Bport interfaces

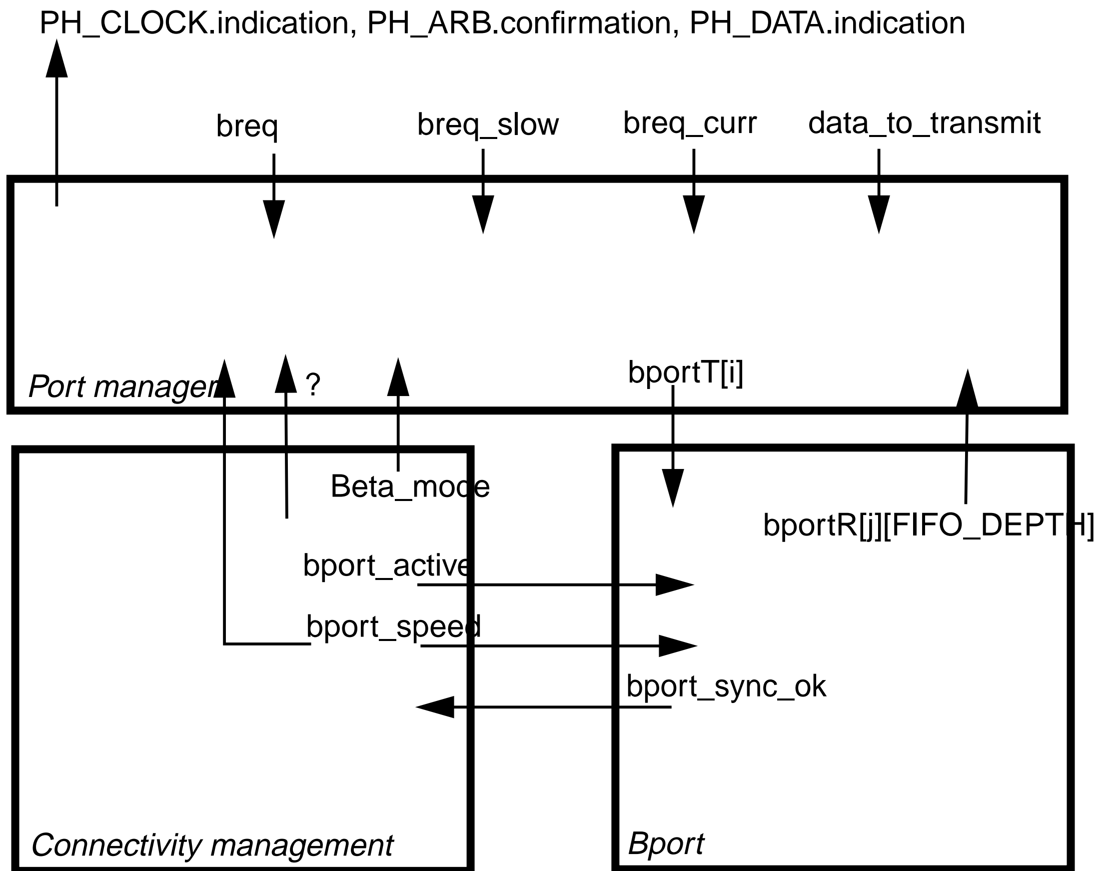
Alistair Coles

Version	Date	Notes
0.1	25/02/99	first draft
0.2	01/03/99	bportT now includes a speedCode member
0.3	31/03/99	Request types renamed ARB_REQUEST and CONFIG_REQUEST. Request type definitions updated to reflect latest BOSS proposal. bport_sync_ok has been added to bport code.

1 Introduction

This document describes the current interfaces betweenthe bport and its neighbour functions. It is recognised that these interfaces may need change. No changes are proposed here but some potential issues and necessary bug fixes are identified.

2 A Picture



3 bportT

The main interface between the bport and the BOSS code is through a shared variable `bportT`. `bportT` is written by the BOSS code to communicate the type of signal the port is to send (i.e. data, control, request type or speed) and the value relating to the signal type. Once written, the variable persists until overwritten. The arb code does not need to rewrite bportT for every symbol the port sends.

bportT contains a member that communicates the effective speed at which each information should be sent. It is the arb code's responsibility to continuously tell the port what the effective speed should be (eg. packet speed should accompany each data bye so that the port does padding correctly, S100 speed should accompany every ARBRST token so that the port stretches it to a length that will enable it to b forwarded to an S100 port). The port no longer remembers the packet speed from the speed signal request during the rest of the packet.

If bportT indicates that a speed code is to be sent, the port generates the sequence of SPEEDa/b codes that comprises the speed signal. The arb code does not need to construct the speed code (which could be different on each port).

The port inserts all padding characters as needed.

bportT is of type bportSymbol, defined as:

```
typedef struct { // This type holds any bport symbol
     bportTag tag;  // The type of symbol
     union { // This part holds symbol data
          char data;            // valid if tag is DATA
          betaCtrl ctrl;        // valid if tag is CTRL
          pktType pkt;          // valid if tag is SPEED
          beta_request_code req;// valid if tag is REQUEST
          legacyReqType sig;    // valid if tag is LEGACY_REQUEST
     };
     speedCode speed; // the effective speed at which the current info is to be sent
} bportSymbol;
```

bportT.tag defines the type of signal to be sent, and is of type bportTag:

```
enum bportTag { DATA, CTRL, SPEED, ARB_REQUEST, CONFIG_REQUEST, INVALID};
```

In the transmit path INVALID would not be used. The second member of bportT is a union of the representations of the various values that the signals may take. The values that each of these signal types may take are defined as:

```
enum pktType { LEGACY, BETA }

enum betaCtrl { DATA_PREFIX, DATA_END, DATA_END_ERR, DATA_PREFIX_ERR, RESET,
ARBRST_GRANT, ARBRST, GRANT };

enum asyncReqType { NONE_EVEN=0, NONE_ODD=1, BORDER_LOW=2, NEXT_EVEN=3, CURRENT=4,
NEXT_ODD=5, BORDER_HIGH=6, CYCLE_START=7 };

enum isochReqType {RESERVED=0, NONE=1, ISOCH_ODD=2, ISOCH_EVEN=3};

enum legacyReqType { TRAINING, STANDBY, CHILD_NOTIFY, PARENT_NOTIFY, DISABLE_NOTIFY,
SUSPEND, OPERATION};

typedef struct {
     enum asyncReqType asynch;
     enum isochReqType isoch;
} beta_request_code;
```

The beta_request_code is itself a structure containing a member for isochronous and asynchronous request types i.e. both aysnc and isoch request info are passed to the port together in a single write to bportT. If either changes, bportT must be rewritten.

Examples:

- Arb code requires DATA_PREFIX to be sent on the port:

```
bportT.tag=CTRL; bportT.ctrl=DATA_PREFIX;
```

- Arb code requires async request for current phase, and no isoch request:

```
bportT.req.async=CURRENT; bportT.req.isoch=NONE; bportT.tag=REQUEST;
```

The third member of the structure defines the speed at which the current information is to be sent. In the case of a speed signal, it defines the particular speed signal that the port should generate. In the case of data it define the amount of padding that may be required. In the case of a control signal it defines the amount of stretching that the port should perform. Note that I am introducing a new value for this speedCode type, DEFAULT. The DEFAULT value is used to tell the port to transmit at its port_speed.

```
enum speedCode { S100, S200, S400, S800, S1600, S3200, NULL, DEFAULT};
```

Issues

- enum betaCtrl, betaReqType and legacyReqType will need updating.
- Not real-world, assumes instantaneous update of buffer.

4 bport_active

bport_active is a boolean variable that is set by the connection management function. All of the operation of the bport transmit functios is conditional upon bport_active being TRUE. However, the value of any interface signal (e.g. the PMD_DATA.request) is not defined for the case when bport_active is FALSE. It is assumed that the actual output of the port is controlled by the connection management function (i.e. that when bport_active is FALSE, the connection management function disconnects the bport output from the port output.

Issues:

- Currently, bport_active does not control the receive function of the bport. This should probably be changed such that the receive function is conditional upon bport_active.

5 port_speed

port_speed is an integer variable that is set by the connection management function. This variable indicates the speed at which the bport should operate (e.g. S800). port_speed should be less than or equal to phy_speed, a variable that indicates the maximum operating speed that a PHY is capable of.

Issues:

- no check in bport code that port_speed <= phy_speed.
- Implication in current code that the max speed of each port on a PHY is equal, and actual speed of individual ports is controlled by phy_speed. What about a PHY that has different max speeds for each port? (see also use of PHY_BIT_CLOCK as base reference for port transmit timing).

6 bportR

bportR is an array of type bportSymbol that represents a FIFO. It is used for passing all receive path information from the port to the arb code. Information is assembled into bportR in a similar manner to that described for bportT, with the addition of the bportTag value of INVALID being permitted.

The FIFO write pointer is incremented during packet reception but not during request reception. This has the effect of allowing request types to be "deleted". Specifically:

The FIFO write pointer starts incrementing when a speed signal is detected (i.e. the speed is decoded, not when a SPEEDa/b is received), or when the first byte of data is received (if no speed signal precedes the packet). The write pointer then increments whenever a new data byte is received. Padding symbols are deleted in the port and never indicated to the arb code.

At the end of a packet, the arb tokens (DATA_END, DATA_PREFIX, ARB_RESET, ARB_RESET_GRANT and GRANT) are indicated once per packet byte period. i.e. although multiple DATA_END symbols may have been sent by the transmitter so that DATA_END lasts a packet byte period, only the first of these symbols causes the DATA_END to be indicated in bportR. The write pointer is incremented for each indication of an arb token.

A result of the FIFO operation is that during packet reception FIFO entries represent events occuring at packet byte time intervals.

The first request type to be received after a packet is written into the FIFO location following the last arb token. Subsequent request notifications are written into the same location, so if the arb code has not read the previous request info before the next arrives then the previous request info is lost ("deleted"). If an isolated arb token arrives (i.e. an arb token separated from a packet), then the first occurence of this token is written into the FIFO and the write pointer is incremented. This ensures that the arb token is not deleted by subsequent request info. Following the first isolated arb token, further tokens are discarded by the PHY until the token value changes or a request types arrives.

Issues:

- The FIFO operation w.r.t. request types is not realisable as it ignores metastability (arb code reading FIFO while FIFO is changing value).
- No memory for request types e.g. if a packet is followed by a single request type and then an arb token, the arb token will delete the request info. Does this matter?
- Code may need some careful thought w.r.t. errors - what behaviour is appropriate when an invalid codeword is received?

7 bport_sync_ok

`bport_sync_ok` is a boolean variable that is written by the bport and read by the connection management function. It is currently represented by a mix of the `signal(SYNC_LOST)` and `signal(SYNC_OK)` functions in the code.

`bport_sync_ok` is assumed to be used by the connection management function to monitor the progress of port training. Connection management function has a timeout on `bport_sync_ok` going TRUE following `bport_active` going TRUE.

`bport_sync_ok` is FALSE during training. `bport_sync_ok` becomes TRUE when a receiver has fully synchronized and has the port has sent some extra operation requests to the peer node to ensure it sees the operation request. `bport_sync_ok` remains TRUE until the port error monitor counter pops and the receive function enters the rx_sync_lost state.

Issues:

- `bport_sync_ok` should be set to FALSE whenever bport_active goes FALSE, so that when the port is turned back on it starts up thinking it is not synchronised.
- Current code does not reflect operation described here (`bport_sync_ok` non-existent!).FIXED 31/03/99