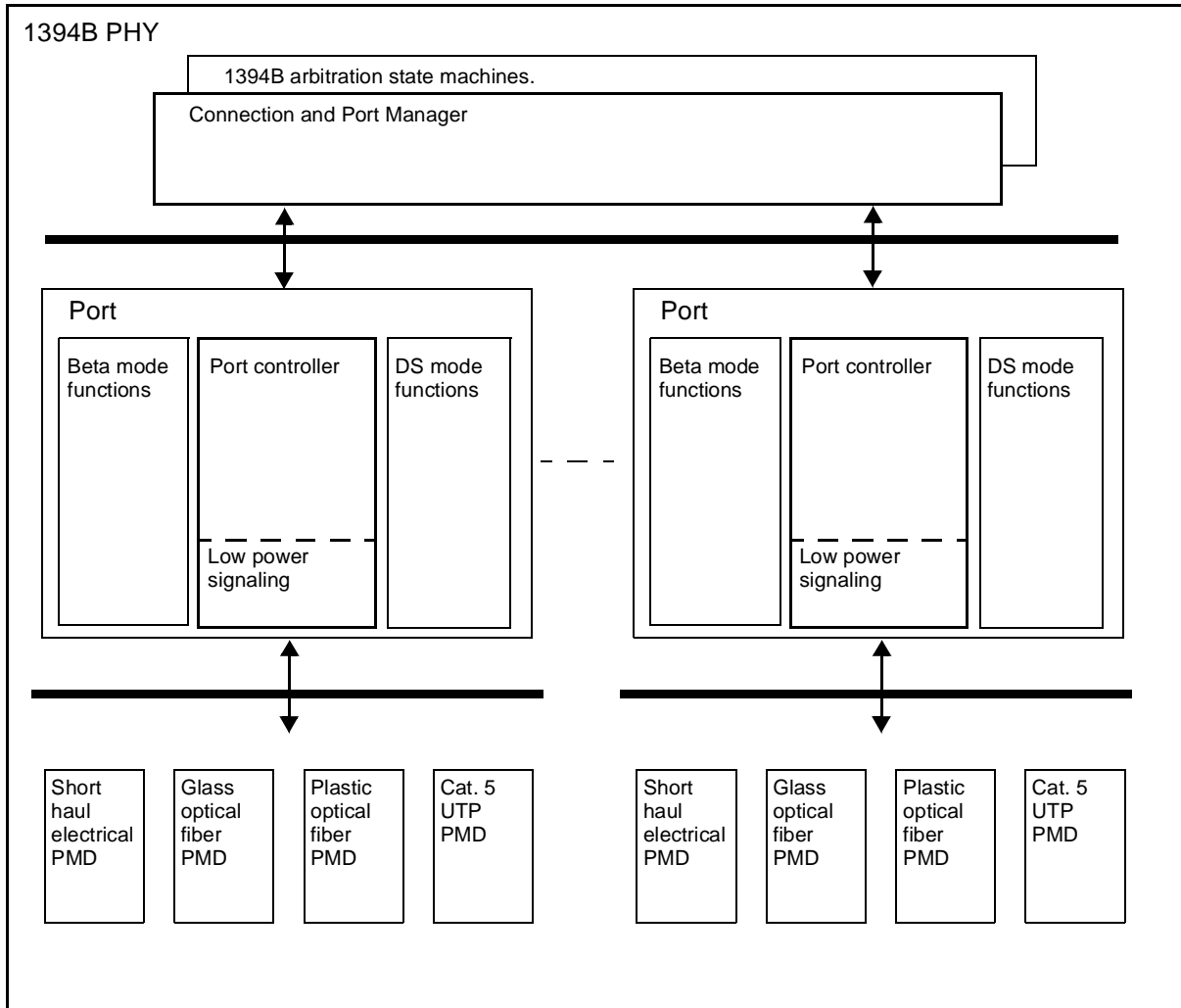


# 11. Connectivity Management

## 11.1 Overview

This section specifies the Connection and Port Manager for the physical layer and the Port Controller for each port, together with the communications used between these two modules, and the services provided by these two modules. The port abstraction is the primary interface inside the node between IEEE Std. 1394-1995 protocols and the P1394b gigabit protocols

The model for a port is shown below



The Connection Management function interacts with the physical media dependent layers, the Beta mode functions, DS mode functions and the repeater function by appropriate ~~chared~~-shared variables.

## 11.2 Port characteristics

The port is the device that controls the flow of data bits onto and off of the 1394 wires. The port state machine must detect the present or absence of its neighbor port, establish reliable communications with the other port in the most suitable mode of signaling, and must manage low-power modes of operation, i.e., sleep modes.

### 11.2.1 Definitions

- Port and port state machine are interchangeable terms when the context make the choice clear
- DS mode means the normal IEEE Std. 1394-1995 form of electrical signaling and handshaking
- Beta mode means the 8b/10b, full duplex form of binary encoded signaling
- Connection means the cabled interconnect between two ports
- Plug Present for a port means that there is a plug present at the local end. It does not imply an end-to-end connection. However, when no plug is present, further power savings are possible.
- Low Power signalling means signalling, based on the exchange of very low duty cycle tones, by which the connectivity status of a port is determined. This is used then the port is not active or disabled.
- Suspend means go into a low power mode of operation while maintaining the ability to process low power connection signalling
- Resume is a signal requiring the port to resume full speed, full power operation

### 11.2.2 Requirements

- a) The port must start up in DS mode if either the port or the port to which it is connected is not beta capable
- b) The port must start up in beta mode if both the ports are beta capable
- c) The port must run (the "operational speed") at the fastest speed of which both ports are capable, or allowed
- d) The port must support all speeds of packet at or below its operational speed through the use of data padding
- e) The port must support Suspend, Resume, and low power connection signalling.
- f) The port must detect and manage all connect/disconnect events

### 11.2.3 Port properties

A P1394b port

- a) shall be capable of operating at any speed range from S100 upwards, or any speed range from S800 upwards
- b) shall be capable of operating in Beta mode
- c) when capable of operating at S100, shall be capable of either DS at S100-S400, or Beta mode at S100-S400, or both
- d) shall not be brought out to a 1394-1995 connector unless capable of operating in DS mode
- e) may be connected directly to a suitable transceiver for long haul connection (thus also providing DC isolation)
- f) may use DC (e.g. capacitive or galvanic) isolation when operating in Beta mode

## 11.3 Connection Management

### 11.3.1 Connection management definition

The Connection management is specified as a continuously running C code machine.

The functions, variables and constants used in this C code have the following descriptions

**Table 11-1—Functions and variables used in Connectivity management**

Function, variable or constant	Description
active	Indicates that the port is in state P2:Active (set by state machine transitions to/from P2)
Beta_mode	Set if the port has determined that it is operating in Beta mode, unset otherwise (i.e. when operating in DS-Mode, or when in P0:disconnected). This is maintained as a read-only bit in the port register map.
bport_active	Variable shared with the Beta mode port. Set to true to indicate to the Beta port that it should commence operating. Set to false to indicate to the Beta port that it should cease operating.
connect_detect_valid	When true, indicates that the DC connect detect comparator will give a valid indication of the connection status.
connected	Connection established with a peer port, and operating speed negotiation completed (Beta mode only)
disabled	PHY register bit, set to disable a port under software control, and indicates that the port is in state P6:Disabled
local_plug_present	Indicates that an external implementation dependent mechanism has determined that there is at least a physical connection from the local node (maybe not connected at the “far end”). Used to avoid performing connection toning if there is definitely no connection. If there is no such mechanism, then this is set permanently to TRUE. (Question, should this be a bit in the port register map?)
max_port_speed	This per-port variable is the maximum speed at which a port is allowed to connect. It is a writable register in the Port register map. The speed is encoded as 1 = S100, 2 = S200, 3 = S400, 4 = S800, 5 = S1600, 6 = S3200. An attempt to write to the register with a value greater than the hardware capability of the port results in the maximum value the port is capable of being stored in the register (or is this too clever by half?). The port uses this register only when a new connection is established in Beta mode.
pmd_tone_on	When true, instructs the PMD to generate a tone (see 5.8.1). On a transition to false, instructs the PMD to cease generating a tone, remove the signalling bias voltage and set the port transmitters to high impedance.
port_speed	Negotiated operating speed of the port. Encoding as max_port_speed above
receive_ok	In DS_mode indicates the reception of a debounced TpBias signal In Beta_mode, indicates the reception of a continuous electrically valid signal. Note, is set to false during the time that only connection tones are detected in Beta mode
sd_detected	Used to latch the PMD signal_detect between signal sampling times
signal_detect	Variable set by PMD (see Clause 5.9) if a valid signal is being received.
signal_detect_OK()	Indicates that an electrically valid signal has been detected on TPA since the last call of this function. Does NOT imply reception of valid 8B10B codes or scrambler synchronization.
tpBias(n)	n = 1 instructs the PMD to generate TpBias on TPA (as defined in IEEE 1394-1995). n = 0 instructs the PMD to cease generating TpBias and set TPA to high impedance.
ACTIVE_SAMPLE_INTERVAL	10.42 us (1024/BASERATE) - allows for chatter on signal detect
BIAS_HANDSHAKE_DELAY	21.33 (2* MAX_BIAS_HANDSHAKE, 2**21/BASERATE)
DISCONNECTED_TONE_INTERVAL	42.67 ms (2**22/BASERATE)
MAX_BIAS_HANDSHAKE	10.67 ms (max value for BIAS_HANDSHAKE, 2**20/BASERATE)
NO_OF_TRIES	4 - number of tries at toning before switching to trying TpBias

**Table 11-1—Functions and variables used in Connectivity management**

Function, variable or constant	Description
RECEIVER_INIT_TIME	1ms. Time from first receipt of signal for a receiver to be operating within the BER objective of $10^{-12}$
RESUME_CHECKS	$DISCONNECTED\_TONE\_INTERVAL / (RESUME\_SAMPLING\_INTERVAL + 2 * TONE\_DURATION)$ - the number of checks that are made between connection tone intervals
RESUME_SAMPLING_INTERVAL	?? must be often enough for the S/R delay to work OK
SPEEDTONE_BIT_INTERVAL:	$TONE\_DURATION * 4$
SYNCHRONIZATION_LENGTH	32768 (maximum number of transmitted bits for scrambler synchronization + maximum number of transmitted bits for byte synchronization rounded up to a power of two)
TONE_DURATION	666.67 us ( $2 * 16 / \text{BASERATE}$ )
TONE_FREQUENCY	61.44 MHz ( $\text{BASERATE} * (10/8) * 0.5$ )

**11.4 Node state machine**

This is defined as continuously running C code, and maintains a record of whether the nodes is an isolated or boundary node. It shares variables with the port state machine. Note that in this C code, the variables are subscripted, whereas in the port state machine and C code, the same variables are unsubscripted.

**Table 11-1—Node state machine**

```

void node_status() { // Continuously monitor node status in all states
    int active_ports = 0, i, suspended_ports = 0;

    isolated_node = TRUE; // Remains TRUE if no active port(s) found
    for (i = 0; i < NPORT; i++) {
        if (active[i]) {
            active_ports++; // Necessary to deduce boundary node status
            isolated_node = FALSE; // ALL ports must be inactive at an isolated node
        } else if (connected[i] && !disabled[i])
            suspended_ports++; // Other part of boundary node definition
        boundary_node = (active_ports > 0 && suspended_ports > 0);
    }
}
    
```

## 11.5 Port State machine

The port connection state machines operate independently for each port. While a port is in the active state its arbitration, data transmission, reception and repeat behaviors are specified by the state machines in ~~XXXXX~~ [Clause 10 if operating in Beta mode, or as in IEEE-Std 1394a-1998](#). 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.

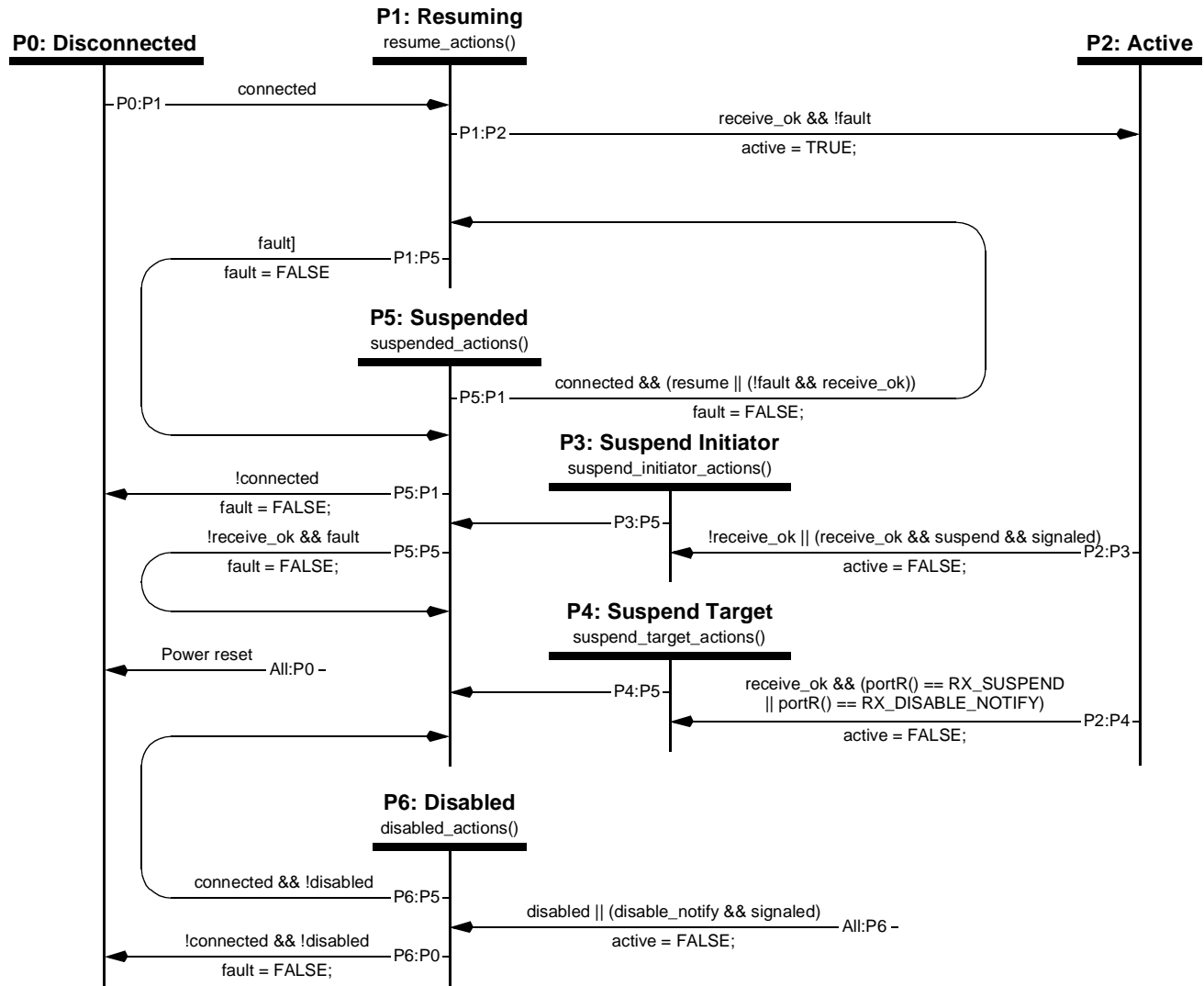


Figure 11-1 — Port connection state machine

### 11.5.1 Port connection state machine notes

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

1 **Transition All:P6.** The local link may immediately disable a port by setting the Disabled bit to one. This transition may  
2 also be caused by a remote command packet, in which case the port, if active, shall transmit TX\_DISABLE\_NOTIFY  
3 before the port is disabled.  
4

5 **State P0: Disconnected.** The generation of TpBias is disabled (DS mode) or continuous transmission is stopped (Beta  
6 mode) and the outputs are in a high-impedance state. The PHY may place most of the port's circuitry in a low-power con-  
7 sumption state. The connection detect mechanism shall be active even if other components of the PHY port are in a low-  
8 power state.  
9

10 **Transition P0:P1.** When a port's connection detect circuitry signals that its peer PHY port is physically connected, the  
11 PHY port transitions to the resuming state.  
12

13 **State P1: Resuming.** In Beta mode, the PHY commences transmission at the operating speed, and attempts to synchron-  
14 ise its receiver. The PHY port tests both the connection status and the presence of receive\_ok to determine if normal oper-  
15 ations may be resumed. If the port is connected, receive\_ok is set and there are no other active ports, the PHY waits seven  
16 RESET\_DETECT intervals before any state transitions. Otherwise, in the case of a boundary node with one or more  
17 active ports, the PHY waits three RESET\_DETECT intervals before any state transitions. A detected bus reset overrides  
18 either of these waits, otherwise, when the wait elapses the PHY initiates a bus reset.  
19

20 **Transition P1:P2.** If the PHY port is both connected and receive\_ok is set, it transitions to the active state.  
21

22 **Transition P1:P5.** A resuming PHY port that remains connected to its peer PHY port but faults during the resume hand-  
23 shake transitions to the suspended state. The fault condition is cleared so that subsequent detection of receive\_ok may  
24 cause the port to resume.  
25

26 **State P2: Active.** The PHY port is fully operational, capable of transmitting or receiving and repeating arbitration signals  
27 or clocked data. While the port remains active, the behavior of this port and the remainder of the PHY are subject to the  
28 cable arbitration states specified in XXXX.  
29

30 **Transition P2:P3.** Upon the loss of *receive\_ok* or the receipt of a PHY remote command packet that sets the *suspend*  
31 variable to one, the PHY port leaves the active state to start functioning as a suspend initiator. A loss of *receive\_ok* is usu-  
32 ally the result of a physical disconnection or the loss of power to the connected peer PHY port. If the transition is the  
33 result of a remote command packet, the PHY transmits a remote confirmation packet with the *ok* bit set to one. In the  
34 meantime, the suspend initiator has signaled TX\_SUSPEND to its connected peer PHY.  
35

36 **Transition P2:P4.** If an active port observes an RX\_DISABLE\_NOTIFY or RX\_SUSPEND signal it becomes a suspend  
37 target and leaves the active state.  
38

39 *Ed: following needs updating to reflect Beta mode operation*  
40

41 **State P3: Suspend Initiator.** A suspend initiator waits for *receive\_ok* to be zero. If Receive\_OK\_HANDSHAKE elapses  
42 and the connected peer PHY has not driven TpBias ~~low~~ low (DS mode) or ceased sending a valid signal (Beta mode), the  
43 suspend operation has faulted and the Fault bit is set to one. In either case the suspend initiator first drives TpBias low for  
44 Receive\_OK\_HANDSHAKE ~~gtime~~ time if in DS mode and then places all outputs in a high-impedance state.  
45

46 **Transition P3:P5.** Upon completion of the actions associated with this state, the PHY port unconditionally transitions to  
47 the suspended state.  
48

49 *Ed: following needs updating to reflect Beta mode operation*  
50

51 **State P4: Suspend Target.** A suspend target sets the *suspend* variable for all the other active ports, which in turn causes  
52 them to propagate the RX\_SUSPEND signal as TX\_SUSPEND. In the meantime the suspend target if in DS mode drives  
53 its TpBias outputs below 0.1 V and if in order to Beta mode ceases sending a valid signal ~~the suspend initiator that~~  
54 ~~RX\_SUSPEND~~ in order to signal the suspend initiator that RX\_SUSPEND was detected. ~~When~~ If in DS mode, either the  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66

node waits a Receive OK HANDSHAKE time to allow the connected peer PHY ~~drives time to drive~~ TpBias low or a  
~~Receive\_OK\_HANDSHAKE time-out expires (whichever occurs first),~~ low, and then the suspend target disables the gener-  
 ation of ~~TpBias and TpBias. It then~~ places the outputs in a high-impedance state.

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

**State P5: Suspended.** The PHY may place most of the port's 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.** Either of two conditions cause a suspended PHY port to transition to the resuming state: a) a nonzero value for the port's *resume* variable or b) the detection of *receive\_ok* if the port's Fault bit is zero. A port's *resume* variable may be set indirectly as the result of the resumption of other PHY ports.

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

**State P6: Disable.** While disabled, the PHY may place most of the port's 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.

## 11.5.2 Port connection actions and conditions

Table 11-1 — Port connection actions and conditions (Sheet ? of ?)

```

31 int connect_timer, bias_timer;           // global timers
32 int isolated_node, boundary_node;       // global node status variables
33 int connect_detectport_speed[NPORT]; // // operating speed of the port, shared
34 with PMDBeta Mode port
35 int biasconnect_detect[NPORT];        // shared with PMD
36 int connect_detect_validbias[NPORT]; // flag to track whether connect_detect
37 can be used at a given timeshared with PMD
38 boolean pm_d_tone_on[NPORT]; // shared with PMD
39 int connect_detect_valid[NPORT]; // flag to track whether connect_detect
40 // can be used at a given time
41 int active[NPORT]; connected[NPORT]; disabled[NPORT];
42
43
44 int active[NPORT]; connected[NPORT]; disabled[NPORT];
45 // the following replaces the P1394a definition
46 enum speedCode {Undefined, S100, S200, S400, S800, S1600, S3200}; // replaces P1394a definition
47 replaces P1394a definition
48 // speed codes are encoded 001 010 011 100 101 110
49
50
51 // is the following still needed? Only used when active.
52 // The following bias filtering code is optional for P1394b connectivity management.
53 // If it is not used, then bias[i] shall reflect the value of bias_detect[i] at all times.
54 // Note that bias filtering is mandatory for operation in P1394a (DS) mode.
55 // Note also that bias will normally be reported as true when the port is
56 // operating in Beta mode, as bias_detect will detect the locally generated bias
57
58
59 void bias_status() { // Continuously running bias update code
60     timer bias_timer; // Timer for bias filter
61     boolean filter_bias[NPORT]; // TRUE when applying hysteresis to bias_detect
62     crivuitcircuit
63     for (i = 0; i < NPORT; i++) {
64         if (bias_filterbias_detect[i] == FALSE) {
65

```

```

1 if (bias_detect[i] == bias[i]) // Has bias detect changed since timer started?
2     bias_timer = 0; // If so, restart the filtering timer
3         bias_filter[i] = FALSE;
4         bias[i] = FALSE; // Report immediately
5     } else if (bias_filter[i]) { // Filtering positive bias transition?
6     else if (bias_timer >= BIAS_FILTER_TIME+) {
7         bias_filter[i] = FALSE; // Done filtering
8         bias[i] = bias_detect[i]TRUE; // Confirm new value in PHY register bit
9     }
10     }
11     }
12     } else if (bias_detect[i] != bias[i]) { // Detected bias differs from reported
13 bias?
14         bias_filter[i] = TRUE; // Yes, start a filtering period
15         bias_timer = 0;
16     }
17     }
18     }
19 }
20 }
21
22 boolean suspend_in_progress() { // TRUE if any port suspending
23     int i;
24     for (i = 0; i < NPORT; i++;)
25         if (suspend[i])
26             return(TRUE);
27     return(FALSE);
28 }
29
30 boolean resume_in_progress() { // TRUE if any port resuming
31     int i;
32     for (i = 0; i < NPORT; i++;)
33         if (resume[i])
34             return(TRUE);
35     return(FALSE);
36 }
37
38 void resume_all_ports() {
39     for (j = 0; j++; j < NPORT)
40         if (!active[j] && !disabled[j] && connected[j])
41             resume[j] = TRUE; // Resume all other suspended ports
42 }
43
44 // Per port code
45 // The code below runs per port
46 // unsubscripted references to connect_detect, etc are to be interpreted
47 // as references to connect_detect[i], where i is the number of the particular port
48
49 int received_speed, speed_ack; // speed_ack says that the other end received OUR speed
50 void signal_detect_monitor() {boolean listening_for_speed; // continuously running latches
51 signal_detect turns on/off the receive_speed_indication routine
52 boolean tried_bias; // indicates that the local port has tried sending TpBias
53 // just in case the far end port was suspended P1394a port
54 // initialised to false on POR, which is the only time this can occur
55 int bias_delay_timer; // used for timing out Bias delays
56 boolean sd_detected; // latches the PMD's signal_detect, initialized to FALSE
57
58 void activate_connect_detect(int delay) {
59     if (Beta_mode) {
60         bport_active = FALSE;
61     }
62 }
63
64
65
66

```

```
2 ?? any need to wait for anything to happen ???
3
4
5
6   _} else {
7       tpBias(0); // Drive TpBias low
8       if (delay != 0) {
9           connect_timer = 0;
10          while (connect_timer < delay) // Enforce minimum hold time for TpBias low
11              ;
12          while (!connect_detect) // Wait for connect_detect circuit to stabilize
13              ;
14          tpBias(Z); // Release TpBias
15      }
16      connect_detect_valid = TRUE; // Signal OK to connection_status()
17  }
18
19 void send_tone() {
20     pmd_tone_on = TRUE;
21     wait (TONE_DURATION);
22     pmd_tone_on = FALSE;
23 }
24
25
26 void signal_detect_monitor() { // continuously running - latches
27     signal_detect
28
29     int sd_detected; // initialized to FALSE
30     if (!sd_detected) sd_detected = signal_detect;
31 }
32
33 boolean signal_detect_OK() { // true if signal_detect set since we last
34     looked
35     int boolean x;
36     wait(ACTIVE_SAMPLE_INTERVAL); // filter for chatter on signal detect
37     x = sd_detected;
38     wait(TONE_DURATION); // to make sure that we don't see the same bit twice
39     waitif (TONE_DURATION)x) sd_detected = FALSE; // to make sure that now we don't see the
40 same can start looking for the next bit twice
41     if (x) sd_detected = FALSE; // now we can start looking for the next bit if we saw
42 a bit last time, but take care not to
43 // but take care not to zero out a bit which started
44 just after we looked
45     return(x);
46 }
47
48
49
50 void receive_speed_indication() { // continuously running
51
52     int count;
53 void send_speed(speed, ack) {
54     if send_tone(listening_for_speed) {;
55
56     received_speed = speed; ack = 0;
57
58     // first find the gap
59     count = 4 wait(SPEEDTONE_BIT_INTERVAL);
60
61
62
63
64
65
66
```



```

1  . . . . . wait(SPEEDTONE_BIT_INTERVAL);
2
3  . . . . . if ((speed & (1 << i)) != 0) send_tone() else wait(TONE_DURATION);
4
5  }
6
7  . . . . . wait(DISCONNECTED_TONE_INTERVAL - 4 * (SPEEDTONE_BIT_INTERVAL + TONE_DURATION));
8
9  }
10
11 void set_beta() { . . . . . // set beta mode and exchange speed signals to establish
12 operating speed
13     int speed_agreed, we_agree, count;
14
15
16 // exchange speed signals and set the variable port_speed
17     port_speed = max_port_speed; . . . . . // our starting point
18     count = 8; . . . . . // four tries
19     speed_agreed = we_agree = FALSE;
20     while ((count > 0) & !speed_agreed) {
21         listening_for_speed = TRUE; // set the autonomous speed listner going;
22         send_speed(port_speed, we_agree); . . . . . // ack if we think the speed is
23 agreed
24         if (received_speed != 0) {
25             if (received_speed += port_speed) {
26                 if (received_speed == we_agree = port_speed) {TRUE;
27                 we_agree = speed_agreed = TRUE; speed_ack; // all agreed when we have the
28 acknowledge
29                 speed_agreed = speed_ack; // all agreed when we have the
30 acknowledge from the other end
31             } else {
32                 if (received_speed < port_speed) {
33                     . . . . . // if the received speed is unacceptable, then we
34 could add code here to give up altogether
35 // then we could add code here to give up altogether
36                 port_speed = received_speed;
37                 count = 8; . . . . . // and try again with a lower speed
38             } else count++; // received speed > port_speed, so allow another go,
39 void set_DS() { // set DS mode and start up Tp Bias but not too many
40 times
41         }
42         count = count - 2;
43     }
44     if (speed_agreed) {
45         disconnected = Beta_mode = FALSE; TRUE;
46         Beta_mode = connected = TRUE;
47         connected = receive_OK = TRUE;
48         receive_OK = TRUE;
49     }
50 }
51
52 void set_DS() { . . . . . // set DS mode and start up Tp Bias (implied by Beta mode
53 remaining false)
54
55     disconnected = FALSE;
56     disconnected = connect_detect_valid = FALSE;
57     connect_detect_valid = connected = FALSE; TRUE;
58     TpBias(1); receive_OK = TRUE;
59     receive_OK = TRUE;
60 }
61
62 void connection_status() { . . . . . // continuously running code for each port

```

```

1   if (~local_plug_present) { // give up
2       receive_ok = FALSE;
3       Beta_mode = FALSE;
4       connected = FALSE;
5       return; // i.e. to start the routine again;
6   }
7   if (disabled) { // give up if Beta mode
8       if (Beta_mode || !connected) return;
9       if (Beta_mode || !connected) return; // look at connection circuit only if still
10  connected in DS mode
11  }
12  if (!connected && local_plug_present) {
13  if (disconnected && local_plug_present) { // look for new
14  connection and determine operating mode
15  int new_connection_detected;
16  new_connection_detected = FALSE;
17  for (i = 0; i < NO_OF_TRIES; i++) {
18  if (bias) break; // don't try to send tones if detect TpBias
19  if (connect_detect && ~new_connection_detected) { // try toning for at least
20  new_connection_detected = TRUE; //
21  try toning for at least two more times on a new connection
22  i = NO_OF_TRIES - 2;
23  i = NO_OF_TRIES - 2;
24  }
25  if signal_detect_OK() {
26  set_beta();
27  return; // to next time round in this routine
28  }
29  send_tone();
30  wait (DISCONNECTED_TONE_INTERVAL);
31  }
32
33  // here if-if (1) tried toning 4 times without detecting a tone;
34  // (2) detected a connection and tried toning twice without detecting
35  a tone;
36  // (3) detected incoming TpBias
37  if (~connect_detect) return; // to next time round in this routine
38  if (bias) { // still seeing Bias
39  bias_delay_timer = 0;
40  waitwhile ((MAX_BIAS_HANDSHAKE+10% || bias_delay_timer < BIAS_HANDSHAKE_DELAY)
41  && bias);
42  // twice as long as a P1394b port will generate it
43  if (bias) { // incoming TpBias persists - set DS mode
44  set_DS();
45  set_DS()return;
46  return;
47  }
48  // Bias not present, try toning again
49  send_tone();
50  wait(DISCONNECTED_TONE_INTERVAL-TONE_DURATION);
51  if signal_detect_OK {
52  set_beta();
53  set_beta()return;
54  return;
55  }
56
57
58
59
60
61
62
63
64
65
66

```

```

1         else-if (!tried_bias) { // try sending BiasBias just once, in case we have powered
2 up whilst the connected
3         tried_bias = TRUE; // whilst the connected P1394a PHY was in suspend
4         connect_detect_valid = FALSE;
5         TpBias(1);
6         wait(MAX_BIAS_HANDSHAKE, bias) bias_delay_timer = 0;
7         if-while (bias_delay_timer < MAX_BIAS_HANDSHAKE, !bias) {
8             set_DS();
9             if (bias) {
10                set_DS();
11                return;
12            }else}
13            activate_connect_detect(0); // includes taking TpBias away
14            return;
15        }
16    } // end of actions whilst disconnected, disconnected or connecting
17
18    // here if connected
19    if (active) {
20        if (Beta_mode) {
21            wait(ACTIVE_SAMPLE_INTERVAL); // filtering for chatter on
22 SD???
23            receive_OK = signal_detect_OK();
24        } else receive_OK = bias; //DS mode
25        return;
26    }
27    if (resume_in_progress) return;
28
29    // here if suspended, or (DS mode and disabled) - look for disconnect or resume
30    if (Beta_mode) {
31        int new_connection_detected know_still_connected = FALSE;
32        new_connection_detected = FALSE send_tone();
33        for (i = 0; i < NO_OF_TRIES RESUME_CHECKS; i++) {
34            if (signal_detects signal_detect_OK()) {
35                if signal_detect_OK() know_still_connected = TRUE;
36                set_beta wait(TONE_DURATION);
37                returns signal_detect_OK(); // to next time round in this routine flush out
38 any residual value from the first detection
39                wait(TONE_DURATION);
40                if (signal_detects signal_detect_OK()) {
41                    receive_OK = TRUE; // to start resuming
42                    return;
43                }
44                wait (RESUME_SAMPLING_INTERVAL-TONE_DURATION);
45                wait (RESUME_SAMPLING_INTERVAL-TONE_DURATION);
46            }
47            connected = know_still_connected;
48        }
49    }
50    } else { // DS mode, suspended or
51 disabled
52        if (!disabled) {
53            receive_OK = bias;
54            if (receive_OK) return;
55        }
56        connected = connect_detect; // see if still connected
57    }
58    if (!connected) { // disconnection detected
59        Beta_mode = FALSE;
60        if (int_enable && !port_event) {
61            port_event = TRUE;
62            if (link_active && LPS)
63                PH_EVENT.indication(INTERRUPT);
64            else

```

```

1         PH_EVENT.indication(LINK_ON);
2     }
3 }
4 }
5
6
7 void disabled_actions {
8     if (int_enable && !port_event) {
9         port_event = TRUE;
10        if (link_active && LPS)
11            PH_EVENT.indication(INTERRUPT);
12        else
13            PH_EVENT.indication(LINK_ON);
14    }
15    disable_notify = signaled = FALSE;
16    disabled = TRUE;
17    activate_connect_detect(i, 0); // Enable the connect detect circuit
18 }
19
20
21 void resume_actions() {
22     while (suspend_in_progress()) // Let any other suspensions complete
23         ; // (we may resume those ports later)
24     connect_timer = 0;
25     if ((int_enable || resume_int) && !port_event) {
26         port_event = TRUE;
27         if (link_active && LPS)
28             PH_EVENT.indication(INTERRUPT);
29         else
30             PH_EVENT.indication(LINK_ON);
31     }
32     connect_detect_valid = FALSE; // Bias renders connect detect circuit useless, stop-
33 toning
34     if (resume == 0 && !boundary_node) resume_all_ports()
35     else resume = TRUE; // Guarantee resume_in_progress() returns TRUE
36     if (Beta_mode) { // start up and train the port
37         bport_active = TRUE;
38         while ((connect_timer < (RECEIVER_INIT_TIME + (SYNCHRONIZATION_LENGTH/
39         (BASE_RATE*2**(port_speed-1))) && !bport_sync_ok)
40         ; // wait until the port tells ut it is synchronized
41         fault = !bport_sync_ok; // Resume attempt failed if failed to synchronize
42         wait_eventif (bport_sync_ok); fault) {
43         activate_connect_detect(0);
44         resume = FALSE;
45         return; // Resume attempt complete
46         port_synchronized = TRUE;
47     } else {
48         tpBias(1); // Generate TpBias
49     }
50
51
52
53 if (resume == 0 && !boundary_node) resume_all_ports()
54 else
55     while (((connect_timer < Receive_OK_HANDSHAKE) && !receive_ok) || bus_initialize_active)
56 while (((connect_timer < Receive_OK_HANDSHAKE) && !receive_ok) || bus_initialize_active)
57 ; // Wait for peer PHY to generate TpReceive_ok
58 if (receive_ok) { // Connection restored to active state?
59     while ((connect_timer < 3 * RESET_DETECT) && !bus_initialize_active)
60         ;
61     if (!bus_initialize_active) { // No other node initiated reset?
62
63
64
65
66

```

```
1  if (boundary_node) // Can we arbitrate?
2  if (boundary_node) isbr = TRUE; // Can we arbitrate? Yes, don't
3  wait any longer
4  else {
5      while ((connect_timer < 7 * RESET_DETECT) && !bus_initialize_active)
6          ; // Let's wait a little longer...
7
8  if (!bus_initialize_active)
9  if (!bus_initialize_active) ibr = TRUE; // Sigh! We'll have to
10 use long reset
11     }
12 }
13 }
14 fault = ~receive_ok; // Resume attempt failed if TpReceive_ok is absent
15 if (fault) // If so, restore usefulness of connect detect circuit
16     activate_connect_detect(i, 0);
17 resume = FALSE; // Resume attempt complete
18 }
19 }
20
21 void suspend_initiator_actions() {
22     connect_timer = 0; // Used to debounce receive_ok or for receive_ok handshake
23     if (!suspend) { // Unexpected loss of receive_ok?
24         suspend = TRUE; // Insure suspend_in_progress() returns TRUE
25         if (child) // Yes, parent still connected?
26             isbr = TRUE; // Arbitrate for short reset
27     }
28     else
29         ibr = TRUE; // Transition to R0 for reset
30     while (connect_timer < CONNECT_TIMEOUT)
31         ; // Time for receive_ok to stabilize
32 }
33 while ((connect_timer < Receive_OK_HANDSHAKE) && receive_ok)
34     ; // Wait for suspend target to deassert receive_ok
35 fault = receive_ok; // Suspend handshake refused by target?
36 activate_connect_detect(i, Receive_OK_HANDSHAKE); // Also guarantees handshake timing
37 activate_connect_detect(Receive_OK_HANDSHAKE); // Also guarantees handshake timing
38 }
39 }
40
41 void suspend_target_actions() {
42     int j;
43     if (resume_in_progress()) // Other ports resuming?
44         resume = TRUE; // OK, do suspend handshake but resume afterwards
45     suspend = TRUE; // Insure suspend_in_progress() returns TRUE
46     if (portR() == RX_DISABLE_NOTIFY) // Is our peer PHY going away?
47         breq = IMMED_REQ; // Topology change! Reset on other (active) ports
48 }
49 }
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
```

## 11.6 Rationale (informative)

### 11.6.1 Connection detection

When the port is disconnected, the procedure described in this section aims to detect a reconnection, then to determine the appropriate mode of operation (DS mode or Beta mode) and, if Beta mode, the appropriate operating speed. The same underlying mechanisms are used during suspend to determine that a port has become disconnected. In both cases, the emphasis has been to provide a very low power mechanism which meets all the appropriate constraints. A simplified algorithm applies if the port is Beta only capable.

P1394b uses connect\_detect\_valid to drive a toning scheme. A tone is transmitted on TPB. A signal\_detect circuit listens on TPA. (The output on TPA is set to high impedance). The ~~signal\_detect~~ signal\_detect\_OK function returns TRUE if an electrically valid signal has been detected on TPA since the last call of the function. The signal\_detect circuit, and therefore the signal\_detect\_OK function does not provide any guarantee of quality of signal, and does not imply scrambler synchronization or the reception of valid 8B10B codes.

### 11.6.2 Tone, Connect detect and Bias properties

The design of the algorithm which is used during disconnection and during suspend in order to detect a change in port status is based on the following fundamental points, given with the reasoning on each.

#1. Can't tone and drive TpBias simultaneously. Proof: P1394a and earlier PHY's would attempt to interpret the tone if TpBias were present. Only safe way to tone into a p1394a mode is when TpBias is deasserted.

#2. Can't wait for remote port on peer PHY to initiate TpBias. Proof: A P1394a node w/ a suspended port won't initiate TpBias as long as the DC connection status is continually active. Consider a P1394a PHY and a P1394b PHY which are connected via a suspended link. Subsequently, if the P1394b experiences a power-on reset (device powered off for a while and then turned back on), it needs to re-establish connectivity with the P1394a node. If the eager beta algorithm waited for the P1394a device to initiate TpBias ... it would never come. Consequently, at least one step in eager Beta must lead to the initiation of a TpBias handshake.

#3. Can't assume initial receipt of TpBias means a non-P1394b device is attached, ~~and must listen for tone even when driving TpBias~~. Proof: If a P1394b port (called X) tries toning first and hears nothing, then according to #2 above, it will have to initiate TpBias. If a connected P1394b port (called Y) on a peer PHY then powers up, it will detect TpBias without a tone. Thus, assuming the presence of TpBias always indicates a P1394a port leads to a false detection of the remote peer. To avoid this, port Y should tone first even if it sees ~~TpBias (courtesy of Mr. LaFollette)~~ TpBias. Port X, which is still driving TpBias, must turn TpBias off first (see #5) and then still listen for a tone. After detecting the tone from Port Y, Port X will revert to toning and the beta mode handshake can commence.

#4. During upstarting, must initiate toning with some frequency. Proof: Assume an AC path between two P1394b ports and allow that the ac path can have passive connection points (RJ-45 jack on the wall or in the patch panel, a optical wall socket, etc.). If the passive connection is not in place, the two P1394b nodes at power-on reset will attempt to initiate a handshake. Both progress to the TpBias assertion phase mandated in #2 above without ever detecting the remote port. If the eager Beta scheme stopped at this point with the continuous assertion of TpBias, a later connection at the passive point won't be detected by either node. For this reason, the eager Beta procedure must attempt toning with some frequency to allow for passive connections. (A plug present feature at the PHY doesn't handle the case of passive connection points in-between the attached ports.) This suggests that the eager Beta sequence would need to alternate between toning and generating TpBias. Alternatively, we might be okay with a single phase of TpBias and then revert to continuously toning.

#5. Can't listen for a tone when generating TpBias. Proof: When generating TpBias, the remote node may be a 1394-1995 node (or indeed a P1394a node). Such a node will interpret the TpBias signal as a connection, and will start sending arbitration signals (such as reset). These will trip the "signal-detect" mechanism, which operates by looking for a differential signal on TPA.

1 Corollary to Axiom 5: A Beta-PHY doesn't need to send a tone when receiving TpBias because the Beta-PHY on the  
2 other end of the cable that's generating TpBias can't listen for the tone anyway.

3  
4 #6. Can't listen for TpBias whilst generating a tone. Proof: When generating a tone, the local PHY will be providing an  
5 internally generated bias level on TPB around which to send the signal. This will, in general, will be detected by the local  
6 TpBias detector. There is no filter on the TpBias detector on sensing loss of TpBias, and it is assumed that the bias bit  
7 will report zero as soon as the transmission of the tone ceases.

8  
9  
10 #7. DC low current connect detect mechanism may give false negative. Proof: An incoming tone on a DC coupled con-  
11 nection will be centered around a common mode bias which may raise the voltage higher than the connect detect thresh-  
12 old (low voltage implies connection).

13  
14  
15 P1394b uses connect\_detect\_valid to drive a toning scheme. A tone is transmitted on TPB. A signal\_detect circuit listens  
16 on TPA. TPA is set to high impedance. The ~~signal\_detects~~signal\_detect\_OK() function returns TRUE if an electrically  
17 valid signal has been detected on TPA since the last call of the function. The ~~signal\_detect function~~signal\_detect, and  
18 hence the signal\_detect-OK() function, does not provide any guarantee of quality of signal, and does not imply  
19 scrambler synchronization or the reception of valid 8B10B codes.

20  
21  
22 A 1394-1995 node at the far end will assert TpBias on its TPA, but will not see TpBias on its TPB, and so will not think  
23 it is connected. A P1394a node on the other end will sense con\_status and start its debounce timeouts.

### 24 25 26 **11.6.3 Connection detection and mode determination algorithm**

27  
28 Based on the above, the algorithm used (provided that plug\_present is true) following power on reset or at any time we  
29 are trying to determine connection status is an "eager Beta mechanism", and operates as follows:-

30  
31 1) Begin toning. If tone is heard, then beta mode is possible. If TpBias is detected, then go straight to 2, otherwise stay  
32 in this state until 10 (???) tones have been generated. If a change from disconnected to connected in connect\_detect is  
33 detected, then tone for two more times, then go to 2.

34  
35 2) Check for "connect\_detect". If not set, then go back to the start of 1. Otherwise, check for TpBias. If TpBias is  
36 detected, wait a period of time (~~BIAS\_HANDSHAKE~~ BIAS\_HANDSHAKE\_DELAY, perhaps, ~~BIAS\_HANDSHAKE~~  
37 +"A little More" twice the maximum for BIAS\_HANDSHAKE) and then sample Bias once again. Wait for it to go away  
38 so that toning can be tried again.

39  
40  
41 If Bias is still present, then the peer port is a legacy PHY, so assert TpBias and DS mode is established.

42  
43 If Bias is NOT present, generate the startup tone. If the attached port is an A-PHY or a 1394-1995 PHY, no response will  
44 occur, otherwise, an attached B-PHY(1) will respond with the startup tone and Beta-Mode will be established.

45  
46 If no response is received as a result of generating the startup tone, asserts TpBias for BIAS\_HANDSHAKE. If Bias is  
47 then detected, DS mode is established, otherwise, when no response, B-PHY(0) resumes alternating between toning and  
48 asserting TpBias.

49  
50 A 1394-1995 port at the far end will assert TpBias on its TPA, but will not see TpBias on its TPB, and so will not think  
51 it is connected. A P1394a node on the other end will sense con\_status and start its debounce timeouts.

52  
53  
54 If an P1394a port is the connection at the other end, and, if just by chance, the local P1394b port does not respond with  
55 TpBias before the P1394a port "times-out" there really is no issue because the P1394a port will enter into a resume-fault  
56 condition, but, because the P1394b port did not receive a tone in reply to its last tone attempt, it will generate TpBias  
57 again causing the P1394a port to "wake" and become a resume target.

58  
59 In the instance that the far port is a 1995 port there will not ever be an issue because it will always be generating TpBias  
60 and when the P1394b port finally generates TpBias it (the B-PHY) will eventually generate a bus-reset because of the  
61 "resume" process if a bus reset is not detected from somewhere else.

The set of possible port connections that a bilingual port may be connected to are:-

**Figure 11-1—Connect Status value in various connection scenarios**

	Connect_status	tone exchange	action
No connection	No	fails	
DC connection to P1394a	Yes	fails	set DS
DC connection to bilingual	Yes	succeeds	set Beta
DC connection to Beta only	probably	succeeds	set Beta
AC connection to bilingual	No	succeeds	set Beta
AC connection to Beta only	No	succeeds	set Beta
DC connection to optical transceiver	No - must be biased so as not to trigger the Con or Bias detector	fails	
AC connection to optical transceiver	No	fails	

In the case that the answer is yes, the connected flag is set to true.

Note that the tone transmission and detection continues through the debounce period. If a tone is detected during this time, then Beta mode is set.

### 11.6.4 Beta mode speed negotiation

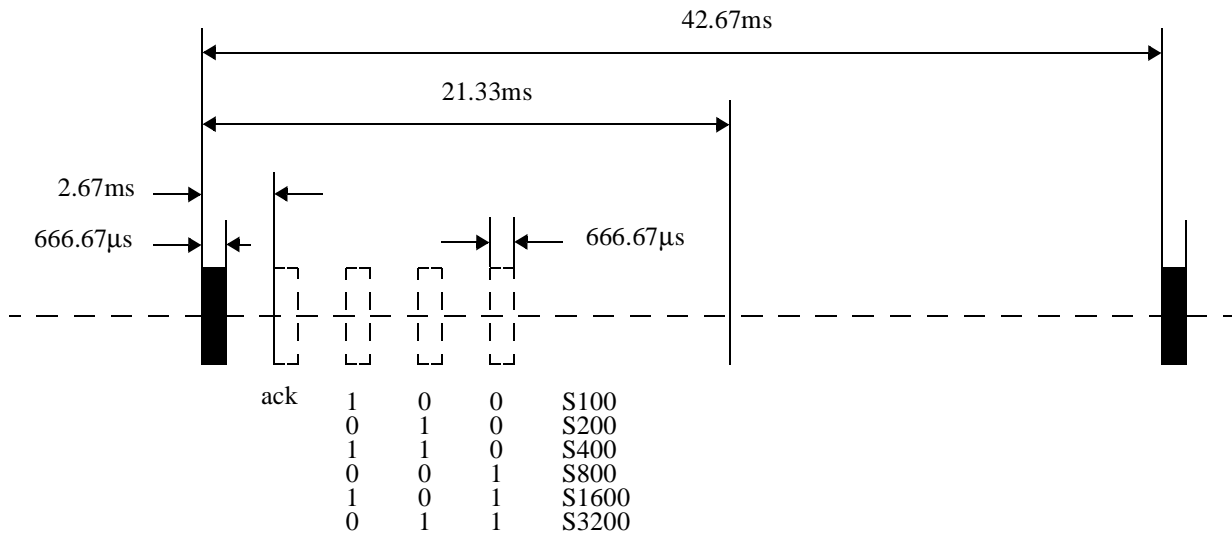
Once a Beta mode connection has been determined, a speed negotiating procedure is used. A single protocol is used to negotiate the speed anywhere in the range S100 to S3200.

Upon connection, the only property of the connecting medium that can be relied upon is that it can support the exchange of ~~125MHz-61.44MHz~~ tones provided they last at least ~~500-667~~ microseconds. This is true until continuous operation is established, (particularly, for example, because of the start-up latencies of the optical components). The aim is to establish continuous operation at the correct operating speed, and to avoid any need for matching configuration options at "both" ends to determine what a "common denominator" operating speed should be.

These aims are achieved by using the same tone as is used for connection detection, differentiating where necessary between the occasional tone to identify connect/disconnect events, a pattern of tones to negotiate the operating speed, and a continuous tone (in suspend) to indicate that it is time to resume. For this latter, we do indeed know the operating speed, but there seems to be no good reason for making this indication any different from the tone used for connect detect apart from its duration (remember, we only want a simple signal detect circuit to be alive during suspend).

Both ports transmit their respective maximum speeds, whilst simultaneously listening for the speed tones from the peer port. The ports then transmit the minimum of the transmitted and received speeds, together with an acknowledge bit (to indicate that the agreed speed has been received). When a port receives the speed tone with the acknowledge bit set, then it ceases to transmit speed tones, and the connection is established. The speed tone timing and encoding is shown in figure 11-2.

**Figure 11-2—Speed tone timing diagram**



1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66