



Figure0-1—Beta mode arbitration state machine

Proposed P1394b Arbitration C-Code, revision 1

```
// Suggested BPORT code

enum bportCode { DATA_PREFIX, DATA_END, DATA_END_ERR, SPEED, RESET, ARBRST_GRANT, ARB_RST,
                GRANT, REQUEST, LEGACY_REQUEST };
enum legacyReqType { REQGNT, CHILD_NOTIFY, PARENT_NOTIFY, DISABLE, SUSPEND, IDLE,
                   TRAINING, OPERATION};
enum betaReqType { NONE, NEXT_SLOW, NEXT_FAST, CURR_SLOW, CURR_FAST, LEGACY, CYCLE_START };
enum speedCode { S100, S200, S400, S800, S1600, S3200, NULL };

typedef struct {
    enum betaReqType asynch;
    enum betaReqType isoch;
} beta_request_code;

typedef union { // This type holds the data, request, or speed code
    char Data; // valid if bportCode is DATA
    beta_request_code req; // valid if bportCode is REQUEST
    legacyReqType sig; // valid if bportCode is LEGACY_REQUEST
    enum speedCode speed; // valid if bportCode is SPEED
} bportData;

bportCode bportT; // type of signal to be sent on port
bportCode bportR; // type of signal received on port
bportData bportTData; // data, request, or speed code to be sent on port
bportData bportRData; // data, request, or speed code to be sent on port

// Suggested Arbitration Code

beta_request_code receive_req[NPORT]; //track latest requests received on each port

typedef union {
    struct {
        union {
            quadlet dataQuadlet;
            dataBit dataBits[32];
            unsigned char dataBytes[4];
            struct { // First self-ID packet
                quadlet type:2;
                quadlet phy_ID:6; // Physical_ID
                quadlet :1; // Always 0 for first self-ID packet
                quadlet L:1; // Link active
                quadlet gap_cnt:6; // Gap count
                quadlet sp:2; // Speed code
                quadlet :2;
                quadlet c:1; // Isochronous resource manager contender
                quadlet pwr:3; // Power class
                quadlet p0:2; // Port 0 connection status
                quadlet p1:2; // Port 1 connection status
                quadlet p2:2; // Port 2 connection status
                quadlet i:1; // Initiated reset
                quadlet m:1; // More self-ID packets...
            };
            struct { // Subsequent self-ID packets
                quadlet :8;
                quadlet ext:1; // Nonzero for second and subsequent self-ID packets
                quadlet n:3; // Sequence number
                quadlet :2;
                quadlet pa:2; // Port connection status...
                quadlet pb:2; // pa pb pc pd pe pf pg ph
                quadlet pc:2; // Self-ID packet 2 P3 P4 P5 P6 P7 P8 P9 P10
                quadlet pd:2; // Self-ID packet 3 P11 P12 P13 P14 P15 --- ---
                quadlet pe:2;
                quadlet pf:2;
                quadlet pg:2;
                quadlet ph:2;
                quadlet :2;
            };
            struct { // PHY configuration packet
                quadlet :2;
                quadlet root_ID:6; // Intended root
            };
        };
    };
};
```

Proposed P1394b Arbitration C-Code, revision 1

```
        quadlet R:1; // If set, root_ID field is valid
        quadlet T:1; // If set, gap_cnt field is valid
        quadlet gap_cnt:6; // Gap count
        quadlet :16;
    };
    struct { // Extended PHY packets (ping and other remote packets)
        quadlet :2;
        quadlet :6; // Physical_ID
        quadlet ext_type:4; // Extended type
        quadlet page:3; // Page_select
        quadlet port:4; // Port_select
        quadlet reg:3; // Register address (add 0b1000 if paged register)
        union {
            quadlet data:8; // Register data (remote reply)
            struct { // Remote confirmation
                quadlet fault:1; // Copies of equivalent PHY register bits...
                quadlet connected:1;
                quadlet bias:1;
                quadlet disabled:1;
                quadlet ok:1; // Confirm command accepted or not
                quadlet cmnd:1; // Remote command
            };
        };
    };
};
union {
    quadlet checkQuadlet;
    dataBit checkBits[32];
};
};
} PHY_PKT;
```

// SHOULD THIS BE COMBINED WITH BPORT RECEIVE ACTIONS?

```
void process_requests() { // continuously running routine on each port to process request signals
    beta_request_code best_req;
    int i,j;

    receive_grant = FALSE;
    arb_reset = FALSE;
    for (i=0; i<NPORT; i++) {
        //gather incoming requests
        switch (bportR[i]) {
            case REQUEST:
                receive_req[i] = bportRData[i].req;
                break;
            case DATA_PREFIX: // Forget about previous requests from this port
            case SPEED:
            case DATA_END:
            case DATA_END_ERR:
                receive_req[i].asynch = NONE;
                receive_req[i].isoch = NONE;
                break;
            case RESET: //NEED TO FURTHER INVESTIGATE RESET HANDLING AND TIMING
                receive_req[i].asynch = NONE;
                receive_req[i].isoch = NONE;
                do_isbr = isbr = TRUE;
                break;
            case LEGACY_REQUEST:
                //need to decide whether isoch or asynch based on timing and update request appropriately
                break;
            case GRANT:
                receive_grant = TRUE;
                break;
            ARBRST_GRANT:
                receive_grant = TRUE;
                arb_reset = TRUE; // need to send ARBRST in grant_actions()
                break;
            case ARBRST:
                arb_reset = TRUE; // need to send ARBRST in idle()
                break;
        }
    }
}
```

Proposed P1394b Arbitration C-Code, revision 1

```
        default:
            break;
    }

    //combine best asynch and isoch request from other ports and own request
    best_req.asynch = own_req.asynch;
    best_req.isoch = own_req.isoch;
    for (j=0; j<NPORT; j++) {
        if (j != i) {
            best_req.asynch = (receive_req[j].asynch > best_req.asynch ? receive_req[j].asynch : best_req.asynch);
            best_req.isoch = (receive_req[j].isoch > best_req.isoch ? receive_req[j].isoch : best_req.isoch);
        }
    }
    if (bportT[i] == REQUEST) // this port is sending a beta request
        bportTData[i].req = best_req; //different for each port
}

}

void idle_actions() {
    while !(new_req || IMM_REQ || receive_grant || data_coming)
        if (arb_reset) {
            PH_DATA.indication(ARBITRATION_RESET_GAP); // Alert link
            if (own_req.asynch == NEXT_FAST) // Update asynch requests
                own_req.asynch = CURR_FAST;
            if (own_req.asynch == NEXT_SLOW)
                own_req.asynch = CURR_SLOW;
            for (i=0; i<NPORT; i++) // Repeat out other ports
                if (active[i] && i != receive_port)
                    bportT[i] = ARBRST;
            arb_reset = FALSE; // mark as done
        } else { //send requests on all ports
            for (i=1; i<NPORT; i++)
                if (active[i])
                    bportT[i] = REQUEST; // each port sends current request state
        }
}

void new_request_actions() { //new request received from link
    //update own_request with new request from link
    //combine latest isoch and asynch requests in own_request
}

void grant_actions() {
    int bip = NPORT; // best isochronous request port number
    int bap = NPORT; // best asynchronous request port number
    int best_port = NPORT; //over all best request port number
    boolean did_arbrst = FALSE;

    // if root, wait for a request
    do {
        //combine best asynch and isoch request from all ports and own request
        best_req.asynch = own_req.asynch;
        best_req.isoch = own_req.isoch;
        for (i=0; i<NPORT; i++) {
            best_req.asynch = (receive_req[i].asynch > best_req.asynch ? receive_req[bap = i].isoch : best_req.asynch);
            best_req.isoch = (receive_req[i].isoch > best_req.isoch ? receive_req[bip = i].isoch : best_req.isoch);
        }
        if ((best_req.asynch < CURR_SLOW) && !did_arbrst) // create arb reset if no current interval requests
            arb_reset = TRUE;
        if (arb_reset && !did_arbrst) {
            PH_DATA.indication(ARBITRATION_RESET_GAP); // Alert link
            if (own_req.asynch == NEXT_FAST) // Update asynch requests
                own_req.asynch = CURR_FAST;
            if (own_req.asynch == NEXT_SLOW)
                own_req.asynch = CURR_SLOW;
            if (root && best_req.asynch == NONE && best_req.isoch == NONE) { // go ahead and send arb_rst
                for (i=0; i<NPORT; i++)
                    bportT[i]=ARBRST;
                arb_reset = FALSE; // mark as done
            }
            did_arbrst = TRUE; // only notify link and update requests in first iteration
        }
    }
}
```

Proposed P1394b Arbitration C-Code, revision 1

```

    }
} while (root && best_req.asynch == NONE && best_req.isoch == NONE);

best_port = NPORT;
if (best_req.asynch == CYCLE_START) {
    best_port = bap;
} else if (best_req.isoch == CURR_FAST || best_req.isoch == CURR_SLOW) {
    best_port = bip;
} else if (new_iso_cycle && (best_req.isoch == NEXT_FAST || best_req.isoch == CURR_SLOW)) {
    // NEED TO DEFINE NEW ISO CYCLE
    best_port = bip;
} else if (best_req.asynch != NONE)
    best_port = bap;
}

if (best_port != NPORT) { // grant the best port
    bportT[best_port] = (arb_reset ? ARBRST_GRANT : GRANT);
    for (i=0; i<NPORT; i++) //send requests on all others
        if (i != best_port)
            bportT[i] = (arb_reset ? ARBRST : REQUEST);
} else if (best_req.isoch != NONE || best_req.asynch != NONE) { // own request is best
    grant_self = TRUE;
    if (arb_reset) // send ARBRST on all ports
        for (i=0; i<NPORT; i++)
            bportT[i] = ARBRST;
} else { // no requests
    bportT[parent_port] = ARBRST_GRANT;
    for (i=0; i<NPORT; i++)
        if (i != parent_port)
            bportT[i] = ARBRST;
}
arb_reset = FALSE;
}

void start_rx_packet() { // Send data prefix and do speed signaling
// currently for beta only environment and doesn't handle all error cases
// S400 and below have different data prefix requirements in mixed environment
int event, i;

    arb_timer = 0;
    bportT[receive_port] = REQUEST; //Immediately begin sending requests on receive port
    event = wait_event(SPEED_SIGNAL_RECEIVED | DATA_STARTED | DATA_END_DETECTED | RESET_DETECTED);
    if ((event & (DATA_END_DETECTED | RESET_DETECTED)) != 0)
        return; // There is no incoming packet
    if ((event & SPEED_SIGNAL_RECEIVED) != 0) { // Send DATA_PREFIX and Repeat the speed signal
        rx_speed = PortRData[receive_port].speed;
        for (i=0; i<NPORT; i++)
            if (active[i] && i != receive_port)
                bportT[i] = DATA_PREFIX; //send out DATA_PREFIX on all other ports
        for (i=0; i<(1<<(port_speed[receive_port]-rx_speed)); i++) // wait long enough to receive one more character
            wait_event(PH_BYTE_CLOCK.indication);
        for (i=0; i<NPORT; i++) { // Repeat the speed signal
            if (active[i] && i != receive_port) {
                speed_OK[i] = (rx_speed <= port_speed[i]);
                if (speed_OK[i]) {
                    bportTData[i].speed = rx_speed; // Receiver can accept
                    bportT[i] = SPEED; // tells port to examine bportTData.speed and send a speed signal
                }
            }
        }
    }
}

void rx_byte(dataByte *data_byte, boolean *end_of_data) {
    int i

    for (i=0; i<(1<<(port_speed[receive_port]-rx_speed)); i++) // wait long enough to receive one more character
        wait_event(PH_BYTE_CLOCK.indication);
    // PH_BYTE_CLOCK is a clock running in the tx clock domain at
    // the phy speed/8, i.e. 100MHz for S800

```

Proposed P1394b Arbitration C-Code, revision 1

```
if (fifo_wr_ptr == fifo_rd_ptr) // FIFO empty?
    *end_of_data = TRUE; // If so, set flag
else {
    *end_of_data = FALSE; // If not, clear flag ...
    *data_byte = FIFO[fifo_rd_ptr]; // ... and get data byte
    ++fifo_rd_ptr %= FIFO_DEPTH; // Advance or wrap FIFO ptr
    for (i=0; i < NPORT; i++) { // Repeat the byte
        if (active[i] && i != receive_port) {
            if (speed_OK[i]) {
                bportTData[i].Data = *data_byte;
                bportT[i] = DATA; // tells port to examine bportTData.Data and send the value
            } else {
                bportT[i] = DATA_PREFIX;
            }
        }
    }
}

void receive_actions() {
// currently doesn't handle several error cases
    unsigned byte_count = 0, i;
    PHY_packet rx_phy_pkt;

    PH_DATA.indication(DATA_PREFIX); // Send notification of bus activity
    start_rx_packet(); // Start up receiver and repeater and repeat data prefix and speed signal, requests on receive_port
    PH_DATA.indication(DATA_START, rx_speed); // Send speed indication
    do {
        rx_byte(&rx_data_byte, &end_of_data); //get character and repeat out other ports
        if (!end_of_data) { // Normal data, send to link layer and repeat
            PH_DATA.indication(data_byte);
            if (byte_count < 8) // Accumulate first 8 bytes
                rx_phy_pkt.bytes[byte_count] = rx_data_byte;
            byte_count++;
        }
    } while (!end_of_data);

    if (!pkt_error) { // Send end of packet indicator
        PH_DATA.indication(DATA_END); // Normal end of packet
        for (i=0; i<NPORT; i++) // Send DATA_END out other ports
            if ((i != receive_port) && active[i])
                bportT[i] = DATA_END;
    } else {
        PH_DATA.indication(DATA_END_ERR); // End of bad packet
        for (i=0; i<NPORT; i++) // Send DATA_END_ERR out other ports
            if ((i != receive_port) && active[i])
                bportT[i] = DATA_END_ERR;
    }

    if ((byte_count == 8) && (rx_phy_pkt.data.Quadlet == ~rx_phy_pkt.check.Quadlet)) // Check PHY packet for good format
        decode_phy_packet(rx_phy_pkt); // Parse valid phy packets
}

void tx_byte(dataByte data_byte) { // Transmit a byte
    int i;

    for (i=0; i<(1<<(port_speed - tx_speed)); i++) // wait one more character period
        wait_event(PH_BYTE_CLOCK.indication);
    // PH_BYTE_CLOCK is a clock running in the tx clock domain at
    // the phy speed/8, i.e. 100MHz for S800
    for (i=0; i < NPORT; i++) {
        if (active[i] && i != receive_port) {
            if (speed_OK[i]) {
                bportTData[i].Data = data_byte;
                bportT[i] = DATA; // tells port to examine bportTData.Data and send the value
            } else {
                bportT[i] = DATA_PREFIX;
            }
        }
    }
}
```

Proposed P1394b Arbitration C-Code, revision 1

```
    }  
}  
  
void start_tx_packet(speedCode speed) { // Send data prefix and do speed signaling  
// currently for beta only environment and doesn't handle error cases  
// S400 and below have different data prefix requirements in mixed environment  
    int signal_port = NPORT, i;  
  
    for (i = 0; i < NPORT; i++)  
        if (active[i])  
            bportT[i] = DATA_PREFIX; //send out DATA_PREFIX on all ports  
    for (i=0; i<(1<<(port_speed - speed)); i++) // wait one character period  
        wait_event(PH_BYTE_CLOCK.indication);  
    for (i = 0; i < NPORT; i++) {  
        if (phy_response) {  
            bportTData[i] = speed; // Almost always S100  
            bportT[i] = SPEED;  
        } else if (disable_notify[i]) {  
            bportT[signal_port=i] = DISABLE_NOTIFY;  
        } else if (suspend[i]) {  
            bportT[signal_port=i] = SUSPEND;  
        } else {  
            bportTData[i] = speed;  
            bportT[i] = SPEED;  
        }  
    }  
}  
}  
  
void transmit_actions() { // Send a packet as link transfers it to the PHY  
    int bit_count = 0, i;  
    boolean end_of_packet = FALSE;  
    phyData data_to_transmit;  
    PHY_packet tx_phy_pkt;  
  
    tx_speed = speed;  
    receive_port = NPORT; // Impossible port number ==> PHY transmitting  
    start_tx_packet(speed); // Send data prefix & speed signal  
    // Assume speed has been set correctly by link from PH_ARB.request  
    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  
        if (data_to_transmit == DATA_END) {  
            end_of_packet = TRUE; // End of packet indicator  
            for (i=0; i<NPORT; i++) // Send DATA_END on all ports  
                bportT[i] = DATA_END;  
        } else {  
            for (i=0; i<NPORT; i++) // Send DATA on all ports  
                tx_byte(data_to_transmit);  
            if (byte_count < 8) // Accumulate possible PHY packet  
                rx_phy_pkt.bytes[byte_count] = data_to_transmit;  
            byte_count++;  
        }  
    }  
    if ((byte_count == 8) && (tx_phy_pkt.data.Quadlet == ~tx_phy_pkt.check.Quadlet)) // Check PHY packet for good format  
        decode_phy_packet(tx_phy_pkt); // Parse valid phy packets  
}
```