

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

```

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

```

```

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

```

```

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

```

```

1   connected connect_detect_valid = TRUEFALSE;
2   receive_OK connected = TRUE;
3   receive_OK = TRUE;
4 }
5
6 void connection_status()      { // continuously running code for each port
7     if (!local_plug_present && !connected) return; // give up
8         // i.e. to start the routine again;
9
10  // Editor's note -
11  // ED—the treatment of disabled is to be reviewed following finalisation of 1394a
12
13     if (disabled) { // give up if Beta mode
14         toning = FALSE; // may have been disabled whilst trying to make a connection
15         // or during suspend
16         if (Beta_mode || !connected) return;
17         // look at connection circuit only if still connected in DS mode
18     }
19
20     if (!connected && local_plug_present) {
21         // look for new connection and determine operating mode
22         int new_connection_detected;
23         toning = TRUE; // set the autonomous toner going
24         new_connection_detected = FALSE;
25         for (i = 0; i < NO_OF_TRIES; i++) {
26             if (dc_connected && !Beta_mode_onlynew_connection_detected) {// only needed if
27 bi-lingual port {
28                 // try toning for at least two more times on a new connection
29                 if (bias) breaknew_connection_detected = TRUE; // don't try to send tones if
30 detect TpBias
31 }
32                 if (dc_connected && !new_connection_detected!Beta_mode_only) {
33                     bias_delay_timer = 0;
34                     while (bias_delay_timer < (MAX_BIAS_HANDSHAKE) && !bias)
35 new_connection_incoming bias to minimize the chance
36                     new_connection_detected = TRUE; // of toning into TpBias
37                 }
38                 i = NO_OF_TRIES - 2;
39             }
40             if (signal_detect_OK()) {
41                 if (set_beta()) return // to next time round in this routine
42                 else if (dc_connected && !Beta_mode_only) { // or fall back to DS
43                     set_dsset_DS();
44                     return;
45                 }
46                 else peer_port_incompatibility = true; // generate PHY status indicn
47                 return;
48             }
49             wait(DISCONNECTED_TONE_INTERVAL); // note that the time to do
50             // signal_detect_OK means that the listening loop
51             // is guaranteed to be longer than the toning loop
52         }
53     }
54     wait(DISCONNECTED_TONE_INTERVAL);
55 }
56
57 // here if (1) tried toning 4 times without detecting a tone;
58 // (2) detected a connection and tried toning twice without detecting a tone;
59 // (3) detected incoming TpBias
60 if (!dc_connected || Beta_mode_only) return; // to next time round in this routine
61 if (bias) { // still seeing Bias
62     bias_delay_timer = 0;
63     while ((bias_delay_timer < BIAS_HANDSHAKE_DELAY) && bias)
64         ; // twice as long as a 1394a or 1394b port will generate it
65     if (bias) { // incoming TpBias persists
66         set_DS(); // peer is a 1394-1995 port
    }
}

```

```

1         return;
2     }
3     tried_bias = FALSE;    // must try bias after hearing bias, as the peer port
4                             // may be a 1394a port which was resuming
5                             // and then went into suspend
6     }
7     // Bias not present, try note that toning again is still going on
8     send_tonewait(DISCONNECTED_TONE_INTERVAL);
9     wait(DISCONNECTED_TONE_INTERVAL_TONE_DURATION);
10    if (signal_detect_OK()) {
11        if (set_beta()) return    // to next time round in this routine
12    else if (dc_connected) {    // or fall back to DS
13        set_dsset_DS();
14        return;
15    }
16    else peer_port_incompatibility = true; // generate PHY status indication
17    return;
18 }
19 if (!tried_bias) {    // try sending Bias just once, in case we have powered up
20     tried_bias = TRUE;    // whilst the connected 1394a PHY was in suspend
21     connect_detect_valid = FALSE;
22     TpBias(1);
23     bias_delay_timer = 0;
24     while ((bias_delay_timer < MAX_BIAS_HANDSHAKE) && !bias)
25         ;
26     if (bias) {
27         set_DS();
28         return;
29     }
30     activate_connect_detect(0); // includes taking TpBias away
31 }
32 return;
33 } // end of actions whilst disconnected or connecting
34 // here if connected
35 if (active || resume_in_progress) {
36     if (Beta_mode) {
37         wait(ACTIVE_SAMPLE_INTERVAL); // filtering for chatter on SD???
38         receive_OK = signal_detect_OK();
39     } else receive_OK = bias;    //DS mode
40     return;
41 }
42 // here if suspended, or (DS mode and disabled) - look for disconnect or resume
43 if (Beta_mode) {
44     if (dc_connected) {
45         if (connect_detect) connected = TRUE;
46     else {
47         signal_detect_OK(); // flush out any old value;
48         waitsignal_detect_OK(TONE_DURATION); // flush out any old value;
49         if (signal_detect_OK()) { continuous tone coming in
50             receive_OK = TRUE:    // to start resuming
51             return;
52         }
53     else connected = FALSE:
54     }
55 }
56 } else // Beta mode AC connected - send a tone at periodic intervals
57 (
58     know_still_connected = FALSE;
59     send_tone();
60     toning = TRUE;    // turn on the autonomous toner
61     for (i = 0; i < RESUME_CHECKS; i++) {
62         if (signal_detect_OK()) {
63             know_still_connected = TRUE;
64             wait(TONE_DURATION);
65             signal_detect_OK();

```

```

1         if (signal_detect_OK()) { // still true, also the time taken
2         // for this inserts an extra delay which
3         // allows for clock frequency differences
4         //flush out any residual value from the first
5         detection/ between transmitter and receiver
6         wait(TONE_DURATION)receive_OK = TRUE; // to start resuming
7         if (signal_detect_OK()) {
8         receive_OK_toning = TRUEFALSE; // to start resumingturn off the
9         autonomous toner
10        return;
11    }
12    }
13    +
14    wait (RESUME_SAMPLING_INTERVAL - TONE_DURATION);
15    connected = know_still_connected;
16    wait (RESUME_SAMPLING_INTERVAL - TONE_DURATION)connected = know_still_connected;
17    }
18    } else { // DS mode, suspended or disabled
19        if (!disabled) {
20            receive_OK = bias;
21            if (receive_OK) return;
22        }
23        connected = connect_detect; // see if still connected
24    }
25    if (!connected) { // disconnection detected
26        Beta_mode = FALSE;
27        dc_connected = FALSE; // may be immediately latched TRUE again!
28        if (int_enable && !port_event) {
29            port_event = TRUE;
30            if (link_active && LPS)
31                PH_EVENT.indication(INTERRUPT);
31            else
32                PH_EVENT.indication(LINK_ON);
33        }
34    }
35 }
36
37 void disabled_actions {
38     if (int_enable && !port_event) {
39         port_event = TRUE;
40         if (link_active && LPS)
41             PH_EVENT.indication(INTERRUPT);
42         else
43             PH_EVENT.indication(LINK_ON);
44     }
45     disable_notify = signaled = FALSE;
46     disabled = TRUE;
47     activate_connect_detect(0); // Enable the connect detect circuit
48 }
49
50 void resume_actions() {
51     while (suspend_in_progress()) // Let any other suspensions complete
52         ; // (we may resume those ports later)
53     connect_timer = 0;
54     if ((int_enable || resume_int) && !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     connect_detect_valid = FALSE; // Bias renders connect detect circuit useless,
62     if (resume == 0 && !boundary_node) resume_all_ports()
63     else resume = TRUE; // Guarantee resume_in_progress() returns TRUE
64     if (Beta_mode) { // start up and train the port
65         bport_active = TRUE;
66     }

```

```

1         while ((connect_timer < (RECEIVER_INIT_TIME + (SYNCHRONIZATION_LENGTH/
2             (BASE_RATE*2**((port_speed-1)))) && !bport_sync_ok)
3             ; // wait until the port tells ut it is synchronized
4         fault = !bport_sync_ok; // Resume attempt failed if failed to synchronize
5         if (fault) {
6             activate_connect_detect(0);
7             resume = FALSE;
8             return; // Resume attempt complete
9         }
10    } else { // DS mode
11        tpBias(1); // Generate TpBias
12    }
13    while (((connect_timer < MAX_BIAS_HANDSHAKE) && !receive_ok) || bus_initialize_active)
14        ; // Wait for peer PHY to generate bias (DS mode)
15    if (receive_ok) { // Connection restored to active state?
16        while ((connect_timer < 3 * RESET_DETECT) && !bus_initialize_active)
17            ;
18        if (!bus_initialize_active) { // No other node initiated reset?
19            if (boundary_node)isbr = TRUE; // Can we arbitrate? Yes, don't wait any longer
20        else {
21            while ((connect_timer < 7 * RESET_DETECT) && !bus_initialize_active)
22                ; // Let's wait a little longer...
23            if (!bus_initialize_active) isbr = TRUE; // Sigh! We'll have to use long reset
24        }
25    }
26    }
27    fault = !receive_ok; // Resume attempt failed if TpReceive_ok is absent
28    if (fault) // If so, restore usefulness of connect detect circuit
29        activate_connect_detect(0);
30    resume = FALSE; // Resume attempt complete
31 }
31
32 void suspend_initiator_actions() {
33     connect_timer = 0; // Used to debounce receive_ok or for receive_ok handshake
34     if (!suspend) { // Unexpected loss of receive_ok?
35         suspend = TRUE; // Insure suspend_in_progress() returns TRUE
36         if (child) // Yes, parent still connected?
37             isbr = TRUE; // Arbitrate for short reset
38     else
39         isbr = TRUE; // Transition to R0 for reset
40     while (connect_timer < CONNECT_TIMEOUT)
41         ; // Time for receive_ok to stabilize
42    }
43    while ((connect_timer < Receive_OK_HANDSHAKE) && receive_ok)
44        ; // Wait for suspend target to deassert receive_ok
45    fault = receive_ok; // Suspend handshake refused by target?
46    activate_connect_detect(Receive_OK_HANDSHAKE); // Also guarantees handshake timing
47 }
48
49 void suspend_target_actions() {
50     int j;
51     if (resume_in_progress()) // Other ports resuming?
52         resume = TRUE; // OK, do suspend handshake but resume afterwards
53     suspend = TRUE; // Insure suspend_in_progress() returns TRUE
54     if (portR() == RX_DISABLE_NOTIFY) { // Is our peer PHY going away?
55         breq = IMMED_REQ; // Topology change! Reset on other (active) ports
56         isbr = TRUE;
57     } else if (portR() == RX_SUSPEND && !resume) { // Don't propagate if resume in progress
58         for (j = 0; j < NPORT; j++)
59             if (active[j]) // Otherwise all active ports become suspend initiators
60                 suspend[j] = TRUE;
61         breq = IMMED_REQ; // Invoke transmitter to propagate TX_SUSPEND
62         isbr = TRUE; // Alert link that we're now isolated
63     }
64     while (portR() == RX_DISABLE_NOTIFY || portR() == RX_SUSPEND)
65         ; // Let signals complete before receive_ok handshake
66 }

```

```
1   activate_connect_detect(Receive_OK_HANDSHAKE);
2 }
3
4 void suspended_actions() {
5     signaled = suspend = FALSE;
6     if (int_enable && !port_event) {
7         port_event = TRUE;
8         if (link_active && LPS)
9             PH_EVENT.indication(INTERRUPT);
10        else
11            PH_EVENT.indication(LINK_ON);
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
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
```