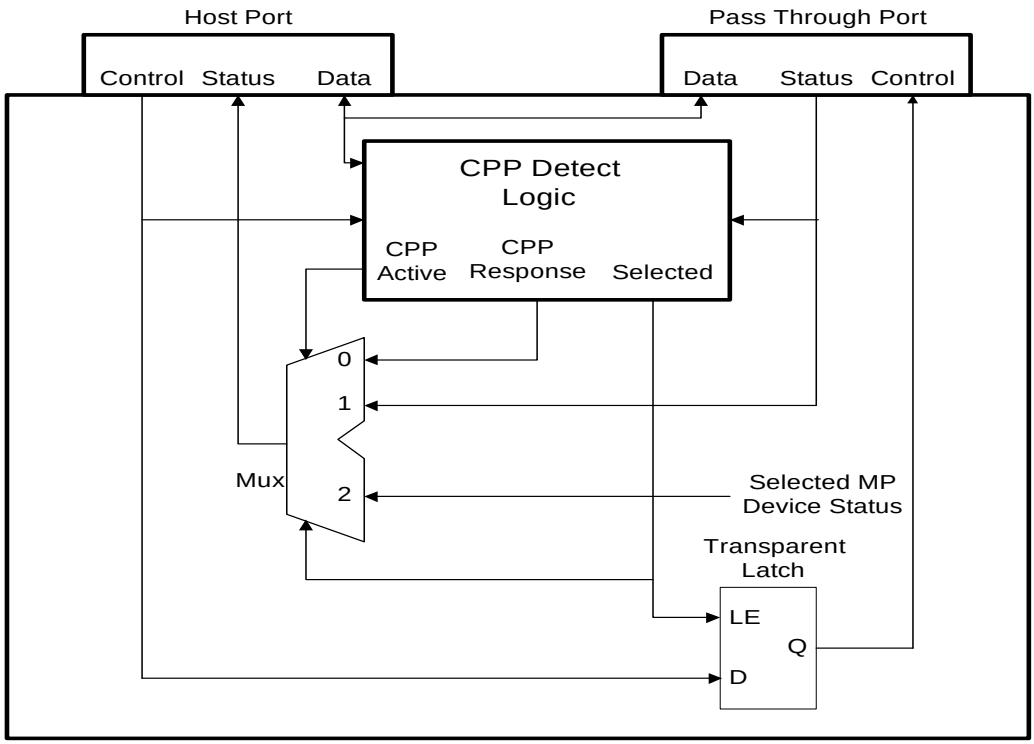


Figure 8—DC device signal flow example

high, event 14. Each device is responsible for monitoring its pass-through port to determine if a downstream DC device is attached. The device does this by comparing its status lines to the pass-through status lines. A DC device detects a downstream device attached if its host and pass-through status lines are in identical states between events 11-13, and between events 15-17. If all the above conditions are not met, then the DC device is the last DC device on the chain.

After a stable 0x78 is detected, the DC device sets PError and Select high, event 18. Busy is set low if other DC devices were detected on the pass-through port, or set high if it determines that it is the last DC device. If the device is the last device, then nFault is set high if the device is currently selected. If the device is not the last device, then nFault is set high if the device is selected or the pass-through nFault is high. The device will maintain these values on the status lines until the falling edge of nStrobe, or until it aborts the packet due to an unexpected control line transition. This handshake allows the host to determine both if a device is present before finishing the communication (so that it will not pulse nStrobe to a device that would misinterpret the data), and if any device is currently selected. The handshake also allows a device to determine that it is the last DC device so that it can block nStrobe to any non-MP device connected to its pass-through port.

The host places the DC Command on the data lines at event 20. The host signal nStrobe is driven low at event 22. The DC Device uses the falling edge of nStrobe to accept the command (event 23) and stop driving the host status line values for last device and selected. The status lines from the pass-through port are driven back to the host beginning at event 26. The signal nStrobe is driven high by the host at event 24.

If a device finds that it is the last DC device in the chain it shall assume that a non-MP Device follows. The signal nStrobe is latched as early as state 18 and as late as state 21, and driven high to the pass-through port

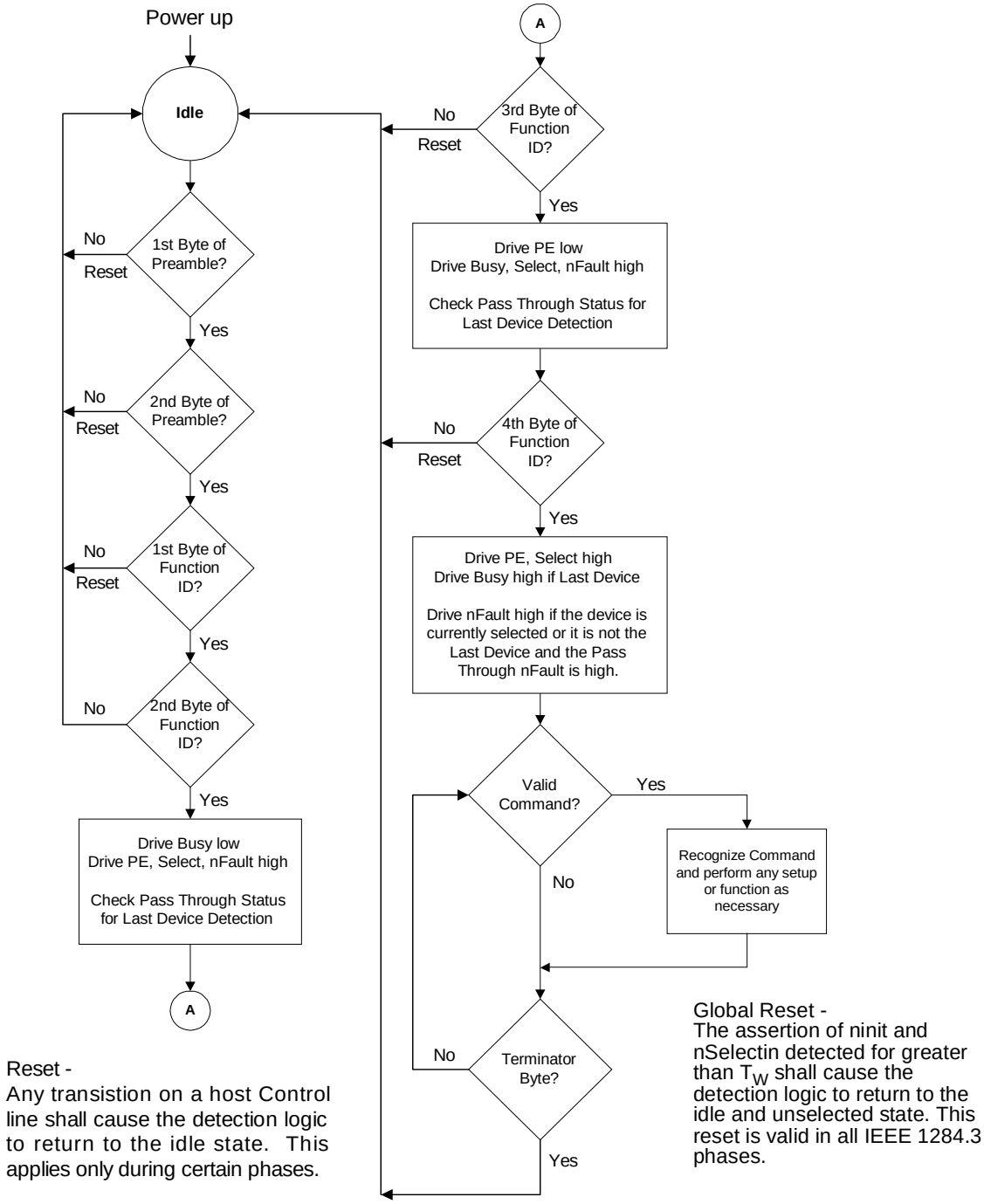
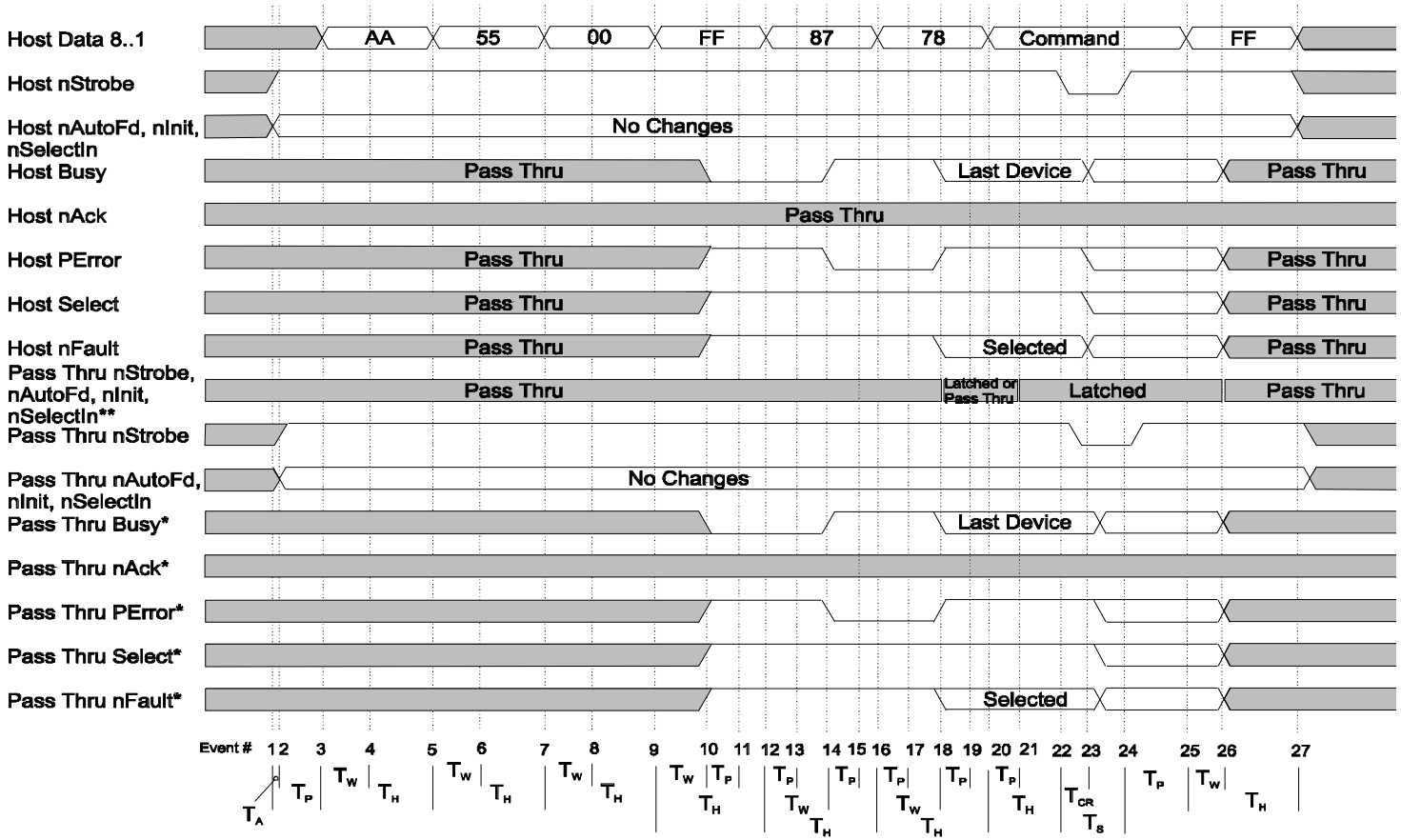


Figure 9—DC CPP command flowchart



*These status lines shall be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

**These control lines shall be driven as shown if the device determines that it is the last IEEE 1284.3-compatible device.

Figure 10—Standard daisy chain CPP sequence

until event 27 when it is allowed to become transparent. The non-MP device shall not see any transitions on any of the control lines during the CPP packet.

5.3.5 DC communications

5.3.5.1 DC CPP command codes

The command byte in the CPP packet represents a command code and possibly an address as well. ‘aa’ refers to DC device address 0–3. The defined codes are shown in Table 6.

All DC devices shall implement the mandatory commands. If implemented, all optional commands must be implemented as a group.

Table 6—DC CPP command codes

DC CPP command codes	CPP command codes	Operation	Implemented
(0x00–0x03)	0000 00aa	Assign Address aa to the current device	Mandatory
(0x08–0x0B)	0000 10aa	Query Interrupt from device aa	Mandatory
(0x30)	0011 0000	De-select device	Mandatory
(0x40)	0100 0000	Disable Daisy Chain Interrupts	Optional
(0x48)	0100 1000	Enable Daisy Chain Interrupts	Optional
(0x50–0x53)	0101 00aa	Clear Interrupt Latches on device aa	Optional
(0x58–0x5B)	0101 10aa	Set Interrupt Latch on device aa	Optional
(0xE0–0xE3)	1110 00aa	Select Device aa	Mandatory

Reserved commands are reserved for future expansion of the specification. “Do not use” commands are commands that have been deprecated from previous drafts of this standard. See Table 7 for Reserved, Do Not Use, and Vendor Specific command definitions.

Table 7—DC CPP miscellaneous command codes

CPP command codes	CPP command codes	Operation
(0x04–0x07)	0000 01xx	Reserved
(0x0C–0x0F)	0000 11xx	Reserved
(0x10–0x1F)	0001 xxxx	Reserved
(0x20–0x27)	0010 0xxx	Do Not Use
(0x28–0x2F)	0010 1xxx	Reserved
(0x31–0x3F)	0011 xxx1	Reserved
(0x41–0x47)	0100 0xx1	Reserved
(0x49–0x4F)	0100 1xx1	Reserved
(0x54–0x57)	0101 01xx	Reserved
(0x5–0x5F)	0101 11xx	Reserved
(0x60–0x7F)	011x xxxx	Reserved
(0x80–0x9F)	100x xxxx	Vendor Specific Commands
(0xA0–0xAF)	1010 xxxx	Vendor Specific Commands
(0xB0–0xBF)	1011 xxxx	Reserved
(0xC0–0xCF)	1100 xxxx	Reserved
(0xD0–0xD7)	1101 00xx	Do Not Use
(0xD8–0xDF)	1101 1xxx	Reserved
(0xE4–0xE7)	1110 01xx	Reserved
(0xE8–0xEF)	1110 1xxx	Reserved
(0xF0–0xFF)	1111 xxxx	Invalid

5.3.5.2 DC CPP command details

5.3.5.2.1 Assign Address (codes 0x00–0x03)

This command is used to assign unique addresses to each device on the daisy chain. The addresses are assigned from 0–3, in the order that the devices are attached to the host. The host will watch the handshake from the peripherals as the CPP packet is sent. After the escape sequence, event 20, the host asserts 0x00 on the data bus (address 0), and pulses the nStrobe line. The first device sees this strobe, and accepts the address, having blocked the nStrobe from passing to the pass-through port, event 28, then becomes

transparent. Status from the pass-through port is now passed through to the host port. The host now sees the next device's status lines. The host assigns the next address and loops assigning addresses in the same manner until all devices are assigned. If the status indicates last device, the host assigns the address then, asserts 0xFF on the data bus to end the transaction.

Figure 11 and Figure 12 show an Assign Address command. Figure 14 shows timing boundaries for status being passed through the device while Figure 12 shows actual status values for the same example. The timing diagrams show device 0 being assigned an address followed in the chain by another three DC devices.

5.3.5.2.2 Enable Daisy Chain Interrupts (code 0x48)

This command sets the interrupt enable latch in all DC devices. DC interrupts may be generated between the time bounded by the Enable Daisy Chain Interrupts command and the Disable Daisy Chain Interrupts command (see 5.3.5.3 for more information on Daisy Chain Interrupts). The host has the option of internally enabling interrupts before giving the qualifying strobe. The DC device shall enable interrupts between events 30 and 32. If a DC device has a pending interrupt, it is asserted between events 30 to 32. The nAck line is driven high from event 30 to 31, then driven low from event 31 to 32 (see Figure 13).

5.3.5.2.3 Disable Daisy Chain Interrupts (code 0x40)

The Disable Daisy Chain Interrupts command is shown in Figure 14. This command clears the interrupt enable latch in all devices, preventing any daisy chain device from raising an interrupt. This command will be issued before selecting any device on the chain to prevent an interrupt from one device corrupting the status lines from another device. The daisy chain device may actually disable interrupts as early as event 22 or as late as event 33. See 5.3.5.3 Interrupt handling for a discussion on interrupt handling.

5.3.5.2.4 De-select Device (code 0x30)

The De-select Device packet is shown in Figure 15. This command causes a selected DC device to become unselected and return to transparent pass-through mode. At event 26 the device releases the pass-through status to the host status lines. The signal nFault is driven low from event 34 to event 26 to signal that the de-selection was successful. After event 26, nFault becomes transparent. The DC device shall be in IEEE 1284 Compatibility Mode before the De-select Device command is issued.

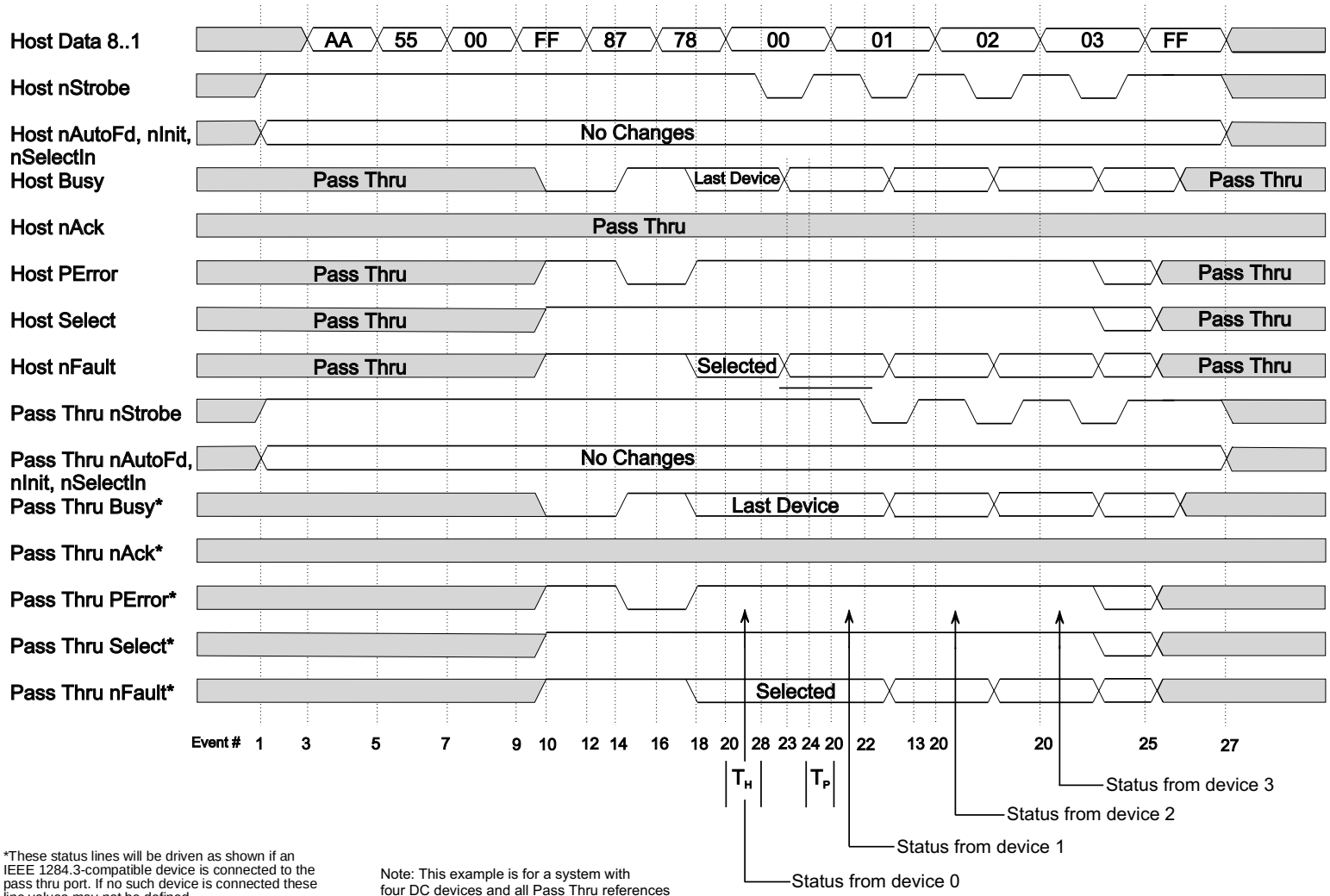
5.3.5.2.5 Select Device (codes 0xE0–0xE3)

Figure 16 shows a generic select sequence of a DC device, which is followed in the chain by another DC device. All DC devices are in transparent mode and not selected before issuing this command. The host sends the Preamble and Function ID sequence, monitoring the handshake from the device. If the device is present, it enters selected mode and verifies the selection by raising nFault at event 35. At event 36 it blocks the status being received on the pass-through port from DC devices or non-MP devices further down the chain.

Once selected, the DC device shall be in Compatibility Mode; the device may optionally negotiate to a different mode per IEEE Std 1284-2000.

5.3.5.2.6 Query Interrupt (codes 0x08–0x0B)

When a device detects the Query Interrupt command to its address, it responds by setting Busy low if it has an internal device interrupt pending, high if not. Additionally, the device sets PError and Select to its address (PError = MSB, Select = LSB), so that the host can distinguish between a valid response and no response (i.e., querying status from a nonexistent device). Refer to events 40–43. The status lines are in pass-through when the Query Interrupt is not directed to the DC device's address. Refer to Figure 17 for an example of a



*These status lines will be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

Note: This example is for a system with four DC devices and all Pass Thru references are referenced to the Pass Thru Port of Device 0.

Figure 11—Assign address timing boundaries

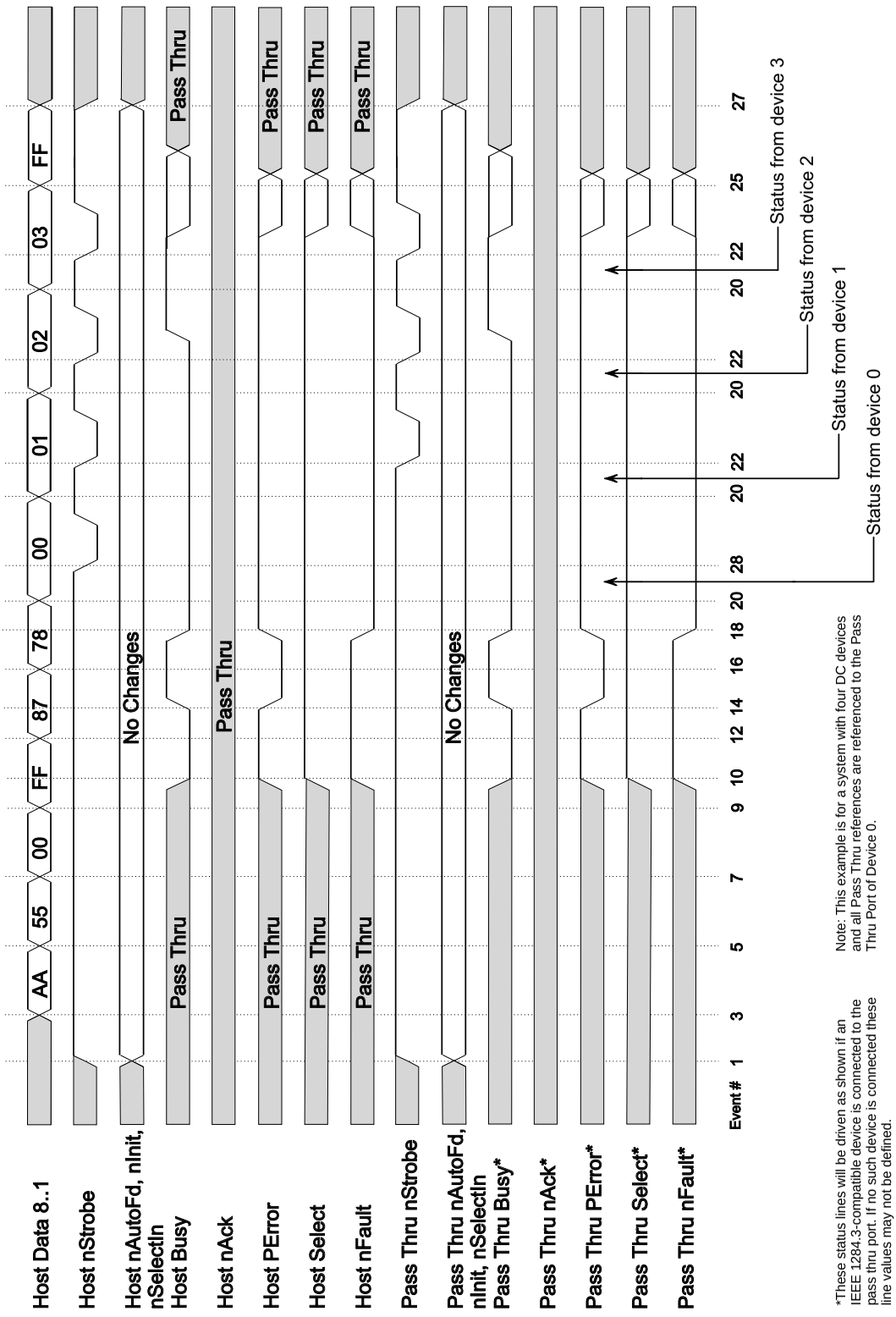
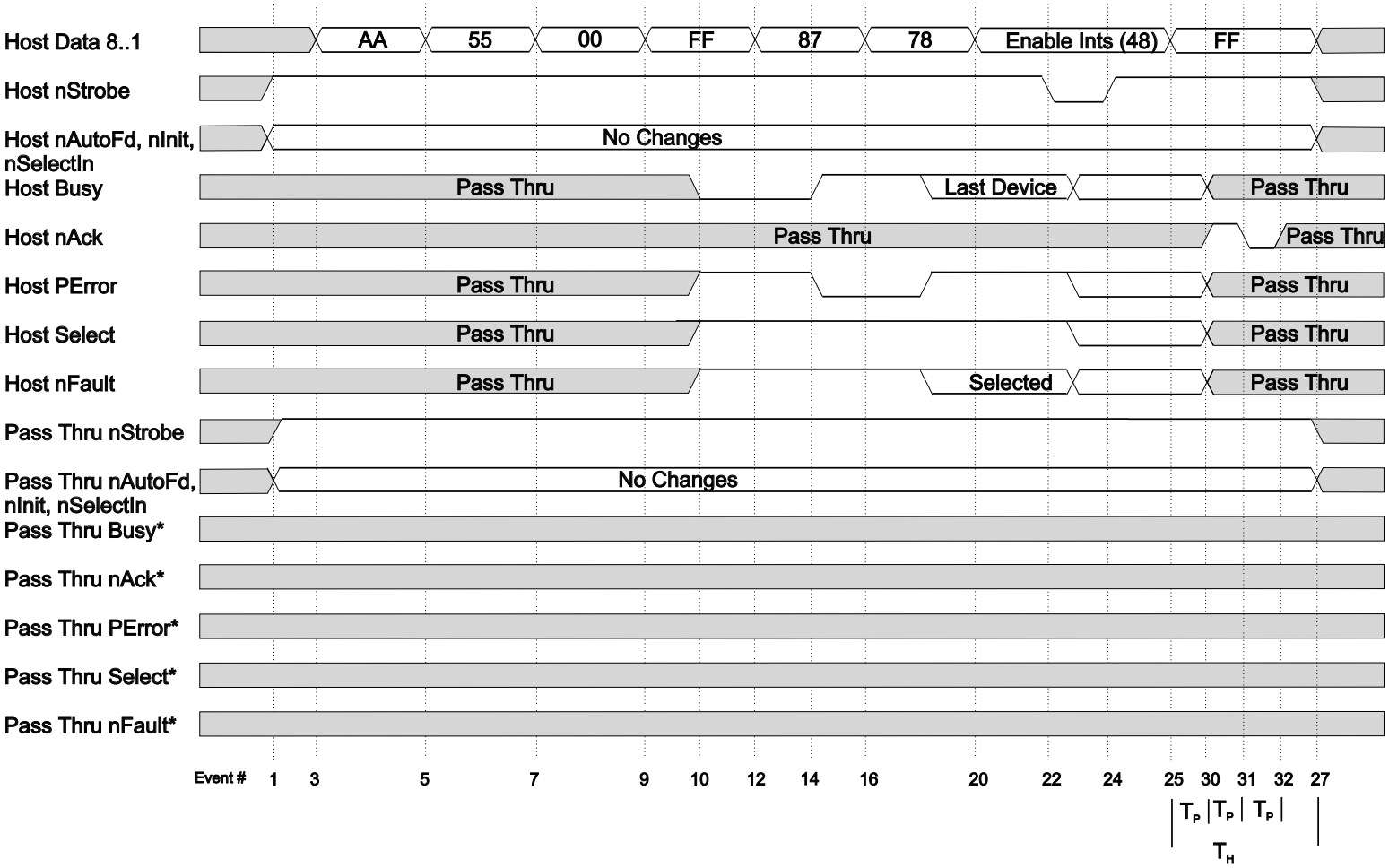
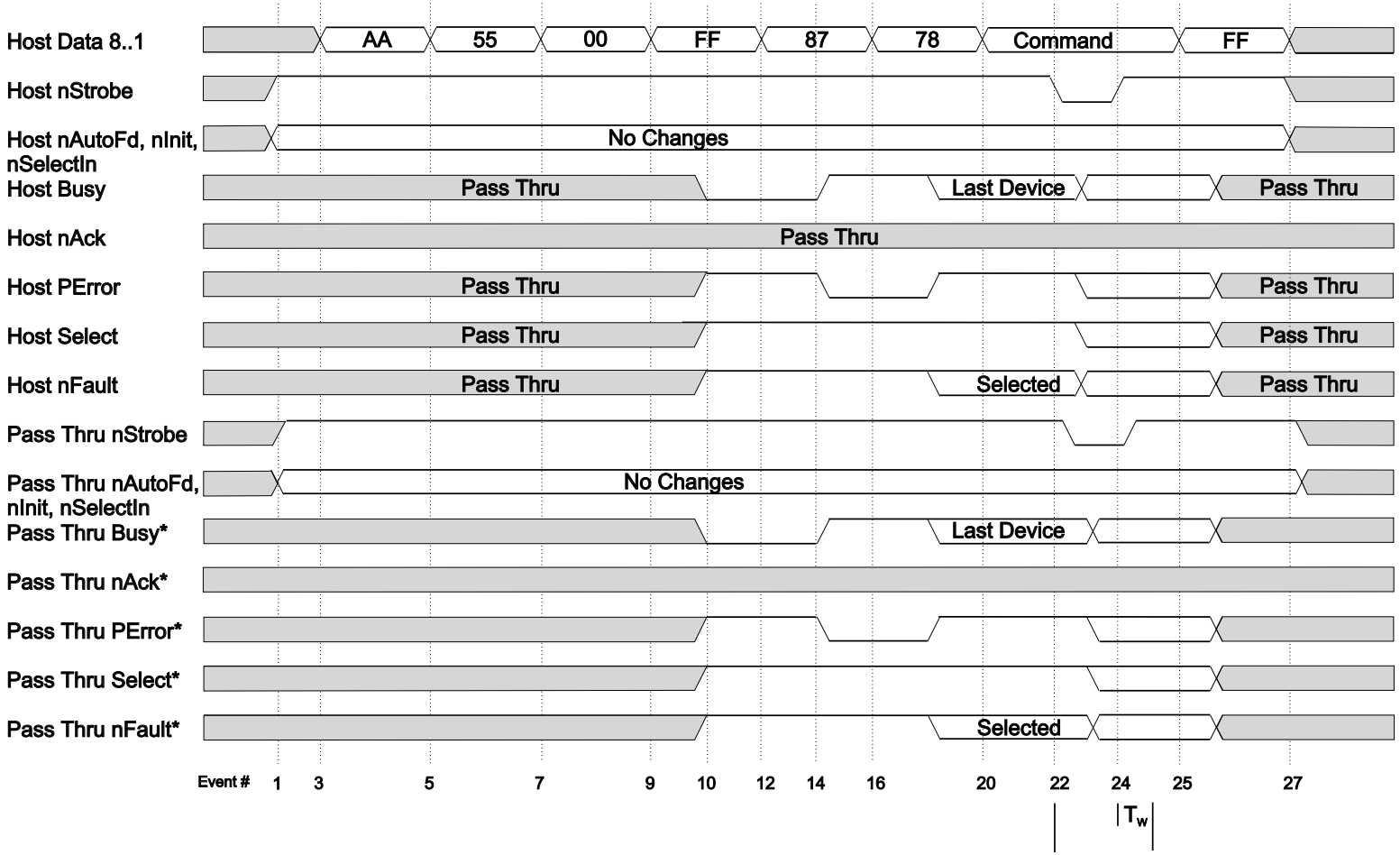


Figure 12—Assign address status example



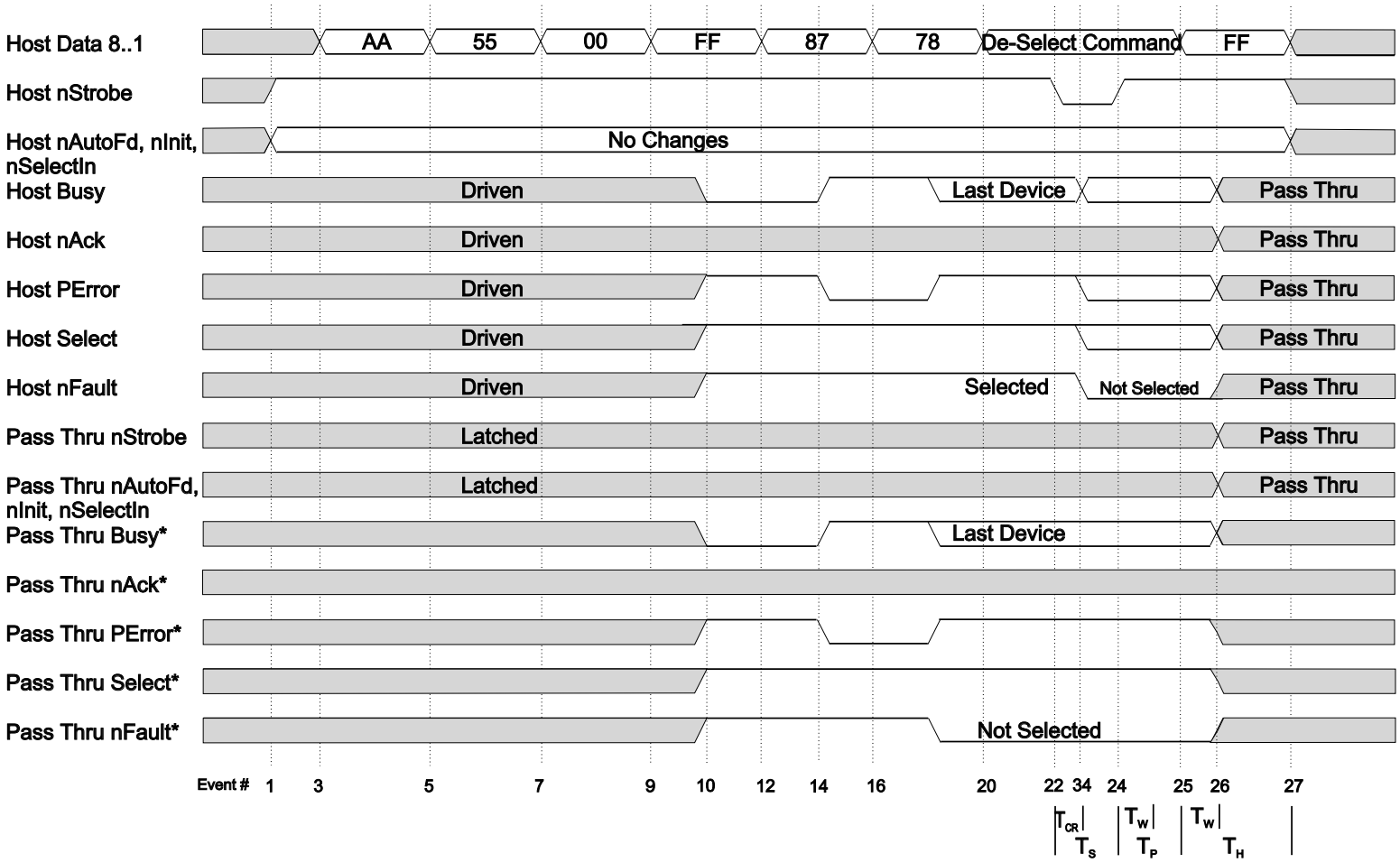
*These status lines will be driven as shown if a non-IEEE 1284.3-compatible device is connected to the pass thru port.

Figure 13—Enable daisy chain interrupts



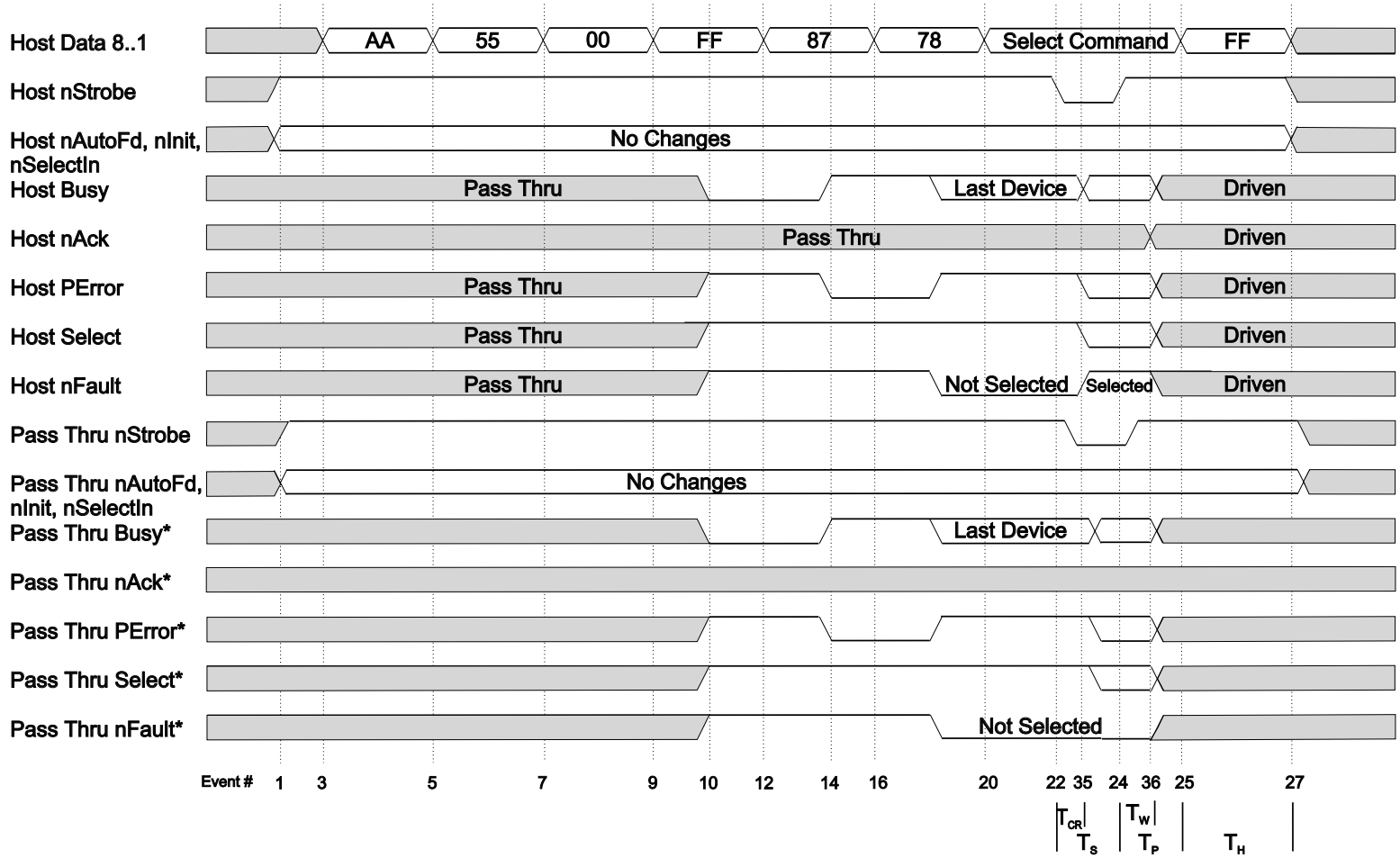
* These status lines will be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

Figure 14—Disable daisy chain interrupts



*These status lines will be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

Figure 15—De-select device packet



* These status lines will be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

Figure 16—Select device packet

query interrupt command. This example shows what device 0 would see if it were followed by three other DC devices. A DC device, which does not support interrupts, must nevertheless respond with no interrupt pending when queried.

5.3.5.2.7 Set Interrupt Latch (codes 0x58–0x5B)

Figure 18 shows the Set Interrupt Latch command packet. This command forces the addressed device to latch an interrupt. It is used primarily as a diagnostic command to test the interrupt path between a host and a DC device. Following this command with an DC Interrupt Enable command (0x48) should cause the system to recognize an interrupt from this device.

5.3.5.2.8 Clear Interrupt Latch (codes 0x50–0x53)

Figure 19 shows the Clear Interrupt Latch command packet. This command will clear the addressed device's interrupt latch. The daisy chain driver issues this command to the device after recognizing its interrupt. The DC device will clear this latch after generating the DC interrupt. This command is used if the host did not enable interrupts but had set this latch.

5.3.5.3 Interrupt handling

The IEEE 1284.3 architecture does not guarantee real-time interrupts. A DC device may only generate an interrupt under the following conditions:

— The DC device is selected: In this case it may generate an interrupt as defined for its current IEEE 1284 mode.
— The DC device is not selected: In this case, it shall only generate a DC interrupt during the time bounded by the reception of the Enable Interrupts command and the Disable Interrupts command.

DC interrupts are treated as Service Request indications. The CPP command "Query Interrupts" is used to poll the daisy chain and determine which devices have outstanding interrupts.

Figure 20 shows how interrupts are enabled and disabled for a typical sequence of selecting and de-selecting a DC device. Upon power up, interrupts from the device are disabled and remain in this state until after addresses on the DC are assigned. The Enable DC Interrupts command allows interrupts to occur. The Enable DC interrupts command shall only be issued when no DC devices are selected. If DC interrupts are enabled, a Disable DC Interrupts command shall be issued before a device may be selected.

5.4 Multiplexor architecture

5.4.1 Interconnection

The multiplexor (Mux) is a port expansion device that can provide up to four parallel port interfaces from a single host parallel port. The Mux has a single host port that connects to the host's parallel port. The Mux shall be connected directly to the host's parallel port. The Mux shall have multiple ports, designated 0 to 3, which transparently connect to the host port when selected. Use of the Mux shall not interfere with any IEEE 1284 communication mode or effect performance.

Mux port 0 is defined as the default port and shall be selected upon power up, after a Reset command, after a De-select command or a Select port 0 command. A non-MP device attached to Mux port 0, either directly attached or at the end of a daisy chain, will be the default device. Non-MP devices may be attached to the other Mux ports and can share the host parallel port by using the appropriate Mux commands.

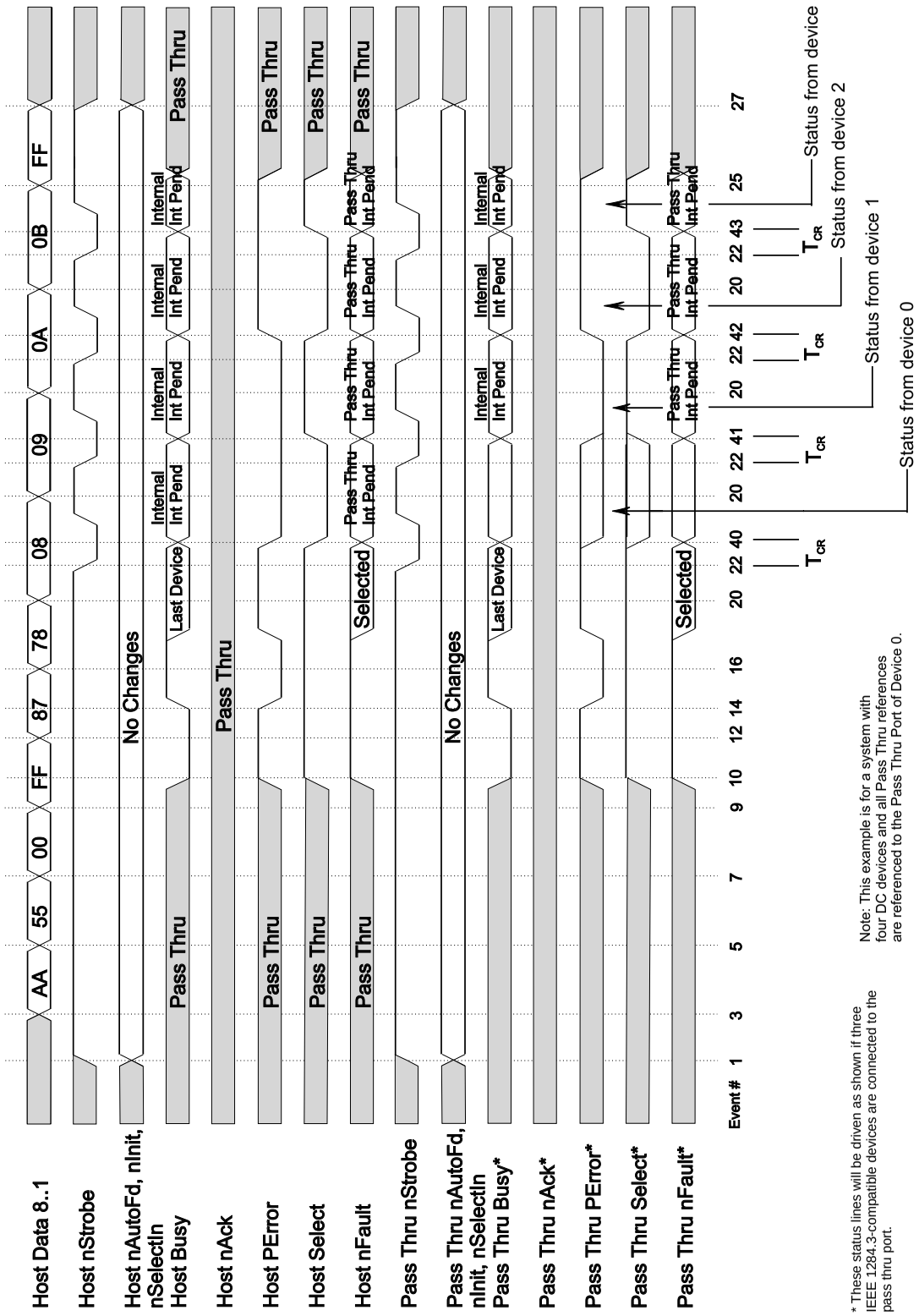
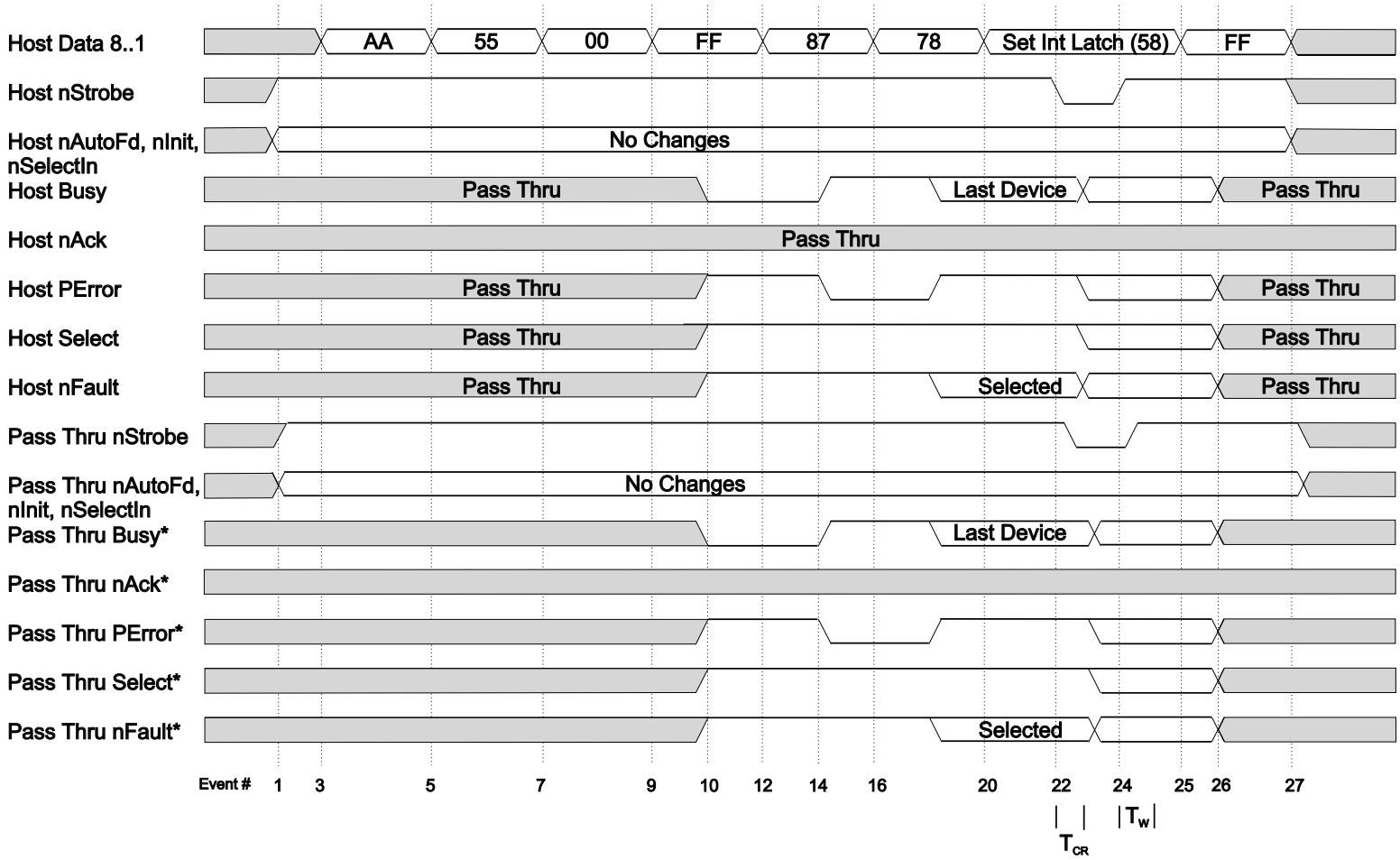
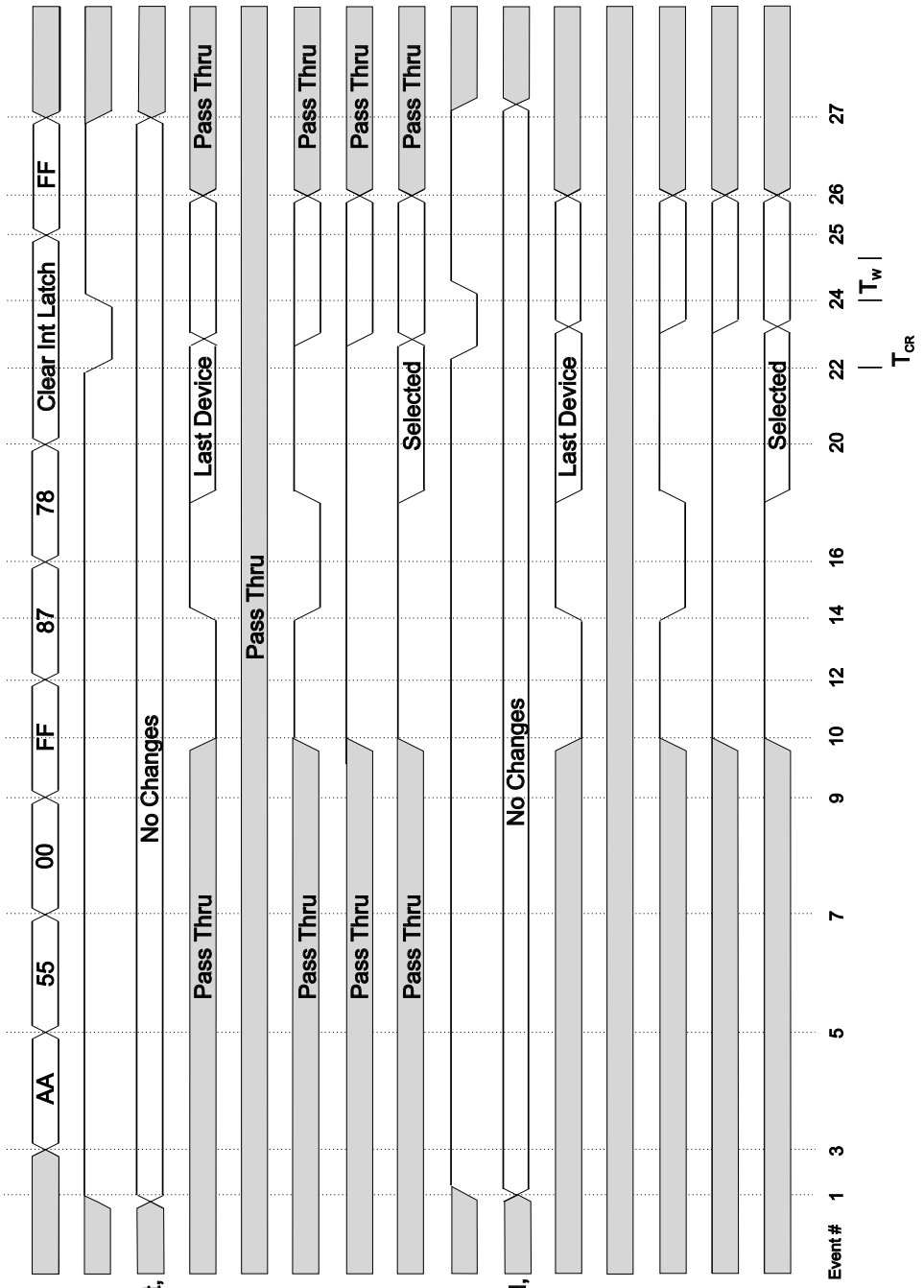


Figure 17—Query interrupt



*These status lines will be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

Figure 18—Set interrupt latch



*These status lines will be driven as shown if an IEEE 1284.3-compatible device is connected to the pass thru port. If no such device is connected these line values may not be defined.

Figure 19—Clear interrupt latch

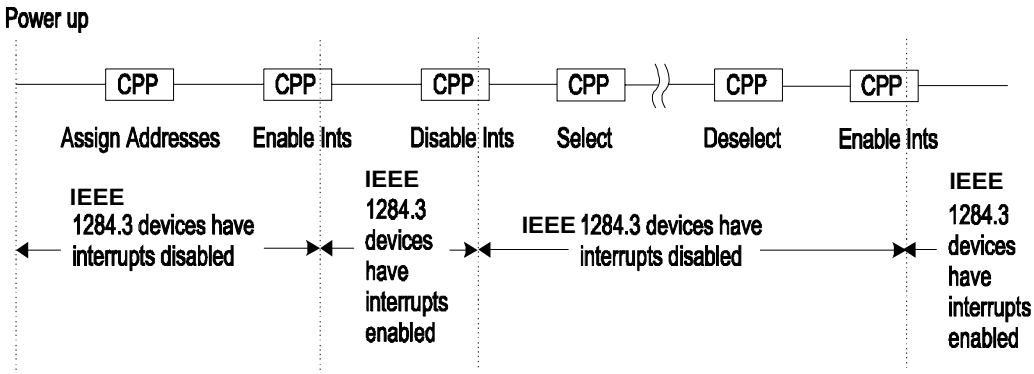


Figure 20—Interrupt commands

5.4.2 Implementation

The parallel port consists of three groups of signals: Status, Control, and Data.

The Status signals (Busy, nAck, PError, Select, and nFault) are inputs to the host. The multiplexor shall be able to pass these signals from the selected Mux port to the host port.

The Control signals (nSelectIn, nInit, nAutoFd, and nStrobe) are outputs from the host port. The multiplexor shall be able to pass these signals from the host port only to the selected Mux port. The Control signals on the non-selected ports shall be latched in the state present when the particular port was last de-selected. At power up, all Mux ports other than Mux port 0 are initially de-selected and the control lines are held in the unasserted state.

The Data signals (Data8–Data1) are bidirectional. The Mux shall be able to establish a transparent path between the host port and selected Mux port, and shall be capable of passing data in either direction. Unselected Mux ports shall isolate their data lines from other Mux ports.

The multiplexor connection shall be electrically compatible with IEEE 1284. The Control and Status lines shall be received and driven in compliance with IEEE Std 1284-2000, the Data lines shall provide for bidirectional communications between the host port and the selected Mux port while maintaining compatibility with IEEE 1284 electrical requirements.

5.4.3 Operation

5.4.3.1 Device selection

The Mux commands are issued using the CPP. Result codes from the Mux commands are returned via the host port Status lines shown in Table 8. These result codes can be seen in Table 9.

Table 8—Max status bit result encoding

Status line	nFault	Busy	PError	Select
Bit position	3	2	1	0 (LSB)

Table 9—Mux commands

Command code	Command code	Operation	Result code
0x50	0101 0000	Reset Mux	0x0
0x51	0101 0001	Mux Present	0x3
0x53	0101 0011	De-select Mux Port	0x0
0x54	0101 0100	Isolate Mux Ports	0xf
0x58	0101 1000	Number of Mux Ports	0x2–0x4
0x5A	0101 1010	Selected Mux Status	0x0–0x3, 0xf
0x60–0x63	0110 00aa	Select Mux Port aa	0x0–0x3, 0xf

The nAck line is used by the Mux to acknowledge Mux CPP commands.

The Mux ports are in one of two states: Selected and De-selected. In the Selected state, the Mux port is transparently connected to the host port. The peripheral on a Selected port will not be able to distinguish between this state and a direct connection to the host port.

In the De-selected state, the Mux port signals shall remain static. The Control lines shall be latched and the Data lines may be latched or high impedance. The de-selected state shall be indistinguishable from the compatible idle forward state.

5.4.3.2 Multiplexor commands

Table 9 shows the commands that a generic Mux device shall implement and respond to.

In contrast to DC commands, nStrobe is not used in conjunction with the Mux CPP command phase. Responses to Mux commands are specified by TPR (see Table 4). The nAck signal is asserted to indicate acknowledgement of all Mux commands (see Figure 21).

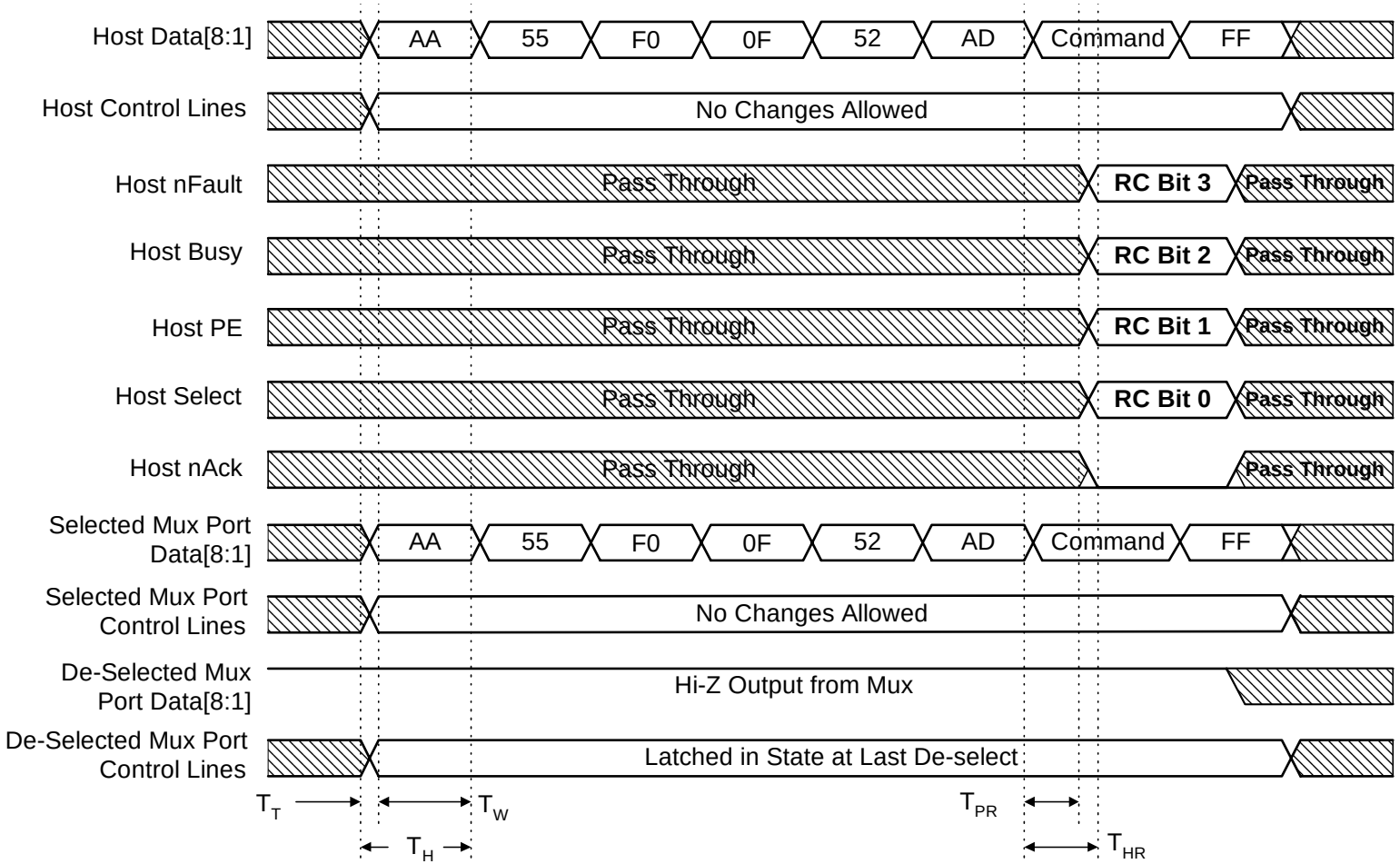
All commands take effect upon recognition of the CPP termination code.

5.4.3.2.1 Reset Mux command (0x50)

The Reset Mux command shall cause any selected Mux port to be de-selected. The default Mux port 0 shall become selected and the Mux set to its power up reset state. All other Mux port control latches shall be set to the state of the Host port control lines at the time when the Reset Mux command is asserted on the data bus. When the command is terminated the control lines for all ports, other than Mux port 0, shall be latched (see Figure 22).

5.4.3.2.2 Mux Present command (0x51)

Upon receipt of the Mux Present command, the Mux shall indicate to the host that a Mux is present by returning the result code as shown in Table 9 (see Figure 23).



"RC Bit[3:0] " refers to the Return Codes defined by Table 5-x, "Mux Commands."

"Pass Through" refers to the Status, Data or Control lines of the currently selected Mux port.

Figure 21—General Mux CPP format

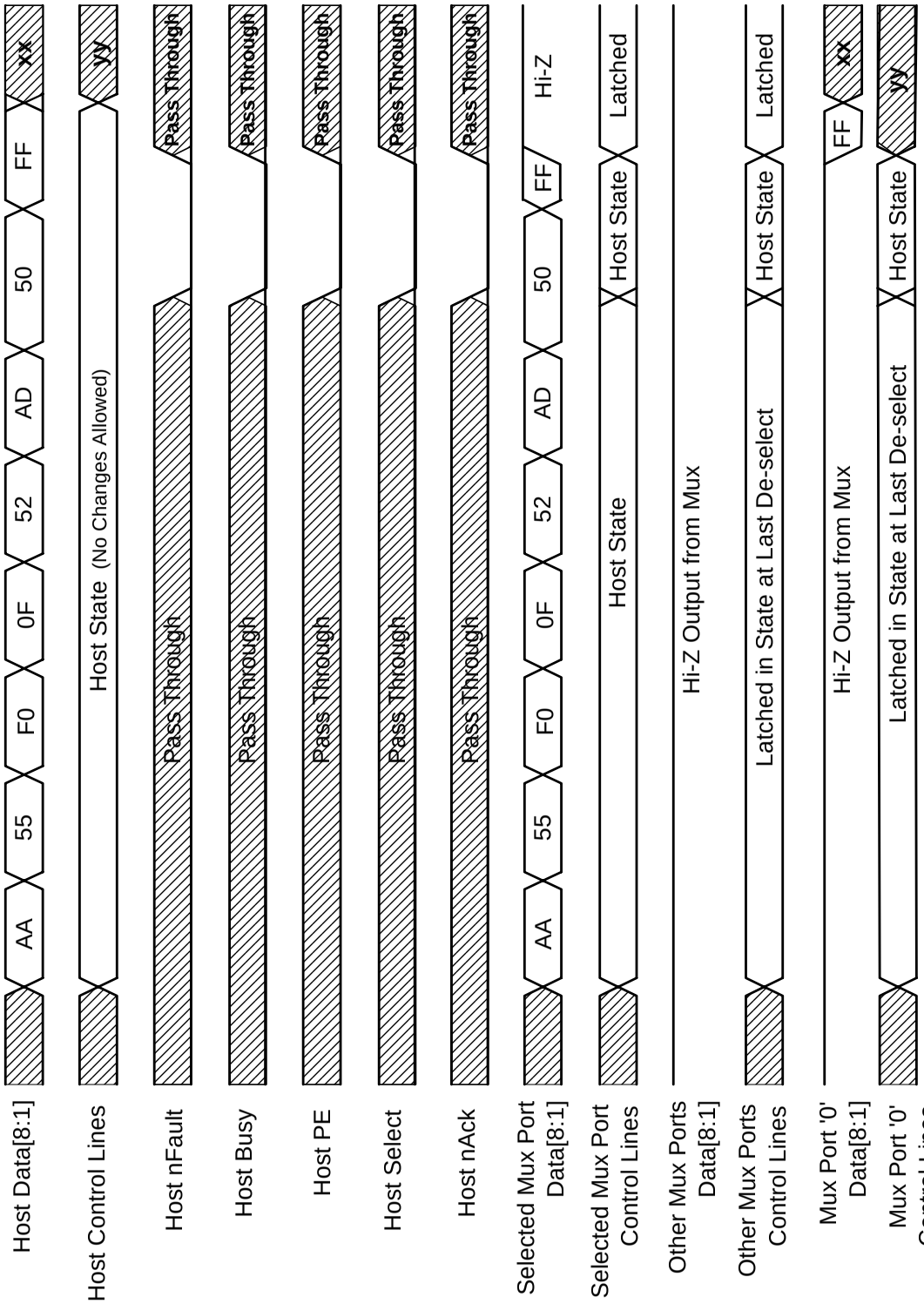
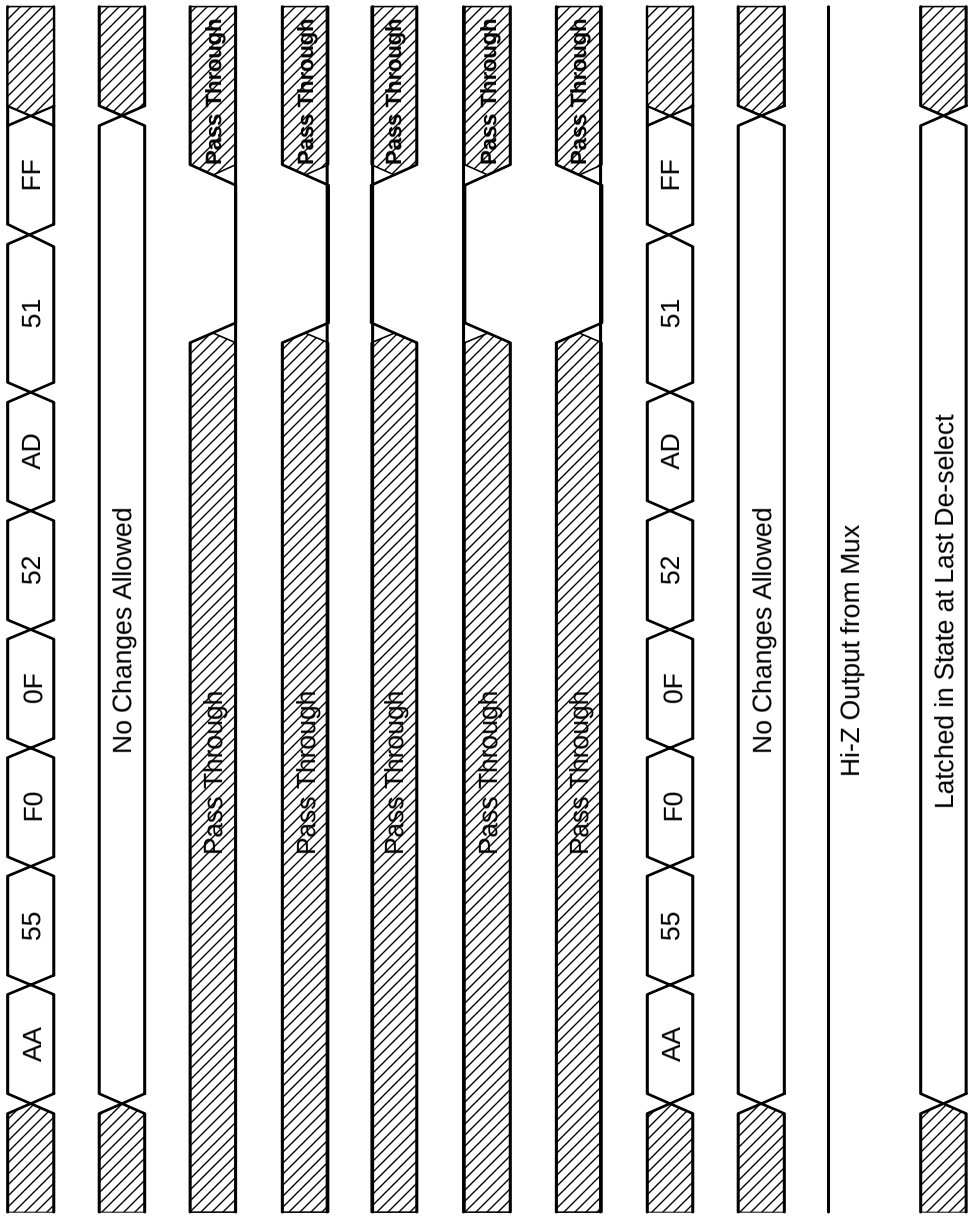


Figure 22—Reset Mux command



"Pass Through" refers to the Status, Data or Control lines of the currently selected Mux port.

Figure 23—Mux Present command

5.4.3.2.3 De-select Mux Port command (0x53)

The De-select Mux Port command shall cause the current Mux port to become de-selected and the default Mux port 0 to become selected (see Figure 24).

5.4.3.2.4 Isolate Mux Ports command (0x54)

The Isolate Mux Ports command shall cause all Mux ports to become de-selected. The Control lines and Data lines shall be put into a high impedance state. This command may be used to implement user initiated “hot plugging” on the MUX port (see Figure 25).

5.4.3.2.5 Number of Mux Ports command (0x58)

This command shall cause the Mux to return the number of Mux ports. The Mux shall return the number encoded as shown in Table 9 (see Figure 26).

5.4.3.2.6 Selected Mux Status command (0x5a)

This command shall cause the Mux to return the Mux port number that is currently selected. If no ports are selected, this command will return 0xf as shown in Table 9 (see Figure 27).

5.4.3.2.7 Select Mux Port aa command (0x60–0x63)

This command shall cause the requested Mux port to become selected. The Mux shall return the requested MUX port number. If an invalid Mux port is requested, the result code 0xf shall be returned (see Figure 28).

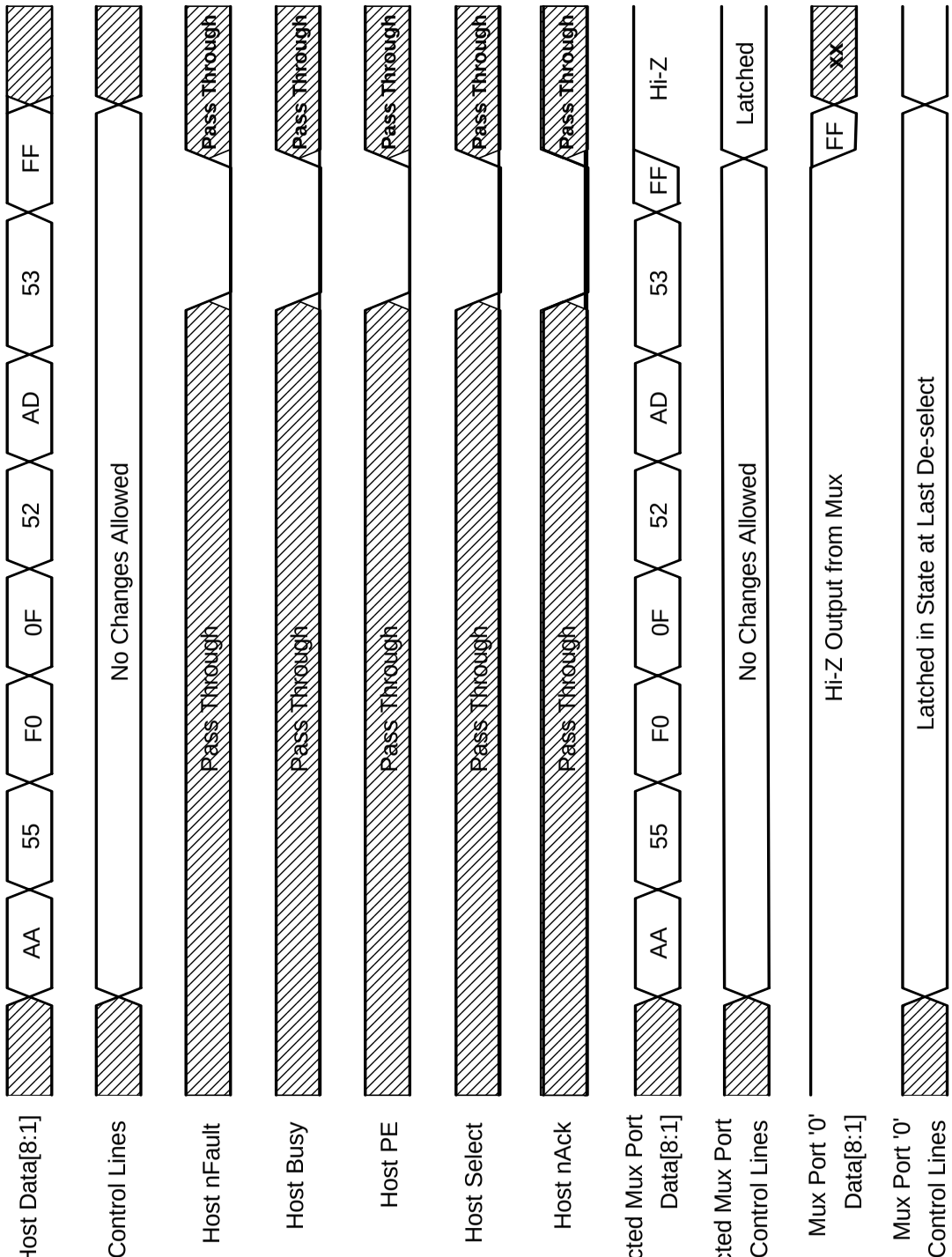


Figure 24—De-select Mux Port command

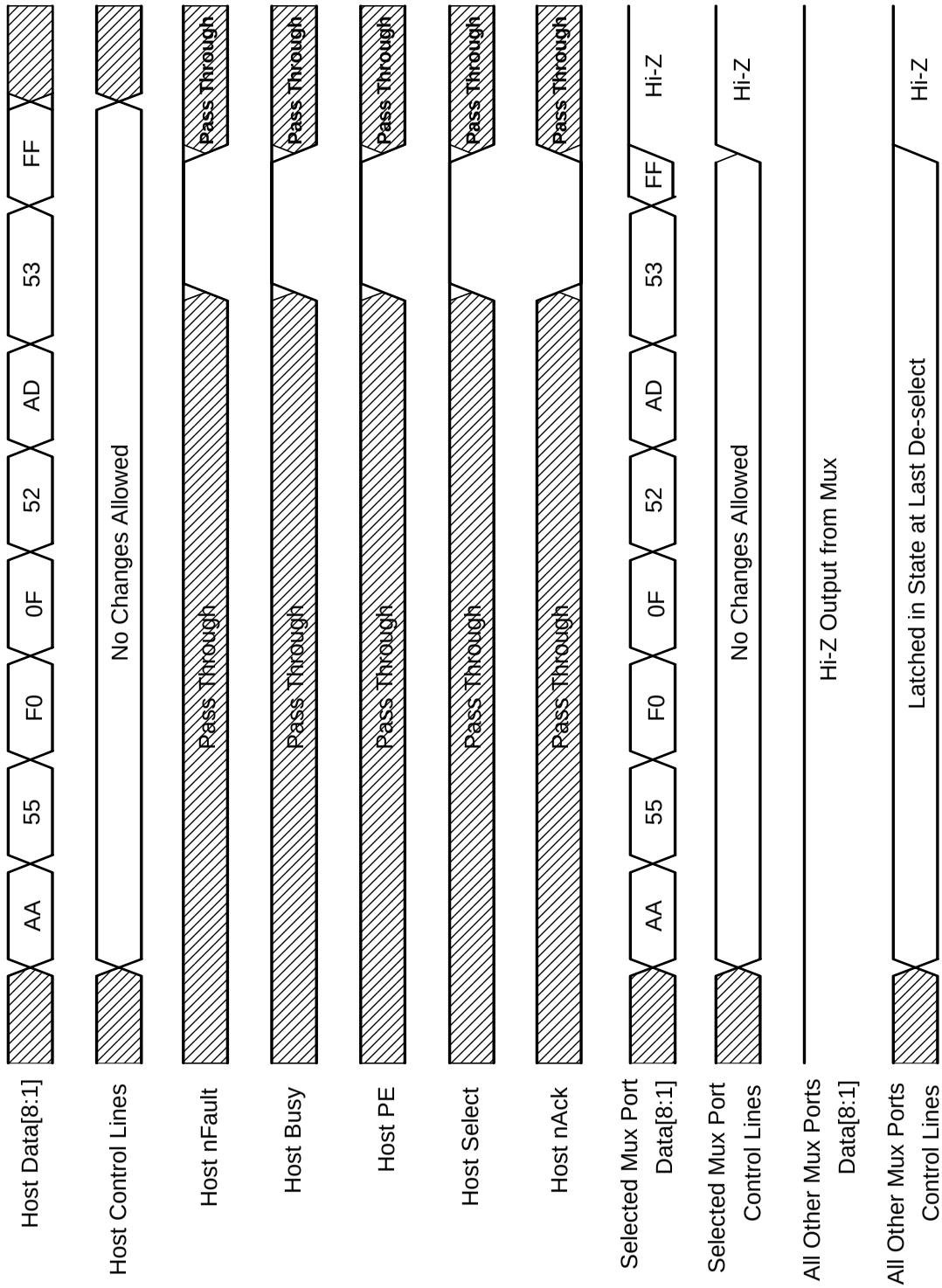


Figure 25—Isolate Mux Ports command

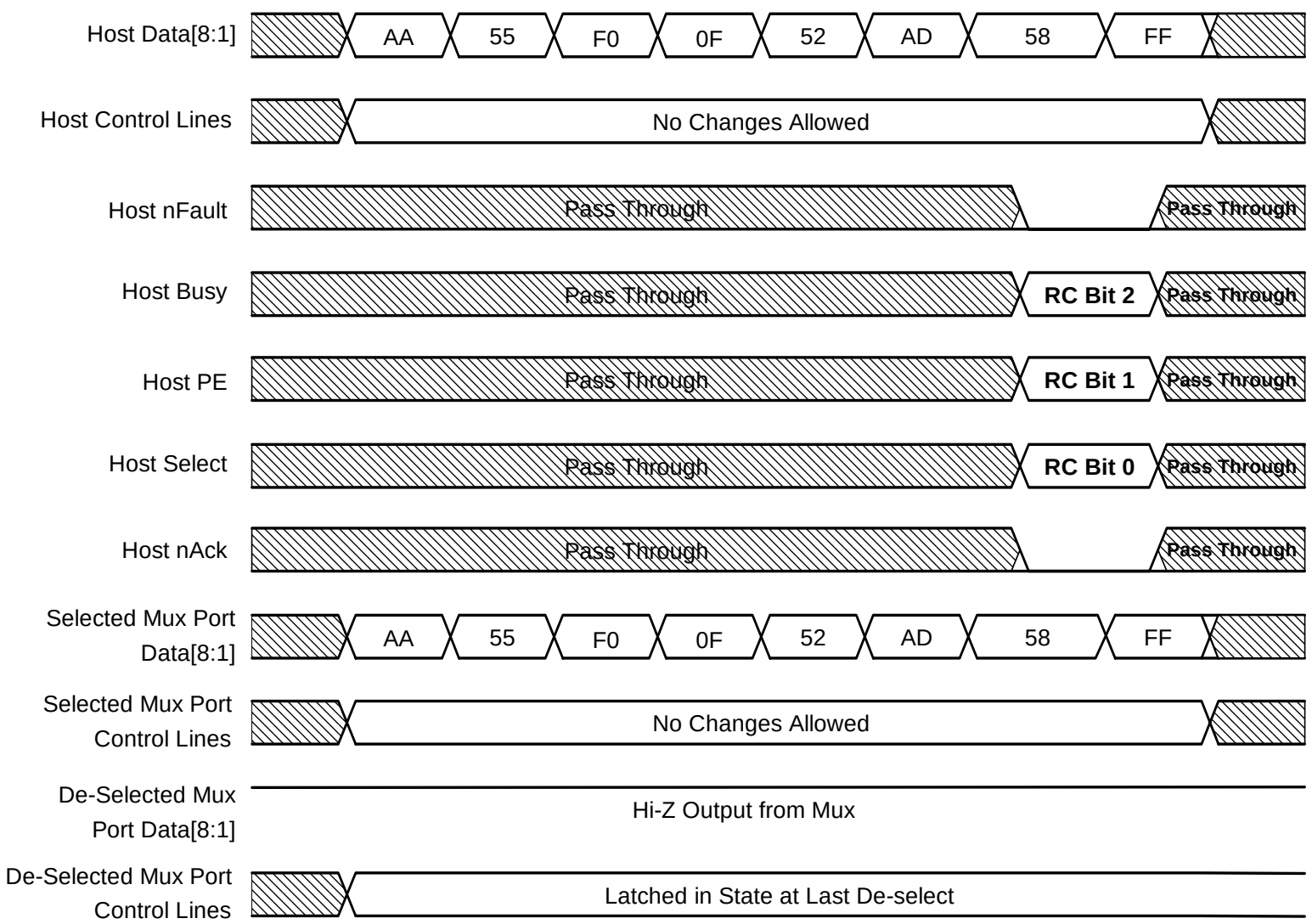


Figure 26—Number of Mux Ports command

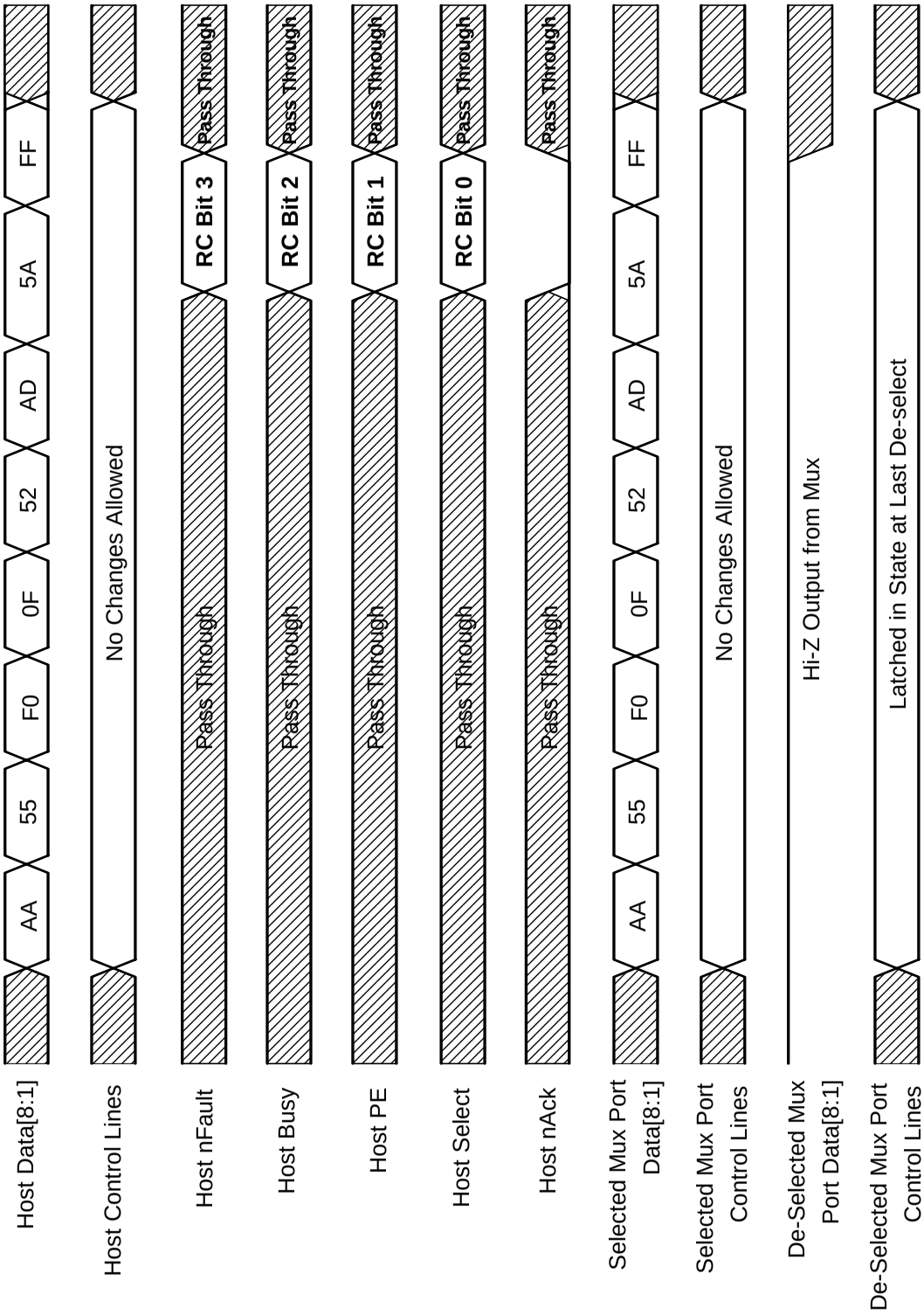
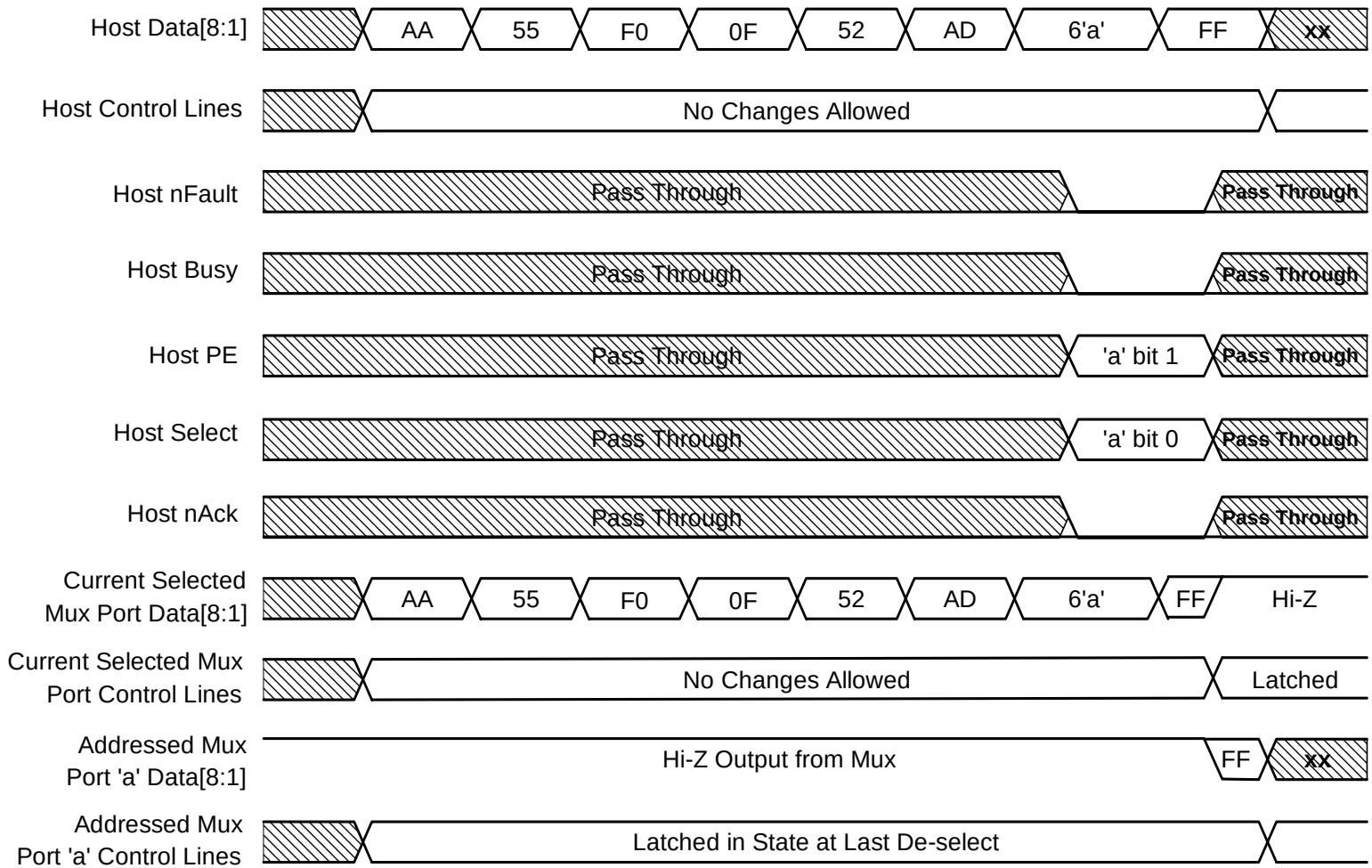


Figure 27—Selected Max Status command



"Pass Through" refers to the Status, Data or Control lines of the currently selected Mux port.

Figure 28—Select Mux Port *aa* command

5.5 Global reset

When a host is reset, it may reinitialize the interface by asserting nInit low in conjunction with nSelectIn low for a minimum of 10 µs. All MP devices must detect this condition for a minimum of 2 µs before returning to pass-through on a DC device or reselect port 0 on a MUX device. This allows for the reset condition to ripple through all MP devices. If assigned, addresses shall be retained after a global reset.

Under normal operations, once selected an IEEE 1284.3-compliant device shall not use the combination of nSelectIn low and nInit low. This signaling combination will cause any MP device to become de-selected and the reset condition to propagate down the chain. The Global reset may be used to recover from a loss of communication.

6. Service provider interface (SPI)

6.1 Overview

This clause provides detailed design semantics associated with SPI functionality. The interface described in this clause is designed for host-side IEEE 1284.3 support. However, throughout this clause, certain aspects of SPI driver design can also be applied to peripheral-side implementations. The purpose of the SPI is to hide the physical topology and present the devices in a unified canonical format.

The SPI is an abstract specification. The SPI specification only outlines functionality requirements with implementation guidelines. Specific calling sequences and actual syntax for procedural or functional device driver entry points is typically operating system (OS) dependent. It is up to individual OS-specific device driver writers to translate the abstract interface specified within this standard to whatever is required to achieve like-functionality within a particular OS environment.

It is expected that OS and other platform vendors will implement the IEEE 1284.3 SPI in slightly different ways. The authors of the SPI layer wanted to stress that the SPI is an abstract interface, even though this standard suggests syntax and parameters for each of the SPI entry points.

Implementations of the SPI are free to “instrument” the SPI using different function call profiles and parameter sets. However, a conforming implementation of the IEEE 1284.3 SPI must not vary the semantic content of the SPI abstraction as defined by this standard.

The SPI provides the following functional classes:

— *Device Enumeration:* Permits SPI clients to query the IEEE 1284.3 topology of physically attached IEEE P1284.3 peripherals.
— *Data Transfer:* Sends and receives data between an SPI client and a peripheral device.
— *Port Contention:* Permits multiple SPI clients to manage access to a single host adapter.
— *Host Adapter Information:* Permits SPI clients to query for host adapter properties, such as the adapter base address, interrupt request level, operational modes, etc.

6.1.1 IEEE P1284.3 Device enumeration

The SPI is responsible for forming a canonical representation of the devices within a particular IEEE 1284.3 topology. The IEEE 1284.3 enumeration capability functions within multipoint (MP) environments as well as traditional single-attach peripheral topologies. SPI_Initialize performs the enumeration, during which each IEEE 1284.3 peripheral in the topology shall be assigned a canonical deviceNumber. These device

numbers are ordered 1 to n, where ‘n’ is the total number of attached IEEE 1284.3 peripheral devices in the topology.

Internally, the SPI may maintain a mapping between a SPI canonical device number and its associated physical IEEE 1284.3 topology coordinates. These coordinates are made up of the following triple: [*hostAdapter, muxPort, deviceAddress*]. The *hostAdapter* identifies the host adapter to which the peripheral is attached (0 - (n-1)). The *muxPort* is the physical Mux port to which the peripheral is attached (if any) (see 5.4.1). The *deviceAddress* is the address assigned to the DC device during the CPP Assign Address phase (see 5.3.5.2.1). The method used by any one OS vendor to map the deviceNumber to the physical IEEE 1284.3 topology coordinates shall be left to that vendor.

6.1.2 CPP services

The SPI is responsible for managing CPP switching for each adapter port, and handles daisy chains and multiplexors. Typically, all CPP commands and handling of responses is managed by the SPI.

6.2 SPI Client interface definition

Each SPI entry point shall be described using the following syntax:

- [return status codes] = [{OUT}]SPI_<name> {IN}|{OUT}[param1 [,param2,…paramN]]
- Where <name> is the symbolic name for the operation or function to be performed.
- Where {IN} is used to indicate that [param1 [param2,…paramN] contains data to be passed into the entry point.
- Where {OUT} is used to indicate that [param1 [param2,…paramN] contains data to be returned from the entry point.
- Where function to be performed does not return status codes, {OUT} shall not be used.
- [param1] through [paramN] may be required or optional parameters depending on the entry point being used.

A quick summary of the operations is listed below, and described in detail in subsequent subclauses:

- SPI_Initialize: Initialize the IEEE 1284.3 device layer to a known state.
- SPI_GetDeviceNumber: Translate physical topology identifiers to SPI canonical device number.
- SPI_GetDeviceCoordinates: Translate SPI canonical device number to physical topology.
- SPI_GetDeviceID: Get IEEE 1284 Device ID using canonical device number.
- SPI_Open: Open connection to a particular device using canonical device number.
- SPI_IO: Perform I/O to and from a device identified by a handle.
- SPI_Close: Close connection to a particular device as identified by a handle.
- SPI_Lock: Obtain exclusive use of a host adapter for subsequent I/O access.
- SPI_Unlock: Release exclusive use of a host adapter.
- SPI_QueryCapablilties: Obtain host adapter hardware information.

6.2.1 SPI_Initialize

The SPI_Initialize function is used to initialize the SPI subsystem to a known state. Typically this is done during a system “boot” process or OS restart. After completion of SPI_Initialize, SPI clients can assume that all IEEE 1284 ports managed by the SPI layer are operating in IEEE 1284 compatibility mode.

Typically, the SPI would also utilize the CPP “Mux Present,” “Number of Mux Ports,” and “AssignAddress” commands to physically enumerate the IEEE 1284.3 topology as part of the SPI_Initialize functionality.

OUT SPI_Initialize <no parameters>

Return status codes
SPI_SUCCESS
SPI_NO_ADAPTERS
SPI_FAIL

6.2.2 SPI_GetDeviceNumber

SPI_GetDeviceNumber is used for clients to map the IEEE 1284.3 physical topology to a SPI canonical deviceNumber.

OUT SPI_GetDeviceNumber IN hostAdapter, IN muxPort, IN deviceAddress, OUT deviceNumber

hostAdapter
0 to (n-1), identifies the physical IEEE 1284 host adapter to which the peripheral is attached.

muxPort
0 to 3

deviceAddress
The CPP assigned address allocated to the device during a previous SPI_Initialize function. -1 to (n-1), with -1 being the address of the end-of-chain peripheral device (EOC).

deviceNumber
The SPI canonical device number (see 6.2.4).

Return status codes
SPI_SUCCESS
SPI_INVALID_ADAPTER
SPI_INVALID_DEVADDR
SPI_INVALID_MUX

6.2.3 SPI_GetDeviceCoordinates

SPI_Get_DeviceCoordinates is used to map the SPI canonical device number to the IEEE 1284.3 physical topology.

OUT SPI_GetDeviceCoordinates IN deviceNumber, OUT hostAdapter, OUT muxPort, OUT deviceAddress

deviceNumber
The SPI canonical device number (see 6.2.4).

hostAdapter
Identifies the physical IEEE 1284 host adapter to which the peripheral is attached (see 6.2.2).

muxPort
See 6.2.2.

deviceAddress
See 6.2.2.

Return status codes

SPI_DEFAULT: Informational status specifying that the device identified by deviceID is the 'end-of-chain' peripheral and is the default device selected on an adapter when no other device is selected. Typically, this device would be a 'legacy,' non-MP IEEE 1284 peripheral (see Figure 2).
SPI_END_CHAIN: Informational status specifying that the device identified by deviceID is the 'end-of-chain' peripheral (see Figure 2).
SPI_END_LIST: The end of the IEEE 1284.3 topology has been reached.
SPI_INVALID_DEVICE: Invalid device number specified
SPI_TIME_OUT

6.2.4 SPI_GetDeviceID

SPI_GetDeviceID shall return the IEEE 1284 deviceID of a specified MPD deviceNumber. If the device-Number is invalid or maps to a non-IEEE 1284 device, then no deviceID shall be returned and the appropriate return status code shall be returned.

This function does not require a handle. If the deviceID information is returned from internal SPI data structures and involves no I/O to the peripheral device, then no lock/unlock operation shall be required. If an I/O operation is required to retrieve the deviceID information, then the lock/unlock operation shall be performed.

OUT SPI_GetDeviceID IN deviceNumber, OUT deviceID, [IN TimeoutBlk]

deviceNumber
The SPI canonical device number from which to obtain the IEEE 1284 device ID string (1 - n).

deviceID
A buffer into which an IEEE 1284 Device ID string will be returned.

TimeoutBlk
The address of a block of memory that specifies some maximum interval of time that the process will wait before successfully completing the I/O operation from a peripheral. This parameter is optionally specified by SPI clients and is not used if the client can wait indefinitely for the SPI_GetDeviceID operation to complete.

Return status codes

SPI_VALID_ID: DeviceID contains a valid IEEE 1284 device ID string.
SPI_END_CHAIN: Informational status specifying that the device identified by 'deviceID' is the 'end-of-chain' peripheral (see Figure 2).
SPI_END_LIST: The end of the IEEE 1284.3 topology has been reached.
SPI_INVALID_DEVICE: Invalid device number specified.
SPI_TIME_OUT
SPI_NO_DEVICE_ID

6.2.5 SPI_Open

SPI_Open is used to establish an instance of a logical connection between a device driver and a peripheral device within the IEEE 1284.3 configuration. This function returns a handle, which is used to reference the connection context in subsequent SPI function calls. It should be noted that SPI_Open only initializes a context internal to the SPI and does not cause any actual I/O between the host and the peripheral identified by the 'deviceNumber' parameter.

OUT SPI_Open IN deviceNumber, OUT handle

deviceNumber
The SPI canonical device number to which a connection should be opened. The deviceNumber is typically determined by calling SPI_GetDeviceID.

handle
The address of a container to hold the device handles. This is the same handle that would be used in any subsequent SPI_Lock, SPI_Unlock, SPI_IO, or SPI_Close operations.

Return status codes
SPI_SUCCESS
SPI_INVALID_DEVICE
SPI_NO_RESOURCES

6.2.6 SPI_IO

The SPI_IO function is an entry point that allows SPI clients to perform I/O operations to a peripheral device. All I/O WRITE operations to a particular SPI device number are serialized. Concurrent I/O READ and I/O WRITE operations to the same handle are supported. During I/O READ/WRITE operations data streams may be broken up unless the SPI_Lock/SPI_Unlock functions are used.

OUT SPI_IO IN handle, IN ioParams, IN ioMode, IN channelAddress, [IN TimeoutBlk], [INOUT AsyncBlk]

handle
See 6.2.5.

ioParams
An OS-specific block of information that describes the I/O transfer characteristics for this operation. Some components of a typical ioParam block would include the following:

— I/O Operation (Read, Write, Ioctl, Control)
— I/O Transfer block address
— I/O Transfer byte count

IoMode
Places Host port and peripheral in the specified mode for SPI_IO operations. Set to one of the following enumerated values:

— SPI_MODE_NIBBLE: IEEE 1284 Nibble mode.
— SPI_MODE_CNIBBLE: IEEE 1284.3 Channelized nibble mode. See clause 8.2.
— SPI_MODE_BYTE: IEEE 1284 Byte mode.
— SPI_MODE_COMPAT: IEEE 1284 Compatibility mode.
— SPI_MODE_BECP: IEEE 1284.3 Bounded ECP mode. See clause 8.1.
— SPI_MODE_ECP: IEEE 1284 ECP mode.
— SPI_MODE_ECPRLE: IEEE 1284 ECP with run-length encoding mode.
— SPI_MODE_ECPSWE: ECP mode, software-emulated.
— SPI_MODE_EPP: IEEE 1284 EPP mode.
— SPI_MODE_EPPSL: (EPP 1.7) Intel® legacy EPP support.
— SPI_MODE_EPPSWE: EPP mode, software-emulated.

channelAddress
ECP channel number or EPP address. ECP addresses are in a range of 0 to 127. EPP addresses fall in the range 0 to 255.

TimeoutBlk
The address of a block of memory that specifies some maximum interval of time that the process will wait before successfully completing the I/O operation with a peripheral. This parameter is optionally specified by SPI clients and is not used if the client can wait indefinitely for the SPI_IO operation to complete.

AsyncBlk
An optional parameter that specifies the address of a block of memory that contains whatever information is needed to complete the SPI_IO operation asynchronously to the calling process thread.

Return status codes
SPI_SUCCESS
SPI_INVALID_ECPCHAN
SPI_INVALID_EPPADDR
SPI_INVALID_HANDLE
SPI_INVALID_MODE
SPI_TIMEOUT_EXCEEDED

6.2.7 SPI_Close

The SPI_Close function is used to terminate an instance of a logical connection between an SPI client and a peripheral IEEE 1284.3 device. The handle parameter is the same handle that was returned in a previous call to SPI_Open. After this call completes the handle is no longer valid.

OUT SPI_Close IN handle

handle
See 6.2.5.

Return status codes
SPI_SUCCESS
SPI_PENDING: A call to SPI_Close was attempted while the client has SPI_IO operations pending.
SPI_HANDLE_LOCKED: Operation failed, Handle is locked.
SPI_INVALID_HANDLE

6.2.8 SPI_Lock

The SPI_Lock function is optional, and may be used to serialize access to a parallel port adapter. SPI clients that require specialized control over IEEE 1284 adapter resources use this function. Specialized control in this context means that the client will be doing either proprietary I/O or the client has real-time requirements with regard to when it must communicate with a peripheral device.

Proprietary I/O means that an application will not be using the standard SPI I/O entry point (SPI_IO), but instead will be using proprietary access directly to the IEEE 1284 adapter hardware registers or resources. Applications that utilize proprietary I/O mechanisms shall participate in the parallel port adapter contention methods employed by the SPI.

It should be noted that SPI contention services (SPI_Lock, SPI_Unlock, and implicitly SPI_IO) operate on a per-adapter basis; meaning that if a particular host environment has multiple physical IEEE 1284 adapters configured, then multiple independent SPI clients may each have a specific adapter “locked.”

Internally, the SPI maintains a mapping from canonical SPI device number to the physical topology coordinates of a peripheral. It is possible for contention to occur even when two applications attempt to access different canonical device numbers, if the two target peripheral devices are reachable from the same IEEE 1284 adapter. Multiple applications use the SPI to contend for access to an IEEE 1284 adapter, and not for access to a particular IEEE 1284.3 device.

OUT SPI_Lock IN handle, [IN TimeoutBlk], [INOUT AsyncBlk]

handle
See 6.2.5.

TimeoutBlk
The address of a block of memory that specifies some maximum interval of time that the process will wait before successfully locking the appropriate IEEE 1284 adapter associated with the specified “handle.” This parameter is optionally specified by SPI clients and is not used if the client can wait indefinitely for the SPI_Lock operation to complete.

AsyncBlk
An optional parameter that specifies the address of a block of memory that contains whatever information is needed to complete the SPI_Lock operation asynchronously to the calling process thread. The 'AsyncBlk' structure should contain the capability for the client to receive the following asynchronous notifications, either through a callback or some other notification method:

— SPI_LOCK_GRANTED: This notification is used to signal an application that a pending SPI_Lock request has been granted.
— SPI_LOCK_REQUESTED: This notification is used to signal the lock owner that another application has requested access to the host adapter, either through an explicit SPI_Lock operation, or implicitly through the use of the SPI_IO operation.

Return status codes
SPI_SUCCESS
SPI_INVALID_HANDLE
SPI_LOCK_PENDING
SPI_TIMEOUT_EXCEEDED

6.2.9 SPI_Unlock

The SPI_Unlock function is used when a client no longer requires exclusive access to a particular peripheral.

OUT SPI_Unlock IN handle

handle
See 6.2.5.

Return status codes
SPI_SUCCESS
SPI_INVALID_HANDLE

6.2.10 SPI_QueryCapabilities

This function is used to determine the various software and hardware properties of the host adapter. Adapter properties (e.g., the interrupt request level) may be needed by the SPI client to perform OS resource mapping and to determine the adapter’s supported IEEE 1284 operating I/O modes.

SPI_QueryCapabilities IN hostAdapter, OUT adapterProperties

hostAdapter
See 6.2.2.

adapterProperties
The adapterProperties parameter is the address of a block of memory that is filled in by the SPI layer with the following information:

- General properties:
  - baseAddress: Base address of parallel port
  - chiptype: Chipset ID
  - DMA: DMA channel
  - ecpPword: ECP bus size
  - irq: Interrupt request level
- Mode properties:
  - FIFORxSize: Receive FIFO Size
  - FIFOTxSize: Transmit FIFO Size
  - FIFOShared: Transmit/Receive FIFOs Shared
  - FIFOAvailable: Supported FIFO properties (bit mask definitions the same as found under ioAttributes)
  - ioAttributes: Supported chip set properties (bit mask)
    - Bit mask definitions:
      - CompatibleNibble: Set if adapter supports Compatibility Mode w/nibble mode reverse channel
      - CompatibleByte: Set if adapter supports Compatibility Mode w/byte mode reverse channel
      - Ecp: Set if adapter supports IEEE 1284 ECP without RLE compression
      - EcpRLE: Set if adapter supports IEEE 1284 ECP with RLE compression
      - EcpDMA: Set if adapter supports ECP DMA
      - Epp1284: Set if adapter supports IEEE 1284 EPP mode
      - Epp16Bit: Set if adapter supports 16 bit EPP I/O transfers
      - Epp32Bit: Set if adapter supports 32 bit EPP I/O transfer
      - EppDMA: Set if adapter supports DMA for EPP
      - EppSL: Set if adapter supports nonstandard (pre-IEEE 1284) EPP mode

Return status codes
SPI_QueryCapabilities only returns known capabilities.

6.3 SPI hardware interface definition

Conceptually, the SPI functions as a multiport access “switch,” receiving requests from clients for access to generic device numbers and switching these requests into operations on specific physical IEEE 1284 adapters. During the switching process, the SPI will ensure that all requests to the same generic device number are serialized. It should be understood that there is only one instance of an SPI component per system. Any and all IEEE 1284 adapters are accessed through this single SPI instance. All SPI clients issue requests to this single SPI instance; in this way, the SPI functions as a true “switch” mapping multiple concurrent client requests into associated low-level requests to hardware adapters.

Since there is only one instance of an SPI per system, there needs to be some way to dynamically extend the SPI for support of new IEEE 1284 hardware adapters. The SPI has no built-in knowledge of any specific vendors' IEEE 1284 adapter register formats, interrupts, or programmed I/O requirements or capabilities.

Explicit vendor-specific hardware support is provided by traditional hardware device drivers, or can be provided by "BIOS helper routines." The layering of IEEE 1284.3 support could be implemented by using the model diagram as seen in Figure 29.

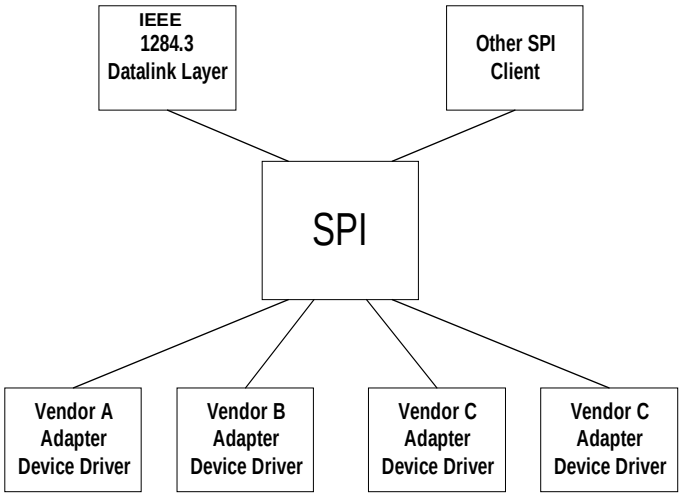


Figure 29—IEEE 1284.3 driver layer model

During the execution of the SPI_Initialize function, the system can use an OS-dependent algorithm to locate installed IEEE 1284.3 hardware device drivers or BIOS routines and "hook" them into the SPI adapter switch logic.

In some operating system environments, the SPI would equate very easily to the concept of an IEEE 1284.3 "class" driver. The IEEE 1284.3 class driver would allow potential IEEE 1284.3 clients generic access to the IEEE 1284.3 facilities available on the system. Since the class driver has no knowledge of specific vendors' hardware adapters, then there needs to be a lower layer of driver support to assist the class driver in completing IEEE 1284 operations. There should be a common lower-level interface that the class driver can depend on for interoperability purposes. This implies that IEEE 1284 hardware adapter vendors should instrument their respective hardware driver code to be SPI compliant. If BIOS routines are to be used by the SPI, then these BIOS routines should also conform to the SPI hardware interface.

The following abstract hardware entry points may be available to the SPI by SPI-compliant hardware drivers or "helper components:"

— SPI_HW_GetMode: This function returns the current IEEE 1284 mode setting of the adapter.
— SPI_HW_Init: This function performs any initialization steps required by the hardware device driver.
— SPI_HW_IO: This function allows the SPI to forward I/O requests to specific adapter drivers for completion. The parameters and semantics should be the same as that for the *SPI_IO* function.
— SPI_HW_Query: Like the *SPI_QueryAdapter* function, this call is used by the SPI to interrogate the adapter's operational capabilities.
— SPI_HW_SetMode: This function allows the SPI to set the IEEE 1284 mode to be used by the adapter.

7. Data link layer

7.1 Overview

The IEEE 1284.3 data link layer provides unacknowledged, connectionless data transfer services to its clients. The client shall be able to send complete packets of data, in order, to the peer client. Corrupted packets shall be discarded. The maximum amount of data that can be transferred within one packet is 65 536 bytes. Client data shall be delivered to the peer client, and only to the peer client. If no peer client exists, data shall be discarded. The peer client is any client of the peer data link registered with the same protocol ID (see 7.2.1).

The data link layer shall not block ("block" means to prevent communication through the physical interface preventing communication with another channel.) Data received for a blocked client shall be discarded if retaining it would cause the data link layer to be unable to receive data bound for another client.

Figure 30 shows the software and hardware components of an IEEE 1284.3-compliant system. Client A uses the IEEE 1284.3 data link layer to communicate with its peer, Client A. Client B also uses the IEEE 1284.3 data link layer to communicate with its peer, Client B. Client A cannot use the IEEE 1284.3 data link layer to communicate with any Client B.

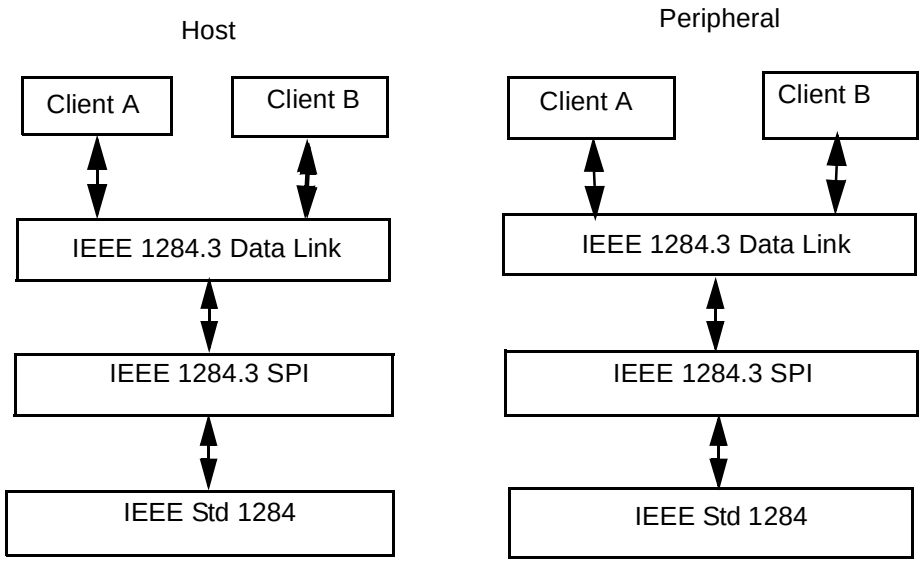


Figure 30—IEEE 1284.3 data link architecture

7.2 Data link SPI

This subclause contains the SPI for the IEEE 1284.3 data link layer. The SPI is for a host, but could be very similar for a device. Data link requests are the requests that a client makes of the data link. The IEEE 1284.3 data link provides requests for registering and unregistering a client, and for delivering data to the client's peer on another device. Data link indications are used by the data link to asynchronously deliver information to a client.

7.2.1 DL_Register

DL_Register is used to register clients with the data link.

DL_Register.request ProtocolID, AsyncBlk, Handle

ProtocolID
The identification number of the protocol served by this client. Only one client shall be allowed to register per ProtocolID. ProtocolIDs are registered in the PPP DLL Protocol Numbers list maintained by the Internet Assigned Numbers Authority (IANA). Please see the IANA website (http://www.iana.org) for current assignments and instructions for registering new protocols.

AsyncBlk
A block of memory that contains whatever information is needed to deliver data link indications to this client. The delivery mechanism is implementation specific.

Handle
The address of a container to hold the client's handle. This handle is used to identify the client in future SPI requests.

7.2.2 DL_Unregister

DL_Unregister is used to unregister clients from the data link.

DL_Unregister.request Handle

Handle
The client's handle as returned in the DL_Register request.

7.2.3 DL_Data

DL_Data is used to exchange packets of data between two peer clients. A client to pass a packet of data to the data link calls DL_Data.request. DL_Data indication is used to deliver the received packet of data to the peer client. Figure 30 illustrates the operation of DL_Data.

DL_Data.request Handle, DeviceNumber, Data

Handle
The client's handle as returned in the DL_Register request.

DeviceNumber
The destination device number for this data. This is the same number as used by the IEEE 1284.3 SPI (see 6.2.4).

Data
The packet of data to be delivered.

DL_Data.indication DeviceNumber, Data

Data
The packet of data passed to the peer data link through DL_Data.request.

DeviceNumber
The source device number for this data. This is the same number as used by the IEEE 1284.3 SPI (see 6.2.4).

7.3 Wire protocol

7.3.1 Overview

The wire protocol shall support the functionality specified in the SPI (see Clause 6).

The wire protocol uses framing to denote the start and end of packets. It shall be able to recover from lost framing due to infrequent byte-missed, byte-doubled, and byte-corrupted errors. Packets with corrupted data may be delivered to a client or discarded. While recovering from these errors, noncorrupted packets may be discarded.

The wire protocol shall also re-establish framing when a partial frame is received when the data link is first initialized. Partial frames left over from a previous conversation are common when hosts and devices reset.

The wire protocol shall enable host DMA by sending the length of the data before sending the data.

The wire protocol is connectionless.

The wire protocol shall use ECP mode to transfer data from the host to the peripheral, and either Bounded ECP or channelized nibble mode to transfer data from the peripheral to the host. If ECP is not supported by the host adapter, the host adapter driver may emulate ECP mode using software. A single ECP channel is used to exchange data.

7.3.2 IEEE 1284.3 data link discovery

The IEEE 1284 device ID is used to discover IEEE 1284.3 data links. A new entry describing the IEEE 1284.3 data link layer is required. It is formatted as follows:

1284.3DL: yy;

Where yy is an indication that the data links shall use ECP channel "yy" to exchange data. The channel is in hex, without 0x prefix.

For example, if the device’s IEEE 1284 device ID contains “1284.3DL:4E;”, the IEEE 1284 host would know to communicate with the IEEE 1284.3 data layer in the device on ECP channel 0x4E. Any data sent on that channel would be routed to the IEEE 1284.3 data link.

The IEEE 1284 device ID can also be used to discover the clients of an IEEE 1284.3 data link. A new entry specifying the clients is required. It is formatted as follows:

1284.3C: aaaa {,bbbb};

Where aaaa and bbbb are ProtocolIDs as defined in 7.2.1. The ProtocolIDs are four hexadecimal characters in length, without 0x prefix. Multiple ProtocolIDs are separated by commas.

For example, if the device’s IEEE 1284 device ID contains “1284.3C:0123,ABCD;”, the IEEE 1284 host would know that two protocols, with the ProtocolIDs of 0x0123 and 0xABCD, are supported by the device’s IEEE 1284.3 data link layer.

7.3.3 Framing

7.3.3.1 Frame format

Each data link frame contains one client data packet and the control information required to detect framing errors and resynchronize the data stream. Table 10 and Table 11 show the format and contents of the frame.

Table 10—7-1 Frame format

StartOfFrame		ProtocolID		LengthOfData		Checksum		Client Data	EndOfFrame	
0x55	0xAA	MSB	LSB	MSB	LSB	MSB	LSB		0x00	0xFF

Table 11—7-2 Frame contents

Field	Byte Offset	Length (in bytes)	Contents
StartOfFrame (SOF)	0x00	2	0x55,0xAA
Protocol ID	0x02	2	Identifies the source and destination client for this packet of data. Big-endian encoded.
LengthOfData	0x04	2	Unsigned length of the client data contained in this frame, minus 1. Big-endian encoded.
Checksum	0x06	2	16 bit one's complement of the one's complement sum of the ProtocolID and LengthOfData fields. The same checksum algorithm is used in Internet standards. Big-endian encoded.
ClientData	0x08	1 to 64K	Data to transfer between the peer clients.
EndOfFrame (EOF)	0x09 + LengthOfData	2	0x00,0xFF

7.3.3.2 Framing recovery procedures

The procedure for receiving a frame is for the data link to read the first four fields of the frame, check their validity, read the client data, and finally read and check the validity of the end of frame marker.

If any of the frame validity checks fail, the byte stream is out of synchronization. To restore synchronization, the data link retries the frame validity checks starting with the next byte after the byte where the previous validity check started. When an SOF is detected, the data link assumes it is at the beginning of a frame, and starts the normal procedure for receiving a frame. Client data for a corrupted frame shall not be delivered.

The client data might contain data that matches the SOF marker. While restoring synchronization, the data link will interpret that data as a valid SOF marker. However, the likelihood of receiving a valid SOF, a protocol ID that matches a registered protocol, a valid checksum, and a valid EOF at the end of the packet is very low. It is therefore unlikely that data not intended for the client will be delivered to the client.

8. Additional IEEE 1284 modes

8.1 Bounded extended capabilities port (BECP) mode

Some ECP host chip implementations have a limitation which can cause the duplication or deletion of a byte of data anytime the bus is negotiated from ECP reverse phase to ECP forward phase. This problem can occur when the peripheral sends a byte of data at the same time as the host changes phases.

To avoid this limitation, the peripheral needs to notify the host when it is safe to switch direction, i.e., when the peripheral will not be sending data. This is achieved through a new mode, BECP. BECP mode is a variation of IEEE 1284 ECP mode. The two differences between BECP mode and ECP mode are as follows:

— The nPeriphRequest signal is used to bound a block of data.
— Only one block of data may be sent per reverse phase negotiation.

The nPeriphRequest signal is defined as a hint to the host in IEEE 1284. BECP mode uses that hint as follows:

— The peripheral shall assert nPeriphRequest to indicate that data is available on the current ECP channel. The peripheral shall transition nPeriphRequest to the asserted state only when the interface is in the ECP forward phase (ECP with framing events 31–37 inclusive; see IEEE 1284).
— Upon negotiation to BECP mode, nPeriphRequest shall be valid by event 31.
— When receiving a channel address command, nPeriphRequest shall be valid for the newly selected channel by event 32.
— The peripheral shall not de-assert nPeriphRequest during BECP forward phase.
— The host shall not negotiate the interface into the BECP reverse phase unless the nPeriphRequest is asserted.
— When the peripheral has transmitted the entire block of data, the peripheral shall de-assert nPeriphRequest. The peripheral shall not de-assert nPeriphRequest between events 43 and 45. The peripheral shall send no more data during the current BECP reverse phase.
— Upon transition from BECP reverse to forward phase the peripheral shall not re-assert the nPeriphRequest prior to event 47.
— The host shall only transition from BECP reverse to forward phase after nPeriphRequest has been deasserted. To provide fair access to other devices on the IEEE 1284 port, the host may transition from BECP reverse to forward phase before the peripheral completes the transfer of the data block. Hosts are recommended to terminate the transfer only if the peripheral delays more than 100 ms between bytes. The remainder of the data block will be transferred during the next BECP reverse phase.

Due to another limitation of legacy host ECP adapters, peripherals shall not send channel addresses in BECP mode.

The extensibility request value for BECP mode is 0x18.

8.2 Channelized nibble mode

The IEEE 1284.3 data link layer requires channels to identify IEEE 1284.3 data link data. Nibble mode does not provide a mechanism for setting channels or addresses.

To indicate channels in nibble mode, a new mode is required—Channelized Nibble mode. In this mode, the current peripheral to host ECP channel is used as the channel from which the peripheral sends its data. If ECP channels have been used, the current channel from the most recent mode is used.

The negotiation extensibility request value for Channelized Nibble is 0x08.

Annex A

(informative)

Enhanced parallel port (EPP) BIOS

This information is provided for historical background and is not part of this standard.

A.1 Background

Concurrent with the development of the original IEEE 1284 specification, the EPP committee, an industry group chartered to standardize and promote the use of EPP, defined a BIOS-level interface to provide easy access to the EPP capability in the host computer. The BIOS interface was important for two reasons:

a) There is not a consistent architecture across various EPP implementations
b) Extensions to this BIOS provided the synchronization point between multiple devices that might be sharing the EPP port through a multiplexor or daisy chain.

A.2 Overview

The EPP BIOS is provided as a hardware independent method of accessing an EPP port.

A.2.1 EPP BIOS application program interface

The EPP BIOS defines a special printer BIOS (interrupt 17 hex) call—*Installation Check*—that returns a far pointer to the EPP BIOS entry point. This pointer is called the *EPP Vector*. This vector serves as the Application Program Interface (API) for all underlying EPP services. All EPP BIOS calls are executed by making a far call using the EPP Vector. The EPP BIOS is intended as a real mode interface, however, unless otherwise noted, EPP BIOS calls can be made while the central processing unit is operating in real or protected mode.

The result code for all BIOS calls is always returned in the AH register. A value of zero (0) indicates that the operation was successful. All values defined in EPP BIOS are in hexadecimal unless otherwise specified.

A.2.2 Earlier versions of EPP BIOS

Various ROM BIOS vendors using revision 3 as a guide have implemented the EPP BIOS. This document contains a subset the earlier revision 3 specification. Additions and changes to the functions between revision 3 and this specification are marked with an asterisk (*) in A.4.

A.3 Multiport operation

The EPP BIOS assumes that the host's parallel port can be used to connect more than one peripheral device. The EPP BIOS supports two kinds of multiport configurations: multiplexor and daisy chain. Peripheral device ports are selected on a time-shared basis; only one device may be selected at a time. Device ports are numbered from one (1) through four (4).

If both a multiplexor and a daisy chain are present, the multiplexor shall be directly attached to the host's EPP port: daisy chains can be attached to the multiplexor's device ports.

A.3.1 Daisy chaining

The daisy chain is controlled by the daisy chain manager (DC manager). When the DC manager is run, it hooks itself in front of the EPP BIOS (and possibly the Mux driver). In this way, the DC manager filters all of the BIOS calls, passing all of the standard (non-DC) commands to the underlying EPP BIOS, executing any DC commands itself. It also virtualizes other EPP BIOS functions, such as Set Mode, so that a specific device driver will not need to care if other devices are being accessed over the same port. The EPP BIOS DC extensions are defined in A.5.

A.3.2 EPP I/O concurrence

Because of its multiport nature, the host EPP port should be thought of as a serially reusable I/O resource. EPP device drivers, or other client programs, shall first lock the EPP BIOS before making any EPP I/O calls. In a multiport environment device drivers will contend for I/O access to the EPP port. Two BIOS services—Lock and Unlock—are provided for this purpose.

Device drivers will typically lock the EPP port, perform one or more EPP I/O cycles (e.g., “Write Block”), and then unlock the port. The time interval between lock and unlock should be kept reasonably small to facilitate EPP port sharing.

In a multiport environment, the “Lock” BIOS call is used to select a particular device port on the multiplexor or daisy chain. Multiplexors use addresses 1–4, and any device may be on any port. Daisy chains are also addressed as 0–3, however, any device that does not support the DC protocol should use zero (0) in the DC field of the device port address.

A.4 EPP BIOS calls

A.4.1 Installation Check

Installation Check is used to test for the presence of an EPP port. This call returns a far pointer to the EPP BIOS entry point—the EPP Vector. For compatibility reasons, this call rides on top of the standard Printer Status BIOS call. Device drivers should call this function in initialization and remember the EPP Vector, because a network redirector or other software, which intercepts printing, may overwrite this call and render the EPP BIOS unrecognizable. This call can only be made while the CPU is operating in real mode.

API: - int 17

input: AH = 2
DX = EPP port number (0-2) - LPT Port 1 - 3
AL = 0
CH = 45 ('E')
BL = 50 ('P')
BH = 50 ('P')

output: AH = - if EPP present
AL = 45 - if EPP present
CX = 505- if EPP present

DX:BX= EPP BIOS entry point - EPP Vector

register usage: AX, BX, CX, DX

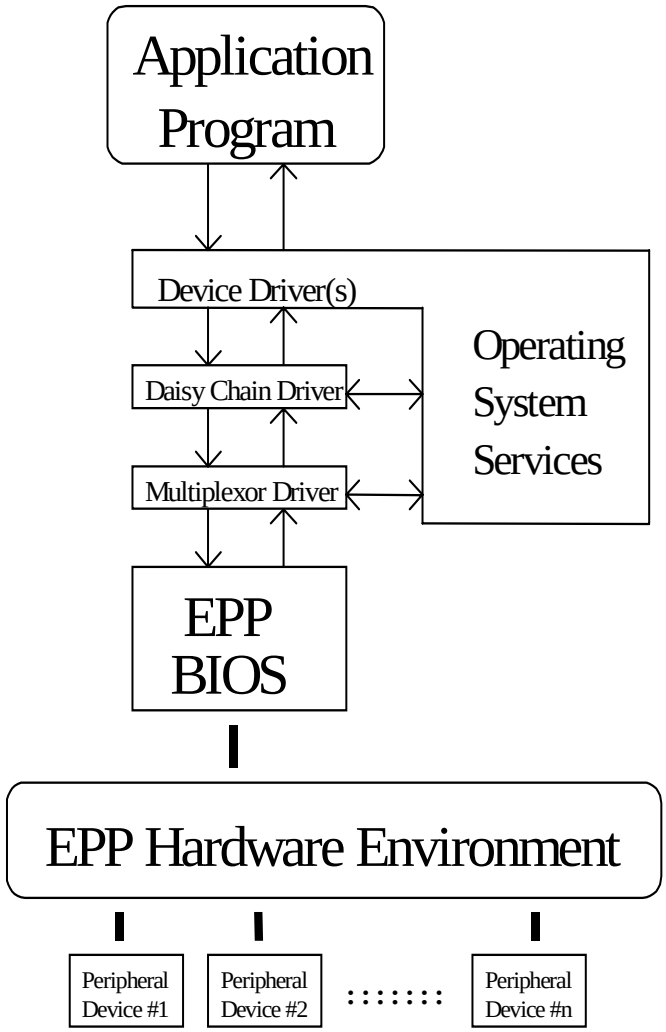


Figure A.1—System architecture

A.4.2 Query Config

Query Config returns the EPP port's configuration.

API - EPP Vector

input: AH = 0
DL = EPP port number (0-2)

output: AH = error code
AL = Interrupt level of EPP port (0-15)
= FF - if interrupts not supported
BH = EPP BIOS revision (MMMMnnnn or M.n) prior to this revision of the EPP BIOS spec, there was no enumeration of this value, so you may not depend on this value for earlier versions of this BIOS.

*90 for versions supporting this document
BL = I/O Capabilities
bit 0 = Multiplexor present
bit 1 = PS/2 bidirectional capable
*bit 2 = EPP 1.9 capable
bit 3 = ECP capable
*bit 4 = Reserved
*bit 5 = Centronics FIFO capable
*bit 6 = EPP 1.7 capable
CX = SPP I/O Base address
ES:DI = EPP BIOS manufacturer's information/version text string (zero terminated)

register usage: AX, BX, CX, DI, ES

A.4.3 Set Mode

Set Mode is used to set the operating mode of the EPP port. *This call can only be made while the CPU is operating in real mode.*

API - EPP Vector

input: AH = 1
DL = EPP port number (0-2)
AL = mode bits
bit 0 = set compatibility mode
bit 1 = set bidirectional mode
bit 2 = set EPP mode
bit 3 = set ECP mode
*bit 4 = Reserved
*bit 5 = set Centronics FIFO mode
*bit 6 = set EPP 1.7 mode

output: AH = error code

register usage: AX, BX

A.4.4 Get Mode

Get Mode returns the current operating mode of the EPP port. *This call can only be made while the CPU is operating in real mode.*

API - EPP Vector

input: AH = 2
DL = EPP port number (0-2)

output: AH = error code
AL = mode bits
bit 0 = in compatibility mode
bit 1 = in bidirectional PS/2 mode
bit 2 = in EPP mode
bit 3 = in ECP mode
*bit 4 = Reserved

*bit 5 = in Centronics FIFO mode
*bit 6 = in EPP 1.7 mode
bit 7 = EPP port interrupts enabled

register usage: AX, BX

A.5 EPP BIOS error codes

When an EPP BIOS call is made the result code is always returned in the AH register. A value of zero (0) indicates SUCCESS. All non-zero values indicates an error. The EPP BIOS error codes defined below are given in hex.

Error code	Description
00	Successful
01	I/O timeout
02	Command/feature not supported
03	Unrecognized EPP Port Number
04	Do not use
05	Parameter Error

Annex B

(informative)

SPI usage examples

B.1 Operating system initialization

Demonstrates how to build a list of installed devices.

```
void BuildIntalledDeviceList()
{
int DeviceNumber = 0;
char DevIDString[ MAX_ID_STRING ];

        SPI_Initialize();

        do {
                sts = SPI_GetDeviceID( DeviceNumber++, DevIDString );
                if ( sts == SPI_VALID_ID ) {
                        Store_OS_Info( DeviceNumber, DevIDString );
                }
        } while ( sts != SPI_END_LIST );
}
```

B.2 To locate a specific device in the configuration

From a device driver for MyDevice with the presumption that the OS has already initialized the Host ports.

```
int DeviceNumber = SPI_START_ENUM;// where caller starts looking for device

char *MyDeviceID = "MyDevice Version 1.0";// Std 1284 device ID string to be found

int ReturnMyDeviceNumber ( int &DeviceNumber, char *MyDeviceID )

{

int found = FALSE;

DevIDString[ MAX_ID_STRING ];

        do {

                sts = SPI_GetDeviceID( DeviceNumber, DevIDString );
                if ( sts == SPI_VALID_ID ) {
                        if ( !strcmp( DevIDString, MyDeviceID )) {
                        found = TRUE;
                        break;  /* Found my device! */
                        }
                }
```



```
                DeviceNumber++;
        } while ( sts != SPI_END_LIST );

return ( found );
}
```

B.3 To perform an output operation using SPI_IO

Used to perform standard, non-proprietary I/O.

```
char *outbuf = "This is a text string to be sent to a remote peripheral";
int handle;
int ecp_channel = 6;
struct OS_ASYNC_BLK asblk;
struct OS_IO_PARAMS ioblk;

sts = SPI_Open(MyDeviceNumber, &handle);
if (sts != SUCCESS)
  return(sts);

ioblk.bufaddr = outbuf;
ioblk.buflen = strlen(outbuf);
ioblk.opcode = IO_WRITE;

sts = SPI_IO(handle, &ioblk, SPI_MODE_ECP, ecp_channel, 0, (struct OS_ASYNC_BLK *));

/* Passing a null pointer for the I/O async block means that the operation is to be completed synchronously.
In this example, we are not performing "proprietary I/O". Rather, we are using the standard generic SPI I/O
facility, which internally handles port contention.  */
```

B.4 Using proprietary I/O for output

Used when the device driver needs to perform proprietary I/O to a device.

```
/* In this example, we are performing "proprietary I/O" operations which occur outside the SPI component.
This would typically involve custom access to IEEE 1284.3 hardware to perform an I/O sequence that might
not strictly conform to the IEEE 1284.3 specification. However, the application needs to participate in the
IEEE 1284.3 port contention environment. */

char *myDeviceID = "MyCompanyName:1284",
     *outbuf = "This is a text string to be sent to a remote peripheral
int handle,
     ecp_channel = 6,
     sts,
     hostAdapter,
     muxPort,
     deviceAddress,
     deviceNumber = SPI_START_ENUM;
struct OS_ITMEOUT_TYPE tmo;
struct OS_ASYNC_BLK asblk;
```

```
/*  Once the deviceID and deviceNumber are found then the physical coordinates for the device can be
located using the SPI_GetDeviceCoordinates call. These coordinates can then be used for the proprietary
I/O sequences. */

sts = ReturnMyDeviceNumber ( &deviceNumber, myDeviceID );
        /* see clause B.2 To locate a specific device in the configuration */

if ( sts )
        SPI_GetDeviceCoordinates( deviceNumber, &hostAdapter, &muxPort, &deviceAddress );
else
        return ( FALSE );

sts = SPI_Open( deviceNumber, &handle );
if ( sts == SUCCESS ) {

  sts = SPI_Lock( handle, ( struct OS_TIMEOUT_TYPE * ) 0,
                  ( struct OS_ASYNC_BLK * ) 0 );

/* Passing NULL pointers in this example specifies that the call to
     SPI_Lock() is to be completed synchronously */

...
/* Do proprietary I/O sequence */
...

SPI_Unlock( handle );
}
```

B.5 To perform input from a remote IEEE 1284 device, asynchronously

```
char input_buf[128];

void io_callback( struct OS_ASYNC_BLK *asblk )
{
  if ( asblk.iostat == SUCCESS ) {
    /* ...process received contents of input_buf... */
} else if ( asblk.iostat == TIMEOUT ) {
/* ...handle timeout status... */
 }
}

struct OS_ASYNC_BLK asblk;
struct OS_IO_PARAMS ioblk;
struct OS_TIMEOUT_TYPE tmo;

ioblk.bufaddr = &input_buf;
ioblk.buflen = sizeof(input_buf);
ioblk.opcode = IO_READ;
asblk.callback = io_callback;

sts = SPI_IO( handle, &ioblk, SPI_MODE_ECP, ecp_channel, 0, &asblk );
```


/* In the asynchronous I/O case, most OS environments return the status of the request ('sts') as one of the following:

FAILURE	One or more of the parameters to the request was invalid.
SUCCESS	The I/O request was initiated successfully, but has not yet completed.

The key point to remember is that, typically, for asynchronous I/O, a successful status return is not an indication that the I/O operation has completed. The I/O operation will have completed only after the specified callback routine has been activated. */

Annex C

(informative)

Bibliography

The following documents were cited in this standard, and provide additional background information useful in conjunction with this standard.

[B1] EPP Bios Specification 3.0.

[B2] Microsoft Extended Capabilities Port: Specification Kit Rev. 1.06, 14 July 1993.

[B3] Microsoft Plug and Play Parallel Port Devices 1.0a.

Annex D

(normative)

Signal transition events

Each signal transition diagram has numbers corresponding to the events that cause the transitions. The following is a list of those events and the associated action. Please note that not all event numbers are used.

1	nStrobe is driven high at the beginning of the DC sequence.
2	Pass-through control signals must be stable by event 2.
3	The host places the first byte of the preamble on the data lines.
4	The point at which the peripheral must detect a stable preamble byte.
5	The host places the second byte of the preamble on the data lines.
6	The point at which the peripheral must detect a stable preamble byte.
7	The host places the first byte of the function ID on the data lines.
8	The point at which the peripheral must detect a stable function ID.
9	The host places the negated value of the first function ID byte on the data lines.
10	The point at which the peripheral must detect a stable function ID. At this point the peripheral drives the status lines setting Busy low, and PE, Select, and nFault high.
11	Status information from the pass-through port is propagated.
12	The host places the second byte of the function ID on the data lines.
13	The point at which the peripheral must detect a stable function ID.
14	The peripheral drives PE low, and Busy, Select and nFault high.
15	Status information from the pass-through port is propagated.
16	The host places the negated value of the second function ID byte on the data lines.
17	The point at which the peripheral must detect a stable function ID.
18	PE and Select are set high. Busy is set low if the device detected a daisy chain peripheral on its pass-through port or set high if it is the last device. If the device is the last device *and* it is currently selected, then it sets nFault high. If it is not the last device then it sets nFault high if it is currently selected or if the pass-through nFault is high. If a device determines that it is the last device in the chain then it shall block the host nStrobe from propagating to its pass-through control port.
19	Status information from the pass-through port is propagated.

20	The host places the command byte on the data lines.
21	The point at which the peripheral must detect a stable Command.
22	The host drive nStrobe low to indicate a valid Command byte.
23	The peripheral accepts the Command and stops driving the Last Device and Selected status on the Busy and nFault lines.
24	The host drives nStrobe high.
25	The host places the termination byte (0xFFh) on the data lines. This indicates the end of the DC sequence.
26	If a peripheral is not selected then it drives the pass-through status lines onto the host status lines. If the peripheral is selected then it blocks the pass-through status lines and drives its' own status information onto the host status lines.
27	Control of the data lines is pass from the DC driver.
28	In the “Assign_Address” command the current device accepts the address at this point.
29	Not used.
30-32	In “Enable_Interrupts,” if a DC device has an interrupt pending then it shall drive nAck high at event 30, low at event 31 and then high again at event 32. If the device does not have an interrupt pending, then the pass-through nAck signal is passed to the host status nAck signal. At event 32 the nAck signal becomes transparent.
33	Not used.
34	Not used.
35	In the “Select_Device” command, the currently addressed device accepts the command and enters the Selected state. It drives nFault high at this point.
36	The Selected device blocks the pass-through status lines at this points and drives its own status onto the host port status lines.
37	Not used.
38	Not used.
39	Not used.
40	In “Query_Interrupt” command, device 0 drives Busy low if it has an interrupt pending, high if it does not. It drives PE and Select low to indicate that it is address 0.
41	In “Query_Interrupt” command, device 0 goes to pass-through and device 1 drives Busy low if it has an interrupt pending, high if it does not. It drives PE low and Select high to indicate that it is address 1.

42	In “Query_Interrupt” command, device 1 goes to pass-through and device 2 drives Busy low if it has an interrupt pending, high if it does not. It drives PE high and Select low to indicate that it is address 2.
43	In “Query_Interrupt” command, device 2 goes to pass-through and device 3 drives Busy low if it has an interrupt pending, high if it does not. It drives PE high and Select high to indicate that it is address 3.