

```
1 timer connect_timer; // global timer
2 int port_speed[NPORT]; // port operating speed, shared with Beta Mode port
3 int connect_detect[NPORT]; // shared with PMD
4 int bias[NPORT]; // shared with PMD
5 boolean pmd_tone_on[NPORT]; // shared with PMD
6 int connect_detect_valid[NPORT]; // flag to track whether connect_detect
7 // can be used at a given time
8 int active[NPORT], connected[NPORT], disabled[NPORT], Beta_mode[NPORT];
9 boolean suspend[NPORT], resume[NPORT], fault[NPORT], receive_OK[NPORT];
10 boolean Beta_mode[NPORT];
11 boolean bport_active[NPORT]; // shared with port machine
12 // the following replaces the 1394a definition
13 enum speedCode {Undefined, S100, S200, S400, S800, S1600, S3200};
14 // speed codes are encoded 001 010 011 100 101 110
15
16 // The following bias filtering code is optional for P1394b connectivity management.
17 // If it is not used, then bias[i] shall reflect the value of bias_detect[i] at all times.
18 // Note that bias filtering is mandatory for operation in 1394a (DS) mode.
19 // Note also that bias will normally be reported as TRUE when the port is
20 // operating in Beta mode, as bias_detect will detect the locally generated bias
21 // If the port is a Beta mode only port and bias_detect is not implemented, then
22 // bias shall be TRUE
23
24 void bias_status() { // Continuously running bias update code
25     timer bias_timer; // Timer for bias filter
26     boolean filter_bias[NPORT]; // TRUE when applying hysteresis to bias_detect circuit
27     for (i = 0; i < NPORT; i++) {
28         if (sending_tone) return; // ignore bias whilst sending a tone
29         if (bias_detect[i] == FALSE) {
30             bias_filter[i] = FALSE;
31             bias[i] = FALSE; // Report immediately
32         } else if (bias_filter[i]) { // Filtering positive bias transition?
33             if (bias_timer >= BIAS_FILTER_TIME) {
34                 bias_filter[i] = FALSE; // Done filtering
35                 bias[i] = TRUE; // Confirm new value in PHY register bit
36             }
37         } else if (bias_detect[i] != bias[i]) { // Detected and reported bias differ?
38             bias_filter[i] = TRUE; // Yes, start a filtering period
39             bias_timer = 0;
40         }
41     }
42 }
43
44 boolean suspend_in_progress() { // TRUE if any port suspending
45     int i;
46     for (i = 0; i < NPORT; i++)
47         if (suspend[i])
48             return(TRUE);
49     return(FALSE);
50 }
51
52 boolean resume_in_progress() { // TRUE if any port resuming
53     int i;
54     for (i = 0; i < NPORT; i++)
55         if (resume[i])
56             return(TRUE);
57     return(FALSE);
58 }
59
60 void resume_all_ports() {
61     for (j = 0; j < NPORT; j++)
62         if (!active[j] && !disabled[j] && connected[j])
63             resume[j] = TRUE; // Resume all other suspended ports
64 }
65
66 // Per port code
```

```

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

```

```

1         return;
2     }
3     sending_tone = TRUE;
4     pmd_tone_on = TRUE;
5     wait (TONE_DURATION);
6     pmd_tone_on = FALSE;
7     sending_tone = FALSE;
8 }
9
10 void signal_detect_monitor() { // continuously running - latches signal_detect
11     if (signal_detect) sd_detected = TRUE;
12 }
13 boolean signal_detect_OK() { // true if signal_detect set since we last looked
14     boolean x;
15     x = sd_detected;
16     wait(TONE_DURATION); // to make sure that we don't see the same bit twice
17     if (x) sd_detected = FALSE; // now we can start looking for the next bit
18                                 // if we saw a bit last time, but take care not to
19                                 // zero out a bit which started just after we looked
20     return(x);
21 }
22
23 void toner() { // continuously running
24     while (toning || t_send_speed) {
25         send_tone();
26         if wait(t_send_speed) {SPEEDTONE_BIT_INTERVAL};
27         wait(SPEEDTONE_BIT_INTERVAL);
28         if (t_send_speed) {
29             if (t_ack) send_tone() else wait(TONE_DURATION);
30             for (i = 0; i++; i < 3) { // send three bits or spaces
31                 wait(SPEEDTONE_BIT_INTERVAL);
31                 if ((t_speed & (1 << i)) != 0) send_tone() else wait(TONE_DURATION);
32             }
33             t_send_speed = FALSE;
34             wait(DISCONNECTED_TONE_INTERVAL -
35                 4*SPEEDTONE_BIT_INTERVAL - 5*TONE_DURATION);
36         } else wait wait(DISCONNECTED_TONE_INTERVAL - TONE_DURATIONTONE_DURATION -
37 SPEEDTONE_BIT_INTERVAL);
38     }
39 }
40
41 void send_speed(speed, ack) {
42     t_speed = speed;
43     t_ack = ack;
44     t_send_speed = TRUE; // initiate sending the speed on the next revolution
45     while (t_send_speed) // and wait for it to be sent
46         ;
47 }
48
49 void receive_speed_indication(){ // continuously running
50     int count;
51     int rs; // accumulate the received speed
52     while (listening_for_speed) {
53         rs = 0;
54         // first find the gap
55         count = TONE_GAP - 2; // maximum gap (number of zero bits) that can occur within a
56                                 // speed signal including possible future codings
57         while count >= 0 {
58             if (signal_detect_OK()) { // saw a bit, so start looking for the gap again
59                 count = TONE_GAP - 2;
60             } else count = count - 1;
61             wait(SPEEDTONE_BIT_INTERVAL);
62         }
63         count = (TONE_INTERVAL - TONE_GAP)*4; // units of TONE_DURATION
64                                 // Start bit should occur within 8 speed tone units
65                                 // (total of 16 in a disconnected_tone_interval)
66     }

```

```

1     found = FALSE;
2     while (count > 0 && !found) {
3         count = count - 1;
4         found = signal_detect_OK();    // seen the start bit for a new speed tone
5     }
6     wait(SPEEDTONE_BIT_INTERVAL + TONE_DURATION/2); // for centering
7     if (found) {
8         speed_ack = signal_detect_OK();    // next bit is the ack bit
9         for (i = 0; i++; i < 3) {        // now look for three speed code bits
10            wait(SPEEDTONE_BIT_INTERVAL);
11            if (signal_detect_OK()) rs = rs | (1 << i);
12        }
13        received_speed = rs;    // must be atomic as it is a shared variable
14    }
15 }
16 }
17
18 boolean set_beta() { // set beta mode and exchange speed signals to establish operating
19                     // speed, returns FALSE if the speed negotiation failed
20     int speed_agreed, we_agree, count;
21     port_speed = (cable_speed < max_port_speed) ? cable_speed : max_port_speed;
22                     // our starting point
23     if (port_speed < min_port_speed) return(FALSE);
24     count = 810; // fourfive tries
25     speed_agreed = we_agree = speed_ack = FALSE;
26     received_speed = 0;
27     listening_for_speed = TRUE; // set the autonomous speed listener going;
28     while ((count > 0) & !speed_agreed) {
29         if (received_speed != 0) { // note, may not be a new indication
30             if (received_speed == port_speed < min_port_speed) {
31                 // our starting point listening_for_speed = FALSE;
32                 return(FALSE);
33             }
34             if (received_speed == port_speed) {
35                 speed_agreed we_agree = speed_ack TRUE; // all agreed when we have the
36                 speed_agreed = speed_ack; // from all agreed when we have the other end
37                 acknowledge
38                 // from the other end
39             } else {
40                 if (received_speed < port_speed) {
41                     // if the received speed is unacceptable,
42                     // then we could add code here to give up altogether
43                     port_speed = received_speed;
44                     port_speed we_agree = received_speed TRUE;
45                     count = 810; // and try again with a lower speed
46                 } else count++; // received speed > port_speed, so allow another go,
47                                 // but not too many times
48             }
49         }
50         send_speed(port_speed, we_agree); // ack if we think the speed is agreed
51         count = count - 2;
52     }
53     listening_for_speed = FALSE; // turn off the autonomous listner (may take some time)
54     if (speed_agreed) {
55         toning = FALSE;
56         Beta_mode = TRUE;
57         connected = TRUE;
58         receive_OK = TRUE;
59     }
60     return (speed_agreed);
61 }
62
63 void set_DS() { // set DS mode (implied by Beta_mode remaining false)
64     toning = FALSE;
65     connect_detect_valid = FALSE;
66     connected = TRUE;

```

```

1     receive_OK = TRUE;
2 }
3
4 void connection_status()      { // continuously running code for each port
5     timer_tone_test_timer;
6     if (!local_plug_present && !connected) return; // give up
7         // i.e. to start the routine again;
8
9 // Editor's note -
10 // the treatment of disabled is to be reviewed following finalisation of 1394a
11
12     if (disabled) { // give up if Beta mode
13         toning = FALSE; // may have been disabled whilst trying to make a connection
14             // or during suspend
15         if (Beta_mode || !connected) return;
16             // look at connection circuit only if still connected in DS mode
17     }
18
19     if (!connected && local_plug_present(local_plug_present || dc_connected)) {
20         // look for new connection and determine operating mode
21         int new_connection_detected;
22         toning = TRUE; // set the autonomous toner going
23         new_connection_detected = FALSE;
24         while (!connected && !disabled && (local_plug_present || dc_connected)) {
25             for (i = 0; i < NO_OF_TRIES; i++) {
26                 if (dc_connected && !new_connection_detected) {
27                     if (!Beta_mode_only) { // only needed if bi-lingual port
28                         if (bias) break; // don't try to send tones if detect TpBias
29                     }
30                 // try toning for at least two more times on a new
31                 connections so break out of the trying to tone loop
32                 new_connection_detected = TRUE;if (dc_connected &&
33                 !new_connection_detected) {
34                     // try toning for at least two more times on a new connection
35                     new_connection_detected = TRUE;
36                     i = NO_OF_TRIES - 2;
37                     if (!Beta_mode_only) {
38                         bias_delay_timer_tone_test_timer = 0;
39                         while ((bias_delay_timer_tone_test_timer <
40                             (MAX_BIAS_HANDSHAKE_DISCONNECTED_TONE_INTERVAL) && !bias_sd_detected
41                             + // allow time for incoming bias to minimize the chance
42                             ; // allow time for incoming bias to minimize the chance
43                         )
44                     }
45                     i = NO_OF_TRIES - 2;
46                 }
47                 if (signal_detect_OK()) {
48                     if (set_beta()) return // to next time round in this routine
49                     else if (dc_connected && !Beta_mode_only) { // or fall back to DS
50                         set_DS();
51                         return;
52                     }
53                     else peer_port_incompatibility = true; // generate PHY status indication
54                         // generate PHY status indication
55                     return; // to continue trying to connect (unless disabled)
56                 }
57                 wait(DISCONNECTED_TONE_INTERVAL); // note that the time to do
58                 // note that the time to do signal_detect_OK means that the
59                 listening loop
60                 // is guaranteed to be longer than the toning loop on the peer connection
61                 } // is guaranteed end of trying to be longer than the toning
62                 tone_loop
63             }
64         }
65
66 // here if (1) tried toning 4 times without detecting a tone;
67 // (2) detected a connection and tried toning twice without detecting a tone;
68 // (3) detected incoming TpBias
69 if (!dc_connected || Beta_mode_only) return; // to next time round in this routine

```

```

1   if (dc_connected && !Beta_mode_only) {
2       if (bias) {     // still seeing Bias
3           bias_delay_timer = 0;
4           while ((bias_delay_timer < BIAS_HANDSHAKE_DELAY) && bias)
5               ;     // twice as long as a 1394a or 1394b port will generate it
6           if (bias) {     // incoming TpBias persists
7               set_DS();     // peer is a 1394-1995 port
8               return;
9           }
10          tried_bias = FALSE;     // must try bias after seeing bias, as the peer
11              port may be a 1394a port which was resuming
12              and then went into suspend with a resume fault,
13              now must to try toning again in case we
14          tried_bias = FALSE;     // must try bias after hearing bias, as the powered up
15          just when a peer port 1394b port was
16              may be a 1394a port which was resuming generating
17          bias
18          } else if (!tried_bias) {     // and then went into suspend
19      }
20          // Bias not present, note that toning is still going on
21          // try sending Bias not present just once, note that toning is still going
22          on in case we have powered up
23          // whilst the connected 1394a PHY was in suspend
24          tried_bias = TRUE;
25          connect_detect_valid = FALSE;
26          wait TpBias(DISCONNECTED_TONE_INTERVAL);
27          bias_delay_timer = 0;
28          while ((bias_delay_timer < MAX_BIAS_HANDSHAKE) && !bias)
29              i
30              if (signal_detect_OK() bias) {
31                  if (set_beta()) return     // to next time round in this routine
32                  else if (dc_connected) {     // or fall back to DSset_DS();
33                      set_DS() return;
34                      return;
35                  }
36                  activate_connect_detect(0); // includes taking TpBias away
37                  toning = FALSE; // the DC connected peer port must be disabled
38                  // or powered off, either way it'll wake us
39              } // end of trying bias
40          else peer_port_incompatibility = true; // generate PHY status indication
41          return;
42      }
43      if (!tried_bias) {     // try sending Bias just once, in case we have powered up
44          tried_bias = TRUE;     // whilst the connected 1394a PHY was in suspend
45          connect_detect_valid = FALSE;
46          TpBias(1);
47          bias_delay_timer = 0;
48          while ((bias_delay_timer < MAX_BIAS_HANDSHAKE) && !bias)
49              +
50          } // end of DC connected actions when toning has failed or bias detected
51      } // main loop whilst not connected return;
52          set_DS();
53          return;
54      }
55      activate_connect_detect(0); // includes taking TpBias away
56      return;
57  } // end of actions whilst disconnected or connecting
58
59  // here if connected
60  if (active || resume_in_progress()) {
61      if (Beta_mode) {
62          wait(ACTIVE_SAMPLE_INTERVAL);
63          receive_OK = signal_detect_OK();
64      } else receive_OK = bias;     //DS mode
65      return;
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       if (signal_detect_OK()) { continuous tone coming in
9         receive_OK = TRUE; // to start resuming
10        return;
11      }
12      else connected = FALSE;
13    }
14  } else // Beta mode AC connected - send a tone at periodic intervals
15  (
16    know_still_connected = FALSE;
17    toning = TRUE; // turn on the autonomous toner
18    for (i = 0; i < RESUME_CHECKS; i++) {
19      if (signal_detect_OK()) {
20        know_still_connected = TRUE;
21        if (signal_detect_OK()) { // still true, also the time taken
22          // for this inserts an extra delay which
23          // allows for clock frequency differences
24          // between transmitter and receiver
25          receive_OK = TRUE; // to start resuming
26          toning = FALSE; // turn off the autonomous toner
27          return;
28        }
29      }
30      wait (RESUME_SAMPLING_INTERVAL - TONE_DURATION);
31    }
31    connected = know_still_connected;
32  }
33  } else { // DS mode, suspended or disabled
34    if (!disabled) {
35      receive_OK = bias;
36      if (receive_OK) return;
37    }
38    connected = connect_detect; // see if still connected
39  }
40  if (!connected) { // disconnection detected
41    Beta_mode = FALSE;
42    dc_connected = FALSE; // may be immediately latched TRUE again!
43    if (int_enable && !port_event) {
44      port_event = TRUE;
45      if (link_active && LPS)
46        PH_EVENT.indication(INTERRUPT);
47      else
48        PH_EVENT.indication(LINK_ON);
49    }
50  }
51 }
52
53 void disabled_actions {
54   if (int_enable && !port_event) {
55     port_event = TRUE;
56     if (link_active && LPS)
57       PH_EVENT.indication(INTERRUPT);
58   } else
59     PH_EVENT.indication(LINK_ON);
60 }
61 disable_notify = signaled = FALSE;
62 disabled = TRUE;
63 activate_connect_detect(0); // Enable the connect detect circuit
64 }
65
66
```

```

1 void resume_actions() {
2     while (suspend_in_progress())          // Let any other suspensions complete
3         ;                                  // (we may resume those ports later)
4     connect_timer = 0;
5     if ((int_enable || resume_int) && !port_event) {
6         port_event = TRUE;
7         if (link_active && LPS)
8             PH_EVENT.indication(INTERRUPT);
9         else
10            PH_EVENT.indication(LINK_ON);
11    }
12    connect_detect_valid = FALSE;          // Bias renders connect detect circuit useless,
13    if (!(resume == 0 && !boundary_node) resume_all_ports()
14    else resume = TRUE;                    // Guarantee resume_in_progress() returns TRUE
15    if (Beta_mode) {                       // start up and train the port
16        bport_active = TRUE;
17        while ((connect_timer < (RECEIVER_INIT_TIME + (SYNCHRONIZATION_LENGTH/
18            (BASE_RATE*2**(port_speed-1))) && !bport_sync_ok)
19            ;                               // wait until the port tells ut it is synchronized
20        fault = !bport_sync_ok;           // Resume attempt failed if failed to synchronize
21        if (fault) {
22            activate_connect_detect(0);
23            resume = FALSE;
24            return;                         // Resume attempt complete
25        }
26    } else { // DS mode
27        tpBias(1);                          // Generate TpBias
28    }
29    while (((connect_timer < MAX_BIAS_HANDSHAKE) && !receive_ok) || bus_initialize_active)
30        ;                                   // Wait for peer PHY to generate bias (DS mode)
31    if (receive_ok) {                       // Connection restored to active state?
32        while ((connect_timer < 3 * RESET_DETECT) && !bus_initialize_active)
33            ;
34        if (!bus_initialize_active) {       // No other node initiated reset?
35            if (boundary_node)isbr = TRUE; // Can we arbitrate? Yes, don't wait any longer
36            else {
37                while ((connect_timer < 7 * RESET_DETECT) && !bus_initialize_active)
38                    ;                       // Let's wait a little longer...
39                if (!bus_initialize_active) ibr = TRUE; // Sigh! We'll have to use long reset
40            }
41        }
42        fault = !receive_ok;                // Resume attempt failed if TpReceive_ok is absent
43        if (fault)                          // If so, restore usefulness of connect detect circuit
44            activate_connect_detect(0);
45        resume = FALSE;                     // Resume attempt complete
46    }
47 }
48
49 void suspend_initiator_actions() {
50     connect_timer = 0;                     // Used to debounce receive_ok or for receive_ok handshake
51     if (!suspend) {                       // Unexpected loss of receive_ok?
52         suspend = TRUE;                   // Insure suspend_in_progress() returns TRUE
53         if (child)                         // Yes, parent still connected?
54             isbr = TRUE;                  // Arbitrate for short reset
55         else
56             ibr = TRUE;                    // Transition to R0 for reset
57         while (connect_timer < CONNECT_TIMEOUT)
58             ;                               // Time for receive_ok to stabilize
59     }
60     while ((connect_timer < Receive_OK_HANDSHAKE) && receive_ok)
61         ;                                   // Wait for suspend target to deassert receive_ok
62     fault = receive_ok;                    // Suspend handshake refused by target?
63     activate_connect_detect(Receive_OK_HANDSHAKE); // Also guarantees handshake timing
64 }
65
66 void suspend_target_actions() {

```

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