NODE SUSPEND/RESUME ACTIONS
Draft 0.41, Sept. 8, 1997
P1394A/97-0054R0

```
void connection_status() { // Continuously monitor port status in all states
    int i;
    isolated_node = TRUE; // Assume true until first connected port found
    for (i = 0; i < NPORT; i++) {
        isolated_node &= !connected[i];
        if (power_down[i])  {  // con_status[i] valid only while bias generator is off
            if (connection_in_progress[i]) {
                if (!con_status[i])  // con_status is unfiltered connect
                    connection_in_progress[i] = FALSE; // Lost attempted connection
                else if (connect_timer >= (isolated_node) ? 2 * CONNECT_TIMEOUT : CONNECT_TIMEOUT) {
                    connection_in_progress[i] = FALSE;
                    connected[i] = TRUE; // Confirmed connection--Set STATUS_A.con
                    if (isolated_node) // Can we arbitrate?
                        ibr = TRUE; // No, transition to R0 for reset
                    else
                        isbr = TRUE; // Yes, arbitrate for short reset
                }
            } else if (!connected[i]) {
                if (con_status[i]) { // Possible new connection?
                    connect_timer = 0; // Start connect timer
                    connection_in_progress[i] = TRUE;
                }
            } else if (!con_status[i]) { // Disconnect?
                connected[i] = FALSE; // Effective immediately!--Clear STATUS_A.con
                if (child[i]) // Parent still connected?
                    isbr = TRUE; // Yes, arbitrate for short reset
                else
                    ibr = TRUE; // No, transition to R0 for reset
            }
        }
    }
}


void reset_start_actions() { // Transmit BUS_RESET for reset_time on all ports
    int i;
    root = FALSE;
    PH_EVENT.indication(BUS_RESET_START);
    ibr = isbr = FALSE; // Don't replicate resets!
    breq = NO_REQ; // Discard any and all link requests
    child_count = physical_ID = 0;
    bus_initialize_active = TRUE:
    if (gap_count_reset_disable) // First reset since setting gap_count?
        gap_count_reset_disable = FALSE; // If so, leave it as is and arm it for next
    else
        gap_count = 0x3F; // Otherwise, set it to the maximum
    for (i = 0; i < NPORT; i++) {
        if (connected[i])  {
            if (port_suspend_status[i] == queued_suspend_initiator)
                portT(i, TX_SUSPEND);  // Propagate suspend signal
            else if (port_status[i] && disabled[i])
                portT(i, TX_DISABLE);  // Propagate disable
            else if (port_status[i] && !fault[i])
                portT(i, BUS_RESET); // Propagate reset signal
            else
                portT(i, IDLE);  // Propagate idle on suspended and faulted ports
        } else
            portT(i, IDLE); // But only on connected ports
        child[i] = FALSE;
        child_ID_complete[i] = FALSE;
    }
    arb_timer = 0; // Start timer
}
```

```
void receive_actions() {
    boolean end_of_data;
    unsigned bit_count = 0, i, rx_data, tx_speed;
    ack = concatenated_packet = FALSE;
    if (!enab_accel && (breq == FAIR_REQ || breq == PRIORITY_REQ)) {
        breq = NO_REQ; // Cancel the request
        PH_ARB.confirmation(LOST); // And let the link know
    }
    PH_DATA.indication(DATA_PREFIX); // Send notification of bus activity
    start_rx_packet(); // Start up receiver and repeater
    tx_speed = rx_speed;
    PH_DATA.indication(DATA_START, rx_speed); // Send speed indication
    do {
        rx_bit(&rx_data, &end_of_data);
        if (!end_of_data) { // Normal data, send to link layer
            PH_DATA.indication(rx_data);
            if (bit_count < 64) { // Accumulate first 64 bits
                rx_phy_pkt.bits[bit_count] = rx_data;
                ack = (bit_count == 7);
                if (bit_count > 7 && (breq == FAIR_REQ || breq == PRIORITY_REQ)) {
                    breq = NO_REQ; // Fly-by impossible
                    PH_ARB.confirmation(LOST); // Let the link know
                }
            }
            bit_count++;
        }
    } while (!end_of_data);
    if (portR(receive_port) == IDLE) { // Unexpected end of data...
        if (bit_count > 8 && (breq == FAIR_REQ || breq == PRIORITY_REQ)) {
            breq = NO_REQ; // Discard (unless link believes there was an ACK)
            PH_ARB.confirmation(LOST);
        }
        ack = FALSE; // Disable fly-by acceleration
        return;
    }
    switch(portR(receive_port)) { // Send appropriate end of packet indicator
        case RX_DATA_PREFIX:
            concatenated_packet = TRUE;
            PH_DATA.indication(DATA_PREFIX); // Concatenated packet coming
            stop_tx_packet(DATA_PREFIX, rx_speed);
            break;
        case RX_DATA_END:
            rx_speed = S100; // Default when no explicit speed code received
            if (fly_by_OK())
                stop_tx_packet(DATA_PREFIX, tx_speed); // Fly-by concatenation
            else {
                PH_DATA.indication(DATA_END); // Normal end of packet
                stop_tx_packet(DATA_END, tx_speed);
            }
            break;
    }
    if (bit_count == 64) { // We have received a PHY packet
        for (i = 0; i < 32; i++) // Check PHY packet for good format
            if (rx_phy_pkt.bits[i] == rx_phy_pkt.checkBits[i])
                return; // Check bits invalid - ignore packet
        switch(rx_phy_pkt.type) { // Process PHY packets by type
            case 0b00: // PHY config packet
                if (rx_phy_pkt.ext_type == 0)  {
                    ping_response = (rx_phy_pkt.phy_ID == physical_ID);
                else if (rx_phy_pkt.ext_type == 1 || rx_phy_pkt.ext_type == 2) {
                    // port reg write or read
                    if (rx_phy_pkt.phy_ID == physical_ID) (
                        phy_access_response = TRUE ;  // flag to transmit phy register
                        addr  = rx_phy_pkt.reg + 0b1000;      // offset into register
                        phy_reg.port_select = rx_phy_pkt.port;             // port number
                        phy_reg.page_select = 0b000;   // port status registersp
                        if (rx_phy_pkt.ext_type == 1)  {       // reg write
                            switch (addr)   {     //write control registers
                                case 0b1010:                          // Control set
                                    phy_reg[addr] = (rx_phy_pkt.value & 0b10011000);
                                    // mask suspend, disable and reserved bits
                                    if (connected[phy_reg.port_select] && !phy_reg[addr].disable &&
                                        !phy_reg[addr].fault) {
                                        if (rx_phy_pkt.disable)  // disable the port
                                            port_suspend_status[phy_reg.port_select] = queued_disable;
```

```
                                else if (rx_phy_pkt.suspend && !phy_reg[addr].suspend)
                                    // check for suspend initiate
                                    port_suspend_status[phy_reg.port_select] = pending_suspend_initiator;
                                } else    // remove the disable mask
                                    phy_reg[addr] = (rx_phy_pkt.value & 0b10111000);
                            case 0b1011:                          // Control clear
                                phy_reg[addr] = (rx_phy_pkt.value && 0b10111000);
                                // mask suspend and reserved bits
                                if (rx_phy_pkt.suspend && phy_reg[addr].suspend &&
                                    !phy_reg[addr].disable && !phy_reg[addr].fault)
                                    // check for resume initiate
                                    port_suspend_status[phy_reg.port_select] = pending_resume;
                            }
                        } else if (rx_phy_pkt.phy_ID == physical_ID && rx_phy_pkt.ext_type == 3 {
                            // place holder for node resume
                        }
                    }
                } else {
                    if (rx_phy_pkt.R) // Set force_root if address matches
                        force_root = (rx_phy_pkt.address == physical_ID)
                    if (rx_phy_pkt.T) { // Set gap_count unconditionally
                        gap_count = rx_phy_pkt.gap_count;
                        gap_count_reset_disable = TRUE;
                    }
                }
                break;
            case 0b01: // Link-on packet
                if (rx_phy_pkt.address == physical_ID)
                    PH_EVENT.indication(LINK_ON);
                break;
        }
    }
}
```

```
void transmit_actions() {
    end_of_packet = FALSE;
    int bit_count = 0, i;
    PHY_packet rx_phy_pkt, tx_phy_pkt;
    phyData data_to_transmit;
    if (breq == FAIR_REQ)
        arb_enable = FALSE;
    breq = NO_REQ;
    tx_speed = speed; // Copy speed from PH_ARB.request
    receive_port = NPORT; // Impossible port number ==> PHY transmitting
    start_tx_packet(tx_speed); // Send data prefix & speed signal
    if (isbr) // Avoid phantom packets...
        return;
    PH_ARB.confirmation(WON); // Signal grant on Ctl[0:1]
    while (!end_of_packet) {
        PH_CLOCK.indication(); // Tell link to send data
        data_to_transmit = PH_DATA.request(); // Wait for data from the link
        switch(data_to_transmit) {
        case DATA_ONE:
        case DATA_ZERO:
            tx_bit(data_to_transmit);
            if (bit_count < 64) // Accumulate possible PHY packet
                rx_phy_pkt.bits[bit_count] = data_to_transmit;
                bit_count++;
                break;
        case DATA_PREFIX:
            end_of_packet = link_concatenation = TRUE;
            stop_tx_packet(DATA_PREFIX, tx_speed); // MIN_PACKET_SEPARATION guaranteed by
            break; // stop_tx_packet() and subsequent start_tx_packet()
        case DATA_END:
            stop_tx_packet(DATA_END, tx_speed);
            end_of_packet = TRUE; // End of packet indicator
            break;
    }
}
ack = (bit_count == 8); // For acceleration purposes, any 8-bit packet is an ACK
if (bit_count == 64) { // We have transmitted a PHY packet
    for (i = 0; i < 32; i++) // Check PHY packet for good format
        if (tx_phy_pkt.bits[i] == tx_phy_pkt.checkBits[i])
            return; // Check bits invalid - ignore packet
        if (tx_phy_pkt.type == 0b00) {
            if (tx_phy_pkt.ext_type == 0)
                ping_response = (tx_phy_pkt.phy_ID == physical_ID);
            else if (rx_phy_pkt.ext_type == 1 || rx_phy_pkt.ext_type == 2) {
                // port reg write or read
                if (rx_phy_pkt.phy_ID == physical_ID) (
                    phy_access_response = TRUE ;  // flag to transmit phy register
                    addr  = rx_phy_pkt.reg + 0b1000;      // offset into register
                    phy_reg.port_select = rx_phy_pkt.port;             // port number
                    phy_reg.page_select = 0b000;   // port status registersp
                    if (rx_phy_pkt.ext_type == 1)  {       // reg write
                        switch (addr)   {     //write control registers
                            case 0b1010:                          // Control set
                                phy_reg[addr] = (rx_phy_pkt.value & 0b10011000);
                                // mask suspend, disable and reserved bits
                                if (connected[phy_reg.port_select] && !phy_reg[addr].disable &&
                                    !phy_reg[addr].fault) {
                                    if (rx_phy_pkt.disable)  // disable the port
                                        port_suspend_status[phy_reg.port_select] = queued_disable;
                                    else if (rx_phy_pkt.suspend && !phy_reg[addr].suspend)
                                        // check for suspend initiate
                                        port_suspend_status[phy_reg.port_select] = pending_suspend_initiator;
                                } else    // remove the disable mask
                                    phy_reg[addr] = (rx_phy_pkt.value & 0b10111000);
                            case 0b1011:                          // Control clear
                                phy_reg[addr] = (rx_phy_pkt.value && 0b10111000);
                                // mask suspend and reserved bits
                                if (rx_phy_pkt.suspend && phy_reg[addr].suspend &&
                                    !phy_reg[addr].disable && !phy_reg[addr].fault)
                                     // check for resume initiate
                                    port_suspend_status[phy_reg.port_select] = pending_resume;
                        }
                    } else if (rx_phy_pkt.phy_ID == physical_ID && rx_phy_pkt.ext_type == 3 {
                        // place holder for node resume
```

```
                }
            }
        }

    } else {
        if (tx_phy_pkt.R)
            force_root = (tx_phy_pkt.root_ID == physical_ID);
        if (tx_phy_pkt.T) {
            gap_count = tx_phy_pkt.gap_cnt;
            gap_count_reset_disable = TRUE;
        }
    }
  }
}


port_register_transmit_actions()  {   // Entered from A0:Idle if (phy_access_response == TRUE)
    phy_access_response = FALSE;
    start_tx_packet(S100);
    phy_access.dataQuadlet = 0;       //clear all zero fields
    phy_access.phy_ID = physical_ID;  //build phy access response packet
    phy_access.ext_type = 0b11111;
    phy_access.port = port_reg;
    phy_access.reg = addr && 0b0111;     // clear most significant bit
    phy_access.value = phy_reg[addr];
    tx_quadlet (phy_access_pct);        // Transmit phy access response packet
    tx_quadlet (~phy_access_pct);      // NOTE: move “void tx_quadlet()” into global space
    if (port_suspend_status[phy_reg.port_select] == pending_suspend_initiator)  {
        isbr = TRUE;                // set flag for short reset and transmission of TX_SUSPEND
        stop_tx_packet(TX_DATA_PREFIX);    // prepare for short reset
        port_suspend_status[phy_reg.port_select] = queued_suspend_initiator;
    } else if (port_suspend_status[phy_reg.port_select] == queued_disable)  {
        isbr = TRUE;                // set flag for short reset and transmission of TX_DISABLE
        stop_tx_packet(TX_DATA_PREFIX);    // prepare for short reset
        port_suspend_status[phy_reg.port_select] = disabled;  // Set CONTROL_SET.disable
    } else
        stop_tx_packet(TX_DATA_END);    // transmit phy access response only
}


node_suspend_target_actions()  {  // entered from RX:Receive upon detection of RX_SUSPEND
    for (i = 0;i < NPORT;i ++)  {
        if (i == receive_port) {  // suspend target port
            portT (i, IDLE);
            port_suspend_status[i] = queued_suspend_target;
        } else if (connected[i]  && port_status[i] && !disabled[i] && !fault[i])   {
            // make remaining connected ports suspend initiators
            portT(i, TX_SUSPEND);
            port_suspend_status[i] = queued_suspend_initiator;
        } else
            portT (i, IDLE);
    }
    wait_time (SHORT_RESET_TIME);
}


void port_change()  {  // Detects change in port register bits con, bias, suspend, disable, fault
    phy_reg.page_select = 0b000;             //select port registers
    for (i = 0; i < NPORT; i++)
        port_chng = port_ctl_chng = port_stat_chng = FALSE;
        phy_reg.port_select = i;
        conBias_stat = phy_reg.Status_A & 0b00000110;       //select con and bias bits
        control_stat = phy_reg.controlSet & 0b01101000;     //select suspend, diable and fault bits
        if (port_suspend_status[i] == pending_resume)  {   //a write to CONTROL_CLEAR.suspend
            if (!port_status[i] && !disable[i] && !fault[i] && connected[i])  {
                port_suspend_status[i] = queued_resume;
                node_status = RESUME_INITIATOR;      //unless there is an active connected port
                for (j = 0; j < NPORT; j++)
                    if (connected[j] && port_status[j] && !disabled[j] && !fault[j])
                        node_status = BOUNDARY_INITIATOR;
                resume_event = TRUE;
            }
        }
        if (conBias_stat != conBias_old[i])          {      //change in con or bias bits
```

```
            port_chng = port_stat_chng = TRUE;
            if (port_status[i] && !conBias_old.bias && connected[i] && !disabled[i] && !fault[i]) {
                resume_event = TRUE;
                port_suspend_status[i] = queued_resume;
                for (j = 0; j < NPORT; j++)   //check for active ports
                    if (i != j && connected[j] && port_status[j] &&
                    !disabled[j] && !fault[j]) {
                        port_suspend_status[j] = connected;
                        node_status = BOUNDARY_TARGET;
                    }
                if (node_status != BOUNDARY_TARGET)  {
                    //no active ports so resume all suspended ports
                    for (j = 0; j < NPORT; j++)       //check for suspended ports
                        if (i != j && connected[j] && !port_status[j] &&
                        !disabled[j] && !fault[j]) {
                            port_suspend_status[j] = resume_initiator;
                                //sets CONTROL_CLEAR_QUEUED.suspend for that port
                            node_status = RESUME_TARGET;
                        }
                }
            }
            phy_reg.port_select = i;
        }
        if (control_stat != control_old[i])                 //change in suspend, disable or fault bits
            port_ctl_chng = port_chng = TRUE;
        if(port_stat_chng && phy_reg.sleep && !LPS)
            LinkOn = TRUE;                              //wake up the LINK
        if (port_chng)
            phy_reg.controlSet = 0b10000000;            //Set the Chg_int_en bit
            if (port_stat_chng)
                PH_EVENT.ind(phy_reg.Status_A);         //send Status A reg. to LINK
            if (port_ctl_chng)
                PH_EVENT.ind(phy_reg.Control_Set);      //send control reg. to LINK
        control_old[i] = control_stat;
        conBias_old[i] = conBias_stat;
}


void node_resume() {       // wake up a node that has all ports suspended or disabled
    if (!phy_reg.sleep && sleep_old)  {       //detect sleep bit being cleared
        sleep_old = phy_reg.sleep;
        for (i = 0; i < NPORT; i++)           // resume all connected ports
            if (!port_status[i] && connected[i] && !disabled[i] && !fault[i])  {
                port_suspend_status[i] = resume_initiator;
                node_status = RESUME_INITIATOR;
                resume_event = TRUE;
            }
    }
}

void resume_event_actions()   {       //entered if resume_event == TRUE
    switch (node_status)  {
        case BOUNDARY_TARGET:  //transmit TX_REQUEST on resume target port
            for (i = 0; i< NPORT; i++)
                if (port_suspend_status[i] == resume_target)
                    portT(i, TX_REQUEST);     //notify suspend domain that this is a boundary node
            wait_time (2 * RESET_DETECT)
            isbr = TRUE;                      //arbitrate for bus and then reset
            wait (data_to_transmit == BUS_RESET);
            while (data_to_transmit == BUS_RESET)    {
                for (i = 0; i < NPORT; i++)   //transmit bus reset on resume target port
                    if (port_suspend_status[i] == resume_target)
                        portT(i, BUS_RESET);
            }
        case BOUNDARY_INITIATOR:      //this boundary node initiated the resume
            node_timer = 0;                   //start timer
            active_domain = FALSE;
            while (node_timer <= 5 * RESET_DETECT)   {
                for (i = 0; i < NPORT; i++)
                    if (port_suspend_status[i] == resume_initiator && portR(i) == RX_REQUEST) {
                        ibr = TRUE;    //two active domains connected to the suspend domain
                        active_domain = TRUE;
                    }
            }
            if (!active_domain)
```

```
            isbr = TRUE;          //didn't detect another active domain; attempt short reset

        case RESUME_TARGET:  //not a boundary node and resume was initiated by another node
            for (i = 0; i < NPORT; i++)
                if (portR(i) == RX_REQUEST)   {
                    for (j = 0; j < NPORT; j++)       //send TX_REQUEST on resume target port
                        if (port_suspend_status[j] == resume_target)
                            portT(j, TX_REQUEST);
                } else if (portR(i) == BUS_RESET) {
                    for (j = 0; j < NPORT; j++)
                        if (port_suspend_status[j] == resume_target ||
                        port_suspend_status[j] == resume_initiator)
                            portT(j, BUS_RESET);
                } else
                    portT(i, IDLE);

        case RESUME_INITIATOR: //not a boundary node and initiated the resume
            node_timer = 0;                   //start timer
            active_domain = FALSE;
            while (node_timer <= 5 * RESET_DETECT)   {
                for (i = 0; i < NPORT; i++)
                    if (port_suspend_status[i] == resume_initiator && portR(i) == RX_REQUEST) {
                        active_domain[i] = TRUE;
                    }
            }
            one_domain = FALSE;
            for (i = 0; i < NPORT; i++)
                if (active_domain[i])
                    one_domain = TRUE;
                if (active_domain[i] && one_domain)
                    phy_reg.IBR = TRUE;       // more than one active domain; do long reset
    }
}
```