NODE SUSPEND/RESUME ACTIONS
Draft 97-0054R2, Oct. 11, 1997

```
void start_tx_packet(speed) // Send data prefix and speed code
    int i;
    for (i = 0; i < NPORT; i++) {
        if (initiate_disable[i])
            portT(i, TX_DISABLE);      // Notify peer node that this port is disabled
                                       // IDLE must preceed TX_DISABLE
        else {
            portT(i, TX_DATA_PREFIX); // Send data prefix
            speed_OK[i] = (tx_speed <= max_peer_speed[i]);
            if (speed_OK[i])
            portTspeed(i, tx_speed); // Receiver can accept, send speed intentions
        }
    }
    wait_time(SPEED_SIGNAL_LENGTH);
    for (i = 0; i < NPORT; i++)
        portTspeed(i, S100); // Go back to normal signal levels
    wait_time(DATA_PREFIX_TIME); // Finish data prefix
}


boolean disable_coming() {  // Detects RX_DISABLE during Idle state
    int i;
    for (i = 0; i < NPORT; i++)
        if (active[i] && (portR(i) == RX_DISABLE) && PHY_state = A0) {
            receive_port = i;          // Remember port for later
            return (TRUE);             // Found a port that is receiving RX_DISABLE
        }
    return (FALSE);
}


void disable_received_actions() { // received RX_DISABLE while Idle
    suspend_target(receive_port) = TRUE;  // Suspend the port that received RX_DISABLE
    start_tx_packet(S100);     // Send data_prefix on active ports
    initiated_reset = TRUE;    // Reset the active domain.
    reset_time = SHORT_RESET_TIME;     // Go to R0: Reset Start
}


boolean reset_detected() { // Qualify BUS_RESET with port status / history
    int i;
    if (PHY_state == R0 || PHY_State == R1) // Ignore while in reset states themselves
        return(FALSE);
    if (DS_clock) // RX data makes it impossible to detect reset
        return(FALSE);
    for (i = 0; i < NPORT; i++)
        if (portR(i) == BUS_RESET) // More than 20 ns (transient DS == 11)
            if (connection_in_progress[i]) {
                reset_time = 0;
                if (isolated_node())
                    reset_time = SHORT_RESET_TIME;
                else if (connect_timer >= RESET_DETECT)
                    reset_time = RESET_TIME;
                if (reset_time != 0) {
                    connection_in_progress[i] = FALSE;
                    connected[i] = TRUE
                    return(TRUE);
                }
            } else if (active[i]) {
                reset_time = (PHY_state == RX) ? SHORT_RESET_TIME : RESET_TIME;
                return(TRUE);
            } else if (initiate_resume[i]) {
                switch (resume_status) {
                    case BOUNDARY_INITIATOR:
                        reset_time = RESET_TIME;      // Probably another boundary node
                    case RESUME_NODE:
                        reset_time = SHORT_RESET_TIME; // Maybe only one boundary node, try short reset
                }
                return(TRUE);
            } else if (resume_target[i]) {
                switch (resume_status) {
                    case: RESUME_NODE:
```

```
                    reset_time = SHORT_RESET_TIME; // None or one boundary nodes
                case: BOUNDARY_TARGET:
                    reset_time = RESET_TIME;      // Took too long to do arbitrated reset
            }
            return(TRUE);
        }
    return(FALSE);
}


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;
    node_status = NULL;
    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 (active[i] && initiate_suspend[i])
            portT(i, TX_SUSPEND);  // Propagate suspend signal
        else if (initiate_disable[i]) {
            portT(i, IDLE);  // Propagate IDLE while port drives TpBias low
            initiate_disable = FALSE;
            disable[i] = TRUE;
        } else if (active[i])
            portT(i, BUS_RESET); // Propagate reset signal on active ports
        else if (port_status[i] && (initiate_resume[i] || resume_target[i])) {
            portT(i, BUS_RESET); // Propagate reset signal on resuming ports
            active_en[i] = TRUE;      // Port transitions to active state
            resume_status = NULL;      // clear resume status
        } else
            portT(i, IDLE); // But only on active or resuming ports
        child[i] = FALSE;
        child_ID_complete[i] = FALSE;
    }
    arb_timer = 0; // Start timer
}


void reset_wait_actions() { // Transmit IDLE, set suspend and disable flags
    int i;
    for (i = 0; i < NPORT; i++)
        if (initiate_suspend[i]) {
            initiate_suspend[i] = FALSE;
            suspend_initiator[i] = TRUE;  // port has been active until end of TX_SUSPEND
        }
        portT(i, IDLE);
    arb_timer = 0; // Restart timer
}

void tree_ID_start_actions() {
int i, temp_count;
arb_timer = 0; // start timer
while(true) { // loop forever
    temp_count = 0; // temporary child counter
    for (i = 0; i < NPORT; i++)
        if (~active[i] || portR(i) == RX_PARENT_NOTIFY) {
            // when not active or receiving "you are my parent"
            child[i] = true;       // set child flag
            temp_count++;          // & increment counter
            child_count = temp_count; // set current child count
        } // end of forever loop
}


void tx_quadlet(quadlet quad_data) { // Send a quadlet...
    int i;
    breq = NO_REQ;  //cancel any request to keep in sync with LINK
    for (i = 0; i < 32; i++) { // ...a bit at a time
        tx_bit(quad_data & 0x80000000); // From the most significant downwards
        PH_DATA.indication(quad_data & 0x80000000); // Copy our own self-ID packet to the link
        quad_data <<= 1; // Shift to next bit
    }
```

```
}



void receive_actions() {
    boolean end_of_data;
    int j, i;
    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) == BUS_RESET || portR(receive_port) == RX_SUSPEND) {
                    // Data prefix followed by reset or suspend
        ack = FALSE;  // Disable fly-by
        return;
    }
    else 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 == 1 || rx_phy_pkt.ext_type == 2) {  // ext_type 0 is reserved
                    // 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
                        rsv_port = FALSE;
                        for (j = NPORT; j < 31; j++)  // check for reserved and unused port numbers
                            if (rx_phy_pkt.port = j) {
                                addr = 0;
                                rsv_port = TRUE;       // reserved port
                            }
                if (rx_phy_pkt.port == 31)            // node register
                    addr = rx_phy_pkt.reg;  // not a port register, read node register
                for (j = 12; j < 16; j++)
                    if (addr == j)
                        rsv_port = TRUE;       // reserved register
                j = 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-- 0 has no effect
                        case 0b1010:                          // Control set
                            phy_reg[addr] = (rx_phy_pkt.value & 0b10011000);
                            // mask suspend, disable and reserved bits
                            if (active[j]) {
                                if (rx_phy_pkt.disable)  // disable the port
                                    initiate_disable[j] = TRUE;
                                else if (rx_phy_pkt.suspend)
                                    // check for suspend initiate
                                    initiate_suspend[j] = TRUE;
                            } 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 (connected[j] && !port_bias[j] && !disable[j] && !fault[j])
                                // check for a suspended port
                                initiate_resume[j] = rx_phy_pkt.suspend; // resume if bit set
                    }
                }
            }
        } else if (rx_phy_pkt.phy_ID == physical_ID && rx_phy_pkt.ext_type == 3
            resume_packet = TRUE;
        } 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, j;
    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 == 1 || tx_phy_pkt.ext_type == 2) {  // ext_type 0 is reserved
                // port reg write or read
                if (tx_phy_pkt.phy_ID == physical_ID) (
                    phy_access_response = TRUE ;  // flag to transmit phy register
                    addr  = tx_phy_pkt.reg + 0b1000;      // offset into register
                    rsv_port = FALSE;
                    for (j = NPORT; j < 31; j++)  // check for reserved and unused port numbers
                        if (tx_phy_pkt.port = j) {
                            addr = 0;
                            rsv_port = TRUE;   // reserved port
                        }
                    if (tx_phy_pkt.port == 31)        // node register
                        addr = tx_phy_pkt.reg;  // not a port register, read node register
                    for (j = 12; j < 16; j++)
                        if (addr == j)
                            rsv_port = TRUE;   // reserved register
                    j = phy_reg.port_select = tx_phy_pkt.port;             // port number
                    phy_reg.page_select = 0b000;   // select port register space
                    if (tx_phy_pkt.ext_type == 1 && !rsv_port)  {       // reg write
                        switch (addr)   {     //write control registers
                            case 0b1010:                        // Control set
                                phy_reg[addr] = (tx_phy_pkt.value & 0b10011000);
                                // mask suspend, disable and reserved bits
                                if (active[j]) {
                                    if (tx_phy_pkt.disable)  // disable the port
                                        initiate_disable[j] = TRUE;
                                    else if (tx_phy_pkt.suspend)
                                        // check for suspend initiate
                                        initiate_suspend[j] = TRUE;
                                } else    // remove the disable mask
                                    phy_reg[addr] = (tx_phy_pkt.value & 0b10111000);
                            case 0b1011:                        // Control clear
                                phy_reg[addr] = (tx_phy_pkt.value & 0b10111000);
                                // mask suspend and reserved bits
                                if (connected[j] && !port_bias[j] && !disable[j] && !fault[j])
```

```
                                        // check for a suspended port
                                       initiate_resume[j] = tx_phy_pkt.suspend; // resume if bit set
                            }
                        }
                    }
                } else if (tx_phy_pkt.phy_ID == physical_ID && tx_phy_pkt.ext_type == 3
                    resume_packet = TRUE;

            } 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;
    PH_DATA.indication(DATA_PREFIX); // Notify LINK that data is coming
    start_tx_packet(S100);
    PH_DATA.indication(DATA_START, S100); // Send speed indication
    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];
    if (rsv_port)                      // reserved port or register
        phy_access.value = 0b00000000;
    tx_quadlet (phy_access);         // Transmit phy access response packet
    tx_quadlet (~phy_access);       // NOTE: move “void tx_quadlet()” into global space
    PH_DATA.indication(DATA_END);
    stop_tx_packet(TX_DATA_END);
    if (initiate_suspend[phy_reg.port_select] ||  initiate_disable[phy_reg.port_select])
        isbr = TRUE;                 // set flag for short reset
}                                      // return to A0: Idle


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);
            suspend_target[i] = TRUE;  // port will do a suspend handshake with its peer
        } else if (active[i])          // make remaining active ports suspend initiators
            initiate_suspend[i] = TRUE;
    }
    initiated_reset = TRUE;
    reset_time = SHORT_RESET_TIME;    // Go to R0: Reset Start
}




void core_resume_actions(){           // wake up all suspended ports and the LINK
                           // entered when sleep() goes FALSE or resume_packet = resume_all = TRUE
                           // or power reset or when a port is to resume
    int j,i;
    wait_event(power_reset | port_resume() | resume_all | resume_packet);
    core_power = TRUE;
    wait (core_functional);    // Wait until clock is running
    resume_timer = 0;          // Start the timer
    resume_status = RESUME_NODE; // assume no active ports
    for (i = 0; i < NPORT; i++)
        if (resume_target[i]) {       // Check for resume target ports
            for (j = 0; j < NPORT; j++)  // Now check for active ports
                if (active[j]) {
                    resume_status = BOUNDARY_TARGET;  // found at least one
                    resume_all = FALSE;        // boundary node-don’t resume suspended ports
                }
        }
    if (resume_packet || resume_all) {
        for (i = 0; i < NPORT; i++)           // resume all suspended and connected ports
            if (!active[i] && connected[i] && !disabled[i] && !fault[i] && !resume_target[i]) {
                port_power[i] = TRUE;  // power up the port
                wait (port_functional[i]);    // wait until port clock is running
                initiate_resume[i] = TRUE;
            }
    }
    resume_packet = resume_all = FALSE;
    if (resume_status == RESUME_NODE) {  // if not boundary target, check for boundary initiator
        for (i = 0; i < NPORT; i++)
            if (initiate_resume[i]) { // Check for initiating ports
                for (j = 0; j < NPORT; j++)  // Now check for active ports
                    if (active[j])
                        resume_status = BOUNDARY_INITIATOR;  // found at least one
            }
    }
    switch (resume_status) {
        case BOUNDARY_TARGET:
            wait (resume_timer == 2 * RESET_DETECT || reset_detected());
            if (!reset_detected())
                isbr = TRUE;           //arbitrate for bus and then reset

        case BOUNDARY_INITIATOR:      //this boundary node initiated the resume
            wait (resume_timer == 5 * RESET_DETECT || reset_detected ());
            if (!reset_detected())
                isbr = TRUE;   //didn't detect another active domain; attempt short reset

        case RESUME_NODE:  //not a boundary node
            wait (resume_timer == 5 * RESET_DETECT || reset_detected ());
            if (!reset_detected()) {
                ibr = TRUE;                       // timed-out issue long reset
}


boolean sleep() {   // put PHY to sleep if all ports power down and no LPS
    int i;
    if (!link_active)  {
        for (i = 0; i < NPORT; i++)  // check for any powered ports
            if (port_power[i])
                return (FALSE);  // returns FALSE if one port is powered
        return (TRUE);
    } else
        return (FALSE);
}


boolean port_resume() {  // TRUE if a port is to initiate a resume
    int i;
    for (i = 0; i < NPORT; i++)  // check for any ports that are to initiate a resume
        if (initiate_resume[i])
            return (TRUE);  // returns TRUE if one port is found
    return (FALSE);
}


boolean isolated_node()  {  // TRUE if no active ports
    for (i = 0; i < NPORT; I++)
        if (active[i])
            return (FALSE);
    return (TRUE);
}
```