CONGRUENT SOFTWARE, INC.

3998 Whittle Avenue
Oakland, CA 94602

(510) 531-5472
(510) 531-2942 FAX

FROM: Peter Johansson

TO: IEEE P1394a Working Group

DATE: December 12, 1997

RE: Suspend / resume

This document is based upon the contributions in 97-031r9, 97-053r2, 97-054r2 and 97-055r2 and is an effort to integrate that work with the P1394a draft standard. Along with editorial changes I have encountered a few areas where technical changes seemed (at least to me) to be in the spirit of the suspend / resume work. There are errors and omissions—most the result of the editor's incomplete comprehension of all details of the suspend / resume design efforts—but I expect them to be easily remedied.

The additions to P1394a are presented with a numbering scheme that matches that of Draft 1.2, November 21, 1997. The changes between that draft and this document are shown in red and with change bars in the margin.

The first draft of this document was reviewed in Ft. Lauderdale; this revision has been prepared for the follow-up PHY designer's review session in Albuquerque.

6. PHY register map (cable environment)

The Port Status page has new register fields to control the suspend and resume process. Because these fields shall be accessible to remote PHY register reads and writes, PHY registers 1010_2 and 1011_2 are defined with special behaviors. Zeros written to either register have no effect. A one written to PHY register 1010_2 sets the corresponding bit to one while a one written to the other PHY register, 1011_2, clears the corresponding bit position to zero. For convenience of reference, the entire definition of the Port Status page from P1394a Draft 1.2 is reproduced below.

In the cable environment, the extended PHY register map illustrated by figure 6-1 shall be implemented by all designs compliant with this supplement. Reserved fields are shown shaded in grey.

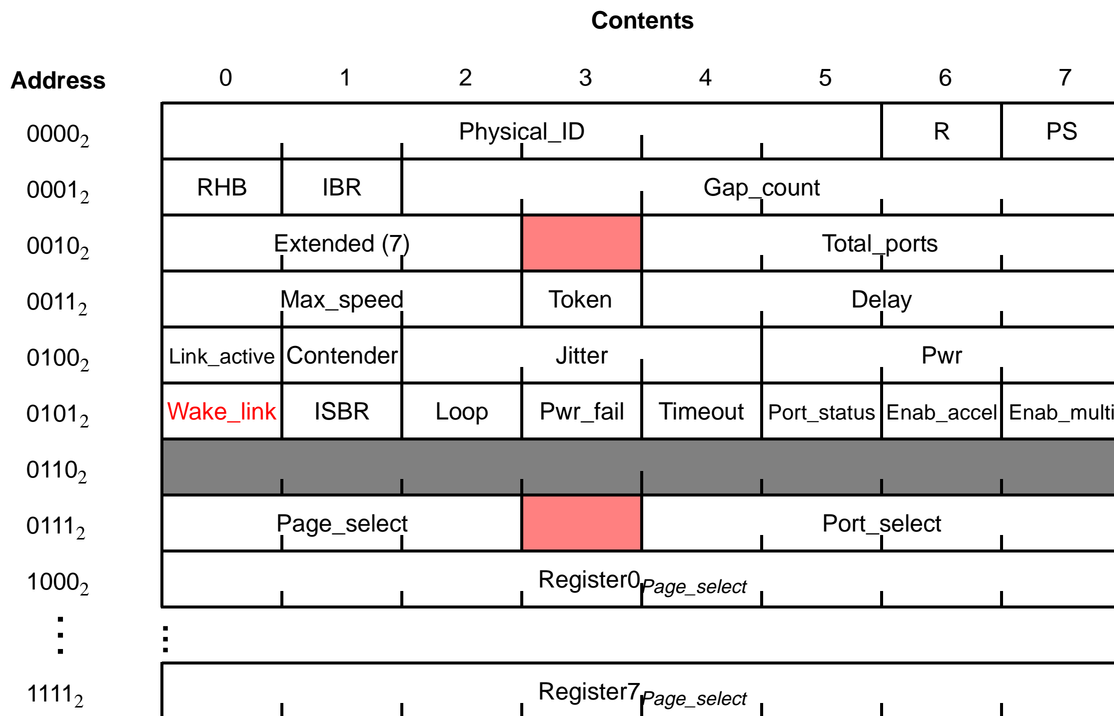

Figure 6-1 — Extended PHY register map for the cable environment

The meaning, encoding and usage of all the fields in the extended PHY register map are summarized by table 6-1. Power reset values not specified are resolved by the operation of the PHY state machines subsequent to a power reset.

Table 6-1 — PHY register fields for the cable environment

Field	Size	Type	Power reset value	Description
Physical_ID	6	r		The address of this node determined during self-identification. A value of 63 indicates a malconfigured bus; the link shall not transmit any packets.
R	1	r		When set to one, indicates that this node is the root.
PS	1	r		Cable power status (see clause 7.2).
RHB	1	rw	0	Root hold-off bit. When set to one, instructs the PHY to attempt to become the root during the next tree identify process.

This is an unapproved standards draft, subject to change
© 1997 IEEE

Table 6-1 — PHY register fields for the cable environment (Continued)

Field	Size	Type	Power reset value	Description
IBR	1	rw	0	Initiate bus reset. When set to one, instructs the PHY to set `ibr` TRUE and `reset_time` to `RESET_TIME`. Unless suspend or resume is in progress for any of the PHY's ports, these values in turn cause the PHY to initiate a bus reset without arbitration; the reset signal is asserted for 166 μs. This bit is self-clearing.
Gap_count	6	rw	$3F_{16}$	Used to configure the arbitration timer setting in order to optimize gap times according to the topology of the bus. See 4.3.6 of IEEE Std 1394-1995 for the encoding of this field.
Extended	3	r	7	This field shall have a constant value of seven, which indicates the extended PHY register map.
Total_ports	4	r	vendor-dependent	The number of ports implemented by this PHY.
Max_speed	3	r	vendor-dependent	Indicates the speed(s) this PHY supports: 000_2 98.304 Mbit/s 001_2 98.304 and 196.608 Mbit/s 010_2 98.304, 196.608 and 393.216 Mbit/s 011_2 98.304, 196.608, 393.216 and 786.43 Mbit/s 100_2 98.304, 196.608, 393.216, 786.432 and 1,572.864 Mbit/s 101_2 98.304, 196.608, 393.216, 786.432, 1,572.864 and 3,145.728 Mbit/s All other values are reserved for future definition
Token	1	r	vendor-dependent	When set to one, indicates that the PHY is capable of token-style arbitration (which shall be separately enabled for each port by the *enab_token* bit).
Delay	4	r	vendor-dependent	Worst-case repeater delay, expressed as 144 + (delay * 20) ns.
Link_active	1	rw	1	Link active. Cleared or set by software to control the value of the L bit transmitted in the node's self-ID packet 0, which shall be the logical AND of this bit and LPS active. If hardware implementation-dependent means are not available to configure the power reset value of the Link_active bit, the power reset value shall be one.
Contender	1	rw	See description	Cleared or set by software to control the value of the C bit transmitted in the self-ID packet. If hardware implementation-dependent means are not available to configure the power reset value of this bit, the power reset value shall be zero.
Jitter	3	r	vendor-dependent	The difference between the fastest and slowest repeater data delay, expressed as (jitter + 1) * 20 ns.
Pwr	3	rw	vendor-dependent	Power class. Controls the value of the pwr field transmitted in the self-ID packet. See clause 7.4.1 for the encoding of this field.
Wake_link	1	rw	0	Wakeup notification. When set to one, if the PHY/link interface is disabled the PHY shall signal LinkOn if any port commences resume operations.
ISBR	1	rw	0	Initiate short (arbitrated) bus reset. A write of one to this bit instructs the PHY to set `isbr` TRUE and `reset_time` to `SHORT_RESET_TIME`. Unless suspend or resume is in progress for any of the PHY's ports, these values in turn cause the PHY to arbitrate and issue a short bus reset. This bit is self-clearing.
Loop	1	rw	0	Loop detect. A write of one to this bit clears it to zero.
Pwr_fail	1	rw	0	Cable power failure detect. Set to one when the PS bit changes from one to zero. A write of one to this bit clears it to zero.
Timeout	1	rw	0	Arbitration state machine timeout. A write of one to this bit clears it to zero.
Port_status	1	rw	0	~~Bias change detect. Set to one when TP bias changes on any disabled port. The state of TP bias for enabled ports does not affect this bit.~~ Port status change detect. The PHY sets this bit to one if any of Connected, Bias, Disabled or Fault change for a port whose Int_enable bit is one. A write of one to this bit clears it to zero.
Enab_accel	1	rw	0	Enable arbitration acceleration. When set to one, the PHY shall use the enhancements specified in clause 7.9.

Table 6-1 — PHY register fields for the cable environment (Continued)

Field	Size	Type	Power reset value	Description
Enab_multi	1	rw	0	Enable multi-speed packet concatenation. When set to one, the link shall signal the speed of all packets to the PHY.
Page_select	3	rw	vendor-dependent	Selects which of eight possible PHY register pages are accessible through the window at PHY register addresses 1000_2 through 1111_2, inclusive.
Port_select	4	rw	vendor-dependent	If the page selected by *Page_select* presents *per* port information, this field selects which port's registers are accessible through the window at PHY register addresses 1000_2 through 1111_2, inclusive. Ports are numbered monotonically starting at zero, p0.

The *RHB* bit should be zero unless it is necessary to establish a particular node as the cycle master. In particular, bus manager- and isochronous resource manager-capable nodes should not set their *RHB* bit(s) to one and should not attempt to become the root unless there is no cycle master. This recommendation is made in anticipation of a requirement for Serial Bus to Serial Bus bridges to become root to distribute the cycle clock.

When any one of the *Loop*, *Pwr_fail*, *Timeout* or *Port_status* bits transitions from zero to one, *PHY_interrupt* shall be set to one. If the link is active, *PHY_interrupt* is reported as S[3] in a PHY status transfer, as specified by clause 5.3; otherwise a PHY interrupt shall cause LinkOn to tbe asserted. These bits in PHY register five are unaffected by writes to the register if the corresponding bit position is zero. When the bit written to the PHY register is one, the corresponding bit is zeroed.

The upper half of the PHY register space, addresses 1000_2 through 1111_2, inclusive, provides a windows through which additional pages of PHY registers may be accessed. This supplement defines pages zero, one and seven: the Port Status page, the Vendor Identification page and a vendor-dependent page. Other pages are reserved.

The Port Status page is used to access configuration and status information for each of the PHY's ports. The port is selected by writing zero to *Page_select* and the desired port number to *Port_select* in the PHY register at address 0111_2. The format of the Port Status page is illustrated by figure 6-2 below; reserved fields are shown shaded in grey.

Address	0	1	2	3	4	5	6	7
1000_2	AStat		BStat		Ch	Connected	Bias	Disabled
1001_2	Negotiated_speed			Int_enable				
1010_2		Suspend	Port_disable	Enab_token	Fault			
1011_2		Resume	Port_enable	Disab_token	Clr_fault			
1100_2								
1101_2								
1110_2								
1111_2								

Figure 6-2 — PHY register page 0: Port Status page

This is an unapproved standards draft, subject to change

© 1997 IEEE

The meanings of the register fields within the Port Status page are defined by the table below.

Table 6-2 — PHY register Port Status page fields

Field	Size	Type	Power reset value	Description
AStat	2	r		TPA line state for the port: 00_2 = invalid 01_2 = 1 10_2 = 0 11_2 = Z
BStat	2	r		TPB line state for the port (same encoding as AStat)
Ch	1	r		If equal to one, the port is a child, else a parent. The meaning of this bit is undefined from the time a bus reset is detected until the PHY transitions to state T1: Child Handshake during the tree identify process (see 4.4.2.2 in IEEE Std 1394-1995).
Connected	1	r	0	If equal to one, the port is connected, else disconnected. This bit reports the value of the connected variable for the port (see the connection_status() function in table 7-18).
Bias	1	r		If equal to one, bias voltage is detected (possible connection). The value reported by this bit is filtered by hysteresis logic, with a time of CONNECT_TIMEOUT, to reduce multiple status changes caused by contact scrape when a connector is inserted or removed.
Disabled	1	r	See description	If equal to one, the port is disabled. The value of this bit subsequent to a power reset is implementation-dependent, but should be a hardware configurable option. A single configuration option may control the power reset value for all ports.
Negotiated_speed	3	r		Indicates the maximum speed negotiated between this PHY port and its immediately connected port; the encoding is the same as for the PHY register Max_speed field.
Int_Enable	1	rw	0	Enable port status change interrupts. When set to one, the PHY shall set Port_status to one if any of Connected, Bias, Disabled or Fault (for this port) change state.
Suspend	1	rw[a]	1	Initiate suspend. If written as one, commence operations as a suspend initiator. While the suspend handshake with the peer PHY is in progress this bit reads as one; otherwise it reads as zero.
Port_disable	1	rw[a]	0	Port disable. When set to one the PHY shall disable the port. While the disable process is active this bit reads as one; otherwise this bit reads as zero.
Enab_token	1	rw[a]	0	Enable token-style arbitration. When set to one, the enhancements specified in clause 7.9 shall be enabled for this port.
Fault	1	r	0	Set to one if an error is detected during a suspend or resume operation.
Resume	1	rw[a]	0	If written as one the PHY shall attempt to resume normal operations on the port. While resuming, this bit reads as one; otherwise it reads as zero.
Port_enable		rw[a]	0	Port enable. If written as one the PHY shall enable the port. While the enable process is active this bit reads as one; otherwise this bit reads as zero.
Disab_token		rw[a]	0	Disable token-style arbitration. If written as one the Enab_token bit shall be cleared to zero. This bit always reads as zero.
Clr_fault		rw[a]	0	Clear fault(s). If written as one, the Fault bit shall be cleared to zero. This bit always reads as zero.

[a] These bits have special behaviors that permit remote PHY register writes to selectively modify them. A write of zero to any of these bits shall have no effect. A write of one shall have the effect specified in the table. More than one bit may be modified in a single register write.

© 1997 IEEE

The *AStat*, *BStat*, *Ch* and *Connected* fields are present in both the legacy and extended PHY registers and have identical meanings, defined by table 6-2 above, in both cases.

PHY registers 1010_2 and 1011_2 in the Port Status page shall be accessible to register read requests made by the link *via* the PHY/link interface. The behavior of a register write request addressed to either of these registers is unspecified; the link should use a remote access packet to modify their contents.

This is an unapproved standards draft, subject to change

© 1997 IEEE

7. Cable physical layer performance enhancement specifications

New extended PHY packets are defined to permit remote read and write access to the PHY registers and to remotely request a PHY port to function as a resume initiator. As a consequence of these changes, the type field (previously 6 bits) has been reduced to 4 bits.

7.4.5 Ping packet

The reception of the cable PHY packet shown in figure 7-5 shall cause the node identified by phy_ID to transmit self-ID packet(s) that reflect the current configuration and status of the PHY. Because of other actions, such as the receipt of a PHY configuration packet, the self-ID packet transmitted may differ from that of the most recent self-identify process.

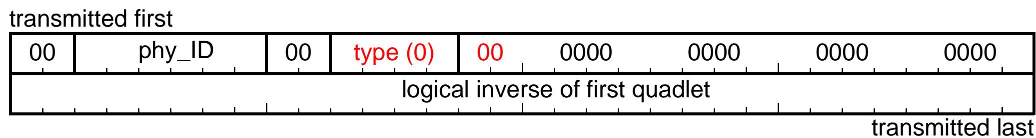

Figure 7-5 — Ping packet format

Table 7-6 — Ping packet fields

Field	Comment
phy_ID	Physical node identifier of the destination of this packet
type	Extended PHY ~~configuration~~ packet type (zero indicates ping packet)

7.4.6 Remote access packet

The reception of the cable PHY packet shown in figure 7-6 shall cause the node identified by phy_ID to either read or write the selected PHY register and subsequently return a remote reply packet that contains the current value of the PHY register (see clause 7.4.7).

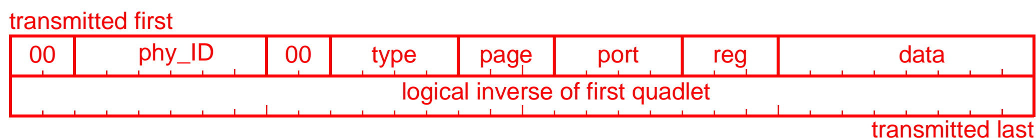

Figure 7-6 — Remote access packet format

Table 7-7 — Remote access packet fields

Field	Comment
phy_ID	Physical node identifier of the destination of this packet.
type	Extended PHY packet type: 1 Register read (base registers) 5 Register read (paged registers) 6 Register write (paged registers)
page	This field corresponds to the *Page_select* field in the PHY registers. The register read or write behaves as if *Page_select* was set to this value.
port	This field corresponds to the *Port_select* field in the PHY registers. The register read or write behaves as if *Port_select* was set to this value.

© 1997 IEEE

Table 7-7 — Remote access packet fields (Continued)

Field	Comment
reg	This field, in combination with page and port, specifies the PHY register. If type indicates a read of the base PHY registers reg directly addresses one of the first eight PHY registers. Otherwise the PHY register address is 1000_2 + reg.
data	This field is meaningful only if type indicates a write, in which case the PHY shall update the addressed PHY register as if a local write request had specified the data value.

7.4.7 Remote reply packet

Subsequent to the reception of a remote access packet, the PHY shall transmit the packet shown in figure 7-7.

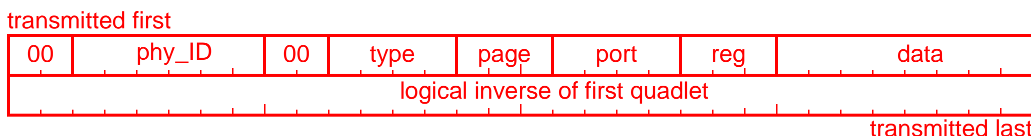

Figure 7-7 — Remote reply packet format

Table 7-8 — Remote reply packet fields

Field	Comment
phy_ID	Physical node identifier of the source of this packet.
type	Extended PHY packet type: 3 Register contents (base registers) 7 Register contents (paged registers)
page	This field corresponds to the *Page_select* field in the PHY registers; in conjunction with port and reg it identifies the register whose contents are returned in data.
port	This field corresponds to the *Port_select* field in the PHY registers; in conjunction with page and reg it identifies the register whose contents are returned in data.
reg	This field, in combination with page and port, identifies the register whose contents are returned in data. If type indicates a base PHY register, reg directly addresses one of the first eight PHY registers. Otherwise the PHY register address is 1000_2 + reg.
data	The current value of the PHY register addressed by the immediately preceding remote access packet. If the register is reserved, data shall be zero.

A PHY shall transmit a remote reply packet within PING_RESPONSE_TIME after the receipt of a remote access packet.

7.4.8 Resume packet

The reception of the cable PHY packet shown in figure 7-8 shall any node to commence resume operations for all PHY ports that are both connected and suspended. This is equivalent to zeroing the Suspend bit in PHY register 1011_2 in the Port Status page for each of these ports.

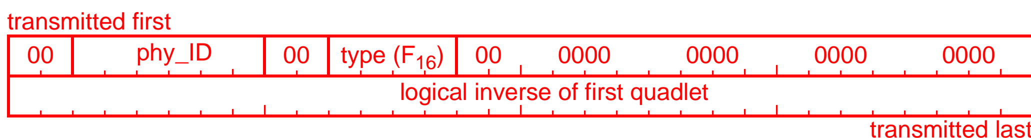

Figure 7-8 — Resume packet format

This is an unapproved standards draft, subject to change
© 1997 IEEE

Table 7-9 — Resume packet fields

Field	Comment
phy_ID	Physical node identifier of the source of this packet
type	Extended PHY configuration packet type (F_{16} indicates resume packet)

7.5 Cable PHY line states

This clause defines new rules by which a PHY decodes the interpreted arbitration signals (Arb_A and Arb_B) into a line state; it is in addition to IEEE Std 1394-1995 clause 4.3.3, “Cable PHY line states.”

Table 7-10 — Cable PHY received arbitration line states

Interpreted arbitration signals			
Arb_A	Arb_B	Line state name	Comment
1	Z	RX_DISABLE	The peer PHY is requesting the recipient to disable the receiving port and to initiate bus reset on all other active ports.
0	0	RX_SUSPEND	Exchange TpBias handshake with the peer PHY and place the connection into the suspended state. Also initiate suspend (*i.e.*, propagate TX_SUSPEND) on all other active ports.
0	Z	RX_TOKEN_GRANT	The parent PHY is granting the bus (although no TX_REQUEST was sent by the child)
Z	1	TX_DISABLE	Request the peer PHY to disable the connected port.
0	0	TX_SUSPEND	Request the peer PHY to handshake TpBias and enter the suspended state. The request is also propagated by the peer PHY to its other active ports.

The RX_TOKEN_GRANT line state is recognized when received by a parent port during the normal arbitration phase.

7.9.1 Data transmission and reception

Data transmission and reception are synchronized to a local clock that shall be accurate within 100 ppm. The nominal data rates are powers of two multiples of 98.304 Mbit/s for the cable environment.

7.9.1.1 Cable environment data transmission

Data transmission entails sending the data bits to the connected PHY along with the appropriately encoded strobe signal using the timing provided by the PHY transmit clock. If the connected port cannot accept data at the requested speed (indicated by the `speed_OK[i]` flag being FALSE), then no data is sent, which leaves the drivers in the "01" data prefix condition.

Table 7-11 — Data transmit actions (Sheet 1 of 2)

```
static dataBit tx_data, tx_strobe;      // Memory of last signal sent

void tx_bit(dataBit bit) {              // Transmit a bit
   int i;

   wait_event(PHY_CLOCK_indication);    // Wait for clock
   if (bit == tx_data)                  // If no change in data
```

Table 7-11 — Data transmit actions (Sheet 2 of 2)

```
      tx_strobe = ~tx_strobe;           // Invert strobe
   tx_data = bit;
   for (i = 0; i < NPORT; i++)
      if (active[i] && i != receivePort)
         if (speed_OK[i]) {
            portData pd = {phyData(tx_strobe), phyData(tx_data)};
            portT(i, pd);
         } else
            portT(i, TX_DATA_PREFIX);
}
```

The edge rates and jitter specifications for the transmitted signal are given in clause 4.2.3 of IEEE Std 1394-1995.

Starting data transmission requires sending a special data prefix signal and a speed code. The `speed_OK[i]` flag for each port is TRUE if the connected PHY has the capabilities to receive the data:

Table 7-12 — Start data transmit actions

```
void start_tx_packet(speed)             // Send data prefix and speed code
   int i;

   for (i = 0; i < NPORT; i++) {
      if (!active[i])
         speed_OK[i] = FALSE;
      else if (disable[i])
         portT(i, TX_DISABLE);
      else {
         portT(i, TX_DATA_PREFIX);      // Send data prefix
         speed_OK[i] = (tx_speed <= max_peer_speed[i]);
         if (speed_OK[i])
            portTspeed(i, tx_speed);    // Receiver can accept, send speed intentions
      }
   }
   wait_time(SPEED_SIGNAL_LENGTH);
   for (i = 0; i < NPORT; i++)
      if (active[i])
         portTspeed(i, S100);           // Go back to normal signal levels
   wait_time(DATA_PREFIX_TIME);         // Finish data prefix
}
```

Ending a data transmission requires sending extra bits (known as “dribble bits”) which flush the last data bit through the receiving circuit. The number of dribble bits required varies with the transmission speed: one, three or seven extra bits for S100, S200 and S400, respectively. An extra bit is required to put the two signals TPA and TPB into the correct state; the value of the bit depends upon whether the bus is being held (PH_DATA.request(DATA_PREFIX) or not (PH_DATA.request(DATA_END)):

Table 7-13 — Stop data transmit actions (Sheet 1 of 2)

```
void stop_tx_packet (phyData ending_status, speedCode tx_speed) {
   switch (tx_speed) {
      case S400:        // Pad with six dribble bits
         tx_bit(1);
         tx_bit(1);
         tx_bit(1);
         tx_bit(1);
      case S200:        // Pad with two dribble bits
         tx_bit(1);
         tx_bit(1);
      default:
         break;
   }
   tx_bit((ending_status == DATA_PREFIX) ? 1 : 0); // Penultimate bit...
```

 This is an unapproved standards draft, subject to change © 1997 IEEE

Table 7-13 — Stop data transmit actions (Sheet 2 of 2)

```
   wait_event(PH_CLOCK.indication());               // Wait for clock
   if (ending_status == DATA_PREFIX) {
      for (i = 0; i < NPORT; i++)
         if (active[i] && i != receive_port)
            portT(i, TX_DATA_PREFIX);               // ...and the last dribble bit
      wait_time(CONCATENATION_PREFIX_TIME);         // Speed signal after this time
   } else if (ending_status == DATA_END) {
      for (i = 0; i < NPORT; i++)
         if (active[i] && i != receive_port)
            portT(i, TX_DATA_END);
      wait_time(DATA_END_TIME);
   }
}
```

NOTE—This algorithm works to force the ending port state to TX_DATA_PREFIX or TX_DATA_END and relies on two characteristics of packet transmission: there are an even number of bits between the beginning and the end of a packet and a packet starts with `tx_strobe` at 0 and `tx_data` at 1. Thus, when `stop_tx_packet` is called the port state is either 01 or 10. If the desired port state is 01 (TX_DATA_PREFIX) and the current port state is 01, this algorithm sets port state to 11 for one bit time, then back to 01. If the desired ending state is 10 (TX_DATA_END) and the current port state is 01, the port state sequence is 00 followed by 10. The process is similar if the current port state is 10.

7.9.1.2 Cable environment data reception and repeat

Data reception for the cable environment physical layer has three major functions: decoding the data-strobe signal to recover a clock, synchronizing the data to a local clock for use by the link layer, and repeating the synchronized data out all other connected ports. This process can be described as two routines communicating *via* a small FIFO:

Table 7-14 — Data reception and repeat actions (Sheet 1 of 2)

```
static tpSig old_data, old_strobe;           // Memory of last signal sent

// Decode data-strobe stream and load FIFO -- this routine is always running
// (speed code recording is also done here)

void decode_bit (void) {
   repeat {
      if (portRspeed(receive_port) > S100) {
         rx_speed = portRspeed(receive_port);
         speed_signalled = TRUE;
         signal(SPEED_SIGNAL_RECEIVED);      // Notify start_rx_packet
      }
      new_signal = tpSignals();              // Get signal
      if (new_signal == IDLE)
         signal(IDLE_DETECTED);
      else {
         new_data = new_signal.TPA;          // Received data is on TPA
         new_strobe = new_signal.TPB;        // Received strobe is on TPB
         if ((new_signal.TPA != old_strobe) || (new_data != old_data)) {
                                             // Either data or strobe changed
            FIFO[fifo_wr_ptr] = new_data;    // Put data in FIFO
            fifo_wr_ptr = ++fifo_wr_ptr % FIFO_DEPTH; // Advance or wrap FIFO pointer
            signal(DATA_STARTED);            // Signal rx_bit to start
         }
         old_strobe = new_strobe;
         old_data = new_data;
      }
   }
}

// Unload FIFO and repeat data (but suppress dribble bits!)

void rx_bit(dataBit *rx_data, boolean *end_of_data) {
   int i;
```

Table 7-14 — Data reception and repeat actions (Sheet 2 of 2)

```
   wait_event(PHY_CLOCK_indication);          // Wait for clock
   if ((fifo_rd_ptr - fifo_wr_ptr) % FIFO_DEPTH) <= rx_dribble_bits) // FIFO empty?
      *end_of_data = TRUE;                    // If so, set flag
   else {
      *end_of_data = FALSE;                   // If not, clear flag...
      *rx_data = FIFO[fifo_rd_ptr];           // ... and get data bit
      fifo_rd_ptr = ++fifo_rd_ptr % FIFO_DEPTH; // Advance or wrap FIFO pointer
      tx_bit(*rx_data);                       // Repeat the data bit
   }
}
```

Starting data reception requires initializing the data resynchronizer and doing the speed signaling with the sender of the data. At the same time, the node must start up the transmitting ports by sending a special data prefix signal and repeating the received speed code. As in the `start_tx_packet()` function, the node must do the speed signaling exchange for each transmitting port:

Table 7-15 — Start data reception and repeat actions

```
void start_rx_packet () {            // Send data prefix and do speed signaling
   int i;

   fifo_rd_ptr = fifo_wr_ptr = 0;             // Reset data resynch buffer
   portT(receive_port, IDLE);                 // Turn off grant, get ready to receive
   for (i = 0; i < NPORT; i++)
      if (active[i] && i != receive_port)
         portT(i, TX_DATA_PREFIX);            // Send data prefix out repeat ports
   wait_event(SPEED_SIGNAL_RECEIVED | DATA_STARTED | IDLE_DETECTED);
   tx_speed = rx_speed;                       // Get speed of packet to repeat
   if (rx_speed == S100)
      rx_dribble_bits = 2;                    // Need for FIFO empty test
   else
      rx_dribble_bits = (rx_speed == S200) ? 4 : 8;
   if (speed_signalled) {                     // Repeat the speed signal...
      for (i = 0; i < NPORT; i++)
         if (active[i] && i != receive_port) {
            speed_OK[i] = (tx_speed <= max_peer_speed[i]);
            if (speed_OK[i])
               portTspeed(i, tx_speed);       // Receiver can accept, send speed intentions
         }
      wait_time(SPEED_SIGNAL_LENGTH);
      for (i = 0; i < NPORT; i++)
         if (active[i] && i != receive_port)
            portTspeed(i, S100);           // Go back to normal signal levels
      wait_time(DATA_PREFIX_TIME);      // Finish data prefix
      wait_event(DATA_STARTED | IDLE_DETECTED); // Wait for decoder to start
   }
   speed_signalled = FALSE;                   // Reset for each packet
   for (i = 0; i < FIFO_DEPTH/2 - 1; i++)
      wait_event(PHY_CLOCK_indication);       // Make sure FIFO is centered
}
```

7.9.2 Cable environment arbitration

The cable environment supports the immediate, priority, isochronous and fair arbitration classes. Immediate arbitration is used to transmit an acknowledge immediately after packet reception; the bus is expected to be available. Priority arbitration is used by the root for cycle start requests or may be used by any node to override fair arbitration. Isochronous arbitration is permitted between the time a cycle start is observed and the subaction gap that concludes an isochronous period; isochronous arbitration commences immediately after packet reception. Fair arbitration is a mechanism whereby a PHY succeeds in winning arbitration only once in the interval between arbitration reset gaps.

This is an unapproved standards draft, subject to change
© 1997 IEEE

Some of these arbitration classes may be enhanced as defined by this supplement. Ack-accelerated arbitration permits a PHY to arbitrate immediately following an observed acknowledge packet; this enhancement can reduce the arbitration delay by a subaction gap time. Fly-by arbitration permits a transmitted packet to be concatenated to the end of a packet for which no acknowledge is permitted: acknowledge packets themselves or isochronous packets. A PHY shall not use fly-by arbitration to concatenate an S100 packet after any packet of a higher speed.

Cable arbitration has two parts: a three phase initialization process (bus reset, tree identify and self identify) and a normal operation phase. Each of these four phases[1] is described using a state machine, state machine notes and a list of actions and conditions. The state machine and the list of actions and conditions are the normative part of the specification. The state machine notes are informative.

7.9.2.1 Bus reset

The bus reset process starts when a bus reset signal is recognized on a connected port or generated locally. Its purpose is to guarantee that all nodes propagate the reset signal. This supplement defines two types of bus reset, long bus reset (identical to that specified by IEEE Std 1394-1995) and arbitrated (short) bus reset. The PHY variable `reset_time` controls the length of the bus reset generated or propagated.

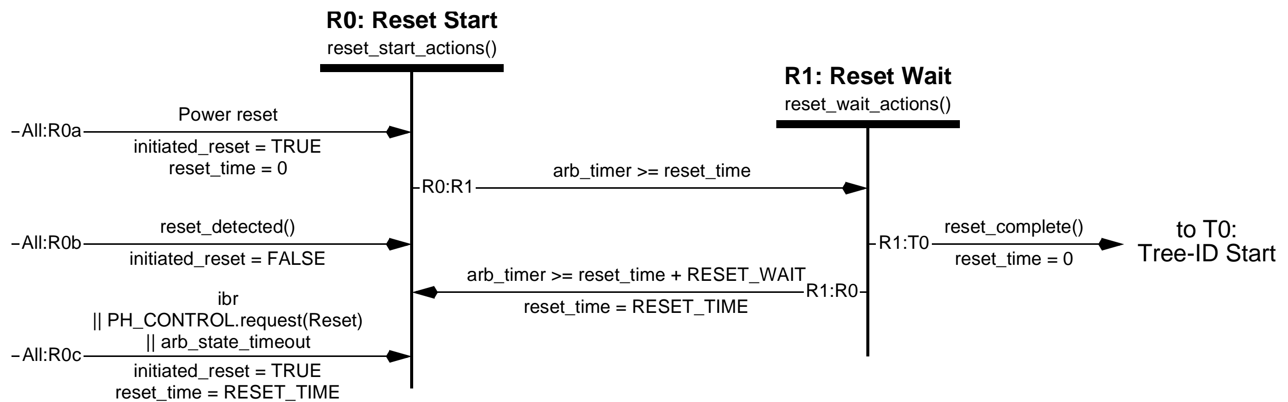

Figure 7-9 — Bus reset state machine

7.9.2.1.1 Bus reset state machine notes

Transition All:R0a. This is the entry point to the bus reset process if the PHY experiences a power reset. On power reset, PHY register values and internal variables are set as specified in this section; in particular all ports are marked disconnected. A solitary node transitions through the reset, tree identify and self-identify states and enters A0: Idle as the root node.

Transition All:R0b. This is the entry point to the bus reset process if the PHY senses BUS_RESET on any connected port's arbitration signal lines (see table 4-28 in IEEE Std 1394-1995).

Transition All:R0c. This is the entry point to the bus reset process if this node is initiating the process. This happens under the following conditions:

1) Serial Bus management makes a PH_CONTROL.*request* that specifies a long reset;
2) The PHY detects a disconnect on its parent port; or
3) The PHY stays in any state (except the idle state or a state that has an explicit time-out) for longer than MAX_ARB_STATE_TIME.

[1] Clause 4.4.2.2 of IEEE Std 1394-1995, which describes the tree identify process, is unchanged and is not reproduced in this supplement.

With the exception of the last condition, the initiation of a bus reset cannot occur until a state's actions have been completed.

State R0:Reset Start. The node sends a BUS_RESET signal whose length is governed by `reset_time`. In the case of a standard bus reset, this is long enough for all other bus activity to settle down (RESET_TIME is longer than the worst case packet transmission plus the worst case bus turn-around time). SHORT_RESET_TIME for an arbitrated (short) bus reset is significantly shorter since the bus is already in a known state following arbitration.

Transition R0:R1. The node has been sending a BUS_RESET signal long enough for all its connected neighbors to detect it.

State R1:Reset Wait. The node sends out IDLEs, waiting for all its active ports to receive IDLE or RX_PARENT_NOTIFY (either condition indicates that the connected PHYs have left their R0 state).

Transition R1:R0. The node has been waiting for its ports to go idle for too long (this can be a transient condition caused by multiple nodes being reset at the same time); return to the R0 state again. This time-out period is a bit longer than the R0:R1 time-out to avoid a theoretically possible oscillation between two nodes in states R0 and R1.

Transition R1:T0. All the connected ports are receiving IDLE or RX_PARENT_NOTIFY (indicating that the connected PHYs are in reset wait or starting the tree ID process).

7.9.2.1.2 Bus reset actions and conditions

Table 7-16 — Bus reset actions and conditions (Sheet 1 of 3)

```
boolean connection_in_progress[NPORT];    // Not referenced outside of the reset state machines
timer connect_timer();                  // Timer for connection status monitor

void connection_status() {              // Continuously monitor port status in all states
   int i;

   isolated_node = TRUE;                // Assume true until first active port found
   for (i = 0; i < NPORT; i++)
      isolated_node &= !active[i];
   for (i = 0; i < NPORT; i++) {
      if (connection_in_progress[i]) {
         if (!connect_detect[i])
            connection_in_progress[i] = FALSE;  // Lost attempted connection
         else if (connect_timer >= (isolated_node) ? 2 * CONNECT_TIMEOUT : CONNECT_TIMEOUT) {
            connection_in_progress[i] = FALSE;
            connected[i] = TRUE;        // Confirmed connection
            if (isolated_node)          // Can we arbitrate?
               ibr = TRUE;              // No, transition to R0 for reset
            else
               isbr = TRUE;             // Yes, arbitrate for short reset
         }
      } else if (!connected[i]) {
         if (connect_detect[i]) {       // Possible new connection?
            connect_timer = 0;          // Start connect timer
            connection_in_progress[i] = TRUE;
         }
      } else if (!connect_detect[i]) { // Disconnect?
         connected[i] = FALSE;          // Effective immediately!
         if (!active[i])                // No resets if disabled or suspended
            continue;                   // Keep examining other ports...
         if (root || child[i])          // Parent still connected?
            isbr = TRUE;                // Yes, arbitrate for short reset
         else
            ibr = TRUE;                 // No, transition to R0 for reset
      }
   }
}
```

 © 1997 IEEE

Table 7-16 — Bus reset actions and conditions (Sheet 2 of 3)

```
boolean reset_detected() {          // Qualify BUS_RESET with port status / history
   int i;

   if (PHY_state == R0 || PHY_State == R1)   // Ignore while in reset states themselves
      return(FALSE);
   for (i = 0; i < NPORT; i++)
      if (portR(i) == BUS_RESET)    // More than 20 ns (transient DS == 11)
         if (connection_in_progress[i]) {
            reset_time = 0;
            if (isolated_node)
               reset_time = SHORT_RESET_TIME;
            else if (connect_timer >= RESET_DETECT)
               reset_time = RESET_TIME;
            if (reset_time != 0) {
               connection_in_progress[i] = FALSE;
               connected[i] = TRUE
               return(TRUE);
            }
         } else if (active[i]) {
            reset_time = (PHY_state == RX) ? SHORT_RESET_TIME : RESET_TIME;
            return(TRUE);
         } else if (resuming[i]) {
            reset_time = (boundary_node) ? RESET_TIME : SHORT_RESET_TIME;
            return(TRUE);
         }
   return(FALSE);
}

void reset_start_actions() {     // Transmit BUS_RESET for reset_time on all ports
   int i;
   root = FALSE;

   PH_EVENT.indication(BUS_RESET_START);
   ibr = isbr = FALSE;           // Don't replicate resets!
   breq = NO_REQ;                // Discard any and all link requests
   child_count = physical_ID = 0;
   bus_initialize_active = TRUE:
   if (gap_count_reset_disable)  // First reset since setting gap_count?
      gap_count_reset_disable = FALSE;  // If so, leave it as is and arm it for next
   else
      gap_count = 0x3F;          // Otherwise, set it to the maximum
   for (i = 0; i < NPORT; i++) {
      if (active[i])             // For active ports, propagate appropriate signal
         portT(i, (suspend) ? TX_SUSPEND : BUS_RESET);
      else if (connect_detect[i] && resuming[i])
         portT(i, BUS_RESET);
      else if (disable[i])
         portT(i, IDLE);         // Maintain IDLE while TpBias driven low
      else
         portT(i, IDLE);         // Ignore disconnected and suspended ports
      child[i] = FALSE;
      child_ID_complete[i] = FALSE;
   }
   arb_timer = 0;                // Start timer
}

void reset_wait_actions() {      // Transmit IDLE
   int i;

   for (i = 0; i < NPORT; i++)
      portT(i, IDLE);
   arb_timer = 0;                // Restart timer
}
```

Table 7-16 — Bus reset actions and conditions (Sheet 3 of 3)

```
boolean reset_complete() {          // TRUE when all ports idle or in tree-ID
   int i;

   for (i = 0; i < NPORT; i ++)
      if ((portR(i) != IDLE) && (portR(i) != RX_PARENT_NOTIFY) && port_status[i])
         return(FALSE);
   rx_speed = S100;                 // For leaf node's self-ID packet(s)
   return(TRUE);                    // Transition to tree identify
}
```

7.9.2.2 Self identify

The self identify process has each node uniquely identify itself and broadcast its characteristics to any management services.

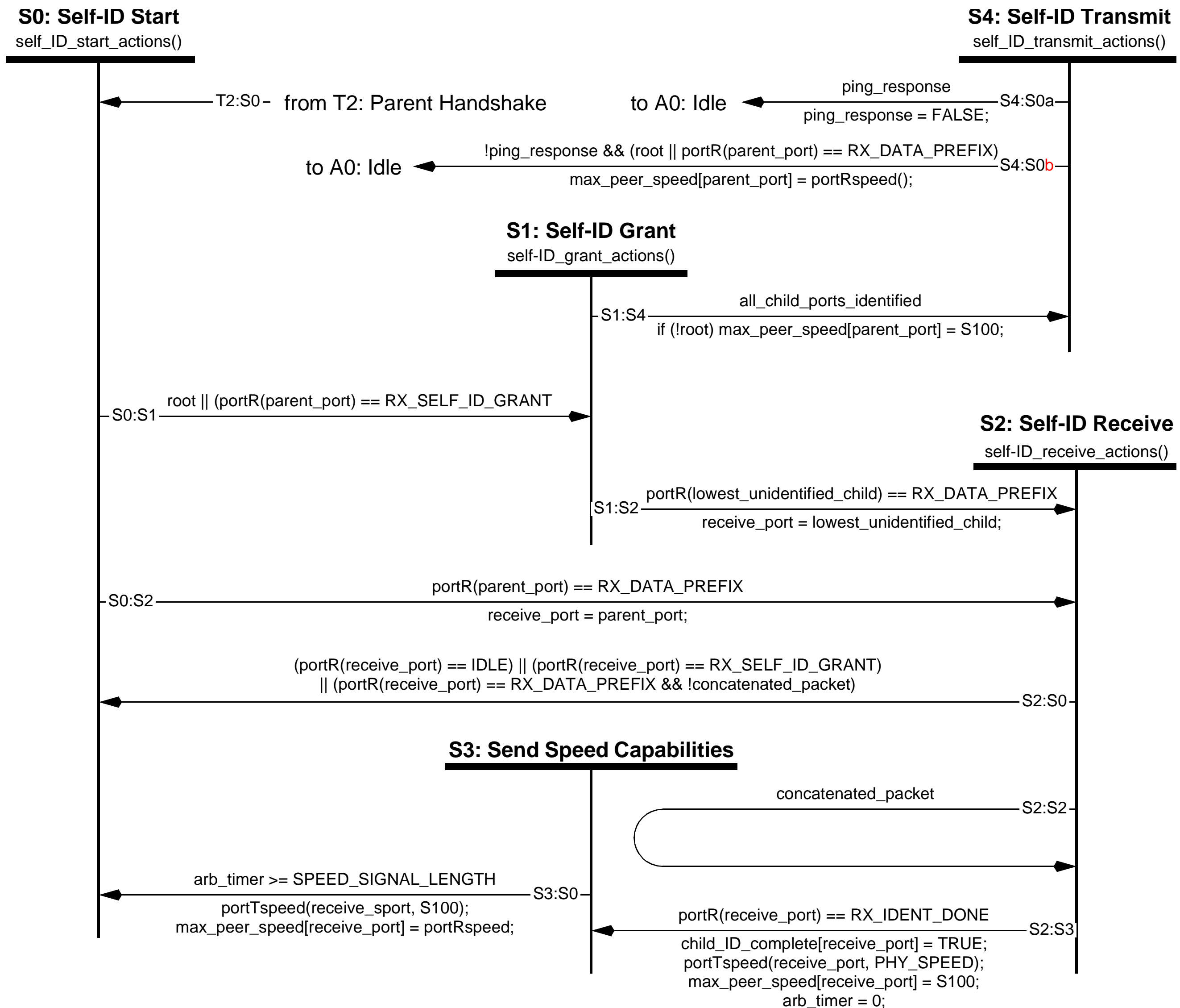

Figure 7-10 — Self-ID state machine

This is an unapproved standards draft, subject to change
© 1997 IEEE

7.9.2.2.1 Self-ID state machine notes

State S0: Self-ID Start. At the start of the self-ID process, the PHY is waiting for a grant from its parent or the start of a self-ID packet from another node. This state is also entered whenever a node is finished receiving a self-ID packet and all its children have not yet finished their self identification.

Transition S0:S1. If a node is the root, or if it receives a RX_SELF_ID_GRANT signal (0Z) from its parent, it enters the Self-ID Grant state.

Transition S0:S2. If a node receives a RX_DATA_PREFIX signal (10) from its parent, it knows that a self-ID packet is coming from a node in another branch in the tree.

State S1: Self-ID Grant. This state is entered when a node is given permission to send a self-ID packet. If it has any unidentified children, it sends a TX_GRANT signal (Z0) to the lowest numbered of those. All other connected ports are sent a TX_DATA_PREFIX signal (01) to warn them of the start of a self-ID packet.

Transition S1:S2. When the PHY receives a RX_DATA_PREFIX signal (10) from its lowest numbered unidentified child, it enters the Self-ID Receive state.

Transition S1:S4. If there are no more unidentified children, it immediately transitions to the Self-ID Transmit state.

State S2: Self-ID Receive. As data bits are received from the bus they are passed on to the link layer as PHY data indications. This process is described in clause 4.4.1.2 of IEEE Std 1394-1995. Note that multiple self-ID packets may be received in this state.

Transition S2:S0. When the receive port goes IDLE (ZZ), gets a RX_SELF_ID_GRANT (0Z) or observes RX_DATA_PREFIX (10) for a unconcatenated packet it enters the Self-ID Start state to continue the self-ID process for the next child. The last case guards against a possible failure to observe IDLE.

Transition S2:S2. Multiple self-ID packets are received by the PHY and self_ID_receive_actions reinvoked for each one.

Transition S2:S3. If the PHY gets an RX_IDENT_DONE (Z1) signal from the receiving port, it flags that port as identified and starts sending the speed capabilities signal. It also starts the speed signaling timer and sets the port speed to the S100 rate.

State S3: Send Speed Capabilities. If a node is capable of sending data at a higher rate that S100, it transmits on the receiving child port its speed capability signals as defined in clause 4.2.2.3 of IEEE Std 1394-1995 for a fixed duration SPEED_SIGNAL_LENGTH.

Transition S3:S0. When the speed signaling timer expires, any signals sent by the child have been latched, so it is safe to continue with the next child port.

State S4: Self-ID Transmit. At this point, all child ports have been flagged as identified, so the PHY can now send its own self-ID packet (see clause 7.4) using the process described in clause 4.4.1.1 of IEEE Std 1394-1995. When a non-root node is finished, it sends a TX_IDENT_DONE signal (1Z) and a speed capability signal as defined in clause 4.2.2.3 of IEEE Std 1394-1995 to its parent and IDLE (ZZ) to its children. The speed capability signal is transmitted for a fixed time duration (SPEED_SIGNAL_LENGTH). Simultaneously it monitors the bus for a speed capability transmission from the parent. The root node just sends IDLE (ZZ) to its children. Note that the children will then enter the Idle state described in the next clause, but they will never start arbitration since an adequate arbitration gap will never open up until the Self-ID process is completed for all nodes.

Transition S4:A0b. The PHY then enters the Idle state described in the next clause when the self-ID packet has been transmitted and if either of the following conditions are met:

1) The node is the root. When the root enters the Idle state, all nodes are now sending IDLE signals (ZZ) and the gap timers will eventually get large enough to allow normal arbitration to start.
2) The node starts to receive a new self-ID packet (RX_DATA_PREFIX – 10). This will be the self-ID packet for the parent node or another child of the parent. This event shall cause the PHY to transition immediately out of A0:Idle into A5:Receive.

7.9.2.2.2 Self-ID actions and conditions

Table 7-17 — Self ID actions and conditions (Sheet 1 of 3)

```
boolean all_child_ports_identified; // Set if all child ports have been identified
int lowest_unidentified_child;      // Lowest numbered active child that has not sent its self-ID

void self_ID_start_actions() {
   int i;

   all_child_ports_identified = TRUE;     // Will be reset if any active children are unidentified
   concatenated_packet = FALSE;           // Prepare in case of multiple self-ID packets
   for (i = 0; i < NPORT; i++)
     if (child_ID_complete[i])
        portT(i, TX_DATA_PREFIX);         // Tell identified children to prepare to receive data
     else {
        portT(i, IDLE);                   // Allow parent to finish
        if (child[i] && active[i]) {      // If active child
           if (all_child_ports_identified)
              lowest_unidentified_child = i;
           all_child_ports_identified = FALSE;
        }
     }
}

void self_ID_grant_actions() {
   int i;

   for (i = 0; i < NPORT; i++)
      if (!all_child_ports_identified && (i == lowest_unidentifed_child))
         portT(i, TX_GRANT);  // Send grant to lowest unidentified child (if any)
      else if (active[i])
         portT(i, TX_DATA_PREFIX);  // Otherwise, tell others to prepare for packet
}

void self_ID_receive_actions() {
   int i;

   portT(receive_port, IDLE);             // Turn off grant, get ready to receive
   receive_actions();                     // Receive (and repeat) packet
   if (!concatenated_packet) {            // Only do this on the first self-ID packet
      if (physical_ID < 63)               // Stop at 63 if malconfigured bus
         physical_ID = physical_ID + 1;   // Otherwise, take next PHY address
      for (i = 0; i < NPORT; i++)
         portT(i, IDLE);                  // Turn off all transmitters
   }
}

void self_ID_transmit_actions() {
   int last_SID_pkt = (NPORT + 4) / 8;
   int SID_pkt_number;              // Packet number counter
   int port_number = 0;             // Port number counter
   quadlet self_ID_pkt, ps;

   receive_port = NPORT;            // Indicate that we are transmitting (no port has this number)
   start_tx_packet(S100);           // Send data prefix and 98.304 Mbit/sec speed code
   PH_DATA.indication(DATA_START, S100);
   for (SID_pkt_number = 0; SID_pkt_number <= last_SID_pkt; SID_pkt_number++) {
```

 © 1997 IEEE

Table 7-17 — Self ID actions and conditions (Sheet 2 of 3)

```
      selfID.dataQuadlet = 0;       // Clear all zero fields in self ID packet
      selfID.type = 0b10;
      selfID.phy_ID = physical_ID;
      if (SID_packet_number == 0) { // First self ID packet?
         selfID.L = LPS && Link_active;   // Link active or not?
         selfID.gap_cnt = gap_count;
         selfID.sp = PHY_SPEED;
         selfID.del = PHY_DELAY;
         selfID.c = CONTENDER;
         selfID.pwr = POWER_CLASS;
         selfID.i = initiated_reset;
      } else {
         selfID.seq = 1;            // Indicates second and subsequent packets
         selfID.n = SID_pkt_number - 1;   // Sequence number
      }
      ps = 0;                   // Initialize for fresh group of ports
      while (port_number < ((SID_pkt_number + 1) * 8 - 5)) {    // Concatenate port status
         if (port_number >= NPORT)
            ;                   // Unimplemented
         else if (!active[port_number])
            ps |= 0b01;         // Unconnected
         else if (child[port_number])
            ps |= 0v11;         // Active child
         else
            ps |= 0b10;         // Active parent
         port_number++;
         ps <<= 2;              // Make room for next port's status
      }
      selfID |= ps;
      if (SID_pkt_number == last_SID_pkt) { // Last packet?
         tx_quadlet(selfID);
         tx_quadlet(~selfID);
         stop_tx_packet(TX_DATA_END, S100); // Yes, signal data end
         PH_DATA.indication(DATA_END);
         breq = NO_REQ;         // Cancel pending requests (only fair and priority possible here)
      } else {
         selfID.m = 1;          // Other packets follow, set "more" bit
         tx_quadlet(self_ID_pkt);
         tx_quadlet(~self_ID_pkt);
         stop_tx_packet(TX_DATA_PREFIX, S100);  // Keep bus for concatenation
         PH_DATA.indication(DATA_PREFIX);
         PH_DATA.indication(DATA_START, S100);
      }
   }
   if (!ping_response) {        // Skip if self-ID packet was in reply to a ping
      for (port_number = 0; port_number < NPORT; port_number++)
         if (root || port_number != parent_port)
            portT(port_number, IDLE);           // Turn off transmitters to children
         else
            portT(port_number, TX_IDENT_DONE);  // Notify parent that self-ID is complete
      if (!root) {                           // If we have a parent...
         portTspeed(parent_port, PHY_SPEED); // Send speed signal (if any)
         wait_time(SPEED_SIGNAL_LENGTH);
         portTspeed(parent_port, S100);      // Stop sending speed signal
      }
      PH_EVENT.indication(SELF_ID_COMPLETE, physical_ID, root);  // Register 0
   }
}
```

Table 7-17 — Self ID actions and conditions (Sheet 3 of 3)

```
void tx_quadlet(quadlet quad_data) {                // Send a quadlet...
   int i;

   breq = NO_REQ;                                  // Cancel any request (keep in step with the link)
   for (i = 0; i < 32; i++) {                      // ...a bit at a time
      tx_bit(quad_data & 0x80000000);              // From the most significant downwards
      PH_DATA.indication(quad_data & 0x80000000);  // Copy our own self-ID packet to the link
      quad_data <<= 1;                             // Shift to next bit
   }
}
```

7.9.2.3 Normal arbitration

Normal arbitration is entered as soon as a node has finished the self identification process. At this point, a simple request-grant handshake process starts between a node and its parent (and all parents up to the root).

This is an unapproved standards draft, subject to change

© 1997 IEEE

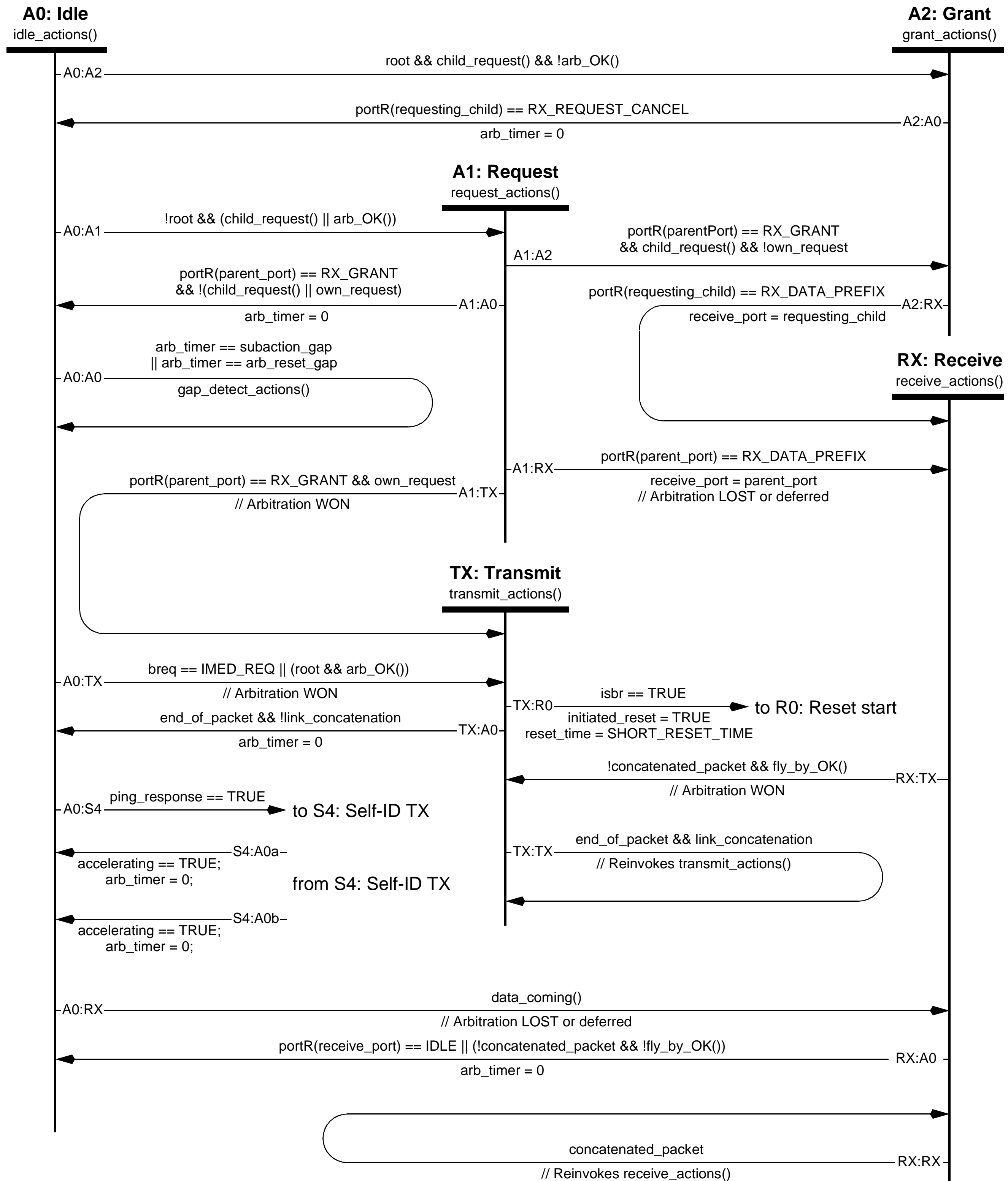

Figure 7-11 — Cable arbitration state machine

7.9.2.3.1 Normal arbitration state machine notes

State A0: Idle. All inactive nodes stay in the idle state until an internal or external event. All ports transmit the IDLE arbitration signal (ZZ). Transitions into this state from states where idle was not being sent reset an idle period timer.

Transition A0:A0. If a subaction gap or arbitration reset gap occurs, the PHY notifies the link layer. In addition, if this is the first subaction gap after a bus reset it signals the completion of the self-identify process and the PHY notifies the node controller. The detection of an arbitration reset gap marks the end of a fairness interval; the PHY sets the arbitration enable flag.

Transition A0:A1. If the PHY has a queued request from its own link or receives an RX_REQUEST signal (0Z) from one of its children (and is not the root), it passes the request on to its parent. The `arb_OK()` function qualifies asynchronous requests according to the time elapsed since A0: Idle was last entered. In particular, notice that the test for a subaction gap is performed for a single value (equality), not a greater than comparison. If arbitration were to be initiated at other times between the detection of a subaction gap and an arbitration reset gap, some nodes could mistakenly observe an arbitration reset gap.

Transition A0:A2. If, on the other hand, the PHY receives a RX_REQUEST signal (0Z) from one of its children, has no queued requests from its own link and is the root, it starts the bus grant process.

Transition A0:RX. If the PHY receives the RX_DATA_PREFIX signal on any of its ports while idle, it shifts into the Receive state and notifies the link layer that any pending arbitration requests have been lost.

Transition A0:TX. If the PHY has a queued isochronous request and is the root or if the PHY has a queued immediate request (generated during packet reception if the link layer needs to send an acknowledge), the PHY notifies the link layer that it is ready to transmit and enters the Transmit state.

Transition A0:S4. In response to the receipt of a PHY “ping” packet, the variable ping_response is set TRUE and a transition is made to the Self-ID Transmit State to send the self-ID packet(s).

State A1: Request. At this point, the PHY sends a TX_REQUEST signal (Z0) to its parent and a data prefix (01) to all its connected children. This will signal all children to get ready to receive a packet.

Transition A1:A0. If the PHY receives a RX_GRANT signal (00) from its parent and the requesting child has withdrawn its request, the PHY returns to Idle state.

Transition A1:A2. If the PHY receives a RX_GRANT signal (00) from its parent and the requesting child is still making a request, the PHY grants the bus to that child.

Transition A1:RX. If the PHY receives a RX_DATA_PREFIX signal (10) from its parent, then it knows that it has lost the arbitration process and prepares to receive a packet. If the link layer was making the request, it is notified.

Transition A1:TX. If the PHY receives a RX_GRANT signal (00) from its parent and the link layer has an outstanding request (asynchronous or isochronous), the PHY notifies the link layer that it can now transmit and enters the Transmit state.

State A2: Grant. During the grant process, the requesting child is sent a TX_GRANT signal (Z0) and the other children are sent a TX_DATA_PREFIX (01) so that they will prepare to receive a packet.

Transition A2:A0. If the requesting child withdraws its request, the granting PHY sees its own TX_GRANT signal coming back as a RX_REQUEST_CANCEL signal (Z0) and returns to the Idle state.

Transition A2:RX. If the data prefix signal is received from the requesting child, the grant handshake is complete and the node goes into the Receive state.

 This is an unapproved standards draft, subject to change © 1997 IEEE

State RX: Receive. When the node starts the receive process, it clears all its request flags (forcing the link layer to send new requests if there were any queued) notifies the link layer that the bus is busy and starts the packet receive process described below. Note that the packet received could be a PHY packet (self-ID, link-on or PHY configuration), acknowledge, or normal data packet. PHY configuration and link-on packets are interpreted by the PHY, as well as being passed on to the link layer.

Transition RX:A0. If transmitting node stops sending any signals (received signal is ZZ) or if a packet ends normally when the received signal is RX_DATA_END, the bus is released and the PHY returns to the idle state.

Transition RX:RX. If a packet ends and the received signal is RX_DATA_PREFIX (10), then there may be another packet coming, so the receive process is restarted.

State TX: Transmit. Unless an arbitrated (short) bus reset has been requested, the transmission of a packet starts by the node sending a TX_DATA_PREFIX and speed signal as described in clause 4.2.2.3 of IEEE Std 1394-1995 for 100 ns, then sending PHY clock indications to the link layer. For each clock indication, the Link sends a PHY data request. The clock indication – data request sequence repeats until the Link sends a DATA_END. Concatenated packets are handled within this state whenever the Link sends at least one data bit followed by a DATA_PREFIX. The arbitration enable flag is cleared if this was a fair request.

Transition TX:A0. If the link layer sends a DATA_END, the PHY shuts down transmission using the procedure described in clause 4.4.1.1 of IEEE Std 1394-1995 and returns to the Idle state.

Transition TX:R0. If arbitration has succeeded and the `reset_time` variable has a nonzero value, there is no packet to transmit. The PHY transition's to the Reset start state to commence a short bus reset.

7.9.2.3.2 Normal arbitration actions and conditions

Table 7-18 — Normal arbitration actions and conditions (Sheet 1 of 3)

```
int requesting_child;               // Lowest numbered requesting child

boolean fly_by_OK() {                  // TRUE if fly-by acceleration OK

   if (!enab_accel)
      return(FALSE);
   else if (receive_port == parent_port)
      return(FALSE);
   else if (speed == S100 && rx_speed != S100)
      return(FALSE);
   else if (breq == ISOCH_REQ)
      return(TRUE);
   else if (ack && accelerating)
      return(breq == PRIORITY_REQ || (breq == FAIR_REQ && arb_enable));
   else
      return(FALSE);
}

boolean child_request() {              // TRUE if a child is requesting the bus
   int i;

   for (i = 0; i < NPORT; i++)
      if (active[i] && child[i] && (portR(i) == RX_REQUEST)) {
         requesting_child = i;         // Found a child that is requesting the bus
         return(TRUE);
      }
   return(FALSE);
}

boolean token_request() {              // TRUE if at least one token-enabled child needs a grant
   int i;
```

Table 7-18 — Normal arbitration actions and conditions (Sheet 2 of 3)

```
   for (i = 0; i < NPORT; i++)
      if (active[i] && child[i] && grant_needed[i]) {
         return(TRUE);
      }
   return(FALSE);
}

boolean data_coming() {                  // TRUE if data prefix is received on any port
   int i;

   for (i = 0; i < NPORT; i++)
      if (active[i] && (portR(i) == RX_DATA_PREFIX)) {
         receive_port = i;               // Remember port for later...
         return(TRUE);                   // Found a port that is sending a data_prefix signal
      }
   return(FALSE);
}

void gap_detect_actions() {

  if (arb_timer >= reset_gap_time) {      // End of fairness interval?
     arb_enable = TRUE;                   // Reenable fair arbitration
     PH_DATA.indication(ARBITRATION_RESET_GAP);  // Alert link
  } else if (arb_timer >= subaction_gap_time) {
     PH_DATA.indication(SUBACTION_GAP);   // Notify link
     if (bus_initialize_active) {         // End of self-identify process for whole bus?
        PH_EVENT.indication(BUS_RESET_COMPLETE);
        bus_initialize_active = FALSE;
     }
  }
}

void idle_actions() {
   int i;

   rx_speed = S100;                  // Default in anticipation of no explicit receive speed code
   for (i = 0; i < NPORT; i++)       // Turn off all transmitters
      portT(i, IDLE);
}


void request_actions() {
   int i;

   for (i = 0; i < NPORT; i++)
      if (active[i] && child[i] && (own_request || i != requesting_child))
         portT(i, TX_DATA_PREFIX);  // Send data prefix to all non-requesting children
   portT(parent_port, TX_REQUEST);  // Send request to parent
}

boolean arb_OK() {                 // TRUE if OK to request the bus
   boolean async_arb_OK = FALSE; // Timing window OK for asynchronous arbtration?

   if (arb_timer < subaction_gap_time + arb_delay)
      async_arb_OK = enab_accel && accelerating && ack;
   else if (arb_timer == subaction_gap_time + arb_delay)
      async_arb_OK = TRUE;
   else if (arb_timer >= arb_reset_gap_time + arb_delay)
      async_arb_OK = TRUE;
   if (breq == ISOCH_REQ)
      own_request = !parent_token_enable;
   else if (breq == PRI_REQ)
      own_request = async_arb_OK;
   else if (breq == FAIR_REQ)
      own_request = async_arb_OK && arb_enable;
```

 © 1997 IEEE

Table 7-18 — Normal arbitration actions and conditions (Sheet 3 of 3)

```
   else if (isbr)
      own_request = async_arb_OK;
   else
      own_request = FALSE;
   return(own_request);
}

void grant_actions() {
   int i;

   for (i = 0; i < NPORT; i++)
      if (i == requesting_child) {
         portT(i, TX_GRANT);         // Send grant to requesting child
         if (token_request[i]) {
            portT(parent_port, TX_DATA_PREFIX]);
            token_request[i] = FALSE;
         }
         token_request[i] = FALSE;
      }
      else if (active[i] && child [i])
         portT(i, TX_DATA_PREFIX);  // Send data prefix to all non-requesting children
}
```

7.9.2.3.3 Receive actions and conditions

Table 7-19 — Receive actions and conditions (Sheet 1 of 2)

```
void receive_actions() {
   boolean end_of_data;
   unsigned bit_count = 0, i, rx_data, tx_speed;

   ack = concatenated_packet = FALSE;
   if (!enab_accel && (breq == FAIR_REQ || breq == PRIORITY_REQ)) {
      breq = NO_REQ;                   // Cancel the request
      PH_ARB.confirmation(LOST);       // And let the link know
   }
   PH_DATA.indication(DATA_PREFIX);    // Send notification of bus activity
   start_rx_packet();                  // Start up receiver and repeater
   tx_speed = rx_speed;
   PH_DATA.indication(DATA_START, rx_speed); // Send speed indication
   do {
      rx_bit(&rx_data, &end_of_data);
      if (!end_of_data) {              // Normal data, send to link layer
         PH_DATA.indication(rx_data);
         if (bit_count < 64)           // Accumulate first 64 bits
            rx_phy_pkt.bits[bit_count] = rx_data;
         bit_count++;
         ack = (bit_count == 8);       // For acceleration, any 8-bit packet is an ack
         if (bit_count > 8 && (breq == FAIR_REQ || breq == PRIORITY_REQ)) {
            breq = NO_REQ;             // Fly-by impossible
            PH_ARB.confirmation(LOST); // Let the link know
         }
      }
   } while (!end_of_data);
   if (portR(receive_port) == IDLE) {  // Unexpected end of data...
      if (bit_count > 8 && (breq == FAIR_REQ || breq == PRIORITY_REQ)) {
         breq = NO_REQ; // Discard (unless link believes there was an ACK)
         PH_ARB.confirmation(LOST);
      }
      ack = FALSE;      // Disable fly-by acceleration
      return;
   }
   switch(portR(receive_port)) {       // Send appropriate end of packet indicator
      case RX_DATA_PREFIX:
```

Table 7-19 — Receive actions and conditions (Sheet 2 of 2)

```
         concatenated_packet = TRUE;
         PH_DATA.indication(DATA_PREFIX);    // Concatenated packet coming
         stop_tx_packet(DATA_PREFIX, rx_speed);
         break;

      case RX_DATA_END:
         if (fly_by_OK())
            stop_tx_packet(DATA_PREFIX, tx_speed); // Fly-by concatenation
         else {
            PH_DATA.indication(DATA_END);    // Normal end of packet
            stop_tx_packet(DATA_END, tx_speed);
         }
         break;
   }
   if (bit_count == 64) {               // We have received a PHY packet
      for (i = 0; i < 32; i++)          // Check PHY packet for good format
         if (rx_phy_pkt.bits[i] == rx_phy_pkt.checkBits[i])
            return;                     // Check bits invalid - ignore packet
      switch(rx_phy_pkt.type) {         // Process PHY packets by type
         case 0b00:                     // PHY config packet
            if (rx_phy_pkt.ext_type == 0)       // Ping packet?
               ping_response = (rx_phy_pkt.phy_ID == physical_ID);
            else if (    (   rx_phy_pkt.ext_type == 1 || rx_phy_pkt.ext_type == 5
                          || rx_phy_pkt.ext_type == 6)
                     && (rx_phy_pkt.phy_ID == physical_ID))
               remote_access(rx_phy_pkt.page, rx_phy_pkt.port,rx_phy_pkt.reg,
                             rx_phy_pkt.data);
            else if (rx_phy_pkt.ext_type == 3)  // Resume packet?
               resuming = (rx_phy_pkt.phy_ID == physical_ID);
            else {                      // Must be PHY configuration packet
               if (rx_phy_pkt.R)        // Set force_root if address matches
                  force_root = (rx_phy_pkt.address == physical_ID)
               if (rx_phy_pkt.T) {      // Set gap_count unconditionally
                  gap_count = rx_phy_pkt.gap_count;
                  gap_count_reset_disable = TRUE;
               }
            }
            break;

         case 0b01:                     // Link-on packet
            if (rx_phy_pkt.address == physical_ID)
               PH_EVENT.indication(LINK_ON);
            break;
      }
   }
}

void remote_access() {                  // Current value of remotely accessed register
   if (     rx_phy_pkt.ext_type == 6 && page == 0
         && (reg == 0b1010 || reg == 0b0111))
      write_phy_reg(page, port, reg, data);
   phy_resp_pkt.dataQuadlet = 0;
   phy_resp_pkt.phy_ID = physical_ID;
   phy_resp_pkt.type = 3;
   phy_resp_pkt.data = read_phy_reg(page, port, reg);
   phy_response = TRUE;
}
```


This is an unapproved standards draft, subject to change
© 1997 IEEE

7.9.2.3.4 Transmit actions and conditions

Table 7-20 — Transmit actions and conditions (Sheet 1 of 2)

```
void transmit_actions() {

   end_of_packet = FALSE;
   int bit_count = 0, i;
   PHY_packet rx_phy_pkt, tx_phy_pkt;
   phyData data_to_transmit;

   if (breq == FAIR_REQ)
      arb_enable = FALSE;
   breq = NO_REQ;
   tx_speed = speed;              // Assume speed has been set correctly...
                                  // (from PH_ARB.request or concatenated packet speed code)
   receive_port = NPORT;          // Impossible port number ==> PHY transmitting
   start_tx_packet(tx_speed);     // Send data prefix & speed signal
   if (isbr)                      // Avoid phantom packets...
      return;
   PH_ARB.confirmation(WON);      // Signal grant on Ctl[0:1]
   while (!end_of_packet) {
      PH_CLOCK.indication();      // Tell link to send data
      data_to_transmit = PH_DATA.request();     // Wait for data from the link
      switch(data_to_transmit) {
         case DATA_ONE:
         case DATA_ZERO:
            tx_bit(data_to_transmit);
            if (bit_count < 64)                 // Accumulate possible PHY packet
               rx_phy_pkt.bits[bit_count] = data_to_transmit;
            bit_count++;
            break;

         case DATA_PREFIX:
            end_of_packet = link_concatenation = TRUE;
            stop_tx_packet(DATA_PREFIX, tx_speed); // MIN_PACKET_SEPARATION needs to be
            break;      // guaranteed by stop_tx_packet() and subsequent start_tx_packet()

         case DATA_END:
            stop_tx_packet(DATA_END, tx_speed);
            end_of_packet = TRUE;          // End of packet indicator
            break;
      }
   }
   ack = (bit_count == 8);     // Used elsewhere to (conditionally) accelerate
   if (bit_count == 64) {      // We have transmitted a PHY packet
      for (i = 0; i < 32; i++)          // Check PHY packet for good format
         if (tx_phy_pkt.bits[i] == tx_phy_pkt.checkBits[i])
            return;                     // Check bits invalid - ignore packet
      if (tx_phy_pkt.type == 0b00)
         if (tx_phy_pkt.ext_type == 0)       // Ping packet?
            ping_response = (tx_phy_pkt.phy_ID == physical_ID);
         else if (    (    tx_phy_pkt.ext_type == 1 || tx_phy_pkt.ext_type == 5
                       || tx_phy_pkt.ext_type == 6)
                  && (tx_phy_pkt.phy_ID == physical_ID))
            remote_access(tx_phy_pkt.page, tx_phy_pkt.port, tx_phy_pkt.reg,
                          tx_phy_pkt.data);
```

Table 7-20 — Transmit actions and conditions (Sheet 2 of 2)

```
         else {                     // Must be PHY configuration packet
            if (tx_phy_pkt.R)       // Set force_root if address matches
               force_root = (tx_phy_pkt.address == physical_ID)
            if (tx_phy_pkt.T) {     // Set gap_count unconditionally
               gap_count = tx_phy_pkt.gap_count;
               gap_count_reset_disable = TRUE;
            }
         }
   }
}
```


This is an unapproved standards draft, subject to change
© 1997 IEEE

7.9.3 Port connection

The port connection state machines operate independently for each port, *i*, where *i* is greater than or equal to zero and less than NPORT. While a port is in the active state its arbitration, data transmission, reception and repeat behaviors are specified by the state machines in clause 7.9.2. When a PHY port is in any state other than active it is permissible for it to lower its power consumption; the only functional component of a PHY that shall be active in all states is the physical connection detect circuitry.

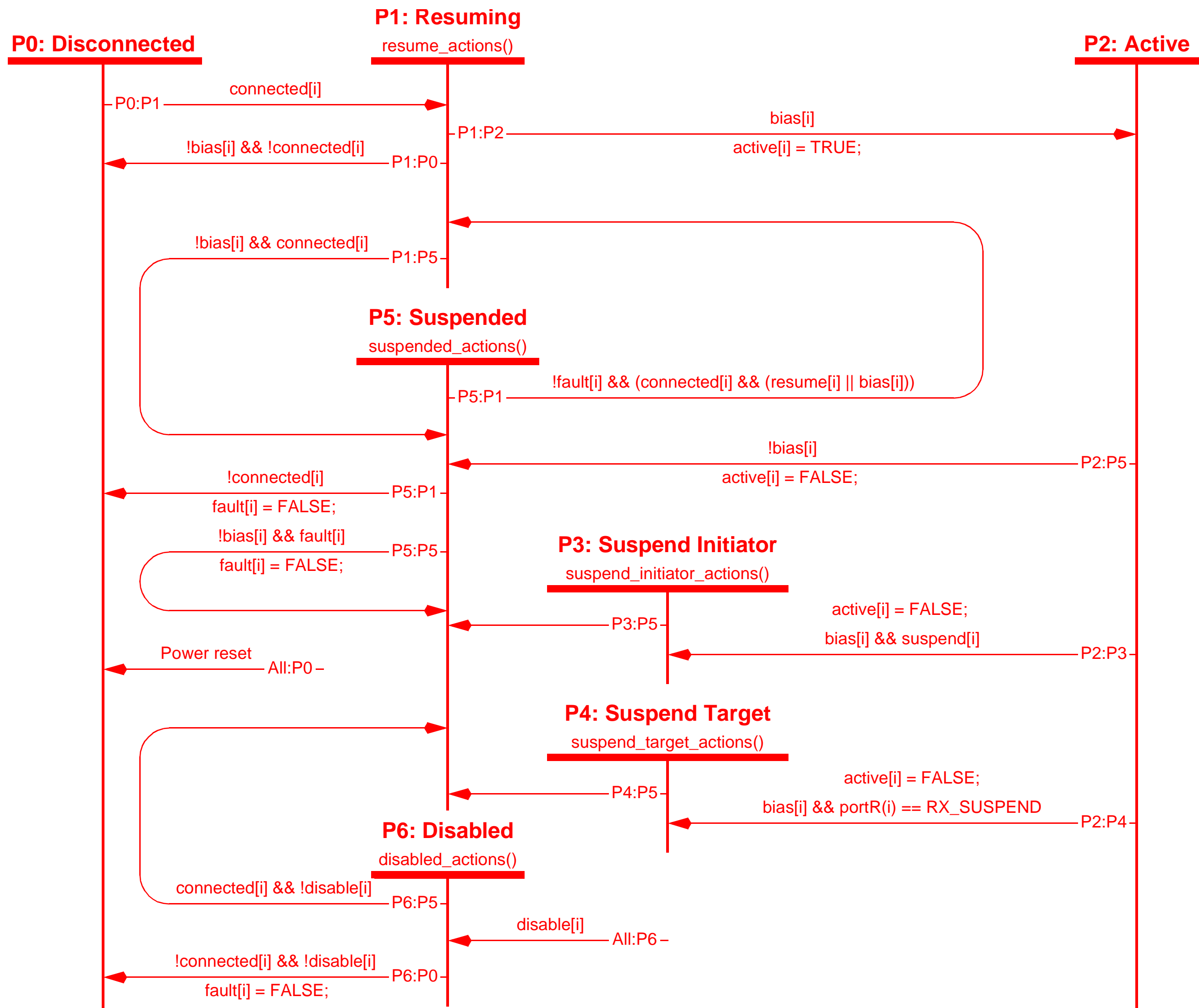

Figure 7-12 — Port connection state machine

7.9.3.1 Port connection state machine notes

Transition All:P0. A power reset of the PHY initializes each port as disconnected.

Transition All:P6. If the PHY port's Disable bit is set to one, either as the result of a register write request from the link or upon receipt of a PHY remote access packet, the PHY port enters the disabled state.

State P0: Disconnected. The generation of TpBias is disabled and the outputs are in a high-impedance state. The PHY may place most of its circuitry in a low-power consumption state. The connection detect circuit. shall be active even if other components of the PHY port are in a low-power state.

Transition P0:P1. When a port's connection detect circuitry signals that its peer PHY port is physically connected, the PHY port transitions to the resuming state.

State P1: Resuming. The PHY port tests both the connection status and the presence of TpBias to determine if normal operations may be resumed. If the port is connected, TpBias is present and there are no other active ports, the PHY waits five RESET_DETECT intervals before any state transitions. Otherwise, in the case of a boundary node with one or more active ports, the PHY waits two RESET_DETECT intervals before any state transitions.

Transition P1:P0. A resuming PHY port that loses its physical connection to its peer PHY port transitions to the disconnected state.

Transition P1:P2. If the PHY port is both connected and observes TpBias, it transitions to the active state.

Transition P1:P5. A resuming PHY port that remains connected to its peer PHY port but fails to observe TpBias transitions to the suspended state.

State P2: Active. The PHY port is fully operational, capable of transmitting or receiving and repeating arbitration signals or clocked data. While the port remains active, the behavior of this port and the remainder of the PHY are subject to the cable arbitration states specified in clause 7.9.

Transition P2:P3. Upon the receipt of a PHY remote access packet that sets the Initiate_suspend bit to one, the PHY port leaves the active state to start functioning as a suspend initiator.

Transition P2:P4. If an active port observes an RX_SUSPEND signal it becomes a suspend target leaves the active state.

Transition P2:P5. An active port that fails to observe TpBias transitions to the suspended state in order to test the cable signals. This transition is usually the result of a physical disconnection or the loss of power to the connected peer PHY port.

State P3: Suspend Initiator. A suspend initiator. responds to the PHY remote access packet by transmitting a PHY remote reply packet that with the Initiate_suspend bit set to one. Since the suspend initiator is no longer active, the connection status monitor triggers the generation of bus reset on all of the other active ports at this PHY. In the meantime, the suspend initiator signals TX_SUSPEND to its connected peer PHY and then waits for TpBias to be driven low. If NOTIFY_HOLD elapses and the connected peer PHY has not driven TpBias low, the suspend operation has faulted and the Fault bit is set to one. In either case the suspend initiator disables the generation of TpBias and places the outputs in a high-impedance state.

Transition P3:P5. Upon completion of the actions associated with this state, the PHY port unconditionally transistions to the suspended state.

State P4: Suspend Target. A suspend target propagates the RX_SUSPEND siganl as TX_SUSPEND on all of the PHY's other, active ports as part of its normal repeating functions. In the meantime the suspend target drives its TpBias outputs below 0.4 V in order to signal the suspend initiator that RX_SUSPEND was detected. The suspend target then waits for TpBias to be drivven low. If BIAS_HOLD elapses and the suspend initiator has not driven TpBias low, the suspend operation has faulted and the Fault bit is set to one. In either case the suspend target disables the generation of TpBias and places the outputs in a high-impedance state.

Transition P4:P5. Upon completion of the actions associated with this state, the PHY port unconditionally transistions to the suspended state.

 © 1997 IEEE

State P5: Suspended. The PHY may place most of its circuitry in a low-power consumption state. The connection detect circuit. shall be active even if other components of the PHY port are in a low-power state.

Transition P5:P0. A suspended PHY port that loses its physical connection to its peer PHY port transitions to the disconnected state.

Transition P5:P1. So long as the port's Fault bit is not one, any one of a number of events cause a suspended PHY port to transition to the resuming state: a) the receipt of a PHY remote access packet that sets the PHY register Initiate_resume bit to one, b) the receipt of a PHY resume packet or c) the detection of TpBias.

Transition P5:P5. If the port entered the suspended state in a faulted condition (*i.e.*, TpBias was still present), the fault is cleared if and when TpBias is removed by the peer PHY.

State P6: Disable. Whenever the Disable bit in the PHY registers is set to one, the PHY port transitions to the disabled state. The Disable bit may be written either by the attached link or by a PHY remote access packet. The PHY may place most of its circuitry in a low-power consumption state. The connection detect circuit. shall be active even if other components of the PHY port are in a low-power state.

Transition P6:P0. If the Disable bit is zero and the PHY port is not physically connected to its peer PHY port, it transitions to the disconnected state.

Transition P6:P5. Otherwise, if the Disable bit is zero and the PHY port is connected it transitions to the suspended state.

7.9.3.2 Port connection actions and conditions

Table 7-21 — Port connection actions and conditions (Sheet 1 of 2)

```
void disabled_actions() {
   if (int_enable[i])
      if (link_active && LPS)
         PH_EVENT.indication(INTERRUPT);
      else if (wakeup)
         PH_EVENT.indication(LINK_ON);
}

void resume_actions() {
   connect_timer = 0;
   tpBias(i, 1);                       // Generate TpBias
   for (j = 0; j < NPORT; j++)         // Activate all other ports as resume initiators
      if (i != j)
         resume[i] = TRUE;
   while (((connect_timer < DETECT_MIN) && !bias[i])  // Wait for TpBias
      ;
   if (fault)
      tpBias(i, Z);                    // Release TpBias
   if (link_active && LPS && int_enable[i])
      PH_EVENT.indication(INTERRUPT);
   else if (!(link_active && LPS) && wakeup)
      PH_EVENT.indication(LINK_ON);
}

void suspend_initiator_actions(int i) {
   connect_timer = 0;
   portT(i, TX_SUSPEND);
   while (connect_timer < SHORT_RESET_TIME)
      ;
   portT(i, IDLE);
   while ((connect_timer < NOTIFY_HOLD) && bias[i])
      ;
   if (!bias[i]) {
      connect_timer = 0;
      tpBias(i, 0);
      while (connect_timer < BIAS_HOLD)
         ;
   }
   tpBias(i, Z);
   if (bias[i])                  // Suspend handshake refused by target?
      fault = connect_detect[i]; // Fault if there's still a physical connection
}

void suspend_target_actions() {
   connect_timer = 0;
   tpBias(i, 0);                 // Drive TpBias low
   while ((connect_timer < BIAS_HOLD) && bias[i])
      ;
   tpBias(i, Z);
   if (bias[i])                  // Suspend initiator reneged?
```

 © 1997 IEEE

Table 7-21 — Port connection actions and conditions (Sheet 2 of 2)

```
      fault = connect_detect[i]; // Fault if initiator still connected
}

void suspended_actions() {
   if (int_enable[i])
      if (link_active && LPS)
         PH_EVENT.indication(INTERRUPT);
      else if (wakeup)
         PH_EVENT.indication(LINK_ON);
}
```

© 1997 IEEE

This is an unapproved standards draft, subject to change