

```

timer connect_timer; // global timer
int port_speed[NPORT]; // port operating speed, shared with Beta Mode port
int boolean connect_detect[NPORT]; // shared with PMD
int bias boolean bias_detect[NPORT]; // shared with PMD
boolean pmd_tone_on[NPORT]; // shared with PMD
int connect_detect_valid boolean sending_tone[NPORT]; // flag to track whether connect_detect
true whilst a tone is being transmitted
boolean connect_detect_valid[NPORT]; // can be used at a given time flag to track whether
connect_detect
// can be used at a given time
boolean active[NPORT], connected[NPORT], disabled[NPORT], Beta_mode[NPORT];
int active[NPORT], connected boolean suspend[NPORT], disabled resume[NPORT], fault[NPORT],
Beta_mode receive_OK[NPORT];
boolean bias[NPORT]; // port register
boolean suspendport_active[NPORT], resume[NPORT]; fault[NPORT], receive_OK[NPORT]; //
shared with port machine
boolean Beta_mode[NPORT];
boolean port_active bias_filter[NPORT]; // shared with port machine TRUE when applying
hysteresis to bias_detect circuit

// the following replaces the 1394a definition
enum speedCode {Undefined, S100, S200, S400, S800, S1600, S3200};
// speed codes are encoded 001 010 011 100 101 110

// The following bias filtering code is optional for P1394b connectivity management.
// If it is not used, then bias[i] shall reflect the value of bias_detect[i] at all times.
// Note that bias filtering is mandatory for operation in 1394a (DS) mode.
// Note also that bias will normally be reported as TRUE when the port is
// operating in Beta mode, as bias_detect will detect the locally generated bias
// If the port is a Beta mode only port and bias_detect is not implemented, then
// bias shall be TRUE

void bias_status() { // Continuously running bias update code
    timer bias_timer; // Timer for bias filter
    int i;
    for (i = 0; i < NPORT; i++) {
        if (Beta_mode_only[i]) {
boolean filter_bias[NPORT]; bias[i] = TRUE; // TRUE when applying hysteresis to bias_detect
circuit no bias processing on beta mode only ports
for (i = 0; i < NPORT; i++) {
            continue; // to next port
}
            if (sending_tone[i]) {
                bias_filter[i] = FALSE; // abandon bias filtering on this port
                if (sending_tone) return continue; // ignore bias whilst sending a tone
            }
if (bias_timer >= BIAS_FILTER_TIME | bias_detect[i]) {
                bias_filter[i] = FALSE; // Done filtering;
} else if (bias_filter[bias[i]]) = FALSE; { // Filtering positive bias
transition? Report immediately
            } else if (bias_filter[i]) { // Filtering positive bias transition?
                if (bias_timer >= BIAS_FILTER_TIME) {
                    bias_filter[i] = FALSE; // Done filtering
                    bias[i] = TRUE; // Confirm new value in PHY register bit
                }
            } else if (bias_detect[i] != bias[i]) { // Detected and reported bias differ?
                bias_filter[i] = TRUE; // Yes, start a filtering period
                bias_timer = 0;
            }
        }
    }
}

boolean suspend_in_progress() { // TRUE if any port suspending
    int i;
    for (i = 0; i < NPORT; i++;)
        if (suspend[i])

```

```

        return(TRUE);
    return(FALSE);
}

boolean resume_in_progress() {           // TRUE if any port resuming
    int i;
    for (i = 0; i < NPORT; i++;)
        if (resume[i])
            return(TRUE);
    return(FALSE);
}

void resume_all_ports() {
    int j;
    for (j = 0; j++; j < NPORT)
        if (!active[j] && !disabled[j] && connected[j])
            resume[j] = TRUE;           // Resume all other suspended ports
}

// Per port code
// The code below runs per port
// unsubscribed references to connect_detect, etc are to be interpreted
// as references to connect_detect[i], where i is the number of the particular port

boolean disable_notify; // Requests transmission of TX_DISABLE at the end of the next packet
boolean signaled;      // Indicates transmission of TX_DISABLE_NOTIFY or TX_SUSPEND
boolean received_speed, speed_ack; // speed_ack says that the other end received OUR speed
boolean listening_for_speed; // just in case turns on/off the far end port was suspended
1394a port receive speed indication routine
boolean toning;        // initialised to false on POR, which is turns on/off the only time this
can occur toner
int bias_delay_timer boolean t_send_speed; // used for timing out Bias delays if true, toner
sends a speed signal
int t_speed;           // speed value to be transmitted
boolean t_ack;         // if true, transmit the ACK bit in the next speed signal
boolean tried_bias;    // indicates that the local port has tried sending TpBias
// just in case the far end port was a suspended 1394a port
// initialised to false on POR, which is the only time this can occur
timer bias_delay_timer // just in case the far end port was suspended 1394a port used for timing
out Bias delays
boolean signal_detect; // initialised to false on POR, which is the only time this can occur PMD
signal
int bias_delay_timer boolean sd_detected; // used for timing out Bias delays latches the PMD's
signal_detect
boolean sd_detected dc_connected; // latches the PMD's signal_detect connect_detect
boolean de_connected local_plug_present; // latches PMD's connect_detect set by optional
hardware - see description in table 11-2
boolean peer_port_incompatibility; // set on Beta mode speed negotiation failure
const Beta_mode_only; // TRUE if the port does not support DS mode
const min_port_speed; // implementation dependent
int max_port_speed, cable_speed;

void power_reset activate_connect_detect(int delay) {
    if (Beta_mode) bport_active = FALSE;
    else if (!Beta_mode_only) tpBias(0); // Drive TpBias low
    if (delay != 0) {
        connect_timer = 0;
        while (connect_timer < delay) // Enforce minimum hold time for TpBias low
            i
    }
    if (!Beta_mode && !Beta_mode_only) {
        while (!connect_detect) // Wait for connect_detect circuit to stabilize
            i
        tpBias(Z); // Release TpBias
    }
    connect_detect_valid = TRUE; // Signal OK to connection_status()
}

```

```

}

void power_reset() {
    connected = dc_connected = sd_detected = receive_OK = FALSE;
    toning_bias = sending_tone = t_send_speed = pmd_tone_on bias_filter = FALSE;
    Beta_mode_toning = bport_active sending_tone = active t_send_speed = suspend = resume
    pmd_tone_on = FALSE;
    Beta_mode = bport_active = active = suspend = resume = FALSE;
    fault = peer_port_incompatibility = tried_bias = FALSE;

    // the variables disabled, max_port_speed
    listening_for_speed = FALSE;

    // the variables disabled, max_port_speed
    // are set to their implementation-dependent power-reset values

    activate_connect_detect(0);
    wait(2*DISCONNECTED_TONE_INTERVAL); // ensure that there is a sufficiently long more than
    one tone
    transmitted // period of silence interval since a signal was
    // node_status(), bias_status() (one copy of each for all ports)
    // at this point, the following continuously running routines are started
    // each routine executes repeatedly and indefinitely
    // so that the peer port moves into disconnected
    // dc_connection_detector() at this point, signal_detect_monitor(), toner(), the following
    continuously running routines are started
    // each routine executes repeatedly and indefinitely
    // node_status(), bias_status() (one copy of each for all ports)
}
// dc_connection_detector(), signal_detect_monitor(), toner(),
// receive_speed_indication(), connection_status()

if (Beta_mode) {
    bport_active = FALSE;
} else if (!Beta_mode_only) {
    tpBias(0); // Drive TpBias low
    if (delay != 0) {
        connect_timer = 0;
        while (connect_timer < delay) // Enforce minimum hold time for TpBias low
        }
    while (!connect_detect) // Wait for connect_detect circuit to stabilize
    while void dc_connection_detector(!connect_detect) { // Wait for connect_detect
    circuit to stabilize Continuously running
    tpBias(Z); // Release TpBias
    }
    connect_detect_valid = TRUE; // Signal OK to connection_status()
}

// Note that an incoming tone may cause connect_detect
// Note that an incoming tone may cause connect_detect to
give a false "disconnect" indication, hence the
// to give need for a false "disconnect" indication, hence
the latch
if (connect_detect_valid) // need for a latch latches connect_detect
if (connect_detect_valid) // latches connect_detect
    if (connect_detect) dc_connected = TRUE;
}

void send_tone() {
    if (!Beta_mode_only)
        if (bias) { // avoid toning into incoming TpBias

```

```

        wait (TONE_DURATION);
        return;
    }
    sending_tone = TRUE;
    pmd_tone_on = TRUE;
    wait (TONE_DURATION);
    pmd_tone_on = FALSE;
    sending_tone = FALSE;
}

void signal_detect_monitor() { // continuously running - latches signal_detect
    if (signal_detect) sd_detected = TRUE;
}

boolean signal_detect_OK() { // true if signal_detect set since we last looked
    boolean x;
    x = sd_detected;
    wait(TONE_DURATION); // to make sure that we don't see the same bit twice
waitif (TONE_DURATION,x) sd_detected = FALSE; // to make sure that we don't see the same
now we can start looking for the next bit twice
if (x) sd_detected = FALSE; // now if we can start looking for the next saw a bit last
time, but take care not to
// if we saw zero out a bit last time, but take care not
to which started just after we looked.
// zero out a bit which started just after we
looked
signal_detect_monitor may immediately reset it TRUE
    return(x);
}

void toner() { // continuously running
    boolean t_ack;
    int i, t_speed; // t_speed latches value of speed to send
    while (toning || t_send_speedsend_speed) {
        send_tone(); // send start bit
        wait(SPEEDTONE_BIT_INTERVAL);
        if (t_send_speedsend_speed) { // get latest status
            t_ack = we_agree; // latch latest status of we_agree
            if (t_ack) send_tone() else wait(TONE_DURATION);
            for (i = 0; i++; i < 3) { // send three bits or spaces
                wait(SPEEDTONE_BIT_INTERVAL);
                if (i == 0) t_speed = port_speed; // latch the latest speed
                if ((t_speed & (1 << i)) != 0) send_tone() else wait(TONE_DURATION);
            }
t_send_speed = FALSE;
signal(EVT_SENT_SPEED); // signal end of effective phase
if (t_ack) signal(EVT_SENT_ACK);
            wait(DISCONNECTED_TONE_INTERVAL -
                4*SPEEDTONE_BIT_INTERVAL - 5*TONE_DURATION);
        } else wait(DISCONNECTED_TONE_INTERVAL - TONE_DURATION - SPEEDTONE_BIT_INTERVAL);
    }
}

void receive_speed_indication(){ // continuously running
    int count, i;
    int rs; // accumulate the received speed
void send_speedwhile (speed, ack,listening_for_speed) {
t_speedrs = speed0;
t_ack = ack;
t_send_speed = TRUE; // initiate sending the speed on the next revolution
while (t_send_speed) // and wait for it to be sent
+
}

void receive_speed_indication(){ // continuously running
int count;
    // first find the gap
    count = (TONE_GAP - 1)*4; // maximum gap (specified in samples separated by

```

```

        count = TONE_GAP - 2; // maximum gap (number of zero bits) TONE_DURATION that
can occur within a valid speed // speed-signal including possible future codings-codings)
        while (count >= 0) {
            // first find the gap
            count = TONE_GAP - 2; // maximum gap (number of zero bits) that can occur within a
            // speed signal including possible future codings
            while count >= 0 {
                if (signal_detect_OK()) { // saw a bit, so start looking for the gap again
                    count = (TONE_GAP - 1)*4;
                } else count = count - 1;
                wait(SPEEDTONE_BIT_INTERVAL);
            }
            count = (TONE_INTERVAL - TONE_GAPTONE_GAP + 1)*4; // units of TONE_DURATIONthe
maximum time (specified in // Start bit should occur within 8 speed tone unitssamples
separated by TONE_DURATION) for a start // (total of 16 in bit to appear after a
disconnected tone interval)previous gap was detected
            found = FALSE;
            while (count > 0 && !found) {
                count = count - 1;
                found = signal_detect_OK(); // seen the start bit for a new speed tone
            }
            wait(SPEEDTONE_BIT_INTERVAL + TONE_DURATION/2); // for centering
            if (found) {
                speed_ack = signal_detect_OK(); // next bit is the ack bit
                for (i = 0; i++; i < 3) { // now look for three speed code bits
                    wait(SPEEDTONE_BIT_INTERVAL);
                    if (signal_detect_OK()) rs = rs | (1 << i);
                }
                received_speed = rs; // must be atomic as it is a shared variable
                signal(EVT_RECEIVED_SPEED);
            }
        }
    }

boolean set_beta() { // set beta mode and exchange speed signals to establish operating
                    // speed, returns FALSE if the speed negotiation failed
    boolean speed_agreed, we_agree, sent_ack;
    int speed_agreedcount, we_agree, countevent;
    port_speed = (cable_speed < max_port_speed) ? cable_speed : max_port_speed;
                    // our starting point
    if (port_speed < min_port_speed) return(FALSE);
    count = 10; // five tries
    speed_agreed = we_agree = speed_ack = FALSE;
    received_speed = 0;
    send_speed = TRUE; // start the autonomous speed sender going with the
                    // updated port_speed and the ack bit initialised to
                    // FALSE
    listening_for_speed = TRUE; // set the autonomous speed listener going;
    while ((count > 0) & !speed_agreed) || (speed_agreed & !sent_ack) {
        event = wait_event(EVT_SENT_SPEED | EVT_RECEIVED_SPEED | EVT_SENT_ACK);
        if (received_speed != 0) { // note, may not be a new indication
            if (received_speed < min_port_speed) {
                if (received_speed < min_port_speed) { listening_for_speed = FALSE;
                listening_for_speed send_speed = FALSE;
                return(FALSE);
            }
        }
        if (received_speed == port_speed) {
            we_agree = TRUE;
            speed_agreed = speed_ack; // all agreed when we have the acknowledge
                    // from the other end
        } else {
            if (port_speed received_speed < min_port_speed port_speed) return(FALSE);
            count = 10; // five tries if the received speed is unacceptable,

```

```

        // then we could add code here to give up altogether
        port_speed = received_speed;
        we_agree = TRUE;
        count = 10; // and try again with a lower speed
    } else count++; // received speed > port_speed, so allow another go,
                    // but not too many times
    }
send_speed(port_speed, we_agree); // ack if we think the speed is agreed
}
if (event & EVT_SENT_SPEED) count = count - 2;
if (event & EVT_SENT_ACK) sent_ack = TRUE;
}
listening_for_speed = FALSE; // turn off the autonomous listner (may take some time)
listening_for_speed_send_speed = FALSE; // turn off the autonomous listner sending of
speed_signal (may take some time);
if (speed_agreed) {
    toning = FALSE;
    Beta_mode = TRUE;
    connected = TRUE;
    receive_OK = TRUE;
}
return (speed_agreed);
}

void set_DS() { // set DS mode (implied by Beta_mode remaining false)
    toning = FALSE;
    connect_detect_valid = FALSE;
    connected = TRUE;
    receive_OK = TRUE;
}

void connection_status() { // continuously running code for each port
    timer tone_test_timer;
    int i;
    boolean know_still_connected; // maintains knowledge of connection if suspended

    if (!local_plug_present && !dc_connected && !connected) return; // give up
        // i.e. to start the routine again;

// Editor's note -
// the treatment of disabled is to be reviewed following finalisation of 1394a

    if (disabled) { // give up if Beta mode
        toning = FALSE; // may have been disabled whilst trying to make a connection
        // or during suspend
        if (Beta_mode || !connected) return;
        // look at connection circuit only if still connected in DS mode
    }

    if (!connected && (local_plug_present || dc_connected)) {
        // look for new connection and determine operating mode
        int new_connection_detected;
        toning = TRUE; // set the autonomous toner going
        new_connection_detected = FALSE;
        while (!connected && !disabled && (local_plug_present || dc_connected)) {
            for (i = 0; i < NO_OF_TRIES; i++) {
                if (!Beta_mode_only) { // only needed if bi-lingual port
                    if (bias) break; // don't try to send tones if detect TpBias
                }
                // so break out of the trying to tone loop
                if (dc_connected && !new_connection_detected) {
                    // try toning for at least two more times on a new connection
                    new_connection_detected = TRUE;
                    i = NO_OF_TRIES - 2;
                }
            }
            tone_test_timer = 0;
            while ((tone_test_timer < DISCONNECTED_TONE_INTERVAL) && !sd_detected)

```

```

;
if (signal_detect_OK()) {
    if (set_beta()) return // to next time round in this routine
    else if (dc_connected && !Beta_mode_only) { // or fall back to DS
        set_DS();
        return;
    }
    else peer_port_incompatibility = true;
        // generate PHY status indication
    return; // to continue trying to connect (unless disabled)
}
// note that the time to do signal_detect_OK means that the listening loop
// is guaranteed to be longer than the toning loop on the peer connection
} // end of trying to tone loop

// here if (1) tried toning 4 times without detecting a tone;
// (2) detected a connection and tried toning twice without detecting a tone;
// (3) detected incoming TpBias
if (dc_connected && !Beta_mode_only) {
    if (bias) {
        bias_delay_timer = 0;
        while ((bias_delay_timer < BIAS_HANDSHAKE_DELAY) && bias)
            ; // twice as long as a 1394a or 1394b port will generate it
        if (bias) { // incoming TpBias persists
            set_DS(); // peer is a 1394-1995 port
            return;
        }
        tried_bias = FALSE; // must try bias after seeing bias, as the peer
            // port may be a 1394a port which was resuming
            // and then went into suspend with a resume fault,
            // now must to try toning again in case we
            // powered up just when a peer p1394b port was
            // generating bias
    } else if (!tried_bias) {
        // Bias not present, note that toning is still going on
        // try sending Bias just once, in case we have powered up
        // whilst the connected 1394a PHY was in suspend
        tried_bias = TRUE;
        connect_detect_valid = FALSE;
        TpBias(1);
        bias_delay_timer = 0;
        while ((bias_delay_timer < MAX_BIAS_HANDSHAKE) && !bias)
            ;
        if (bias) {
            set_DS();
            return;
        }
        activate_connect_detect(0); // includes taking TpBias away
        toning = FALSE; // the DC connected peer port must be disabled
            // or powered off, either way it'll wake us
    } // end of trying bias
} // end of DC connected actions when toning has failed or bias detected
} // main loop whilst not connected
} // end of actions whilst disconnected or connecting

// here if connected
if (active || resume_in_progress() || suspend_in_progress()) {
    if (Beta_mode) {
        waitreceive_OK = signal_detect_OK(ACTIVE_SAMPLE_INTERVAL);
        receive_OK = signal_detect_OK();
    } else receive_OK = bias; //DS mode
    return;
}

// here if suspended, or (DS mode and disabled) - look for disconnect or resume
if (Beta_mode) {

```

```

    if (dc_connected) { // look for either a continuous tone or a disconnection
        if (connect_detect) connected = TRUE;
        else {
            signal_detect_OK(); // flush out any old value;
            if (signal_detect_OK()) { continuous tone coming in
                receive_OK = TRUE: // to start resuming
            }
            else connected = FALSE;
        }
        connected = connect_detect; // note that the value of connect_detect
        // is unreliable if there is a tone coming in, so it is
        // only tested if we are sure there is no tone coming in
    } else // Beta mode AC connected - send a tone at periodic intervals
    {
        know_still_connected = FALSE;
        toning = TRUE; // turn on the autonomous toner
        for (i = 0; i < RESUME_CHECKS; i++) {
            if (signal_detect_OK()) {
                know_still_connected = TRUE;
                if (signal_detect_OK()) { // still true, also the time taken
                    // for this inserts an extra delay which
                    // allows allows for clock frequency
                    // between transmitter and receiver
                }
                receive_OK = TRUE; // to start resuming
                toning = FALSE; // turn off the autonomous toner
                return;
            }
        }
        wait (RESUME_SAMPLING_INTERVAL - TONE_DURATION);
        connected = know_still_connected;
    }
} else { // DS mode, suspended or disabled
    if (!disabled) {
        receive_OK = bias;
        if (receive_OK) return;
    }
    connected = connect_detect; // see if still connected
}
if (!connected) { // disconnection detected
    Beta_mode = FALSE;
    dc_connected = FALSE; // may be immediately latched TRUE again!
    if (int_enable && !port_event) {
        port_event = TRUE;
        if (link_active && LPS)
            PH_EVENT.indication(INTERRUPT);
        else
            PH_EVENT.indication(LINK_ON);
    }
}
}

void disabled_actions {
    if (int_enable && !port_event) {
        port_event = TRUE;
        if (link_active && LPS)
            PH_EVENT.indication(INTERRUPT);
        else
            PH_EVENT.indication(LINK_ON);
    }
    disable_notify = signaled = FALSE;
    disabled = TRUE;
    activate_connect_detect(0); // Enable the connect detect circuit
}

```

```

void resume_actions() {
    while (suspend_in_progress())          // Let any other suspensions complete
        ;                                  // (we may resume those ports later)
    connect_timer = 0;
    if ((int_enable || resume_int) && !port_event) {
        port_event = TRUE;
        if (link_active && LPS)
            PH_EVENT.indication(INTERRUPT);
        else
            PH_EVENT.indication(LINK_ON);
    }
    connect_detect_valid = FALSE;          // Bias renders connect detect circuit useless,
    if !(resume || boundary_node) resume_all_ports()
    else resume = TRUE;                    // Guarantee resume_in_progress() returns TRUE
    if (Beta_mode) {                       // start up and train the port
        bport_active = TRUE;
        while ((connect_timer < (RESUME_SAMPLING_INTERVAL + RECEIVER_INIT_TIME +
-(SYNCHRONIZATION_LENGTH/
-(SYNCHRONIZATION_LENGTH/(BASE_RATE*2**((port_speed-1))))) &&
!bport_sync_ok)
            ;                               // wait until the port tells ut-us it is synchronized
        fault = !bport_sync_ok;            // Resume attempt failed if failed to synchronize
        if (fault) {
            activate_connect_detect(0);
            resume = FALSE;
            return;                          // Resume attempt complete
        }
    } else { // DS mode
        tpBias(1);                          // Generate TpBias
    }
    while (((connect_timer < MAX_BIAS_HANDSHAKE) && !receive_ok) || bus_initialize_active)
        ;                                   // Wait for peer PHY to generate bias (DS mode)
    if (receive_ok) {                       // Connection restored to active state?
        while ((connect_timer < 3 * RESET_DETECT) && !bus_initialize_active)
            ;
        if (!bus_initialize_active) {       // No other node initiated reset?
            if (boundary_node)isbr = TRUE; // Can we arbitrate? Yes, don't wait any longer
            else {
                while ((connect_timer < 7 * RESET_DETECT) && !bus_initialize_active)
                    ;                       // Let's wait a little longer...
                if (!bus_initialize_active) ibr = TRUE; // Sigh! We'll have to use long reset
            }
        }
    }
    fault = !receive_ok;                    // Resume attempt failed if TpReceive_ok is absent
    if (fault)                              // If so, restore usefulness of connect detect circuit
        activate_connect_detect(0);
    resume = FALSE;                          // Resume attempt complete
}

void suspend_initiator_actions() {
    connect_timer = 0;                      // Used to debounce receive_ok or for receive_ok handshake
    if (!suspend) {                        // Unexpected loss of receive_ok?
        suspend = TRUE;                    // Insure suspend_in_progress() returns TRUE
        if (child)                          // Yes, parent still connected?
            isbr = TRUE;                    // Arbitrate for short reset
        else
            ibr = TRUE;                     // Transition to R0 for reset
        while (connect_timer < CONNECT_TIMEOUT)
            ;                               // Time for receive_ok to stabilize
    }
    while ((connect_timer < Receive_OK_HANDSHAKERECEIVE_OK_HANDSHAKE) && receive_ok)
        ;                                   // Wait for suspend target to deassert receive_ok
    fault = receive_ok;                     // Suspend handshake refused by target?
    activate_connect_detect(Receive_OK_HANDSHAKERECEIVE_OK_HANDSHAKE); // Also guarantees

```

```
handshake timing
}
```

```
void suspend_target_actions() {
  int j;
  if (resume_in_progress()) // Other ports resuming?
    resume = TRUE; // OK, do suspend handshake but resume afterwards
  suspend = TRUE; // Insure suspend_in_progress() returns TRUE
  if (portR() == RX_DISABLE_NOTIFY) { // Is our peer PHY going away?
    breq = IMMED_REQ; // Topology change! Reset on other (active) ports
    isbr = TRUE;
  } else if (portR() == RX_SUSPEND && !resume) { // Don't propagate if resume in progress
    for (j = 0; j < NPORT; j++)
      if (active[j]) // Otherwise all active ports become suspend initiators
        suspend[j] = TRUE;
    breq = IMMED_REQ; // Invoke transmitter to propagate TX_SUSPEND
    isbr = TRUE; // Alert link that we're now isolated
  }
  while (portR() == RX_DISABLE_NOTIFY || portR() == RX_SUSPEND)
    ; // Let signals complete before receive_ok handshake
  activate_connect_detect(Receive_OK_HANDSHAKERECEIVE_OK_HANDSHAKE);
}
```

```
void suspended_actions() {
  signaled = suspend = FALSE;
  if (int_enable && !port_event) {
    port_event = TRUE;
    if (link_active && LPS)
      PH_EVENT.indication(INTERRUPT);
    else
      PH_EVENT.indication(LINK_ON);
  }
}
```

```
.
.
+
```