1. Error Reporting and Synchronization Recovery

Bporters,

Following on from discussions in Santa Cruz and some recent reflector traffic, I would like to propose in some more detail how we specify error reporting, beta port training and loss of synchronization detection. As I have suggested before, this is based heavily on the fibrechannel methods as far as the loss of sync algorithm is concerned.

The training signals are yet to be defined, but will hopefully be provided in a similar way to the yet to be defined prioritised request format that Dave LaFollette has requested.

The port is only concerned with error rates for the purposes of determining when it has lost sync. The error rate in this case is very large and the fibrechannel algorithm is designed to reveal this.

The port is not concerned with anything other than sync loss e.g. poor but not catastrophic error rates. The port merely reports that an error occurs. A higher level may or may not monitor, average and/or take some action on the basis of the errors that the port reports.

I'd like the task group to review the test herein with a view to refining and adopting these procedures at our meeting in Tempe.

Alistair Coles

1.1 Error reporting

Whenever the port receives an INVALID codeword, it shall increment the value of the port_error_reg register. This 8 bit [TBD] read-only PHY register shall wrap from 255 to zero. The port takes no action based on the value of this register. A codeword shall be considered INVALID if:

a) it does not belong to the set of data codewords nor the set of control codewords, or
b) it has incorrect disparity.

NOTE—The register may be read by higher level applications, which may take action on the basis of the register values. For example, a higher level application might disable a port that reports a large error rate. However, the specification of such functionality is beyond the scope of this standard.

1.2 Port training

When a beta mode port is initially activated, and also whenever loss of synchronization occurs, the port must synchronize itself with the received codeword stream. The port must acquire bit synchronization and also determine the boundary between 10-bit codewords (codeword synchronization).

1.2.1 Loss of sync procedure

The port determines that it has lost synchronization with the received codeword stream by monitoring the validity of received codewords. The port maintains a count of the number of invalid codewords received. Whenever an invalid codeword is received, this count is incremented. Whenever two consecutive valid codewords are received, the count is decremented (to a minimum value of zero). If the invalid codeword count reaches four then the port attempts to resynchronize.

For the purposes of this procedure, the control symbol REQUEST(training) shall be treated in the same way as an INVALID. This provides a means for a connected transmitting port to force the receiving port to enter the synchronization procedure.[see below]

Note - it is envisaged that this 'training request' control symbol may be provided by one of the request types currently under discussion for the 'BOSS' accelerations scheme.

The algorithm used to determine a loss of synchronization is defined by the portions of the bport_receive_actions highlighted below. This does not yet include the handling of received REQUEST(training) symbols, as these are not yet defined.

```
void bport_receive_actions() {

//running continuously on each port - equivalent partly to 1394-1995 decode_bit routine.
//the pkt_speed parameter determines the rate at which a port reports events - during a padded packet the port will only
// report one event per port_speed/pkt_speed.
pad_count=0;
repeat {
	speed_err=0;
	rx_character();
	switch (char_type) {
		case CONTROL:
			if(rx_control_state==SPEEDa | SPEEDb ) {
				if (~pkt) {
					pkt_speed=speed_decode();			//this is the start of a speed signal
					BPORT_SPEED.indication(pkt_speed);
					pkt=1;
					pkt_prefix=1;					//may not have been precedeed by a data prefix
					pad_count=port_speed/pkt_speed;			//reset pad counter
					}
				else {
					if(pad_count == 0) {
						//out of place pad symbol error
						}
					}
				}
			else if(rx_control_state==DATA_PREFIX & pkt_prefix==0) {		//either first data prefix
				pkt_speed=port_speed;					//or end of a packet,
				pkt=0;							//so reset these values
				pkt_prefix=1;
				BPORT_CONTROL.indication(DATA_PREFIX);
				}
			else if((rx_control_state==DATA_END | rx_control_state==DATA_END_ERR) ) {
				pkt=0;							// could be a null packet
				pkt_speed=port_speed;
				BPORT_CONTROL.indication(rx_control_state)
				}
			else if(pad_count==0) {						//i.e. always outside of packet or
				BPORT_CONTROL.indication(rx_control_state);		//once per padded payload in packet
				}
			break;
		case DATA:
			if(~pkt_prefix & ~pkt) {
				//shouldn't get data without a packet prefix first (speed or data prefix) unless its a request type
				if(rx_control_state==REQUEST & type_check==1) {	//type check indicates that the data codeword
					rx_request_type=xor(data_byte, type_mask);	//was in the set of valid type codewords
					}
				else {
					//error!
					}
				}
			else if (~pkt) {
				pkt_speed=S100;						// a packet starting w/out speed signal
				BPORT_SPEED.indication(pkt_speed);			// must be S100
				pkt=1;
				pkt_prefix=0;
				BPORT_DATA.indication(data_byte);
				}
			else {
				BPORT_DATA.indication(data_byte);
				}
			break;
		case INVALID:
			char_check=BAD;
			if (pkt & pad_count==0) {					//if error occurs when data is expected
				BPORT_DATA.indication(error);				//need to report something
				}
```

```
                break;                                              //Last control indication remains valid
            }
        pad_count++;
        if (pad_count>=port_speed/pkt_speed) {
            pad_count=0;
            }
        switch(char_check) {
            case GOOD:
                if(invalid_count>0) {
                    valid_count=valid_count++;
                    if(valid_count>1) {
                        invalid_count=invalid_count-1; // If second consecutive valid then decrement invalid counter
                        valid_count=0;
                        }
                    }
                break;
            default:
                port_error_reg=(port_error_reg==255? 0 : port_error_reg++);     //increment error counter
                invalid_count==invalid_count++;
                valid_count=0;
                break;
            }
        if (invalid_count==4) {
            sync_lost=1;                        // Sync lost on reaching threshold of four, causing receiver to enter
                                                // rx sync lost state
            }
        }
}
```

1.2.2 Synchronization procedure

Whenever the port determines that synchronization has been lost, it transmits a training request signal and attempts to resynchronize with the incoming codeword stream. The training request signal causes the connected port to enter the synchronization procedure and return a training request signal. Once the training request signal is received from the connected port, the port is able to synchronize.

The training request signal, REQUEST(training), shall be TBD.

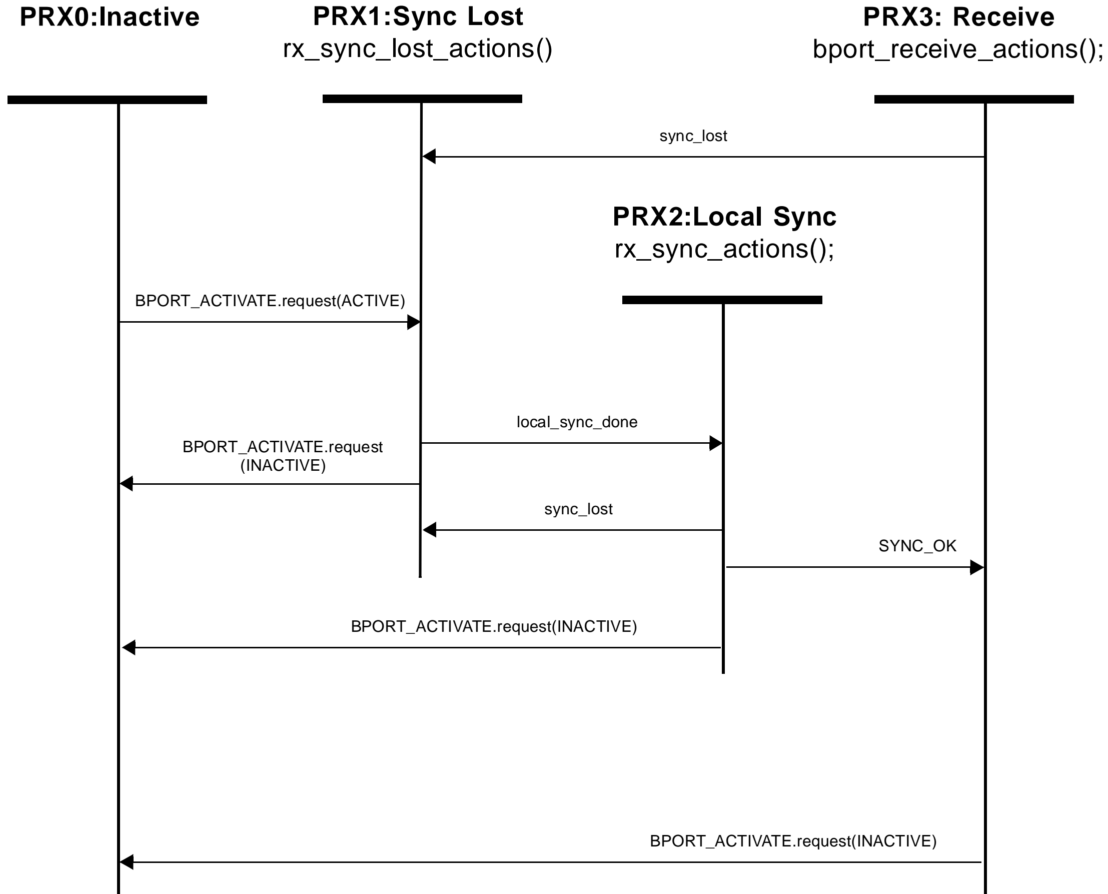
NOTE—The training request signal is treated in the same way as an INVALID symbol by the receiving port, and therefore causes that port to enter the synchronization procedure.

The port shall synchronize within TBD microseconds of receiving the first codeword of a valid training request signal. The port shall consider itself to be synchronized when it has received and successfully decoded at least SYNC_CONFIDENCE consecutive training request symbols or when it has received and successfully decoded at least SYNC_CONFIDENCE consecutive operation request symbols. Upon achieving synchronization, the port shall begin to transmit an operation request signal.

The operation request signal, REQUEST(operation) shall be TBD.

The port shall continue to transmit an operation request signal until it has received and successfully decoded at least 2*SYNC_CONFIDENCE consecutive operation request symbols since the beginning of sending an operation request signal, or it is deactivated by the connection management function. The port shall then exit the synchronization procedure and indicate that it has achieved synchronization by setting the sync_ok register.

1.2.2.1 Receiver synchronization state machine



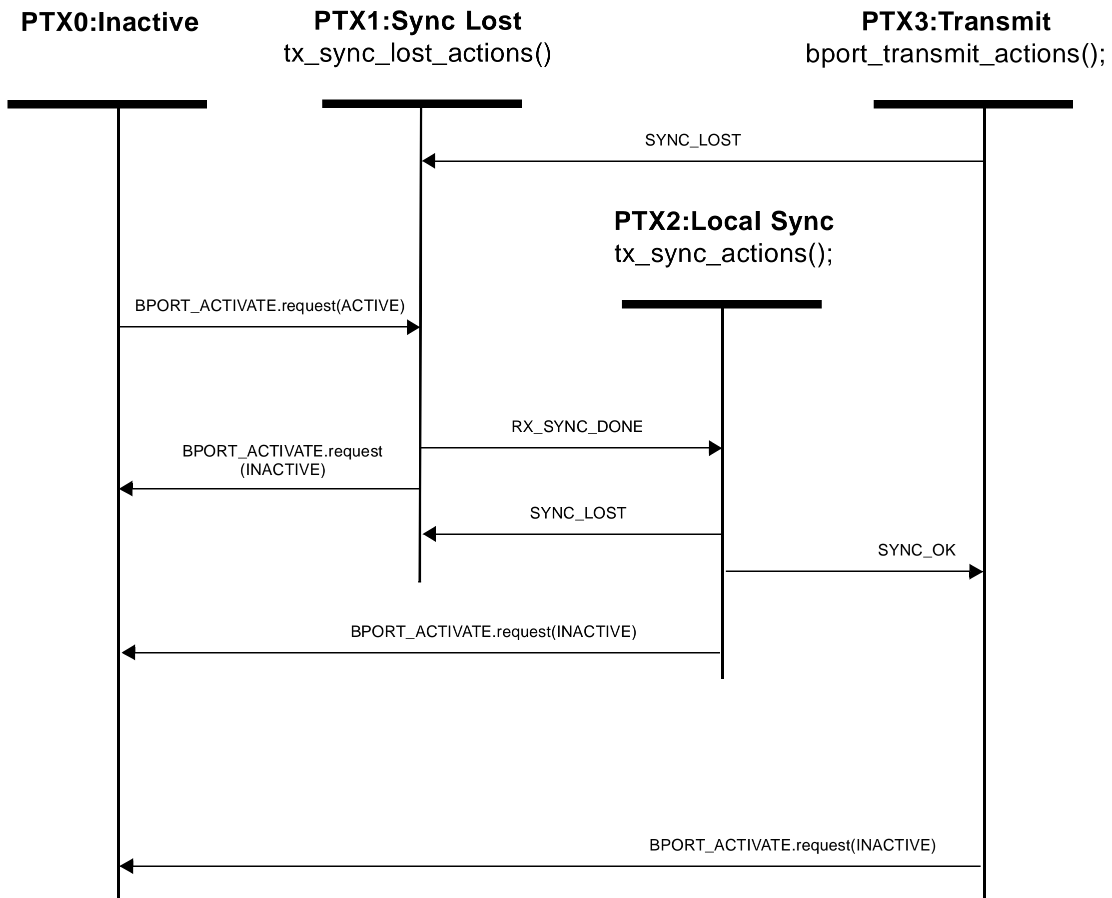
void **rx_sync_lost_actions**() {

```
signal(SYNC_LOST);              //Causes transmitter to send REQUEST(training)
                                //Indicates to receiver that it should acquire bit, codeword and scrambler sync.
                                // Method of acquiring sync. is beyond scope of this standard

                                //Following routine checks for a sequence of valid REQUEST(training) or REQUEST(operation)
                                //symbols to occur before receiver transitions to local_sync_done state.
sync_counter=0;
local_sync_done=0;
while(~local_sync_done) {
        rx_character();
        if (char_type==CONTROL) {
                switch(rx_control_state) {
                        case REQUEST(training):
                                if (last_rx_control_state==REQUEST(training)) {         //i.e. part of a consecutive stream
                                        sync_counter++;
                                        }
                                else {
                                        sync_counter=0;                                 // i.e. isolated ocurrence
                                        }
                                break;
                        case REQUEST(operation):
                                if (last_rx_control_state==REQUEST(operation)) {        //i.e. part of a consecutive stream
```

```
                    sync_counter++;
                    }
                else {
                    sync_counter=0;                         // i.e. isolated ocurrence
                    }
                break;
            default:
                sync_counter=0;
                break;
            }

                    //If there have been SYNC_CONFIDENCE consecutive occurences of TRAINING(request) received,
                    // or if there have been SYNC_CONFIDENCE consecutive occurences of TRAINING(operation)
                    //received, then receiver is considered to be synchronized.

        if(sync_counter==SYNC_CONFIDENCE) {
                local_sync_done=1;      //condition for transition to rx_sync_done state.
                sync_counter=0;
                }
            }
        }
    else {                                  //If an INVALID or data character is received, reset counter
        sync_counter=0;
        }
    }
}


void rx_sync_actions() {

signal(RX_SYNC_DONE);   //causes transmitter to send REQUEST(operation) and indicates to receiver that lock has been achieved

                    //Following routine checks for a sequence of valid REQUEST(operation)
                    //symbols. This indicates that the connected port is also synchronized and must
                    //occur before receiver transitions to receive state.
sync_counter=0;
sync_test=0;
while(~sync_test) {
    rx_character();
    if (char_type==CONTROL) {
        switch(rx_control_state) {
            case REQUEST(training):
                sync_counter=0;
                break;
            case REQUEST(operation):
                if (last_rx_control_state==REQUEST(operation)) {
                    sync_counter++;
                    }
                else {
                    sync_counter=0;
                    }
                break;
            default:
                sync_lost=1;
            }
        if(sync_counter==SYNC_CONFIDENCE) {
            sync_test=1;
            }
        }
    else {                                  //If an INVALID or data character is received, then return to sync_lost state
        sync_lost=1;
        }
    }

                    //Once connected port is seen to be synchronized, allow some extra time for remote port to receive
```

```
                                        //sufficient operation requests.
sync_counter=0;
for(sync_counter==0; sync_counter<SYNC_CONFIDENCE; sync_counter++) {
        rx_character();
        }
signal(SYNC_OK);                        //This causes transmitter to resume normal operation and receiver transitions to receive state.
}
```

1.2.2.2 Transmitter synchronization state machine



```
void bport_transmit_actions() {

while(~SYNC_LOST & ~INACTIVE) {
        switch(bport_transmit_type) {
                case CONTROL:
                        tx_character(CONTROL, tx_control_state, null);
                        break;
                case DATA:
                        tx_character(DATA, null, tx_data_byte);          //send the data byte
                        for(i=pkt_speed;i<port_speed;i=i+pkt_speed) {
                                tx_character(CONTROL, SPEEDa, null);     //send padding if required
                                }
                        break;
                case SPEED_SIGNAL:
                        service_speed_request(pkt_speed, port_speed);
                        break;
                }
        }
}


void service_speed_request() {
if (pkt_speed > port_speed | pkt_speed==S100) {          //if packet is faster than port then don't send any speed signal
        tx_character(CONTROL, DATA_PREFIX, null);
        }
else {j=pkt_speed;                                       //otherwise send appropriate combination of SPEEDa, SPEEDb
        for(i=pkt_speed;i<=port_speed;i=i+pkt_speed) {
                if(j=port_speed)
                        {tx_character(CONTROL, SPEEDb, null);
                        }
                else
                        {tx_character(CONTROL, SPEEDa, null);
                        }
                j=j*2;
                }
        }
}
void tx_sync_lost_actions() {

while(~INACTIVE & ~RX_SYNC_DONE) {
        tx_character(CONTROL, REQUEST(training), null);
        }
}

void tx_sync_actions() {

while(~INACTIVE & ~SYNC_OK) {
        tx_character(CONTROL, REQUEST(operation), null);
        }
}
```

```
void tx_character(tx_char_type, tx_control_state, tx_data_byte) {
switch(tx_char_type) {
	case: CONTROL
		switch(tx_control_state) {
			case(IDLE):
				control_sym=0000; break;
			case(TX_REQUEST):
				control_sym=0001; break;
			case(TX_GRANT):
				control_sym=0001; break;
			case(TX_PARENT_NOTIFY):
				control_sym=0010; break;
			case(TX_CHILD_NOTIFY):
				control_sym=0011; break;
			case(TX_IDENT_DONE):
				control_sym=0011; break
			case(SPEEDa):
				control_sym=1101; break
			case(SPEEDb):
				switch(disparity) {
					case(0):			//rds<0
						control_sym=1000;
					case(1):			//rds>0
						control_sym=0100;
					}
				break;
			case(TX_DATA_PREFIX) :
				switch(disparity) {
					case(0):			//rds<0
						control_sym=0101;
					case(1):			//rds>0
						control_sym=1001;
					}
				break;
			case(TX_DATA_END) :
				switch(disparity) {
					case(0):			//rds<0
						control_sym=0110; break;
					case(1):			//rds>0
						control_sym=1010; break
					}
				break;
			case(TX_DATA_END_ERR) :
				switch(disparity) {
					case(0):			//rds<0
						control_sym=0111; break;
					case(1):			//rds>0
						control_sym=1011; break
					}
				break;
			case(BUS_RESET)
				control_sym=0000;
				break;
			case(REQUEST(training)):
								//TBD
				break;
			case(REQUEST(operation)):
								//TBD
				break;
			}
		scram_control=xor(control_sym,scram([7:2:0));		//check scrambler bits****
		character=control_codetable[scram_control];
		break;
		}
	case DATA:
		scram_data=xor(data_value,scram_state);			//scramble the data byte
```

```
        [character,disparity]=data_codetable[scram_data,disparity];     //lookup codeword and update disparity
    }

for(i=0;i<10;i++) {
    PMD_DATA.request(character(i));
    wait_event(BPORT_CLOCK.indication)     //wait for next port bit clock cycle
    }
}
```