PPE Interface and Functional Specification [1]

Mark R. Swanson
L. Brad Stoller
Terry T. Tateyama

UUCS-95-013

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

August 8, 1995

Abstract

This document describes the interface and functional specification of a Protocol Processing Engine (PPE) for workstation clusters. The PPE is intended to provide the *support* necessary to implement low latency protocols requiring only low resource (cpu and bus bandwidth) consumption.

1 Introduction

We describe the function of and the interface to a device to *aid* in protocol processing for workstation cluster multicomputers: the device is referred to (perhaps a bit inappropriately) as the Protocol Processing Engine (PPE). The protocol base which we envision the PPE supporting is a *sender-based* protocol. The bulk of the actual protocol implementation is intended to be done in (kernel-level) software.

2 Overview

The specification will begin with short descriptions of the majority of the control and status registers of the PPE. The function of these registers will be explained in greater detail in later sections. Next we deal with message transmission, defining the set(s) of registers used to initiate a particular transmission, the data structures that the PPE must interpret, and the expected software-visible behavior entailed in a transmission. A similar discussion of message reception follows. Following this are sections dealing with initialization and fault handling which complete the specification.

[1]This work was supported by a contract from Hewlett Packard, and by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

```
struct ppe_csr0 {
    unsigned
        reset           : 1,
        enable          : 1,
        reserved        :10,
        incarnation     : 4,
        local_node_num  :16;
};
```

Figure 1: The PPE control register.

3 Global PPE Control Registers

A number of registers are used by the software to configure the PPE and to provide global addresses to it. Unless otherwise specified, the registers are assumed to be 32 bits in width. Specific bit assignments within control registers are intended merely as suggestions; aggregation of functions within control registers is also merely suggestive.

Note that additional registers were specified as described in Section A.2.3.

3.1 PPE control register (PCSR0)

The PPE control register (see Figure 1) is set by the kernel and is read-only to the PPE. The fields of PCSR0 have the following meanings:

- reset – when set by the kernel, the PPE should reset to its initial state; see Section 7.
- enable – when set by the kernel, the PPE should process incoming and outgoing messages; when not set, the PPE should refuse any packets ejected from the fabric and refrain from injecting any packets into the fabric; the PPE will complete any injection or ejection ongoing at the time `enable` is reset.
- incarnation – a quantity specified by the kernel which the PPE uses as part of incoming packet validation; see Section 6.1;
- local_node_num – contains the node's unique number within the cluster (16 bits should be adequate).

3.2 PPE status register (PCSR1)

All fields in the PCSR1 (see Figure 2) are to be written by the PPE and are read-only to the kernel (except where noted). The fields have the following meanings:

- ready – the network interface is ready to accept packets from the PPE and the PPE itself is ready to process incoming and outgoing messages.

```
struct ppe_csr1 {
    unsigned
        ready          :1,
        int_hi         :1,
        int_lo         :1,
        idle           :1,
        nbqr_empty     :1,
        nbqr_full      :1,
        reserved_1     :6,
        send_desc_cnt  :4,
        reserved_2     :16,
};
```

Figure 2: The PPE status register.

- int_hi – set by the PPE when the AQR transitions from empty to non-empty; cleared by the software when it is ready to accept more interrupts; this is both a status and control field.
- int_lo – set by the PPE when the NQR transitions from empty to non-empty; cleared by the software when it is ready to accept more interrupts; this is both a status and control field.
- idle – set by the PPE when it is neither injecting nor ejecting a packet; used in conjunction with `enable` in `PCSR0` to allow the software to determine when shared state in PPE memory may be safely modified.
- nbqr_empty – set by the PPE when the `NBQR` goes empty. An interrupt should be generated as a signal to the kernel to check this register and replenish the pointers in the `NBQR`.
- nbqr_full – set by the PPE when the `NBQR` reaches its maximum capacity. Writes to a full `NBQR` on the PPE are ignored.
- send_desc_cnt – the number of send descriptors (see Section 5.2) provided by the PPE.

3.3 Notification List Heads Register (`NLHR`)

The Notification List Heads Register (`NLHR`) – contains the base physical address (in PPE memory) of the table of pointers to notification lists.

3.4 PPE-Maintained Queues (`NQR,AQR`)

The PPE will also present two registers which represent the heads of PPE-maintained queues of interrupt "tokens". The width of the registers and the size of the "tokens" are both 32 bits. The two registers are:

- message event *Notification Queue Register* (NQR); the size of the queue backing NQR should be 256 entries.
- message *Acknowledgment Queue Register* (AQR); the size of the queue backing AQR should be 256 entries.

The PPE will place items on these queues on the occurrence of events described in later sections. Whenever the enqueuing of an item causes a queue to transition from empty to non-empty and the corresponding interrupt bit in PCSR1 is off, the PPE must interrupt the host. A read of one of these queue registers serves to remove the item read from the queue. I.e., reads by the software are the mechanism for draining these queues. Reading from an empty queue should return zero and not otherwise change the state of the queue. The PPE is allowed to stall on an attempt to enqueue an item onto a full queue; it is not allowed to discard queue entries. The queues should all be empty on a reset of the PPE.

3.5 The Software Maintained Queue (NBQR)

The PPE will provide a *Notification Block Queue Register* (NBQR), writable by the software, which should be backed by a queue on the PPE. The size of this queue should be 256 entries. This queue will hold the physical addresses of empty notification structures in host memory.

4 Notification

The PPE provides a general notification mechanism to inform the processor of several kinds of events. The general mechanism is described here; specific details for each case can be found in:

- Section 5.3.2 (Transmission Completion).
- Section 6.2 (Message Reception).
- Section 5.4.2 (Ack Packets).
- Section 8 (Faults).
- Section A.2.6 (Miscellaneous Packets).

The PPE uses a **note_index** to index into a table of notification list heads (see Figure 3). The software initializes the entries of this table to contain unique tokens and pointers to empty notification objects. The software never subsequently writes the **head** fields of the table entries; those pointers are strictly changed only by the PPE thereafter. The software will occasionally change the **token** fields. The software may read the **head** fields of the entries to detect when the PPE has added an item to a notification list.

Although the notification list heads are in PPE memory, the actual notification objects are located in host main memory. The PPE *should* be able to write an entire notification object in one bus transaction, whereas the cpu (and hence the software) would likely require a transaction per word to read the notification from PPE memory.

```
struct ppe_notelist_t {
        note_t *                head;
        struct {
               unsigned         token   :31,
               unsigned         enqueue :1;
        }
};
```

Figure 3: An entry in the notification list table.

```
struct notification {
        short                   type_specific;
        struct _note_control {
               type_specific      :14,
               software           :2;
         } control;
        unsigned                type_specific;
        unsigned                type_specific;
        note_t *                next;
        unsigned                metadata[4]; /* optional */
} *note_t;
```

Figure 4: The common notification entry structure.

The source of the `note_index` varies with the type of event causing the notification. The base of the table is at the address specified in the `NLHR`. The table entry includes a pointer, in the `head` field, to an empty notification object (see Figure 4).

The PPE forms a notification object as follows:

- The fields labeled `type_specific` are filled according to the type of event.
- the `next` field is a pointer to a new empty notification object; it comes from the PPE's notification block queue. The address of this empty notification must also be written back to the location in the PPE Notetable specified by `node_index`.

The PPE procedure described above maintains the invariant that the table entries always point to empty notification objects at the *end* of the notification lists.

The software views any notification with a zero `next` field as empty. When the PPE fills a `next` field with a pointer to a new empty object, the software assumes it can process the now non-empty notification object. Thus, it is required that the PPE always fill in the `next` field of a notification last – when the rest of the fields are valid. The software maintains its own table of notification list heads that parallels the PPE's table, but may point at earlier entries in the lists. Notifications on a given list are always processed in the order in which the PPE posts them. The

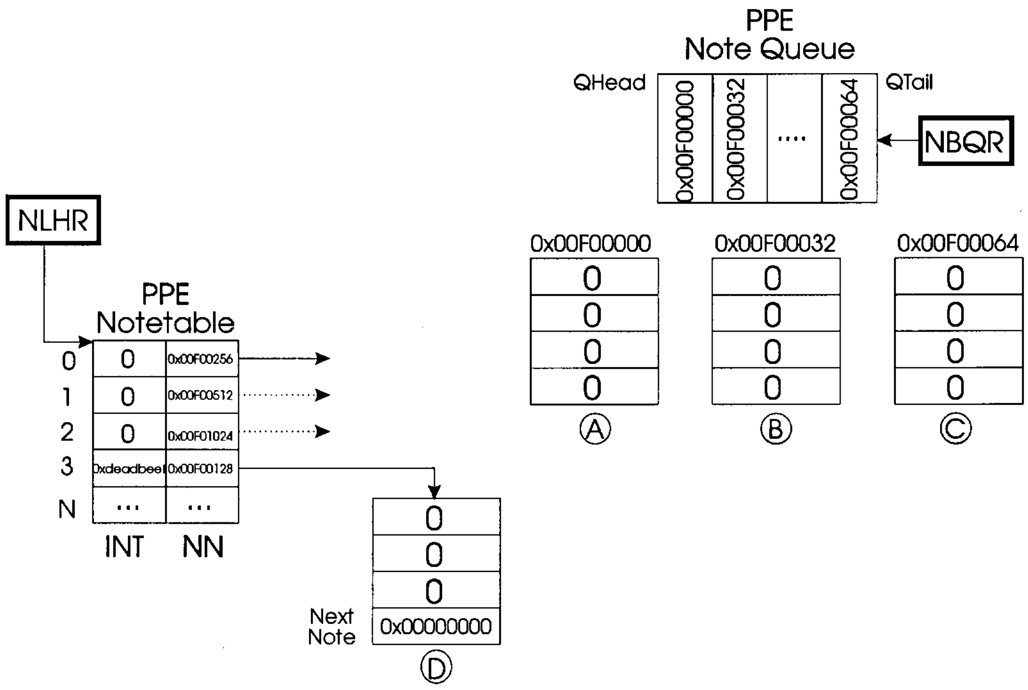

Figure 5: An empty notification list

only synchronization necessary between the software and the PPE is provided by the setting of the **next** field. The software "consumes" notifications; it performs the following actions after processing a given notification:

- the pointer in the object's **next** field is stored into the appropriate slot of the software's notification list table;
- the object's **next** slot is zeroed;
- the address of the object is enqueued on the PPE's notification object queue by writing it to the NBQR register.

Finally, the PPE pushes the **token** field of the **note_list_hdr** onto the message event notification queue *iff* **enqueue** is set in the **note_list_hdr**. If this causes the queue to become non-empty, and if **int_lo** in the PCSR1 is not already set, the PPE issues an interrupt to the CPU. The software will subsequently read NQR to remove entries from the queue for processing. As noted earlier, the software may change these token fields as necessary to obtain correct interrupt behavior.

4.1 An Example of Posting and Consuming a Notification

In Figure 5, the notification list head at index 3 in the table (labeled "PPE Notetable") points at an "empty" notification object, labeled D. At the head of the notification object queue (labeled "PPE Note Queue") is a pointer to another "empty" notification object, labeled A.

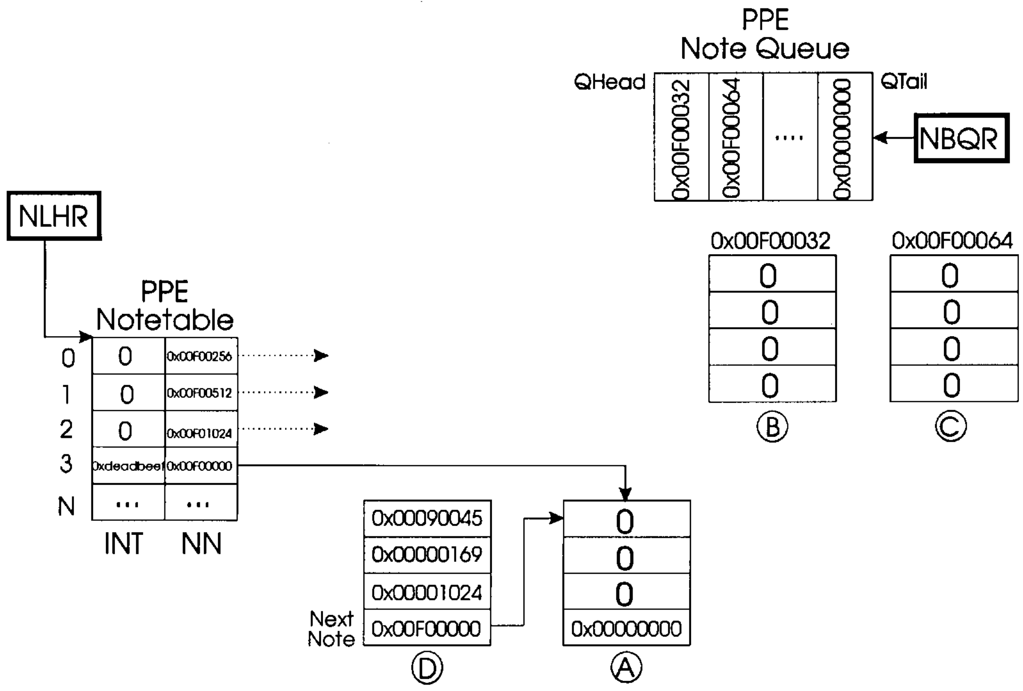

Figure 6: A notification list with one entry

In Figure 6, the PPE has added a notification object to the list at index 3. It formed the notification in the previously empty object, D. The **next** field contains the pointer to A, the empty object that was previously at the head of the notification object queue; A has been removed from that queue by the PPE. Also note that the entry in the 3rd slot of the PPE Notetable has been updated to point at the empty object A.

In Figure 7, the software has consumed the notification. It has returned the object it consumed (D) to the notification object queue, after zeroing its **next** field.

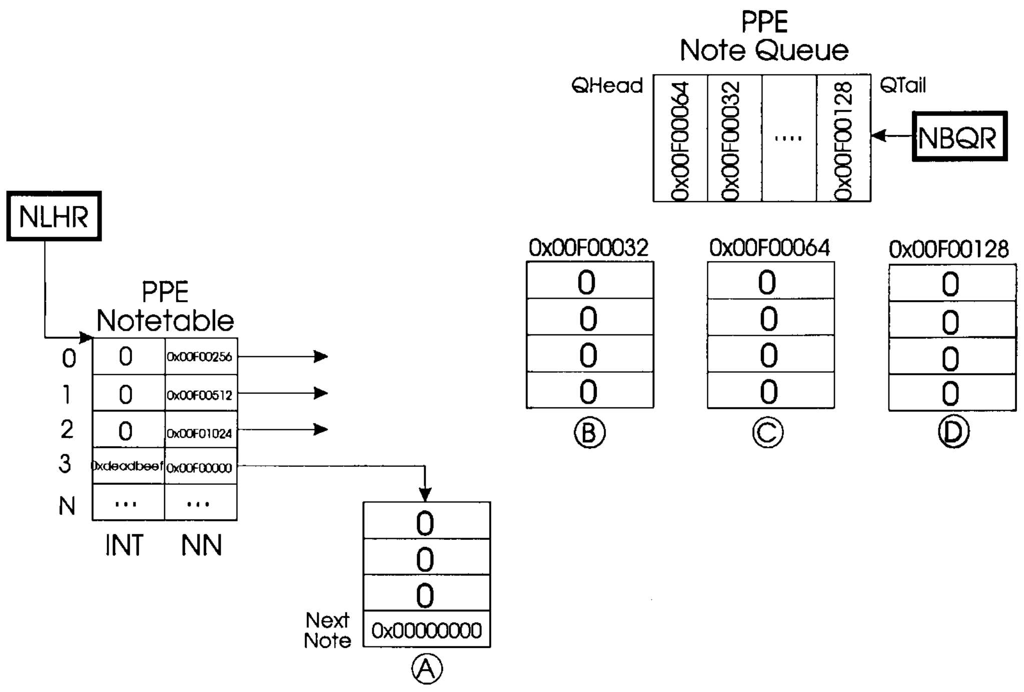

Figure 7: The notification list after the entry is consumed

5 Message Transmission

At the highest level, the PPE should provide the functionality of accepting a pointer (in the form of a physical address) to a message body, a message length in word-aligned bytes, and packet header information. It should then packetize that message as necessary, inject the packets into the network with properly formed headers, and finally notify the sender when the entire message has been injected.

We begin with a proposed format for the packets, since the information it carries is central to the discussions that follow. We then describe the *send descriptor sets* used to control the packet/message transmission processes. Finally, we describe the details of typical message transmission sequences for the DMA, DIO, and special cases.

5.1 Packet Format

The packet header must transport all of the information necessary to implement the sender-based protocols. An example format is given in Figure 8. The size of some fields is open to debate and the location of most fields within the header is immaterial to the protocol implementation. More detailed discussion of the use of the packet header fields will appear in later sections.

5.2 Send Descriptor Register Sets

The interface for message transmission should be implemented by a number of send descriptor register *sets*. We encourage the inclusion of more than one set, and urge the implementation of at least four. We expect fairness in the allocation of packet injection opportunities when multiple sets are provided. For example, round robin injection of one packet from each active send descriptor would be acceptable. More complex scheduling/prioritization of message transmission is the responsibility of the software.

Each descriptor set consists of 16 registers. (see Figure 9) The control registers supply all of the information describing a message to be transmitted.

- The `msg_address` register receives the physical address of the message body to be transmitted. This is either a pointer into main memory (in the DMA case) or a pointer into PPE memory (in the DIO case) depending upon the **`direct_io`** bit in the send descriptor's **`control1`** register.
- The **`address1`** register gets the slot number (**`dst_slot`**) used by the receiving PPE to uniquely identify the receiver channel. It also stores the **`note_index`** the PPE must use when configured to post a notification object after the entire message has been injected into the interconnect fabric.
- The **`control0`** register receives initial values for certain packet header fields, which are used by the receiving PPE, as well as the **`offset`** within the receiving buffer.
- The **`control1`** register contains fields controlling the sending PPE's handling of the message, as well as the message size in bytes. Note that both the **`msg_size`** and receiving buffer **`offset`** are limited to 16 Mbytes.

```
struct pkt_hdr {
        struct {
                unsigned fab_bits       :4
                         dst_node       :12,
                         more_fab_bits  :16;
        } address0;
        struct {
                unsigned dst_slot       :16,
                         src_node       :12,
                         reserved       : 4;
        } address1;
        struct {
                unsigned use_msg_size   : 1,
                         use_msg_offset : 1,
                         ack_pkt        : 1,
                         reserved       : 1,
                         incarnation    : 4,
                         remote_offset  :24;
        } control0;
        struct {
                unsigned reserved       : 4,
                         meta_data_flag : 1,
                         meta_len       : 3,
                         msg_size       :24;
        } control1;
        struct {
                unsigned hdr_checksum   :16,
                         reserved       :16;
        } control2;
        unsigned long    meta_data[4];  /* optional */
}
unsigned                 pkt_data[0..32]; /* arbitrary length */
unsigned short           pkt_checksum;
unsigned short           padding;
```

Figure 8: The packet format.

The one-bit control definitions (in the `control1` register) are as follows:

  - `direct_io` - the PPE should function in a DIO manner; see Section 5.3;
  - `control_pkt` - used to form special payload-free packets; see Section 5.4.1
  - `notify` - when set, the PPE is instructed to form a notification and enqueue it after it sets the `done` bit; see Section 5.3.2;
  - go - when set, the PPE is expected to use the send_descriptor for transmitting; when not set, the PPE should ignore the send_descriptor.
  - `meta_data` - when set, the PPE is expected to include the data from the four metadata fields of the send descriptor in the first outgoing packet. The metadata is to be stored in the receiving PPE's appropriate `rslot` and included in a "message received notification" when the entire message has been reassembled in the receiver's main memory.

- The `metadata` registers can be used to convey additional information about the message (for example, a message type) to be interpreted by the receiver.
- The status registers are associated with each send descriptor by which the PPE communicates status back to the software. The `status` field definitions are as follows:
  - `bytes_to_go` - the number of bytes of the message not yet injected.
  - `busy` - the PPE sets `busy` when it is actively using the send_descriptor to inject a packet; it is set at the initiation of each packet injection and reset when the packet injection completes or aborts.
  - `stalled` - the PPE sets `stalled` whenever the ongoing send has been stalled by the fabric for flow control reasons; the PPE resets it when it again attempts to inject a packet; the PPE must also reset `stalled` whenever the software sets `go` in the associated control register.
  - `done` - the PPE sets `done` when it has finished injecting all packets specified by the send descriptor; the PPE will never inject any packets using a send descriptor with `done` set; the PPE must reset `done` whenever the software sets `go` in the associated `control1` register.

The remaining registers are reserved for future use.

5.3 Message Transmission Mechanisms

There are two mechanisms for configuring the send descriptors to transmit messages (and two special forms of data-less messages, in which case either mechanism can be used). The software can configure the PPE to operate in Direct Memory Access (DMA) mode or DIO mode.

In **DMA mode**, the software writes a pointer to the message body (in the host's main memory) to the `msg_address` of an available send descriptor. The PPE is then responsible for pulling the data across the host's data bus to be packetized and injected into the network fabric.

In **DIO mode**, the software downloads the message body into PPE memory before initiating the packetization/injection process through an available send descriptor. While the `msg_address` field still points to the address of the message body, in this case, the message body has already been downloaded to the PPE's private memory. DIO mode is actually a misnomer. It is more properly viewed as PPE-buffered output mode.

Thus, a message transmission sequence is as follows:

1. (DIO mode only) the kernel allocates a region of PPE memory and copies the message body there.
2. the kernel selects an available send descriptor; the PPE provides no direct support for this, the software performs all necessary accounting.
3. the kernel initializes the appropriate registers of the send descriptor;
4. the kernel writes the `control1` register last, setting go and, if appropriate, **`notify`**. The `msg_size` field must be set with the same write.

The order given is mandatory: the last step only makes sense after all of the other registers are set. When go is set by the kernel, both `stalled` and `done` must be reset by the PPE. It is an error for the software to set go when `busy` is in the set state.

The PPE must also be able to accommodate data-less messages (see Section 5.4), which should result in the formation and injection of a packet consisting only of a header.

5.3.1 Packet Formation

The PPE is expected to packetize messages, inserting appropriate headers and checksums. The bulk of the information needed for the packet header comes from the send descriptor registers.

- `dst_node`, `dst_slot`, and `msg_size` come directly from the send descriptor;
- `remote_offset` is *initialized* from the `offset` field of the `control0` register for the initial packet of the message; as each packet is sent, `remote_offset` must be incremented by the PPE by the size of that packet; the PPE may keep this incremented value in private state or it may use the `control0` register itself;
- `use_msg_size` and `use_msg_offset` of `control0` are used only for the like-named fields of the first packet of a message; the PPE should send zeroes for subsequent packets; this may be accomplished by having the PPE zero these bits in the `control0` register after the first packet is injected.
- `ack_pkt` in the packet header is simply a copy of that field in the `control0` register; the PPE should never modify this bit.

The other information necessary to form a complete packet header comes from other sources:

- `src_node` comes from `local_node_num` in `PCSR0`

- `hdr_checksum` is generated by the PPE on a per-packet basis. *[Header checksums are neither seen nor generated by the software. While we think they are a good idea, we leave them entirely in the hands of the PPE implementors.]*

Depending upon the `direct_io` bit, the body of the packet will either be pulled from the host's main memory across the data bus (DMA mode) or simply read from the PPE's own memory (DIO mode).

As packets are successfully injected, the PPE is responsible for updating the `msg_address`, `control0`, `status1`, and `status2` registers in the corresponding send descriptor.

In order to minimize communication overhead, packets should be as large as the network can reasonably accept.

5.3.2 Transmission Completion

When a message transmission completes, the PPE is expected to post a notification if `notify` is set in the `control1` register. The PPE should use `note_index` field of the send descriptor to locate the appropriate notification list head. (See also Section A.2.7.)

There is currently no type-specific information in a transmission completion notification object.

5.3.3 Control of Ongoing DMA Transmissions

The kernel may reset `go` while DMA is in progress. The kernel might do this to:

- timeshare the available send descriptors,
- to safely free up a wired send buffer,
- or as part of the termination actions for a process.

The PPE *may* complete sending any partially sent packet or it may abort it. In order to facilitate saving the state of an interrupted message transmission, the kernel must be able to determine how much of the message has been transmitted. We define the situation where the kernel requires this information be correct as: when `busy` is reset (and `go` has been reset by the kernel) by the PPE on completion (or abortion) of the current packet injection. The kernel needs "message progress state"; we suggest that this information be provided via the `bytes_to_go` field of the send descriptor status register.

It is the kernel's responsibility to wait for `busy` to be in the reset state before utilizing the message progress state and before manipulating other control bits. This information would also be useful to the kernel in monitoring the progress of very long transmissions; serving this purpose would require updating the message progress state more frequently, perhaps on every packet injection. The software depends on one assumption about the packetization process. Packets must be formed from data from monotonically increasing addresses; i.e., if packet I was taken from address X, packet I+1 will start at location X+N, where N is the size of a packet.

If the software resets **go** while **busy** is in the set state, it is possible that injection of the current packet completes and **done** is set. In that case, the PPE should perform all normal transmission completion actions described in Section 5.3.2.

When the **go** bit is reset and **busy** is set, it is possible that injection of the current packet is/will be stalled. In that case, when the stalled attempt to inject a packet terminates, **stalled** should be set at the same time that **busy** is reset. The PPE should make no further attempts to inject the packet until go is once again set. *[This allows the kernel to perform whatever higher level flow control actions it might implement. Simply allowing busy to be be reset might lead the kernel to attempt continuing a stalled transmission prematurely.]*

5.4 Data-Less Packets

This section describes the two anticipated types of data-less packets the sender-based protocols will use.

5.4.1 Control Packets

When `control_pkt` is set in the send descriptor, the PPE should form a packet header in the normal manner from the send descriptor registers, but it should transmit only the header, regardless of the specified **msg_size**, and immediately set **done** in the status register.

The software currently uses `control_pkt` for only one purpose: when the transmission of a message actually consisted of a number of DMA transmissions and an empty packet with the total message size (along with the **use_msg_size** bit) must be sent to complete transmission of the message.

5.4.2 Ack Packets

The ack-packet is a part of the fast message reception/acknowledgment mechanism. It has no payload, but may carry general data in the **remote_offset** and **msg_size** fields of its header. Reception of such a packet should immediately result in the posting of a "message received notification" (see Figure 11) using **ack_note_index** of the `rslot` and filling in the **msg_offset** and **msg_size** fields from the packet header. The **ack_pkt** bit in the notification should be set to 1.

It is possible that an ack packet will arrive in the middle of another message reception destined for the same `rslot`. The processing of the ack packet should not disturb the message reception by causing changes to `rslot` state.

```
unsigned            *msg_address;
struct {
    unsigned        fab_bits       : 4
                    dst_node       :12,
                    more_fab_bits  :16;
} address0;         /* fabric destination */
struct {
    unsigned        dst_slot       :16,
                    note_index     :16;
} address1;         /* node destination */
struct  {
    unsigned        use_msg_size   : 1,
                    use_msg_offset : 1,
                    ack_pkt        : 1,
                    reserved       : 1,
                    incarnation    : 4,
                    offset         :24;
} control0;
struct {
    unsigned        reserved       : 8,
                    bytes_to_go    :24;
} status1;
struct {
    unsigned        direct_io      : 1,
                    control_pkt    : 1,
                    notify         : 1,
                    go             : 1,
                    meta_data      : 1,
                    meta_cnt       : 3,
                    msg_size;      :24;
} control1;
struct {
    unsigned        busy           : 1,
                    stalled        : 1,
                    done           : 1,
                    reserved       :29;
} status2;
unsigned            reserved;
unsigned            metadata[4];
unsigned            reserved[4];
```

Figure 9: A PPE send descriptor control register set.

```
struct rslot {
                unsigned        buffer_base_phys;
                unsigned        buffer_size;
                short           note_index;
                struct _r_control {
                        u_short valid        :1,
                                indirect     :1,
                                do_acks      :1,
                                notify       :1,
                                reserve      :12;
                } control;
                signed          bytes_to_go;
                unsigned        msg_size;
                unsigned        msg_offset;
                short           ack_note_index;
}
```

Figure 10: The receive slot.

6 Message Reception Mechanisms

At a high level, the function of the PPE for message reception is to accept a (possibly misordered) stream of packets, validate each packet header, and, based on information in the packet header and additional information fetched from PPE memory, deposit the payloads of those packets into appropriate locations in host memory. The PPE is expected to determine when a complete message has arrived and to perform appropriate notification actions. We will first discuss packet reception and then describe message reception and notification.

6.1 Packet Reception

When a packet is received by the PPE, its payload is to be deposited into host memory at a location specified by an `rslot` (see Figure 10).

The `dst_slot` field of the packet header is used as an index into the table of `rslots` in PPE memory. The `rslot` contains state of interest to the PPE for the message in progress as well as information of a more static nature provided by the software. The inclusion of the per-message information within the `rslot` is allowable because the software guarantees that at most one message will be in flight to a given `rslot` at any time.

The specific tasks performed by the PPE on packet arrival are:

1. the PPE reads the packet header from the fabric, computes the checksum, and compares it to the packet header checksum; if they are unequal, the packet is an error packet.
2. it checks whether the `dst_node` is equal to the `local_node_num` in `PCSR0`; if they are not

equal, the packet is an error packet.

3. it range checks the `rslot` index (`dst_slot` in the packet header); if it is greater than the the configured number of `rslots`, the packet is an error packet.
4. it reads the `rslot`, selected by `dst_slot` in the packet header, from PPE-resident memory;
5. it checks whether `valid` is set in the `rslot`; if it is zero, the `rslot` is invalid and the packet is an error packet.
6. it checks whether `incarnation` in the packet header is equal to `incarnation` in the `PCSR0`; if they are not equal, it is an error packet.
7. it checks whether `ack_pkt` is set in the packet header; if it is, the PPE forms and posts a notification and does no further processing with the packet; in particular, the following range checks are not performed; see Section 5.4.2;
8. it compares `remote_offset` from the packet header against `buffer_size` in the `rslot`; if the `remote_offset` is greater, the packet is an error packet.
9. it compares the sum of `remote_offset` and `pkt_size` from the packet header against `buffer_size` in the `rslot`; if the sum is greater, the packet is an error packet.
10. it copies the packet data into main memory at the location formed by the sum of `remote_offset` from the packet header and `buffer_base_phys` from the `rslot`;
11. if `use_msg_size` is set in the packet header, `msg_size` from the packet header is stored into the `msg_size` in the `rslot`; also, `msg_size` is added to `bytes_to_go` of the `rslot`.
12. if `use_msg_offset` is set in the packet header, `offset` from the packet header is stored into the `msg_offset` in the `rslot`.
13. `bytes_to_go` in the `rslot` is decremented by the incoming packet's size and is written back to the `rslot`; when it becomes zero, a complete message has arrived and notification action must be taken (see Section 6.2). `bytes_to_go` must be written back to the `rslot` even when it goes to zero. This ensures that `bytes_to_go` has the proper initial value, zero, for the next message.

6.2 Message Reception

When a complete message has arrived, the PPE is responsible for notifying the host. The notification can involve one or both of the following actions:

1. If `do_acks` is set in the `rslot` of the completed message, the PPE must enqueue the *rslot index* onto the `AQR`.
2. If `notify` is set in the `rslot`, the PPE should form and post a notification object.

```
struct notification {
        short               dst_slot;
        struct _note_control {
            unsigned
                src_node       :12,
                reserved       : 4;
        } control;
        struct {
            unsigned
                reserved       : 4,
                incarnation    : 4,
                offset         :24;
        } control0;
        struct {
            unsigned
                reserved       : 4,
                meta_data      : 1,
                meta_len       : 3,
                msg_size       :24;
        } control1;
        note_t *            next;
        unsigned            metadata[4]; /* optional */
} *note_t;
```

Figure 11: The message reception notification object structure.

The PPE posts a notification using `note_index` of the `rslot`. The specific structure of a message reception notification object is shown in Figure 11. The PPE should form the type-specific parts of the notification object as follows:

- the `dst_slot` field can come from the last (or any) packet of the message;
- the `msg_offset` and `msg_size` must come from the `rslot`;
- The `ack_pkt` bit in the notification should be 0.

7 Reset and Initial State

When the kernel sets **`reset`** in the `PCSR0`, the PPE should initialize itself. Any internal copies of control registers should be refreshed from the software visible registers. Internal state relating to messages in progress, either incoming or outgoing, should be discarded. The queues of interrupt tokens should be reset to an empty state. The status registers for all send descriptors should have `busy` and `stalled` reset and `done` set. Only when all these actions are completed should the PPE set `ready` in the `PCSR1`. Software will be responsible for reinitializing `rslots` and the notification list heads table.

8 Faults

Whenever an error packet is detected, the PPE is expected to post a notification, using zero (0) as the `note_index`. The packet should be discarded without further processing. The specific format for an error packet notification is shown in Figure 12. The PPE should form the type-specific parts of the notification object as follows:

1. error_status - status bits set by the PPE to indicate the type of error;
2. dst_node - for error packets, this *may* not be the local node and thus must be supplied in the notification;
3. src_node - used in error handling to return error messages to the sender.

The software is responsible for taking appropriate actions based on the kind of error the packet represents.

```
struct notification {
        short                   dst_slot;
        struct  {
               invalid_rslot   :1,
               rslot_range     :1,
               bad_dst_node    :1,
               bad_offset      :1,
               bad_size        :1,
               bad_hdr_chksum  :1,
               bad_body_chksum :1,
               bad_incarnation :1,
               reserved        :8;
         } error_status;
        unsigned                filler;
        short                   dst_node;
        short                   src_node;
        note_t *                next;
        unsigned                metadata[4]; /* unused */
} *note_t;
```

Figure 12: The individual error notification entry structure.

Structure	Count/Entries	Size	Space
rslots	1024	32	32K
dio bufs	64	128	8K
notification list heads table	1024	8	8K
PPE queues			12K
other			4K
Total			64K

Figure 13: Example partition of 64K of PPE resident memory.

A Appendixes

A.1 On-board Memory

The provision of a modest amount of memory on-board the PPE *appears* to offer noticeable performance advantages. The advantages come from three sources:

1. reduced waiting by the PPE for GSC+/host memory bus cycles;
2. reduced consumption of host memory bus cycles by the PPE;
3. avoidance of cache interactions for structures shared by PPE and software.

Figure 13 is one possible partition of 64K of memory into pools/tables of structures described in this document. The numbers in the figure would be adequate for a cluster of modest size, perhaps 32 or 64 nodes. The allocations are also somewhat flexible, for example, the sizes of the rslot table and notification list heads table may, in practice, be modified from those shown here. `dio_bufs` have not previously been discussed; they are part of an alternative approach to direct output outlined in Section A.1.3.

A.1.1 Rslots

The inclusion of `rslots` on the PPE simplifies both the PPE and the software. The PPE would otherwise be required to cache some `rslots`; this, in turn, would necessitate some mechanism to allow the kernel to force the PPE to refresh its cache from host memory resident `rslots`. This also simplifies kernel software, since the kernel can write directly to the PPE-resident `rslots` without concern for cache interactions. The frequency and scope of changes to `rslots` is such that the cost of having to do single word writes to IO space should not be a problem. On the other hand, the frequency with which the PPE must access `rslots` – one read of the entire structure and (at least) one write, per packet received – is a powerful motivation for putting `rslots` on the PPE.

A.1.2 Notification List Heads

This table is primarily used by the PPE, after being initialized by the kernel. The PPE will read and write the table on every message complete reception. The kernel will occasionally (infrequently) change the `token` field of an entry; it should never need to read or write the **head** field after the initialization phase.

A.1.3 Buffered Direct Output

This may potentially result in greater latency for the direct output case, since the copy to PPE memory and the injection of the packet are inherently serialized. We expect, however, that direct output will be useful primarily for short messages (in the range of 1 to 2 data cache lines), so the non-overlapped injection time should be insignificant and the bus cycles to write the data to the PPE should remain constant (as compared to the other direct output design).

A.2 Implementation Notes

This section describes the refinements, compromises, and actual parameters used in the physical realization of the PPE.

A.2.1 Multicomputer Architecture

The PPE prototype was designed to interface HP715 workstations with a new interconnect fabric from HP Labs called FedEx.

The HP715 offers a high-capacity, multi-mastered internal bus with an expansion slot.

The FedEx interconnect interface provides two bi-directional high-bandwidth I/O ports which we use as dedicated one-way transmission and reception channels.

A.2.2 PPE Functional Partitioning

This document was primarily partitioned into three main functional blocks for implementation in three separate Actel Field Programmable Gate Arrays (FPGAs). The partitions are shown in Figure 14.

Descriptions for the main functional blocks can be found in

- Section 5 (Message Transmission)
- Section 4 (Notification)
- Section 6 (Message Reception)

The other functional blocks include:

- Slipstream – another HP proprietary ASIC which implements the necessary HPPA functions and protocols over the GSC+ bus.
- Sender Dual-Port RAM – contains the Send Descriptors as well as DIO Buffers.
- Receiver Dual-Port RAM – contains the `Rslots`, the Notification Table, and some scratch pad locations for processing received packets.
- Sync FIFO – adds speed matching for the FedEx network. The CRC16 checksums are calculated in this block because the Actel FPGA used to implement the sender function was too slow.
- RCV Pkt – deposits packets into the Receiver dual-port RAM to be copied into the host's main memory. The CRC16 checksums are checked in this block.

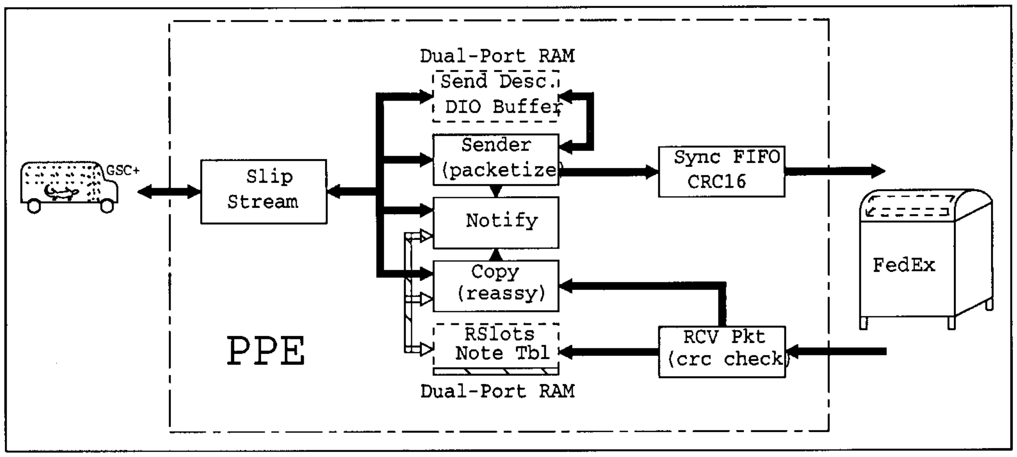

Figure 14: The functional partitions

A.2.3 The Implemented Register/Memory Interface

Because the Slipstream ASIC only passes 8 usable bits of address information to the PPE while the PPE uses 96KB of on-board memory, we added some additional registers and pseudo-registers to allow the kernel software to address the entire PPE memory.

The host accesses the PPE's XMT_RAM by writing a pointer in the `PXR_Ptr` register (at offset address 0x040). Once this pointer is set, there are three methods to access the XMT_RAM:

- Static Address (through `PXR_MEM` at offset 0x044)
- Auto-Increment (through `PXR_MEM_INC` at offset 0x048) for copying large blocks of contiguous data.
- Paged Addresses (at offsets 0x200 through 0x2FC) in which the PPE uses bits 15 through 8 (of `PXR_Ptr`) as a page number and splices in the low byte of the address (the offset) to generate the effective address into the `XMT_RAM`.

The RCV_RAM works much the same way using registers `PRR_Ptr`, `PRR_MEM`, `PRR_MEM_INC`, and addresses 0x300-0x3FC.

Because much of the kernel interaction will involve the maintenance of the send descriptors we've nailed down a page in the XMT_RAM exclusively for paged access to them through offset addresses 0x100-0x1FC.

```
        +---------------------------------------------------------------+
0x000   |R|E|: : : : : : : : : :|incrntn|         local_node_num        | PCSR0
      0 +---------------------------------------------------------------+
0x004   |R|H|L|I|E|F|: : : : : :|dsc_cnt|: : : : : : : : : : : : : : : :| PCSR1
      1 +---------------------------------------------------------------+
0x008   |: : : : : : : : : : : : : : : :|Notification List Heads Reg|0 0| NLHR
      2 +---------------------------------------------------------------+
0x014   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :| NQR
      5 +---------------------------------------------------------------+
0x018   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :| AQR
      6 +---------------------------------------------------------------+
0x01C   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :| NBQR
      7 +---------------------------------------------------------------+
                                        :
        +---------------------------------------------------------------+
0x040   |: : : : : : : : : : : : : : : :|               |           |0 0| PXR_Ptr
     10 +---------------------------------------------------------------+  /Page
0x044   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|: :| PXR_MEM
     11 +---------------------------------------------------------------+
0x048   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|: :| PXR_MEM
     12 +---------------------------------------------------------------+   _INC
                                        :
        +---------------------------------------------------------------+
0x080   |: : : : : : : : : : : : : : : :|               |           |0 0| PRR_Ptr
     20 +---------------------------------------------------------------+  /Page
0x084   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|: :| PRR_MEM
     21 +---------------------------------------------------------------+
0x088   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|: :| PRR_MEM
     22 +---------------------------------------------------------------+   _INC
                                        :
        +---------------------------------------------------------------+
0x100   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :| Send
     40                                 :                                 Desc's
0x1FC   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :| (x4)
     7F +---------------------------------------------------------------+
0x200   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|
     80                                 :                                 XMT_RAM
0x2FC   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|
     BF +---------------------------------------------------------------+
0x300   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|
     C0                                 :                                 RCV_RAM
0x3FC   |: : : : : : : : : : : : : : : : : : : : : : : : : : : : : : : :|
     FF +---------------------------------------------------------------+
```

A.2.4 PPE XMT RAM Memory Map

The PPE XMT RAM Map has been specified as follows:

```
    0x0000 - 0x003C     Send Descriptor 0
    0x0040 - 0x007C     Send Descriptor 1
    0x0080 - 0x00BC     Send Descriptor 2
    0x00C0 - 0x00FC     Send Descriptor 3

    0x0100 - 0x7FFC     DIO Buffers
```

```
Address Map:
         1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
         5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
        +----------------------------+---+
        |0 0 0 0|0 0 0 0|0 0|: : : :|0 0|       SD0
        +----------------------------+---+
        |0 0 0 0|0 0 0 0|0 1|: : : :|0 0|       SD1
        +----------------------------+---+
        |0 0 0 0|0 0 0 0|1 0|: : : :|0 0|       SD2
        +----------------------------+---+
        |0 0 0 0|0 0 0 0|1 1|: : : :|0 0|       SD3
        +----------------------------+---+
        |0| else                     |0 0|       DIO Buffers
        +----------------------------+---+
```

A.2.5 PPE RCV RAM Memory Map

The PPE RCV RAM Map has been specified as follows:

```
    0x0000 - 0xEFFC      RSlots/NLHR (Note List Heads table)
                         (size to be configured by software)

    0xF000 - 0xF3FC      NQR  (256 entries) message Notification Queue
    0xF400 - 0xF7FC      AQR  (256 entries) message Acknowledgment Queue
    0xF800 - 0xFBFC      NBQR (256 entries) Notification Block Queue
    0xFC00 - 0xFF7C
    0xFF80 - 0xFFFC      RCV_PKT

    NOTE:  The 0xF... registers have been reserved for use by the hardware.
```

```
Address Map:
         1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
         5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
        +---------------------------+---+
        |       dst_slot      | off |0 0|       RSlots
        +---------------------------+---+
        |    NLHR + note_index      |  |0 0|       NLHR
        +---------------------------+---+
        |1 1 1 1|0 0|               |0 0|       NQR              (256)
        +---------------------------+---+
        |1 1 1 1|0 1|               |0 0|       AQR              (256)
        +---------------------------+---+
        |1 1 1 1|1 0|               |0 0|       NBQR             (256)
        +---------------------------+---+
                      :
        +---------------------------+---+
        |1 1 1 1|1 1 1 1|0 0 0 : : :|0 0|       RCV_PKT_HEADER  (4/8 words)
        +---------------------------+---+
        |1 1 1 1|1 1 1 1|1 : : : : :|0 0|       RCV_PKT_PAYLOAD (up to 32)
        +---------------------------+---+
```

A.2.6 Miscellaneous Packets

In order to handle non-data type incoming FedEx packets, we've agreed that after the packet has been received in the RCV_RAM packet buffer, the PPE should:

- Post an Error Notification (which contains the packet type from the FedEx header).
- Signal an interrupt through the SlipStream interface.
- Reset the `PCSR0:enable` bit (to prevent additional incoming packets from overwriting the packet buffer before the software processes it).

The software will be able to find the non-data packet payload (without the associated FedEx header) at RCV RAM locations 0xFF80 -¿ 0xFFFC. (The header will be in the initial notification, and can be found at RCV RAM location 0xFF00).

A.2.7 Send Descriptor `note_index` Restriction

One final implementation compromise was made to accommodate the pinout limitations of the chosen Actel FPGAs in the interface between the sender FPGA and the notification FPGA. Since there were only 9 otherwise unused pins between the FPGAs, only 9-bit `note_index`es can be used. `Rslot note_index`es are only limited by the size of the notification table in the PPE's RCV_RAM.