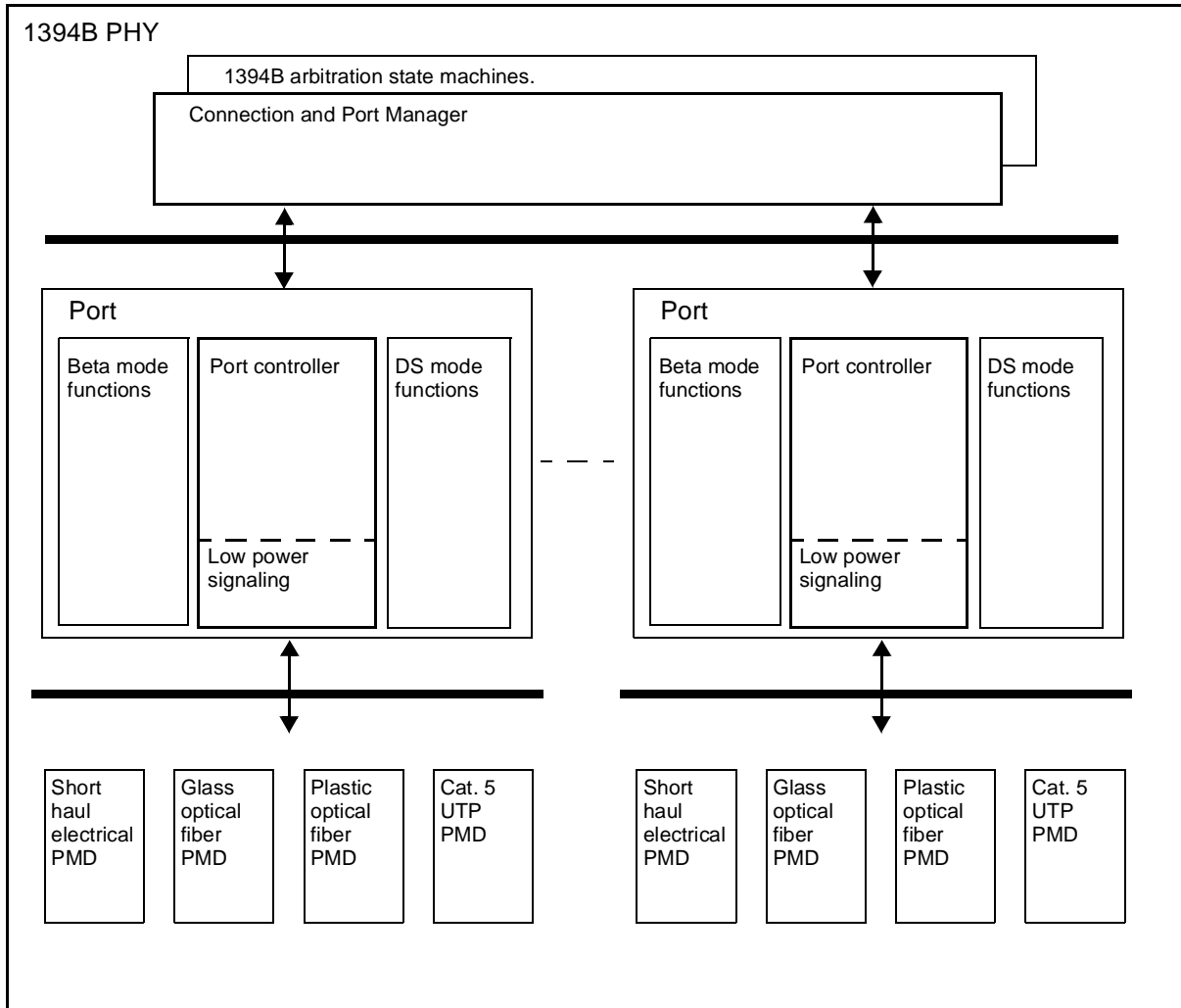


11. Connectivity Management

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 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.
Beta_mode_only	Implementation dependent constant - indicates that a port is not capable of DS mode operation and identifies the code which need not be implemented in such ports.
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)
dc_connected	TRUE if the port has detected a DC connection to the peer port
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.
peer_port_incompatibility	PHY register bit, set true if the Beta mode speed negotiation fails and the port is not DC coupled to its peer. Reset by software after appropriate higher level action has been taken.
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 drive the common mode voltage on TPA to VG. n = Z 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

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

Function, variable or constant	Description
BIAS_HANDSHAKE_DELAY	21.33 (2* MAX_BIAS_HANDSHAKE, 2**21/BASERATE)
DISCONNECTED_TONE_INTERVAL	TONE_DURATION * 4 * TONE_INTERVAL (42.67 ms, or 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
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	1.333ms (2**17/BASERATE) (must be often enough for the S/R delay to work OK)
SPEEDTONE_BIT_INTERVAL:	TONE_DURATION * 3 (the time between the end of one tone position and the start of the next)
SYNCHRONIZATION_LENGTH	16384 (maximum number of transmitted bits for scrambler synchronization + byte synchronization rounded up to a power of two)
TONE_DURATION	666.67 us (2**16/BASERATE)
TONE_FREQUENCY	61.44 MHz (BASERATE*(10/8)*0.5)
TONE_GAP	TONE_INTERVAL / 2 (number of tone units in the gap before the start bit)
TONE_INTERVAL	16 (number of tone units in a tone interval)

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 (by read-only access) the variables `active[NPORT]`, `connected[NPORT]` and `disabled[NPORT]` 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 11-2—Node state machine

```

int isolated_node, boundary_node; // global node status variables
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 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.

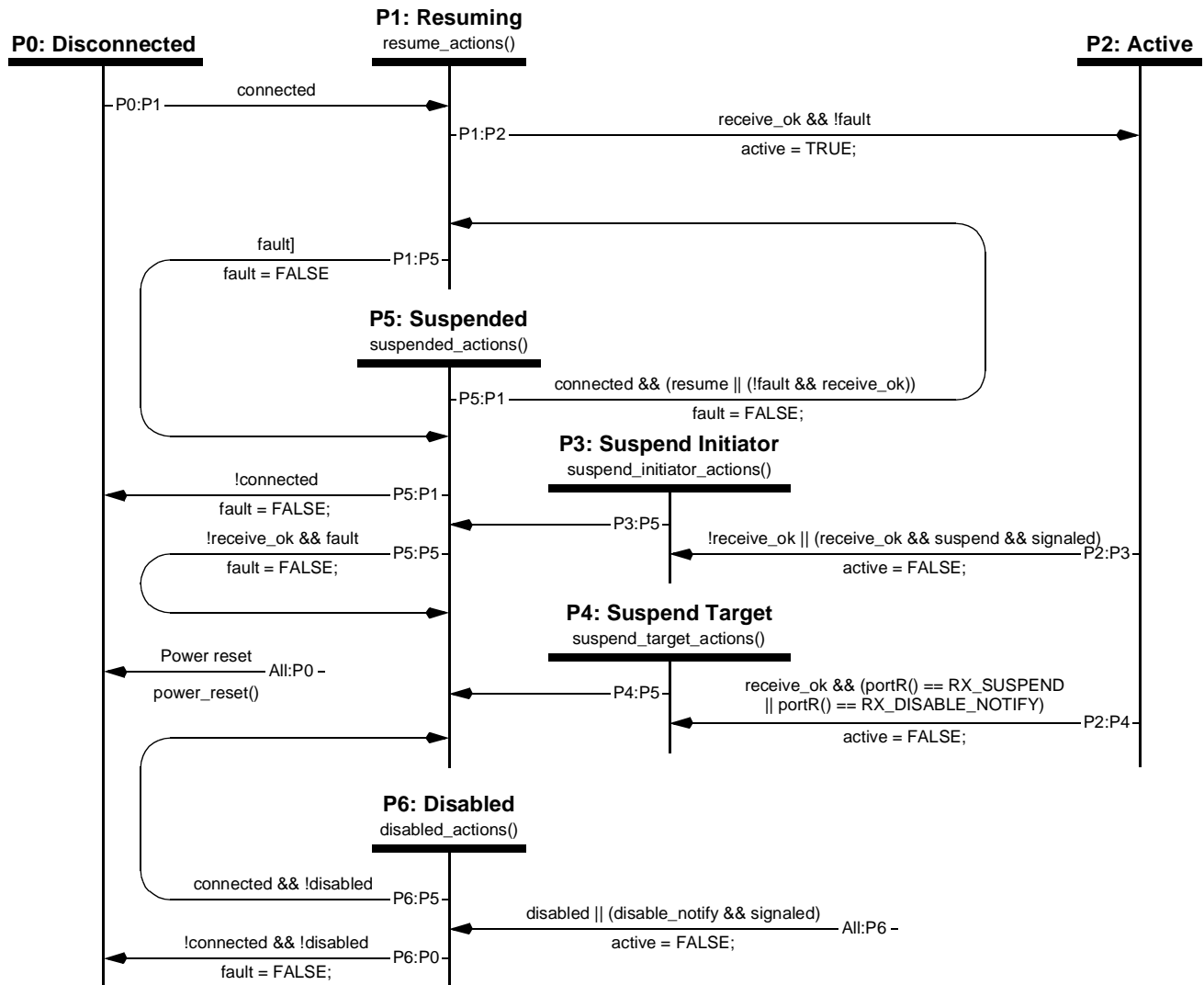


Figure 11-1 — Port connection state machine

11.5.1 Port connection state machine notes

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

Transition All:P6. The local link may immediately disable a port by setting the Disabled bit to one. This transition may also be caused by a remote command packet, in which case the port, if active, shall transmit TX_DISABLE_NOTIFY before the port is disabled.

1 **State P0: Disconnected.** The generation of TpBias is disabled (DS mode) or continuous transmission is stopped (Beta
2 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-
3 sumption state. The connection detect mechanism shall be active even if other components of the PHY port are in a low-
4 power state.
5

6
7 **Transition P0:P1.** When a port's connection detect circuitry signals that its peer PHY port is physically connected, the
8 PHY port transitions to the resuming state.
9

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

18
19 **Transition P1:P2.** If the PHY port is both connected and receive_ok is set, it transitions to the active state.
20

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

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
31 **Transition P2:P3.** Upon the loss of *receive_ok* or the receipt of a PHY remote command packet that sets the *suspend*
32 variable to one, the PHY port leaves the active state to start functioning as a suspend initiator. A loss of *receive_ok* is usu-
33 ally the result of a physical disconnection or the loss of power to the connected peer PHY port. If the transition is the
34 result of a remote command packet, the PHY transmits a remote confirmation packet with the *ok* bit set to one. In the
35 meantime, the suspend initiator has signaled TX_SUSPEND to its connected peer PHY.
36

37 **Transition P2:P4.** If an active port observes an RX_DISABLE_NOTIFY or RX_SUSPEND signal it becomes a suspend
38 target and leaves the active state.
39

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 (DS mode) or ceased sending a valid signal (Beta mode), the sus-
43 pend 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 time if in DS mode and then places all outputs in a high-impedance state.
45
46

47
48 **Transition P3:P5.** Upon completion of the actions associated with this state, the PHY port unconditionally transitions to
49 the suspended state.
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 Beta mode ceases sending a valid signal in order to signal the suspend initiator
54 that RX_SUSPEND was detected. If in DS mode, the node waits a Receive_OK_HANDSHAKE time to allow the con-
55 nected peer PHY time to drive TpBias low, and then the suspend target disables the generation of TpBias. It then places
56 the outputs in a high-impedance state.
57
58

59
60 **Transition P4:P5.** Upon completion of the actions associated with this state, the PHY port unconditionally transitions to
61 the suspended state.
62

63 **State P5: Suspended.** The PHY may place most of the port's circuitry in a low-power consumption state. The connection
64 detect circuit. shall be active even if other components of the PHY port are in a low-power state.
65
66

1 **Transition P5:P0.** A suspended PHY port that loses its physical connection to its peer PHY port transitions to the disconnected state.
2
3

4 **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.
5
6
7
8

9 **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.
10
11
12

13 **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.
14
15

16 **Transition P6:P0.** If the Disabled bit is zero and the PHY port is not physically connected to its peer PHY port, it transitions to the disconnected state.
17
18
19

20 **Transition P6:P5.** Otherwise, if the Disabled bit is zero and the PHY port is connected it transitions to the suspended state.
21
22
23
24
25

26 11.5.2 Port connection actions and conditions

27
28 **Table 11-3 — Port connection actions and conditions (Sheet ? of ?)**
29

```
30 int connect_timer, bias_timer; // global timers  
31 int port_speed[NPORT]; // operating speed of the port, shared with Beta Mode port  
31 int connect_detect[NPORT]; // shared with PMD  
32 int bias[NPORT]; // shared with PMD  
33 boolean pmd_tone_on[NPORT]; // shared with PMD  
34 int connect_detect_valid[NPORT]; // flag to track whether connect_detect  
35 // can be used at a given time  
36 int active[NPORT]; connected[NPORT]; disabled[NPORT];  
37  
38 // the following replaces the 1394a definition  
39 enum speedCode {Undefined, S100, S200, S400, S800, S1600, S3200};  
40 // speed codes are encoded 001 010 011 100 101 110  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66
```

```

1 // The following bias filtering code is optional for P1394b connectivity management.
2 // If it is not used, then bias[i] shall reflect the value of bias_detect[i] at all times.
3 // Note that bias filtering is mandatory for operation in 1394a (DS) mode.
4 // Note also that bias will normally be reported as TRUE when the port is
5 // operating in Beta mode, as bias_detect will detect the locally generated bias
6 // If the port is a Beta mode only port and bias_detect is not implemented, then
7 // bias shall be TRUE
8
9 void bias_status() { // Continuously running bias update code
10     timer bias_timer; // Timer for bias filter
11     boolean filter_bias[NPORT]; // TRUE when applying hysteresis to bias_detect circuit
12     for (i = 0; i < NPORT; i++) {
13         if (bias_detect[i] == FALSE) {
14             bias_filter[i] = FALSE;
15             bias[i] = FALSE; // Report immediately
16         } else if (bias_filter[i]) { // Filtering positive bias transition?
17             if (bias_timer >= BIAS_FILTER_TIME) {
18                 bias_filter[i] = FALSE; // Done filtering
19                 bias[i] = TRUE; // Confirm new value in PHY register bit
20             }
21         } else if (bias_detect[i] != bias[i]) { // Detected bias differs from reported bias?
22             bias_filter[i] = TRUE; // Yes, start a filtering period
23             bias_timer = 0;
24         }
25     }
26 }
27
28 boolean suspend_in_progress() { // TRUE if any port suspending
29     int i;
30     for (i = 0; i < NPORT; i++;)
31         if (suspend[i])
32             return(TRUE);
33     return(FALSE);
34 }
35
36 boolean resume_in_progress() { // TRUE if any port resuming
37     int i;
38     for (i = 0; i < NPORT; i++;)
39         if (resume[i])
40             return(TRUE);
41     return(FALSE);
42 }
43
44 void resume_all_ports() {
45     for (j = 0; j++; j < NPORT)
46         if (!active[j] && !disabled[j] && connected[j])
47             resume[j] = TRUE; // Resume all other suspended ports
48 }
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

```

1 // Per port code
2 // The code below runs per port
3 // unsubscripted references to connect_detect, etc are to be interpreted
4 // as references to connect_detect[i], where i is the number of the particular port
5
6 int received_speed, speed_ack; // speed_ack says that the other end received OUR speed
7 boolean listening_for_speed: // turns on/off the receive_speed_indication routine
8 boolean tried_bias; // indicates that the local port has tried sending TpBias
9 // just in case the far end port was suspended 1394a port
10 // initialised to false on POR, which is the only time this can occur
11 int bias_delay_timer; // used for timing out Bias delays
12 boolean sd_detected; // latches the PMD's signal_detect
13 boolean dc_connected; // latches PMD's connect_detect
14 boolean peer_port_incompatibility; // set on Beta mode speed negotiation failure
15 const Beta_mode_only; // TRUE if the port does not support DS mode
16 const max_port_speed; // Implementation dependent
17
18 void power_reset() {
19     connected = dc_connected = sd_detected = receive_OK = FALSE;
20     Beta_mode = active = disabled = suspend = resume = FALSE;
21     fault = peer_port_incompatibility = tried_bias = FALSE;
22     activate_connect_detect(0);
23 }
24
25 void activate_connect_detect(int delay) {
26     if (Beta_mode) {
27         bport_active = FALSE;
28     } else if (!Beta_mode_only) {
29         tpBias(0); // Drive TpBias low
30         if (delay != 0) {
31             connect_timer = 0;
32             while (connect_timer < delay) // Enforce minimum hold time for TpBias low
33                 ;
34             while (!connect_detect) // Wait for connect_detect circuit to stabilize
35                 ;
36             tpBias(Z); // Release TpBias
37         }
38         connect_detect_valid = TRUE; // Signal OK to connection_status()
39     }
40
41 void dc_connection_detector() { // Continuously running
42     // Note that an incoming tone may cause connect_detect
43     // to give a false "disconnect" indication, hence the
44     // need for a latch
45     if (connect_detect_valid) // latches connect_detect
46         if (connect_detect) dc_connected = TRUE;
47
48 }
49
50
51 void send_tone() {
52     pmd_tone_on = TRUE;
53     wait (TONE_DURATION);
54     pmd_tone_on = FALSE;
55 }
56
57 void signal_detect_monitor() { // continuously running - latches signal_detect
58     if (!sd_detected) sd_detected = signal_detect;
59 }
60
61
62
63
64
65
66

```

```

1  boolean signal_detect_OK() {          // true if signal_detect set since we last looked
2      boolean x;
3      wait(ACTIVE_SAMPLE_INTERVAL); // filter for chatter on signal detect
4      x = sd_detected;
5      wait(TONE_DURATION);           // to make sure that we don't see the same bit twice
6      if (x) sd_detected = FALSE;    // now we can start looking for the next bit
7                                     // if we saw a bit last time, but take care not to
8                                     // zero out a bit which started just after we looked
9      return(x);
10 }
11
12 void send_speed(speed, ack) {
13     send_tone();
14     wait(SPEEDTONE_BIT_INTERVAL);
15     if (ack) send_tone() else wait(TONE_DURATION);
16     for (i = 0; i++; i < 3) {        // send three bits or spaces
17         wait(SPEEDTONE_BIT_INTERVAL);
18         if ((speed & (1 << i)) != 0) send_tone() else wait(TONE_DURATION);
19     }
20     wait(DISCONNECTED_TONE_INTERVAL - 4 * SPEEDTONE_BIT_INTERVAL - 5 * TONE_DURATION);
21 }
22
23 void receive_speed_indication(){ // continuously running
24     int count;
25     int rs;                       // accumulate the received speed
26     while (listening_for_speed) {
27         rs = 0;
28         // first find the gap
29         count = TONE_GAP - 2; // maximum gap that can occur within a speed signal
30                                 // including possible future codings
31         while count > 0 {
32             if signal_detect_OK() count = TONE_GAP -2;
33                                 // saw a bit, so start looking for the gap again
34             wait(SPEEDTONE_BIT_INTERVAL);
35             count = count - 1;
36         }
37         count = (TONE_INTERVAL - TONE+GAP)*4;
38                                 // Start bit should occur within 8 speed tone units
39                                 // (total of 16 in a disconnected_tone_interval)
40         found = FALSE;
41         while (count > 0 && !found) {
42             count = count - 1;
43             found = signal_detect_OK(); // seen the start bit for a new speed tone
44         }
45         wait(SPEEDTONE_BIT_INTERVAL/2); // for centering
46         if (found) {
47             speed_ack = signal_detect_OK(); // next bit is the ack bit
48             for (i = 0; i++; i < 3) { // now look for three speed code bits
49                 wait(SPEEDTONE_BIT_INTERVAL);
50                 if signal_detect_OK() rs = rs | (1 << i));
51             }
52             received_speed = rs; // must be atomic as it is a shared variable
53         }
54     }
55 }
56
57 boolean set_beta() { // set beta mode and exchange speed signals to establish operating speed
58     // returns FALSE if the speed negotiation failed
59     int speed_agreed, we_agree, count;
60     port_speed = max_port_speed; // our starting point
61     count = 8; // four tries
62     speed_agreed = we_agree = speed_ack = FALSE;
63     received_speed = 0;
64     listening_for_speed = TRUE: // set the autonomous speed listener going;
65
66

```

```

1   while ((count > 0) & !speed_agreed) {
2       send_speed(port_speed, we_agree); // ack if we think the speed is agreed
3       if (received_speed != 0) { // note, may not be a new indication
4           if (received_speed == port_speed) {
5               we_agree = TRUE;
6               speed_agreed = speed_ack; // all agreed when we have the acknowledge
7                                           // from the other end
8           } else {
9               if (received_speed < port_speed) {
10                  // if the received speed is unacceptable,
11                  // then we could add code here to give up altogether
12                  port_speed = received_speed;
13                  count = 8; // and try again with a lower speed
14              } else count++; // received speed > port_speed, so allow another go,
15                              // but not too many times
16          }
17      }
18      count = count - 2;
19  }
20  listen_for_speed = FALSE; // turn off the autonomou listner (may take some time)
21  if (speed_agreed) {
22      Beta_mode = TRUE;
23      connected = TRUE;
24      receive_OK = TRUE;
25  }
26  return (speed_agreed);
27 }
28
29
30 void set_DS() { // set DS mode (implied by Beta_mode remaining false)
31     connect_detect_valid = FALSE;
31     connected = TRUE;
32     receive_OK = TRUE;
33 }
34
35 void connection_status() { // continuously running code for each port
36     if (!local_plug_present && !connected) return; // give up
37         // i.e. to start the routine again;
38
39 // ED - the treatment of disabled is to be reviewed following finalisation of 1394a
40     if (disabled) { // give up if Beta mode
41         if (Beta_mode || !connected) return;
42             // look at connection circuit only if still connected in DS mode
43     }
44
45     if (!connected && local_plug_present) {
46         // look for new connection and determine operating mode
47         int new_connection_detected;
48         new_connection_detected = FALSE;
49         for (i = 0; i < NO_OF_TRIES; i++) {
50             if (!Beta_mode_only) { // only needed if bi-lingual port
51                 if (bias) break; // don't try to send tones if detect TpBias
52             }
53             if (dc_connected && !new_connection_detected) {
54                 // try toning for at least two more times on a new connection
55                 new_connection_detected = TRUE;
56                 i = NO_OF_TRIES - 2;
57             }
58         }
59         if signal_detect_OK() {
60
61
62
63
64
65
66

```

```

1         if set_beta() return          // to next time round in this routine
2         else if (dc_connected && !Beta_mode_only) { // or fall back to DS
3             set_ds();
4             return;
5         }
6         else peer_port_incompatibility = true; // generate PHY status indication
7         return;
8     }
9     send_tone();
10    wait (DISCONNECTED_TONE_INTERVAL);
11 }
12
13 // here if (1) tried toning 4 times without detecting a tone;
14 // (2) detected a connection and tried toning twice without detecting a tone;
15 // (3) detected incoming TpBias
16 if (!dc_connected || Beta_mode_only) return; // to next time round in this routine
17 if (bias) { // still seeing Bias
18     bias_delay_timer = 0;
19     while ((bias_delay_timer < BIAS_HANDSHAKE_DELAY) && bias)
20         ; // twice as long as a 1394a or 1394b port will generate it
21     if (bias) { // incoming TpBias persists
22         set_DS(); // peer is a 1394-1995 port
23         return;
24     }
25     tried_bias = FALSE; // must try bias after hearing bias, as the peer port may
26                         // be a 1394a port which was resuming and then went into suspend
27 }
28 // Bias not present, try toning again
29 send_tone();
30 wait(DISCONNECTED_TONE_INTERVAL-TONE_DURATION);
31 if (signal_detect_OK()) {
32     if (set_beta()) return // to next time round in this routine
33     else if (dc_connected) { // or fall back to DS
34         set_ds();
35         return;
36     }
37     else peer_port_incompatibility = true; // generate PHY status indication
38     return;
39 }
40 if (!tried_bias) { // try sending Bias just once, in case we have powered up
41     tried_bias = TRUE; // whilst the connected 1394a PHY was in suspend
42     connect_detect_valid = FALSE;
43     TpBias(1);
44     bias_delay_timer = 0;
45     while ((bias_delay_timer < MAX_BIAS_HANDSHAKE) && !bias)
46         ;
47     if (bias) {
48         set_DS();
49         return;
50     }
51     activate_connect_detect(0); // includes taking TpBias away
52 }
53 return;
54 } // end of actions whilst disconnected or connecting
55
56 // here if connected
57 if (active || resume_in_progress) {
58     if (Beta_mode) {
59         wait(ACTIVE_SAMPLE_INTERVAL); // filtering for chatter on SD???
60         receive_OK = signal_detect_OK();
61     } else receive_OK = bias; //DS mode
62     return;
63 }
64 }
65 }
66 }

```

```

1
2 // here if suspended, or (DS mode and disabled) - look for disconnect or resume
3   if (Beta_mode) {
4       if (dc_connected) {
5           if (connect_detect) connected = TRUE;
6           else {
7               signal_detect_OK(); // flush out any old value;
8               wait(TONE_DURATION);
9               if (signal_detect_OK()) { continuous tone coming in
10                  receive_OK = TRUE: // to start resuming
11                  return;
12              }
13              else connected = FALSE;
14          }
15      } else // Beta mode AC connected - send a tone at periodic intervals
16      (
17          know_still_connected = FALSE;
18          send_tone();
19          for (i = 0; i < RESUME_CHECKS; i++) {
20              if (signal_detect_OK()) {
21                  know_still_connected = TRUE;
22                  wait(TONE_DURATION);
23                  signal_detect_OK();
24                  //flush out any residual value from the first detection
25                  wait(TONE_DURATION);
26                  if (signal_detect_OK()) {
27                      receive_OK = TRUE; // to start resuming
28                      return;
29                  }
30              }
31          }
32          connected = know_still_connected;
33          wait (RESUME_SAMPLING_INTERVAL-TONE_DURATION);
34      )
35  } else { // DS mode, suspended or disabled
36      if (!disabled) {
37          receive_OK = bias;
38          if (receive_OK) return;
39      }
40      connected = connect_detect; // see if still connected
41  }
42  if (!connected) { // disconnection detected
43      Beta_mode = FALSE;
44      dc_connected = FALSE; // may be immediately latched TRUE again!
45      if (int_enable && !port_event) {
46          port_event = TRUE;
47          if (link_active && LPS)
48              PH_EVENT.indication(INTERRUPT);
49          else
50              PH_EVENT.indication(LINK_ON);
51      }
52  }
53  }
54  }
55  }
56  }
57
58 void disabled_actions {
59     if (int_enable && !port_event) {
60         port_event = TRUE;
61         if (link_active && LPS)
62             PH_EVENT.indication(INTERRUPT);
63         else
64             PH_EVENT.indication(LINK_ON);
65     }
66 }

```

```

1   disable_notify = signaled = FALSE;
2   disabled = TRUE;
3   activate_connect_detect(0);      // Enable the connect detect circuit
4 }
5
6 void resume_actions() {
7   while (suspend_in_progress())    // Let any other suspensions complete
8     ;                               // (we may resume those ports later)
9   connect_timer = 0;
10  if ((int_enable || resume_int) && !port_event) {
11    port_event = TRUE;
12    if (link_active && LPS)
13      PH_EVENT.indication(INTERRUPT);
14    else
15      PH_EVENT.indication(LINK_ON);
16  }
17  connect_detect_valid = FALSE;    // Bias renders connect detect circuit useless,
18  if (resume == 0 && !boundary_node) resume_all_ports()
19  else resume = TRUE;              // Guarantee resume_in_progress() returns TRUE
20  if (Beta_mode) {                 // start up and train the port
21    bport_active = TRUE;
22    while ((connect_timer < (RECEIVER_INIT_TIME + (SYNCHRONIZATION_LENGTH/
23      (BASE_RATE*2**(port_speed-1)))) && !bport_sync_ok)
24      ;                               // wait until the port tells us it is synchronized
25    fault = !bport_sync_ok;        // Resume attempt failed if failed to synchronize
26    if (fault) {
27      activate_connect_detect(0);
28      resume = FALSE;
29      return;                        // Resume attempt complete
30    }
31  } else { // DS mode
32    tpBias(1);                      // Generate TpBias
33  }
34  while (((connect_timer < MAX_BIAS_HANDSHAKE) && !receive_ok) || bus_initialize_active)
35    ;                               // Wait for peer PHY to generate bias (DS mode)
36  if (receive_ok) {                // Connection restored to active state?
37    while ((connect_timer < 3 * RESET_DETECT) && !bus_initialize_active)
38      ;
39    if (!bus_initialize_active) {   // No other node initiated reset?
40      if (boundary_node) isbr = TRUE; // Can we arbitrate? Yes, don't wait any longer
41    } else {
42      while ((connect_timer < 7 * RESET_DETECT) && !bus_initialize_active)
43        ;                               // Let's wait a little longer...
44      if (!bus_initialize_active) ibr = TRUE; // Sigh! We'll have to use long reset
45    }
46  }
47  fault = !receive_ok;              // Resume attempt failed if TpReceive_ok is absent
48  if (fault)                        // If so, restore usefulness of connect detect circuit
49    activate_connect_detect(0);
50  resume = FALSE;                  // Resume attempt complete
51 }
52
53 void suspend_initiator_actions() {
54   connect_timer = 0;               // Used to debounce receive_ok or for receive_ok handshake
55   if (!suspend) {                 // Unexpected loss of receive_ok?
56     suspend = TRUE;               // Insure suspend_in_progress() returns TRUE
57     if (child)                    // Yes, parent still connected?
58       isbr = TRUE;                // Arbitrate for short reset
59   } else
60     ibr = TRUE;                    // Transition to R0 for reset
61 }

```

```

1      while (connect_timer < CONNECT_TIMEOUT)
2          ; // Time for receive_ok to stabilize
3      }
4      while ((connect_timer < Receive_OK_HANDSHAKE) && receive_ok)
5          ; // Wait for suspend target to deassert receive_ok
6      fault = receive_ok; // Suspend handshake refused by target?
7      activate_connect_detect(Receive_OK_HANDSHAKE); // Also guarantees handshake timing
8  }
9
10
11 void suspend_target_actions() {
12     int j;
13     if (resume_in_progress()) // Other ports resuming?
14         resume = TRUE; // OK, do suspend handshake but resume afterwards
15     suspend = TRUE; // Insure suspend_in_progress() returns TRUE
16     if (portR() == RX_DISABLE_NOTIFY) { // Is our peer PHY going away?
17         breq = IMMEDIATE_REQ; // Topology change! Reset on other (active) ports
18         isbr = TRUE;
19     } else if (portR() == RX_SUSPEND && !resume) { // Don't propagate if resume in progress
20         for (j = 0; j < NPORT; j++)
21             if (active[j]) // Otherwise all active ports become suspend initiators
22                 suspend[j] = TRUE;
23         breq = IMMEDIATE_REQ; // Invoke transmitter to propagate TX_SUSPEND
24         isbr = TRUE; // Alert link that we're now isolated
25     }
26     while (portR() == RX_DISABLE_NOTIFY || portR() == RX_SUSPEND)
27         ; // Let signals complete before receive_ok handshake
28     activate_connect_detect(Receive_OK_HANDSHAKE);
29 }
30
31 void suspended_actions() {
32     signaled = suspend = FALSE;
33     if (int_enable && !port_event) {
34         port_event = TRUE;
35         if (link_active && LPS)
36             PH_EVENT.indication(INTERRUPT);
37         else
38             PH_EVENT.indication(LINK_ON);
39     }
40 }
41 }
42 }
43
44
45

```

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_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: 1394a and earlier PHY's would attempt to interpret the tone if TpBias were present. Only safe way to tone into a 1394a mode is when TpBias is deasserted.

#2. Can't wait for remote port on peer PHY to initiate TpBias. Proof: A 1394a node w/ a suspended port won't initiate TpBias as long as the DC connection status is continually active. Consider a 1394a 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 1394a node. If the eager beta algorithm waited for the 1394a 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. 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 1394a port leads to a false detection of the remote peer. To avoid this, port Y should tone first even if it sees 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 generate TpBias. After a single phase of TpBias the procedure then reverts 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 1394a 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.

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

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

#7. DC low current connect detect mechanism may give false negative. Proof: An incoming tone on a DC coupled connection will be centered around a common mode bias which may raise the voltage higher than the connect detect threshold (low voltage implies connection).

11.6.3 Connection detection and mode determination algorithm

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

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

2) Check for "connect_detect". If not set, then go back to the start of 1. Otherwise, check for TpBias. If TpBias is detected, wait a period of time (BIAS_HANDSHAKE_DELAY, twice the maximum for BIAS_HANDSHAKE) and then sample Bias once again. Wait for it to go away so that toning can be tried again.

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

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 occur, otherwise, an attached B-PHY(1) will respond with the startup tone and Beta-Mode will be established.

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

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 it is connected. A 1394a node on the other end will sense con_status and start its debounce timeouts.

If an 1394a port is the connection at the other end, and, if just by chance, the local P1394b port does not respond with TpBias before the 1394a port "times-out" there really is no issue because the 1394a port will enter into a resume-fault condition, but, because the P1394b port did not receive a tone in reply to its last tone attempt, it will generate TpBias again causing the 1394a port to "wake" and become a resume target.

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 and when the P1394b port finally generates TpBias it (the B-PHY) will eventually generate a bus-reset because of the "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 11-2—Connect Status value in various connection scenarios

	Connect_status	tone exchange	action
No connection	No	fails	
DC connection to 1394a	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

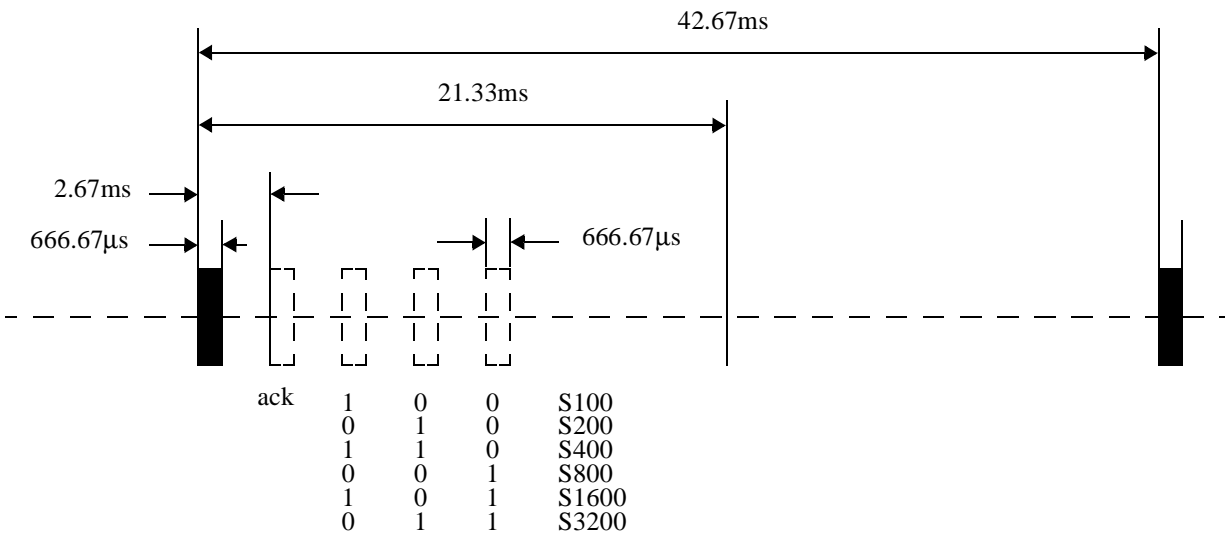
Once a Beta mode connection has been determined, a speed negotiating procedure is used. A single protocol is used to negotiate the speed anywhere in the range S100 to S3200.

Upon connection, the only property of the connecting medium that can be relied upon is that it can support the exchange of 61.44MHz tones provided they last at least 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 11-3.

Figure 11-3—Speed tone timing diagram



Note that in order to recognize the "start" tone (the first tone in the sequence which is always present), the encoding of the speed and any other information is constrained to last for less than half DISCONNECTED_TONE_INTERVAL (i.e. less than 21.33 ms).