Programmer's Reference Manual for AALI Interface

Part # (MANU0023-02)

(May, 1997)

FORE Systems, Inc.

1000 FORE Drive

Warrendale, PA 15086-7502

Phone: 412-742-4444

FAX: 412-742-7742

Legal Notices

TRADEMARKS

List of Figures

List of Figures

List of Tables

List of Tables

Preface

Technical Support

If your equipment is under warranty or a support contract with FORE Systems, please reference the following information for technical support issues.

In the U.S.A., you can contact FORE Systems' Technical Support by any one of four methods:

1. If you have access to Internet, you may contact FORE Systems' Technical Support via E-Mail at the following address:

   support@fore.com

2. You may FAX your questions to "support" at:

   412-742-7900

3. You may send questions, via U. S. Mail, to the following address:

   FORE Systems, Inc.
   1000 FORE Drive
   Warrendale, PA 15086-7502

4. You may telephone your questions to "support" at:

   1-800-671-FORE(3673)

   or

   412-635-3700

Technical support for non-U.S.A. customers should be handled through your local distributor.

When contacting Technical Support, please be prepared to provide you customer support ID number, the serial number(s) of the product(s) and as much information as possible describing your problem or question.

Typographical Styles

Throughout this manual, specific commands to be entered by the user appear on a separate line in bold typeface. In addition, use of the Enter or Return key is represented as <ENTER>. The following example demonstrates this convention:

cd /usr/FORE <ENTER>

Commands, parameters, menu items, or file names that appear within the text of this manual are represented in the following style: "...the Configure button will access the next menu item."

As in the following example, any messages or code appearing on a screen will appear in `Courier` font to distinguish this text from the rest of the text.

```
.... Are all four conditions true?
```

Important Information Indicators

To call your attention to safety and otherwise important information that must be reviewed to ensure correct and complete installation, as well as to avoid damage to the adapter or your system, FORE Systems utilizes the following *WARNING/CAUTION/NOTE* indicators.

WARNING statements contain information that is critical to the safety of the operator and/or the system. Do not proceed beyond a ***WARNING*** statement until the indicated conditions are fully understood or met. This information could prevent serious injury to the operator and/or damage to the adapter, the system, or currently loaded software, and will be indicated as:

WARNING! Hazardous voltages are present. To lessen the risk of electrical shock and danger to personal health, follow the instructions carefully.

Information contained in **CAUTION** statements is important for proper installation/operation. Compliance with **CAUTION** statements can prevent possible equipment damage and/or loss of data and will be indicated as:

CAUTION You risk damaging your equipment and/or software if you do not follow these instructions.

Information contained in **NOTE** statements has been found important enough to be called to the special attention of the operator and will be set off from the text as follows:

NOTE: Steps 1, 3, and 5 are similar to the installation for the computer type above. Review the previous installation procedure before installation in your particular model.

CHAPTER 1 Introduction

1.1 Introduction

This document specifies an ATM Adaptation Layer Interface (AALI) for host device drivers and signaling protocol modules. The AALI is a programming interface which is built upon the FORE Systems 200-series hardware, specifically, the on-board RAM and the board control register (BCR).

This document refers to a 200-series adapter and the firmware executing on the 200-series as the cell processor (CP).

1.1.1 Overview

The CP implements the AAL4 and AAL5 protocol standards. A null AAL interface is also available, providing access to the ATM layer. This interface allows complete 53-byte ATM cells to be transferred to and from the host.

Communication between the host processor and the CP is primarily through shared CP memory and host memory. CP to host interrupts can also be enabled. The CP accesses host memory using bus master direct memory access (DMA). The host uses memory-mapped accesses to write the CP memory and the BCR. The CP memory only supports 32-bit word accesses.

The CP is capable of scatter / gather DMA, allowing the host buffers to be discontiguous. The buffer descriptors allow cell-payloads to be transferred to and from the host operating systems' native network buffers. The CP does not buffer cell-payloads in CP memory, instead they are transferred to and from the network using fly-by DMA.

CAUTION This document describes a specific version (1.8) of the AALI firmware. The driver developer is cautioned that the versions of header files that are being used to build a driver must match the version of the AALI firmware that is being used.

1.2 CP and Host Interaction

A series of CP resident queues are used by the host to initiate operations:

- PDU receive queue
- PDU transmit queue
- Command queue
- Receive buffer supply queues

Queue accesses by the host are limited to a single write-only access per operation. For example, to transmit a PDU, the host writes a single pointer to the tail of the transmit queue. Host to CP interrupts are not used to notify the CP of a host-initiated operation, instead the CP polls each queue head.

Two types of queues exist. They differ only in the location of the (I/O block). The transmit, receive and buffer supply queues reference a host resident I/O block, while the command queue has a CP resident I/O block.

All queues reference a location in host memory where a queue status entry is maintained. The host initializes the status word before initiating an operation and the CP overwrites the status word on completion. The host uses the value of the status word to determine if an operation has completed and whether a queue is full or not-full before initiating an operation. The status word must be four byte aligned in host memory.

The CP writes stat_complete to the status word on completion and may logically or stat_error with stat_complete if an error has occurred (i.e. (stat_complete | stat_error)).

```
enum status {
            stat_pending= 0x01,/* initial status (written by host)  */
            stat_complete= 0x02,/* completion status (written by cp) */
            stat_free= 0x04,/* initial status (written by host)  */
            stat_error= 0x08/* completion status (written by cp) */
};
typedef enum status Status;
```

1.3 Host Resident Block

The transmit, receive and buffer supply queues have a host resident I/O block. For these three types of queues, a single queue entry has the following structure:

```
struct queue_entry {
Haddr  ioblock;       /* host DMA address of i/o block */
Haddr  stataddr;      /* host DMA address of completion status */
};
typedef volatile struct queue_entry Qcard;
```

The ioblock member can reference a transmit PDU descriptor (Tpd), a receive PDU descriptor (Rpd) or a receive block descriptor (Rbd) located in host memory. The stataddr member references a location in host memory where the CP can write completion status. Both the ioblock and stataddr members reference a location in host memory. This address must be accessible from the CP side of the hosts I/O bus via DMA.

Below is a general description of CP and host interaction. See the section on specific queue types for more details.

- The host maintains a host resident tail pointer to each queue and the CP maintains a CP resident head pointer. For each CP resident queue entry, the host must maintain a host resident status word. Each queue wraps around from the end to the beginning forming a logical ring.
- For each queue entry, the host initializes the host resident status word to stat_free. The host then writes the stataddr member of the CP resident queue entry with the DMA address of that status word. This should be done only once at initialization (i.e. the value of the stataddr member remains constant).
- To initiate an operation, the host first initializes the host resident status word to stat_pending and then writes the ioblock member of the queue entry with the DMA address of the host resident I/O block. The host then advances its head pointer. It is the act of writing the I/O block entry with a new address that signals to the CP that the queue entry is ready for processing.

- The CP notices that the ioblock member of the queue entry has changed and starts the operation associated with that queue. At the end of the operation, the CP will zero the ioblock member of the queue entry. It will then overwrite the status word with stat_complete using the DMA address found in the stataddr member of the queue entry. If the queue is the receive queue or a command queue and interrupts are not masked then the host is interrupted.

NOTE: The CP zeros the ioblock member of the queue entry to recognize a queue empty condition. The host should never read CP memory to determine when the CP has completed an operation but should read the host resident status word. This CP action is documented only to aid in host device driver debugging.

1.3.1 Queue Scheduling

In addition to the various shared memory queues the CP also examines the network receive FIFO for available cells. The receive FIFO is treated as yet another queue by the CP. The receive FIFO is 16K deep and can hold up to 290 host formatted cells.

If cells are available then some number of cells are transferred to host memory before scanning the other queues.

The CP queues are serviced in the following order:

1. Buffer resupply queue
2. Network cell receive queue
3. PDU transmit queue
4. Host command queue

CHAPTER 2 PDU Transmission and Reception

2.1 PDU Transmission

The host transmits a PDU by first initializing a host resident transmit PDU descriptor (Tpd), then setting the host resident status word to stat_pending and finally writing the transmit queue with the DMA address of the Tpd.

The size of a Tpd must be a multiple 32 bytes and be aligned on a 32 byte boundary. The least significant 5 bits of the DMA address are used by the host to encode the size of the Tpd in 32 byte blocks. See Figure 2.1.

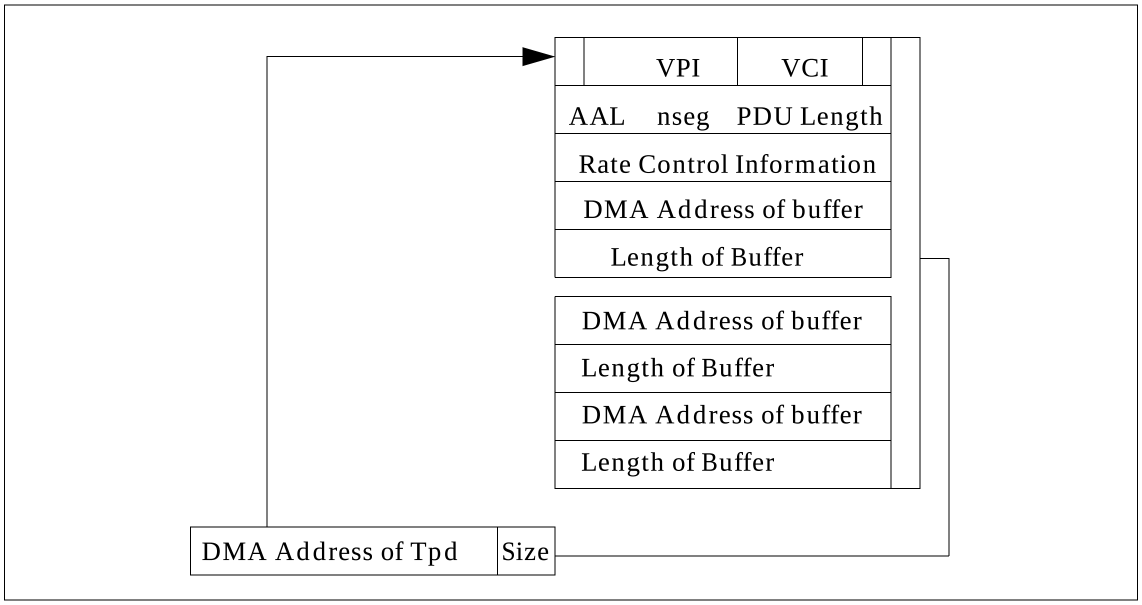


Figure 2.1 - Transmit PDU Descriptor (Tpd) & Tpd DMA Address Encoding

For each PDU segment, data buffering addresses and lengths are encoded in a Tsd. The host must ensure that data for each segment starts on a 32-bit word address and that the length of each segment is a multiple of four bytes.

2.1.1 Tsd Description

Each PDU segment is described by a Tsd.

```
struct transmit_segment {
       Haddr  buffer;   /* transmit buffer DMA address */
       int    length;   /* number of bytes in buffer */
};
typedef struct transmit_segment Tsd;
```

2.1.2 Tpd Description

The entire PDU to be transmitted is described by a `Tpd`. The first word of the `Tpd` contains the VC used to transmit this PDU.

The first four bytes of the `Tpd` (atm_header) are a representation of the ATM Header that will be used to transmit cells for the PDU. It is in the form of a standard ATM Header minus the HEC byte. The fields of this header used during PDU transmission are the VPI and VCI fields.

```
/*
 *  Tsd specification (spec) word format:
 *
 *  31   28 24 23    16 15      8 7       0
 *  +----+---+--------+---------+--------+
 *  |intr|aal|num seg |          PDU len |
 *  +----+---+--------+---------+--------+
 *
 *  Transmit ioblock member encoding (the tpd
 *  dma address written to the transmit queue).
 *
 *  31                   5 4 3 2 1 0
 *  +--------------------+-+-------+
 *  |      ioblock addr   |R|     N |
 *  ---------------------------------
 *
 *  R = reserved (zero)
 *  N = size of Tpd in 8 word blocks
 */
struct tx_pd {
    u_int  atm_header;
    u_int  spec;
    u_int  rc_stream_desc;      /* Number of data cells/interval */
    u_int  pad;
    Tsd    segment[NUM_TXSEG];
};
typedef struct tx_pd Tpd;
```

2.1.3 Transmit Process

When issuing a transmit the host checks the status of any previous transmit requests by examining the host resident status word associated with that request for the value `stat_complete`. If complete, the host can free host resources used for the transmitted PDU. One factor in determining the transmission speed is rate control.

Rate control information is encoded as a thirty-two bit field comprised of two sixteen bit sub-fields which form a ratio of data cells to idle cells. The high order sixteen bits represent the number of data cells to send, and the low order sixteen bits specify the number of idle cells. For example, a value of `0x000a0008` instructs the firmware to transmit ten data cells and eight idle cells per eighteen cells (shown graphically in Figure 2.2). Specifying `0x0` for both sixteen bit fields disables the rate-control mechanism. Rate control is applied on a per-TPD basis.

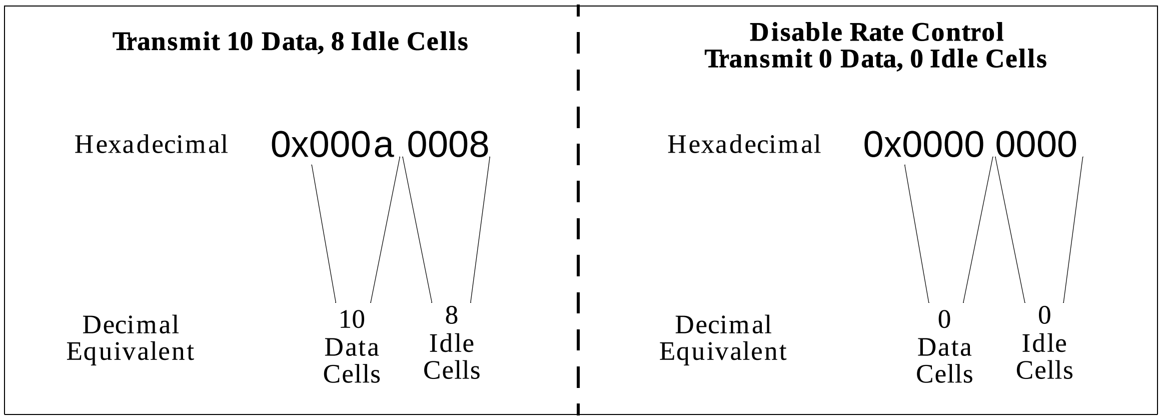


Figure 2.2 - Rate Control as Data and Idle Cells

Figure 2.3 depicts a 3-element transmit queue where the host has issued a transmit for Tpd 3, advanced the request pointer, issued a transmit for Tpd 1, and advanced the request pointer again. The CP has tranmitted the PDU referenced by Tpd 3 and Tpd 1 is still pending.

As you can see, the host has a queue which is needed to keep track of the queue the CP references to emit PDUs. The host transmit queue is set up during initialization to contain pointers to all the relevent pieces of data, of each transmit element. For each QHOST element, there is an associated Tpd, a pointer to a corresponding QCARD element (which exists on the transmit queue of the CP), and a pointer to the status word element (host-resident in this example). Once these essential pieces are set up, no reallocation is performed; i.e., each free Tpd can be thought of as a fixed sized container, holding PDU information. When the Tpd container has been filled, the status word for that Tpd is set to STAT_PENDING, and the DMA address of the Tpd is transferred to the CP's transmit queue. When this DMA address is set, the CP will then recognize that the host has given another PDU for transmit. (Note that the status word DMA addresses within the CP's transmit queue are never changed after initialization.)

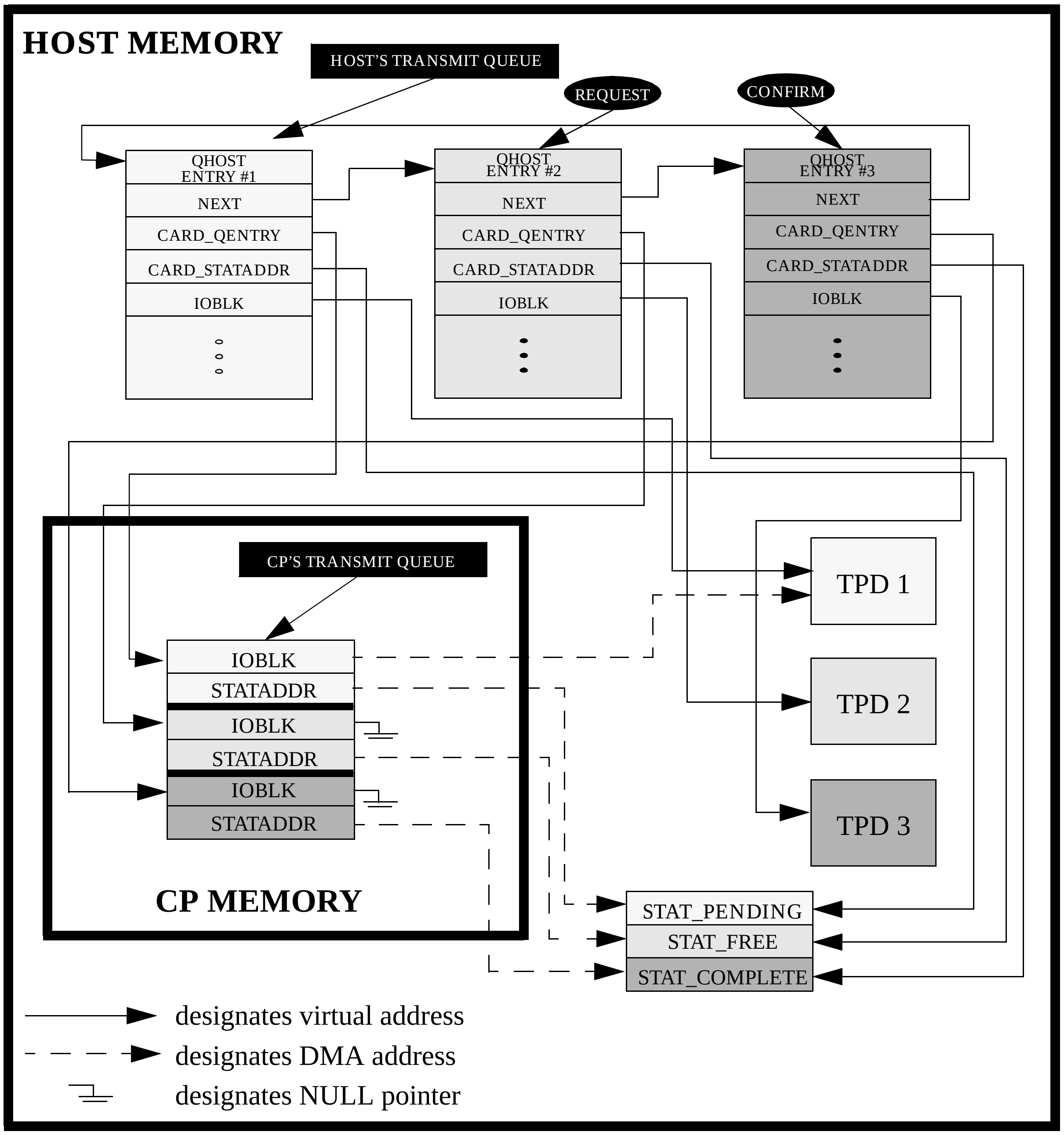


Figure 2.3 - Transmit Queue Snapshot One

Figure 2.4 shows the state of the transmit queue after the CP has transmitted `Tpd 1` and the host has cleaned up after `Tpd 3` and `Tpd 1`, advancing the confirmation pointer twice. The host has also issued a transmit for `Tpd 2` advancing the request pointer once.

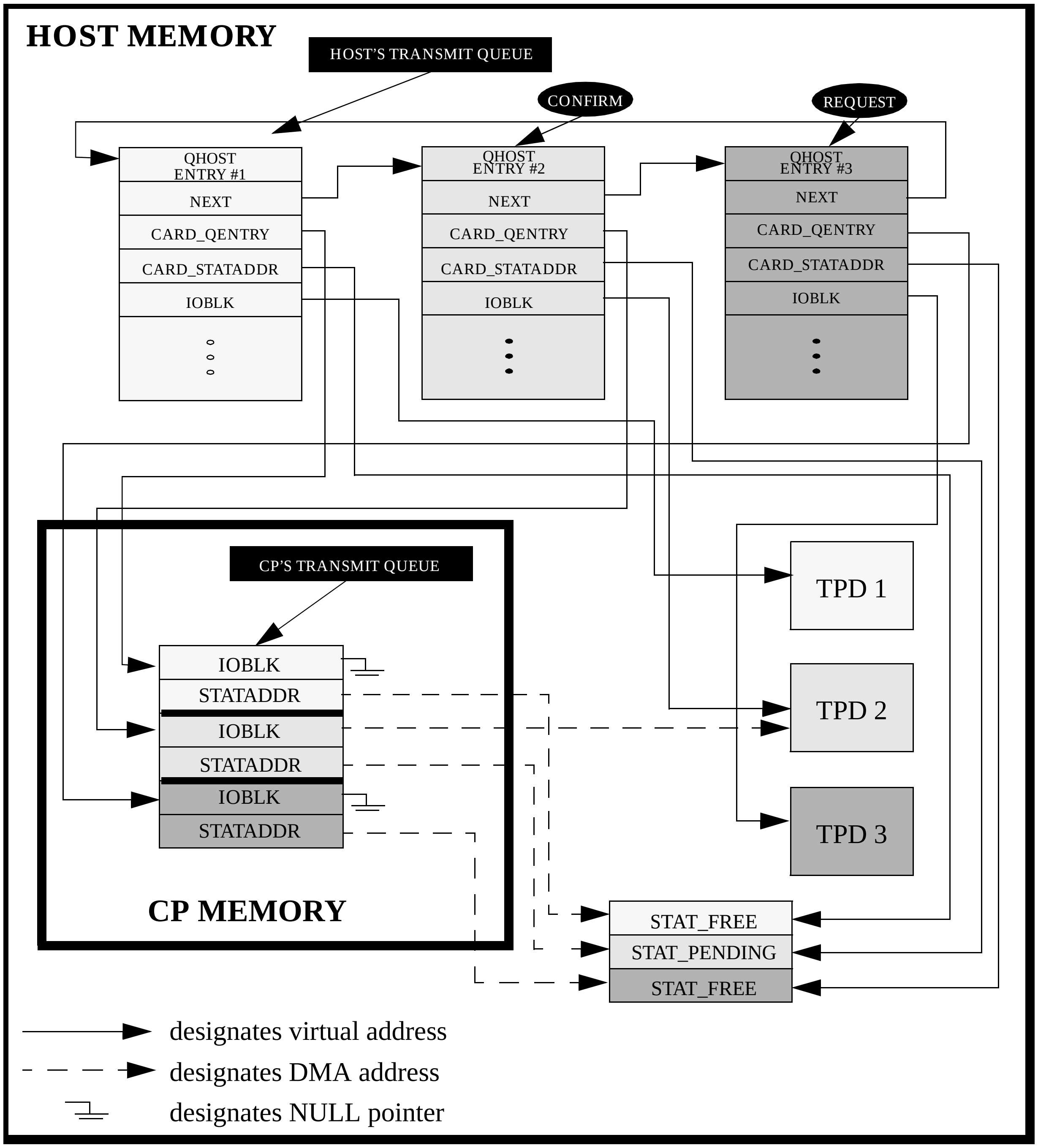


Figure 2.4 - Transmit Queue Snapshot Two

2.2 PDU Reception

The host enables PDU reception by initializing the receive queue, supplying the CP with buffers and activating a VCI (i.e. opening an ATM connection).

The CP carries out the reassembly of cells into a PDU, buffering the cell-payloads in host memory. For each partially reassembled PDU, the CP maintains a CP resident copy of a receive PDU descriptor (Rpd). An Rpd specifies the VPI/VCI, PDU length and buffers comprising a PDU. The CP copy of the Rpd is initialized by the CP during the reassembly process. When the last cell of a PDU is transferred to host memory, then the Rpd is also transferred to host memory and the host is interrupted.

For connections with an AAL of type null, cell payloads will be written contiguously into the reassembly buffers. For this AAL type, the CP implements header coalescing, where cells arriving with identical headers are grouped together and delivered to the host as a batch. The maximum number of cells in a group is controlled by the MTU (or batch size) for that connection. So the net effect is that the host is notified that cells have arrived on an AAL null connection only when either a cell arrives on that connection with a different header, or the number of cells which have arrived equals the current batch size.

Each entry of the receive queue is initialized by the host to reference a location in host memory equal to the size of an Rpd. This location can be viewed as a fixed sized container supplied by the host containing uninitialized values. The Rpd container will eventually hold an initialized Rpd put there by the CP. See Figure 2.5.

As with the transmit queue paradigm, PDU reception is accomplished via the interaction of a CP receive queue and its counterpart within the host driver. Each QHOST entry within the host's receive queue contains pointers to pairings of status words and Rpd structures. When a status word is set to STAT_COMPLETE, the corresponding Rpd contains a PDU that has been received by the CP; otherwise, the status word isn't set to completion and the Rpd is empty.

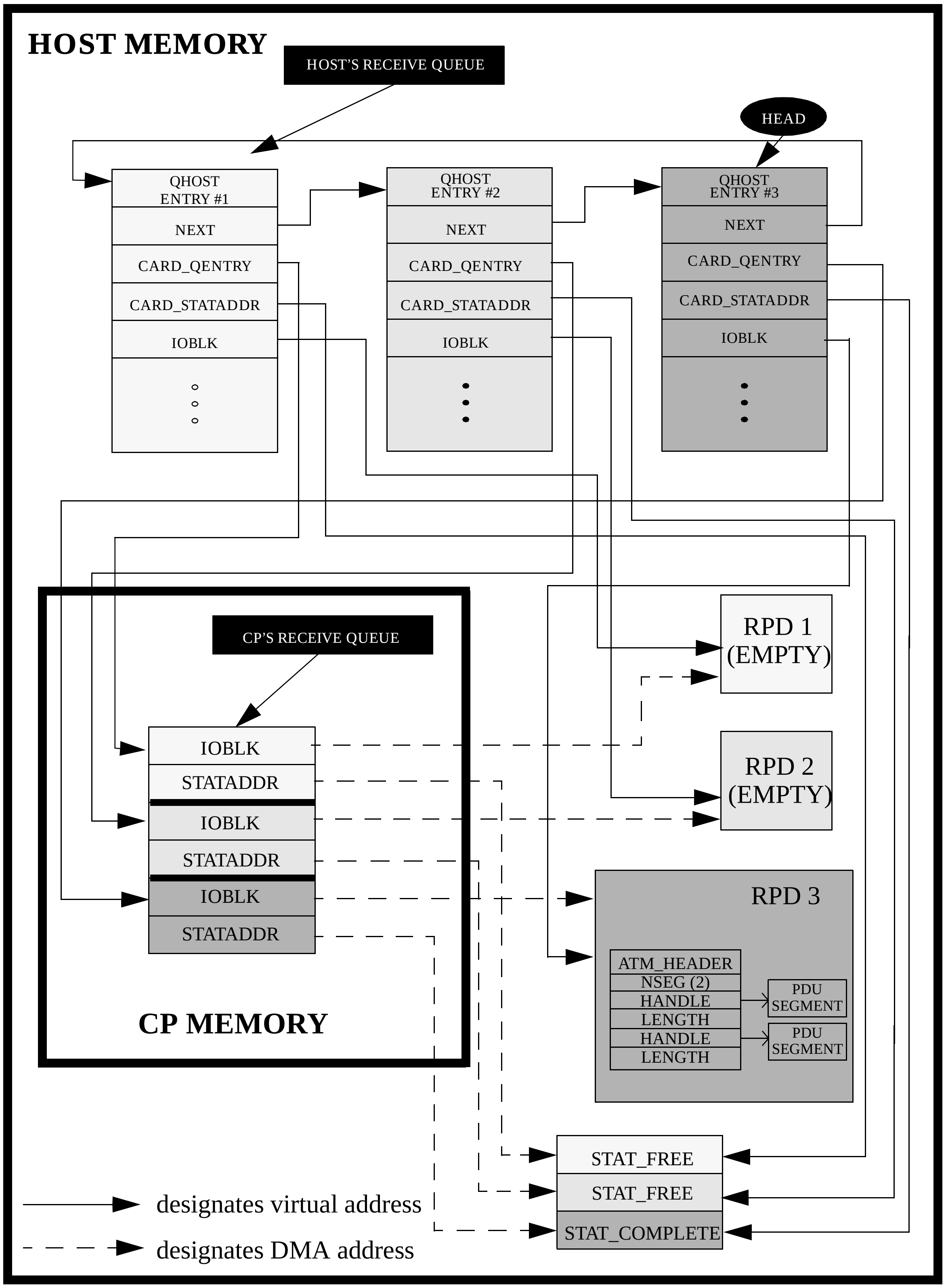


Figure 2.5 - Receive Queue

The length of the receive queue is configurable by the host and represents the maximum number of PDUs that can be queued for the host before the CP starts discarding cells.

The host maintains a host resident head pointer to the receive queue and the CP maintains a CP resident tail pointer to the receive queue.

A receive PDU interrupt indicates that one or more PDUs are ready to be processed by the host. The host need only check the head of queue status word for the value `stat_complete`. After processing a PDU, the host must then make a new `Rpd` container available to the CP by re-writing the receive queue with the DMA address of the same or a new `Rpd` container.

2.2.1 Receive Buffer Descriptors

The CP can support multiple host buffer schemes (e.g. BSD mbuf, System V STREAM buffer, frame buffer, etc.) through a host buffer independent buffer descriptor. The buffer descriptor gives the CP a uniform view of all host buffer schemes.

A receive buffer descriptor contains two parts, referencing the same host buffer structure. One part is the DMA address used by the CP for writing cell-payloads to host memory. The second part is a handle which is used by the host CPU to locate the buffer while processing a received PDU. The handle part migrates from the host supplied buffer descriptor to a receive PDU descriptor. It is simply passed along during the reassembly process, uninterpreted by the CP.

```
struct receive_buffer_descriptor {
    Haddr  handle;   /* host CPU control address of buffer */
    Haddr  buffer;   /* DMA address of host buffer */
};
typedef struct receive_buffer_descriptor Rbd;
```

2.2.2 Receive Buffer Schemes and Sizes

The CP maintains a buffer pool for each buffer scheme and buffer size (for example, number of pools = number of schemes * number of magnitudes). For each connection opened, the host specifies which buffer scheme the CP must use to buffer the cell-payloads for incoming cells.

The binding of a particular buffer scheme (`Bscheme`) to a particular host buffer scheme is done by the host with the `activate` command. The CP has no knowledge of the binding. For example `bscheme_one` may be bound to BSD mbufs or System V STREAMs buffers.

Host buffer schemes The firmware has a uniform view of all host buffer schemes through the buffer descriptors. (e.g. `Bscheme` may be designated by the host to be an mbuf/loaned mbuf, a streams buffer etc.)

```
enum buffer_scheme {
      bscheme_one =     0,
      bscheme_two =     1
};
typedef enum buffer_scheme Bscheme;
```

Within each host buffer scheme, the CP will optionally support multiple buffer magnitudes. Currently small and large buffers are supported. The host chooses the length of small buffers and large buffers with the `initialize` command. All buffers for a particular buffer scheme and magnitude must be the same size. It is recommended that for efficiency reasons these be a multiple of 48 bytes.

Host buffer magnitude What is designated to be small or large by the host is arbitrary. (They can be equal).

```
enum buffer_magnitude {
bmag_small =    0,
      bmag_large =    1
};
typedef enum buffer_magnitude Bmag;
```

The use of small buffers can be disabled by the host at initialization. If small buffers will be used, then the CP allocates one to hold the first cell-payload of a new PDU. If the remaining cells of the PDU will not fit into the original small buffer, then large buffers are used to complete the reassembly process.

If the CP runs out of small buffers for first cell-payload processing, then it will attempt to allocate a large buffer. However, if the CP has made the transition from using small buffers to large buffers and then runs out of large buffers, it will not attempt to allocate small buffers, instead the PDU reassembly process is aborted.

2.2.3 Receive PDU Descriptors

Receive PDU descriptors (`Rpd`) reside in host memory. They are supplied by the host and used by the CP. An `Rpd` can be in one of two states; empty or not-empty. When empty, it can be viewed as a fixed size container supplied by the host containing uninitialized values. In the not-empty state, the `Rpd` has been initialized by the CP (DMA write) with values that describe a received PDU.

The size of an `Rpd` is fixed at eight 32-bit words (3 receive segments). The host can extend the size of an `Rpd` with the initialize command in increments of eight 32-bit words (4 receive segments). The extension is based of the size of host buffers and the MTU size. For example, if the largest MTU accepted is 9180 bytes and a small buffer is 256 bytes and a large buffer is 4K, then the `Rpd` would have to be extended.

For example:

$$(256 + 2 \times 4K) < 9180$$

Therefore, 3 receive segments would be insufficient.

```
struct rx_pd {
u_int atm_header;
int    nseg;
Rsd    segment[NUM_RXSEG];
};
typedef struct rx_pd Rpd;
```

The receive segment descriptor (Rsd) describes a segment of a received PDU. The handle part is used by the host to locate the host buffer while processing a received PDU. The handle is passed along by the CP from a host supplied receive buffer descriptor (Rbd) to the Rsd.

```
struct receive_segment {
    Haddr  handle;   /* receive buffer handle */
    int    length;   /* number of bytes in buffer */
};
typedef struct receive_segment Rsd;
```

2.2.4 Buffer Supply Protocol

For each buffer scheme and magnitude, there is a supply queue and a CP resident buffer descriptor pool (see Figure 2.6). The host supplies the new buffer descriptors to the CP in fixed size blocks of buffer descriptors. The ioblock member of the queue entry points to an area in host memory that contains a fixed number of Rbd's.

If there are entries in particular buffer supply queue and the CP has space in the corresponding descriptor pool, the CP will transfer the block of descriptors to that pool. The CP then notifies the host that it has read the set of Rbd's by writing stat_complete to the host status word associated with that buffer resupply queue entry.

This protocol allows the CP to maintain a group of usable buffer segments in an area that is quickly accessible to it while allowing the host to supply the CP with more descriptors when convenient (e.g. in the host interrupt routine and / or when transmitting a PDU).

The length of each supply queue and the number of descriptors per block are configurable by the host with the initialize command. The supply queue and transmit queue function identically with the exception that the ioblock member references a host resident supply block instead of a Tpd.

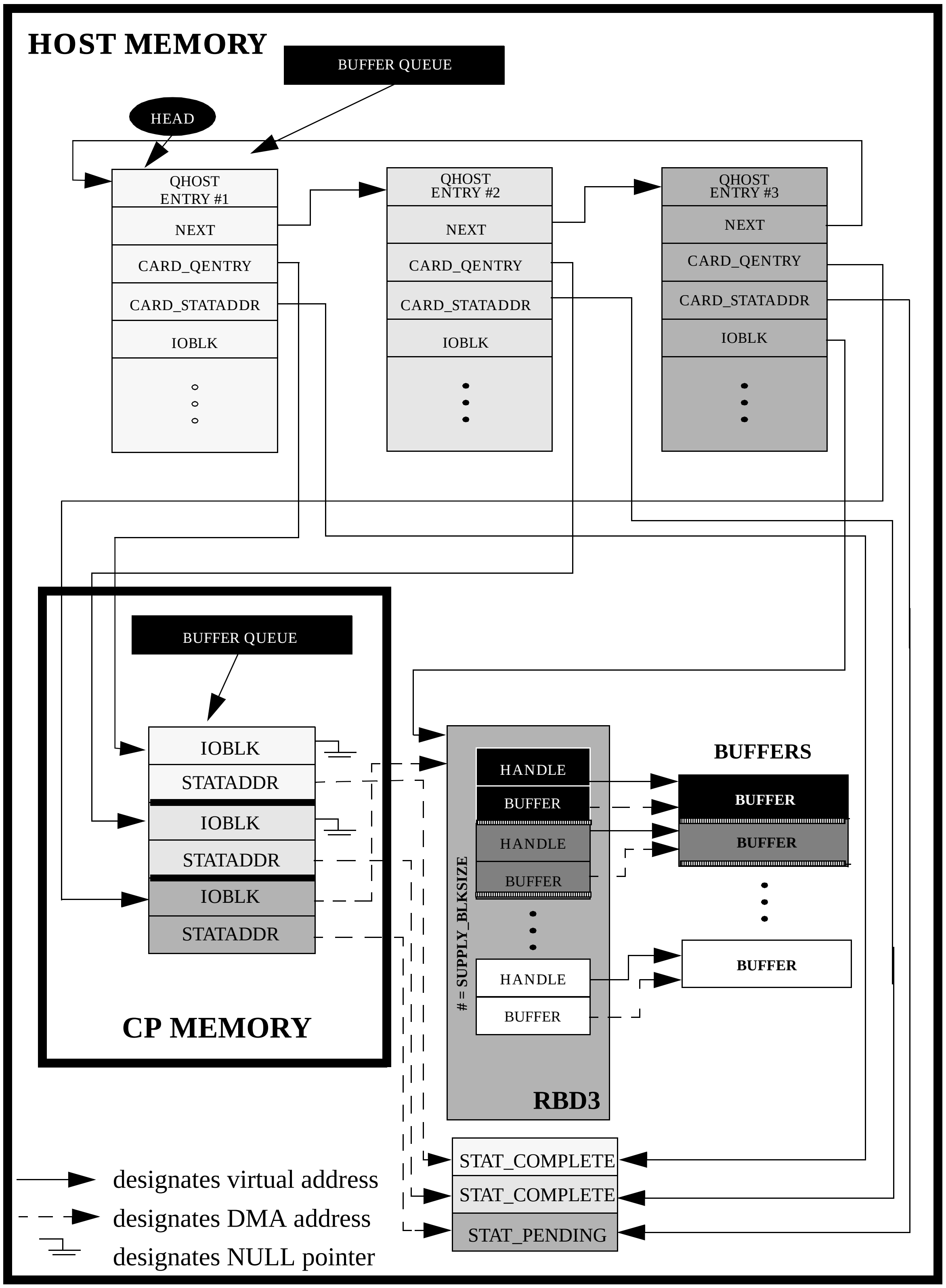


Figure 2.6 - Buffer Queue

CHAPTER 3 Commands

The command queue is slightly different than the other queues. Instead of the queue entry referencing an I/O block in host memory, the I/O block is located within the command queue. The host writes the command arguments directly to the queue entry. Each command queue entry has the following structure:

```
union command {
      Opcode              op;
      Activate_block      activate_param;
      Deactivate_block    deactivate_param;
      Request_stats       stats_param;
      Oc3_reg_op          oc3_reg_param;
      int force_multiple4[4]; /* gcc960 does this */
};
typedef volatile union command Command;

struct command_block {
      Command cmd;
      Haddr stataddr;
      int force_multiple4[3]; /* gcc960 does this */
};
typedef volatile struct command_block Qcard_cmd;
```

The host issues commands by first writing the command arguments into the queue entry at the head of the queue and then by writing the opcode. The order is important because the CP polls the op location within the command block in order to detect the presence of a new command. When an opcode is detected, the command arguments are assumed to be valid. On command completion, the CP notifies the host by writing stat_complete to the host status word (stataddr). The host can request that the CP interrupt the host on command completion. Any opcode can be modified with the opcode modifier op_interrupt_sel by logically ORing it with an opcode (e.g. (op_activate | op_interrupt_sel).

After completion of a command, the CP zeros the `op` member of a command queue entry in order recognize a queue empty condition. This is documented to aid in device driver development. The host should not read CP memory to determine when the CP has completed a command but should either request an interrupt or read the host resident status word.

```
/*
 * Command opcodes.
 */
enum opcodes {
   op_initialize       = 0x01, /* init 200-series              */
   op_activate_vcin    = 0x02, /* activate incoming VCI        */
   op_activate_vcout   = 0x03, /* activate outgoing VCI        */
   op_deactivate_vcin  = 0x04, /* deactivate incoming VCI      */
   op_deactivate_vcout = 0x05, /* deactivate outgoing VCI      */
   op_request_stats    = 0x06, /* return AAL and buffer stats  */
   op_oc3_set_reg      = 0x07, /* modify SUNI OC3 register     */
   op_oc3_get_reg      = 0x08, /* return SUNI OC3 registers    */
   op_zero_stats       = 0x09, /* zero card statistics         */
   op_get_prom_data    = 0x0a, /* return expansion rom info    */
   op_setvpi_bits      = 0x0b  /* set x bits of those decoded by*/
                               /* the firmware to be low order  */
                               /* bits from the VPI field of the*/
                               /* ATM cell header.              */
   op_interrupt_sel    = 0x80  /* interrupt select             */
};
typedef volatile enum opcodes Opcode;
```

The “`op_get_prom_data`” command is specific only to a PCI card implementation.

3.1 Activate VCIN Command

This command is used by the host to open an incoming ATM connection. Once a connection is opened, the CP will start reassembling cells having opened the vpvc into PDUs.

The vpvc is used by the host to specify the VPI and VCI to be used for the connection. The VPI must be zero. The VCI must be in the range of zero to the number of connections supported. The host uses the initialize command to select the number of connections supported.

The host also selects the AAL for the incoming cell stream and the buffer scheme to be used to buffer cell-payloads during the reassembly process.

For AAL null connections, the MTU is the amount of data (in bytes) that should be received before notifying the host. Since AAL null cells are grouped into batches and then delivered to the host, the MTU is alternatively referred to as the batch size.

The batch size of an AAL null connection may be changed while the connection is open by simply issuing another Activate VCIN command and specifying a new MTU. Issuing an Activate VCIN command for a connection which is already open is only permitted for AAL null connections.

```
/*
 *  Activate VCI; used by the host to enable the reassembly
 *  of cells with the specified VCI. Op word format:
 *
 *  31     24 23    16 15      8 7       0
 *  +--------+--------+---------+--------+
 *  |reserved| bscheme|     aal | opcode |
 *  +--------+--------+---------+--------+
 */
struct activate_block {
    Opcode  op;
    Vpvc    vpvc;
    u_int   mtu;/* For AAL null only */
};
typedef volatile struct activate_block Activate_block;
```

3.2 Deactivate VCIN Command

This command is used by the host to close an incoming ATM connection. Cells which arrive for the specified VCI after the connection is closed are discarded.

```
/*
 * Deactivate VCI command block.
 */
struct deactivate_block {
    Opcode  op;
    Vpvc    vpvc;
};
typedef volatile struct deactivate_block Deactivate_block;
```

3.3 Deactivate VCIO Command

This command is reserved for future use.

3.4 Activate VCIO Command

This is command is reserved for future use.

3.5 SUNI OC3 Set Register Command

This command is used by the host to set the value of a SUNI OC3 register (see PMC-Sierra, Inc. PM5345 SUNI Saturn User Network Interface).

The `oc3_op` field of `Oc3_reg_op` differs from other firmware commands with opcodes in that `oc3_op` is used to encode more information than just the specified operation. This field is used to encode a mask, value and register as well as the intended operation.

To alter the value of a SUNI register, the host must set `opcode` to `op_oc3_set_reg` and `reg` to the appropriate SUNI OC3 register number (refer to comment titled 'OC3 Register Command'). Both the `value` and `mask` fields must also be set in the following way:

The `mask` field specifies which bits of the `value` field are interpreted by the firmware. For example, if only the least significant two bits of a write to `reg` should be modified, set `mask` to 0x3.

The `value` field specifies the logical value to place at each bit position of `reg` that `mask` states is interpreted. This scheme facilitates the setting of specific bit positions within each register without the need to read, modify, then write the specified register.

The `oc3_buff` field is unused for a set operation.

```
/*
 * OC3 Register Command: used by the host to get/set specific
 *  SUNI register values. Op word format:
 *
 *      31     24 23    16 15      8 7       0
 *      +--------+--------+---------+--------+
 *      |  mask  | value  |   reg   | opcode |
 *      +--------+--------+---------+--------+
 */

struct oc3_reg_op {
          Opcode          oc3_op;
        Oc3_reg_set200  *oc3_buff; /* DMA address of oc3 buffer */
};
typedef volatile struct oc3_reg_op Oc3_reg_op;
struct oc3_reg_set200 {
          u_int   reg[128];
};
typedef struct oc3_reg_set200 Oc3_reg_set200;
```

SUNI Write Example:

Set the two high order bits of SUNI reg 0x34 to 0, not altering the low order 6 bits.

```
mask = 0xC0 (high order 2 bits set to 1)
value = 0x00 (high order 2 bits set to 0)
reg = 0x34
opcode = op_oc3_set_reg
```

For example, only consider the two high order bits of `value` when writing to register 0x34 leaving the other 6 bits alone, and the value of these two bits are 0.

3.6 SUNI OC3 Get Register Command

This command is used by the host to retrieve the value of SUNI OC3 registers (see PMC-Sierra, Inc. PM5345 SUNI Saturn User Network Interface).

To retrieve the value of the SUNI registers, the host must set `opcode` to `op_oc3_get_reg`. The `reg`, `value`, and `mask` fields are not interpreted.

All registers are returned into the buffer pointed to by `oc3_buff`. Unimplemented registers will have the value -1. `oc3_buff` must be 32 byte aligned.

```
/*
 * OC3 Register Command; used by the host to set and
 * retrieve SUNI OC3 register values. Op word format:
 *
 *      31     24 23    16 15      8 7       0
 *      +--------+--------+---------+--------+
 *      |  mask  | value  |   reg   | opcode |
 *      +--------+--------+---------+--------+
 *
 */

struct oc3_reg_op {
          Opcode          oc3_op;
        Oc3_reg_set200  *oc3_buff; /* DMA address of oc3 buffer */
};
typedef volatile struct oc3_reg_op Oc3_reg_op_t;

struct oc3_reg_set200 {
          u_int   reg[128];
};
typedef struct oc3_reg_set200 Oc3_reg_set200;
```

3.7 Statistics Command

This command causes the CP to transfers the contents of the CP resident Atm_stats structure to host memory. The stats_buff structure must be thirty-two byte aligned in host memory.

```
/*
 * Request statistics command block.
 */
struct request_stats {
    Opcode   op;
    Haddr    stats_buff; /* DMA address of stats buffer */
};
typedef volatile struct request_stats Request_stats;

/*
 * Statistics.
 */
struct phy_4b5b_stats200 {
    u_long   crc_header_errors; /* cells with bad header CRC */
    u_long   framing_errors;    /* cells with bad framing */
    u_long   pad[2];            /* pad to gcc960 boundary */
};
typedef struct phy_4b5b_stats200 Phy_4b5b_stats200;

struct phy_oc3_stats200 {      /* OC3 PHY */
 u_int  section_bip8_errors;/* section 8 bit interleaved parity */
 u_int  path_bip8_errors;  /* path 8 bit interleaved parity */
 u_int  line_bip24_errors; /* line 24 bit interleaved parity */
 u_int  line_febe_errors;  /* line far end block errors */
 u_int  path_febe_errors;  /* path far end block errors */
 u_int  corr_hcs_errors;   /* correctable header check sequence */
 u_int  ucorr_hcs_errors;  /* uncorrectable header check sequence */
 u_int    pad[1];           /* pad to gcc960 boundary */
};
typedef struct phy_oc3_stats200 Phy_oc3_stats200;
struct atm_stats200 {        /* ATM layer */
    u_long   cells_transmitted; /* cells transmitted */
    u_long   cells_received;    /* cells received */
    u_long   vpi_bad_range;     /* cell drops: VPI out of range */
    u_long   vpi_no_conn;       /* cell drops: no conn for VPI */
    u_long   vci_bad_range;     /* cell drops: VCI out of range */
    u_long   vci_no_conn;       /* cell drops: no conn for VCI */
    u_long   pad[2];            /* pad to gcc960 boundary */
};
typedef struct atm_stats200 Atm_stats200;

struct aal0_stats200 {           /* Null AAL */
    u_long   cells_transmitted;     /* cells transmitted */
    u_long   cells_received;        /* cells received */
    u_long   cells_dropped;         /* cells dropped */
    u_long   pad[1];                /* pad to gcc960 boundary */
};
typedef struct aal0_stats200 Aal0_stats200;

struct aal4_stats200 {              /* AAL4 */
    u_long   cells_transmitted;     /* cells xmited from PDUs */
    u_long   cells_received;       /* cells assembled in PDUs */
    u_long   cells_crc_errors;     /* payload CRC error count */
    u_long   cells_protocol_errors; /* SAR or CS layer errors */
    u_long   cells_dropped;         /* cells dropped */
    u_long   cspdus_transmitted;    /* CS PDUs transmitted */
    u_long   cspdus_received;       /* CS PDUs received */
   u_long   cspdus_protocol_errors;/* CS layer protocol errors */
    u_long   cspdus_dropped;       /* PDUs dropped (in cells) */
    u_long   pad[3];                /* pad to gcc960 boundary */
};
typedef struct aal4_stats200 Aal4_stats200;

struct aal5_stats200 {          /* AAL5 */
   u_long   cells_transmitted; /* cells transmitted from  SDUs */
    u_long   cells_received;    /* cells reassembled into SDUs */
    u_long   congestion_experienced; /* CRC err & length wrong */
    u_long   cells_dropped;     /* PDUs dropped (in cells) */
    u_long   cspdus_transmitted;/* CS PDUs transmitted */
    u_long   cspdus_received;   /* CS PDUs received */
    u_long   cspdus_crc_errors; /* PDU CRC errors */
    u_long   cspdus_protocol_errors; /* CS protocol errors */
    u_long   cspdus_dropped;    /* reassembled PDUs dropped */
    u_long   pad[3];            /* pad to gcc960 boundary */
};
typedef struct aal5_stats200 Aal5_stats200;

struct auxiliary_stats200  {
    u_long   small_b1_failed;    /* rcv BD allocation failures */
    u_long   large_b1_failed;    /* rcv BD allocation failures */
   u_long   small_b2_failed;    /* rcv BD allocation failures */
   u_long   large_b2_failed;    /* rcv BD allocation failures */
   u_long   rpd_alloc_failed;   /* rcv PDU allocation failures */
   u_long   receive_carrier;    /* no carrier = 0, carrier = 1 */
   u_long   pad[2];              /* pad to gcc960 boundary */
};
typedef struct auxiliary_stats200 Auxiliary_stats200;


struct stats200 {
    Phy_4b5b_stats200  phy4b5b;
    Phy_oc3_stats200   phyoc3;
    Atm_stats200       atm;
    Aal0_stats200      aal0;
    Aal4_stats200      aal4;
    Aal5_stats200      aal5;
    Auxiliary_stats200 aux200;
};
typedef struct stats200 Stats200;
```

CHAPTER 4 Initialization

This chapter provides some details on how the CP is loaded with the AAL interface firmware and how that firmware is initialized.

4.1 Downloading the AAL Interface Firmware

Firmware images can be loaded using one of two methods. In the past (refer to any ForeThought 4.0.x source distribution for more details), FORE's drivers used a virtual terminal interface to download the firmware through shared memory, byte by byte. Briefly, after executing a series of self-tests the CP starts to execute a version of the mon960 program from Intel. This software interacts with a "terminal" through a "UART" (Universal Asynchronous Transmitter/Receiver). In the case of the FORE CP, that UART is the soft uart implemented through shared memory.

To download a program onto the CP, a `do` command is sent to the CP over the soft uart. This command instructs the mon960 program to accept a new executable image. This image must be in b.out or Common Object File Format (COFF) format, and is sent to the CP using the xmodem protocol--firmware images used in this method use a .ucode extension. During the transmission, the format is translated to stripped COFF format. After successful completion, mon960 is sent the `go` command. The CP will then begin execution at the address provided as a parameter to this command.

The faster method, used by ForeThought 4.1 drivers, is to load the firmware directly into the adapter card's RAM--it was originally designed for use within the PC drivers. In this case, no translations are necessary (beyond dealing with the "endianness" of the target processor). Its method is implemented in a utility called ncomm, and is described next.

4.1.1 Fast Downloading

A .bin file is a firmware image file that begins with a four word header, which defines the following: a firmware identification number, its version, the offset from the top of RAM where the firmware is to be loaded, and the entry point for the firmware. Loading a .bin file into a FORE adapter card's RAM requires essential three steps:

- Read the top four words into a data structure so that these values may be used during the download procedure:

```
struct {
    u_int32_t firmId;          /* 0x65726f66 ==> "fore" */
    u_int32_t firmVersion ID;
    u_int32_t startOffset;
    u_int32_t entryAddr;
} firmHeader;

int headSize = sizeof(firmHeader);

handle= OPEN(<device>.bin, "r");
error= READ_FILE (handle, &firmHeader, headSize);
```

Any error checking can be done next using the first two fields in the header. Note that the header is part of the image, and therefore must also be downloaded to the card.

- Transfer the header just read and the rest of the image into the card’s memory, starting at the offset given in the header:

```
startP= au->au_ram + firmHeader.startOffset;
error = COPY(startP, &firmHeader, headSize);
iSize = firmSize - headSize;
error = READ_FILE(handle, startP+headSize, iSize);
```

- Because the host and the I960 may have different byte orderings, a reordering of the entire file may be necessary. Specifically, big-endian hosts require a reordering of each word because the file is read as bytes and written to RAM as words. To reorder, swap bytes 0 and 3 and bytes 1 and 2 of the 4-byte word (see SLAVE_XFER macro):

```
#if defined (SLAVE_SWAP_BYTES)
u_int32_t *swapp;

swapp = (u_int32_t *) startP;
for ( i = 0 ; i < (long)(firmSize/sizeof(u_int32_t)
+ 1) ; i++ ) {
   /*   SLAVE_XFER   is   defined   in   driver/
fore_atm_drv.h */
   SLAVE_XFER(*swapp, *swapp);
   swapp++;
}
#endif
```

The last thing to do is to start the firmware’s execution. You can do this by issuing characters into the monitor’s address location:

```
static u_int8_t *Hexchs = 0123456789ABCDEF”;

/*  Send  the  firmware  the  start  command  (“go
<addr>“) */
put_char (‘\r’);
put_char (‘g’);
put_char (‘o’);
put_char (‘ ‘);

entryAddr = firmHeader.entryAddr;
entryAddr <== 16; /* only lower 2-bytes are signifi-
cant */
for ( i = 4 ; i > 0 ; i-- ) {
   put_char ( Hexchs[ (entryAddr & 0xf0000000) >>
28 ] );
   get_char();
   entryAddr <<= 4;
}
put_char (‘\r’);
```

At this point, wait for a period (check every 10 milliseconds for 5000 milliseconds, or something more to your liking) for the monitor.bstat flag to be set to ‘cp_running’. Fail if this takes too long.

4.1.2 Notes on the Firmware

Header files included as part of this package define structures used to interface to the firmware. Therefore the structures defined in these headers must match those used internally by the firmware. The developer should use the firmware images included as part of this package to ensure that these structures match.

Some manuals that may be of interest to developers are “I960 Processor Software Utilities User’s Guide” and “Using the GNU/960 Tools” available from Intel.

4.1.3 The Host/CP Endian Description

This description presupposes a host processor and associated memory and an cell processsor with a local memory region. The host processor can access the cell processor local memory via a bus interface, and the cell processor can access the host memory also by a bus interface. All transfers on the bus occur as 32-bit (4-byte) words. These transfers both begin and end on physical word boundaries in both the host and cell processor memories. The host processor memory may be either little or big endian byte order. (The cell processor memory regions are little endian byte order.) The cell processor sends cells to and receives cells from the network interface as a series of words.

The network interface transmits the bytes from each word to the physical layer and receives the bytes into each word from the physical layer beginning with the most significant byte and ending with the least significant byte. The bytes in an ATM cell payload follow the order that those same bytes were in the original PDU buffer (which is in the host processor memory).

If the host processor memory is defined as little endian byte order, the bus interface must convert the bytes in words that are transferred between the host processor memory and the cell processor. An example of a PDU transmission from a little endian byte order memory will demonstrate why this is necesssary. Suppose a host processor with little endian byte order memory. We would like to transmit a twelve-byte buffer containing the string “Hello World!" that begins at a physical memory word with address B.

Buffer in host processor memory:

```
B+00: 'H'   B+04: 'o'   B+08: 'r'
B+01: 'e'   B+05: ' '   B+09: 'l'
B+02: 'l'   B+06: 'W'   B+10: 'd'
B+03: 'l'   B+07: 'o'   B+11: '!'
```

The cell processor will transfer this buffer to the network interface. Since the bus only permits word transfers between the host processor memory and the cell processor must read three words.

Words read by the cell processor from the host processor memory (little endian byte ordering):

```
                Most                           Least
                          significant bytes
Word 0:  |  'l' | 'l'|  'e'|  'H' |
Word 1:  |  'o'|  ‘W' | ' '|  'o' |
Word 2:  |  '!'|  'd' | 'l'|  'r' |
```

These words are directly transferred into the network interface or transmission beginning with the most significant byte of Word 0 and ending with the least significant byte of Word 2. Thus the byte stream presented to the physical layer would be “lleHoW o!dlr“. This is not the defined output byte stream.

Since PDU transmission and reception represent the greatest volume of transfers between the host processor and the cell processor, the bus interface is modified to invert the byte order during a memory word read or write. In the previous example, with the bus interface performing byte order inversion, the cell processor will read the following three words:

```
                 Most                                    Least
                                 significant bytes
   Word 0:  |  'H' | 'e'|  'l'|  'l' |
   Word 1:  |  'o'|  ' ' | 'W'|  'o' |
   Word 2:  |  'r'|  'l' | 'd'|  '!' |
```

When these three words are transferred to the network interface, the byte stream presented to the physical layer would be “Hello World!”. This is the required output byte stream.

There is a side-effect to the bus interface performing this byte order inversion: When the host processor and cell processor exchange word data, a second byte order inversion must be performed to restore the original value to the data word. By definition, only the host processor will perform the byte order inversion operations (so that the firmware on the cell processor can remain consistent for all implementations). For example, the host processor initializes the firmware parameters, the opcode word must be set to the value op_initialize (0x00000001). In order for the cell processor to read the opcode word with the value op_initialize, the host processor must actually store the value 0x01000000. During the read operation, the bus interface will invert the byte order in this word. As a result, the cell processor will read the opcode word with the value 0x00000001.

If the host processor memory is defined as big endian byte order, the bus interface does not perform byte order version and no byte order inversions are required from the host processor.

4.1 Downloading the AAL Interface Firmware

When the CP is first started during the power up cycle, it will run a number of simple self-tests that are in ROM. After completion of these tests, the CP starts to execute a version of the `mon960` program from Intel. This software interacts with a "terminal" through a "UART" (Universal Asynchronous Transmitter/Receiver). In the case of the FORE CP, that UART is the soft uart implemented through shared memory.

To download a program on to the CP, a `do` command is sent to the CP over the `soft uart`. This command instructs the `mon960` program to accept a new executable image. This image must be in COFF (Common Object File Format). It is sent to the CP using the xmodem protocol. After successful completion, `mon960` is sent the `go` command. The CP will start execution at the address provided as a parameter to this command.

For additional details on the latest release of the Intel i960 processor and the associated software, please refer to the applicable Intel documentation.

4.2 The Cp_queue Structure

The Cp_queue structure is CP resident and is read by the host during the boot sequence. After the boot sequence host accesses to the structure should be limited to writes. These write operations are used to place new entries into the queues that are used to initiate CP operations.

The host maintains host resident pointers to each of the CP resident queues. When initializing the queue pointers, the host treats all pointer members in the Cp_queue structure as offsets from the beginning of CP memory (e.g. the host base address of CP memory plus the value of transmit_queue yields the host addressable first entry of the transmit queue). See Figure 4.1 for details.

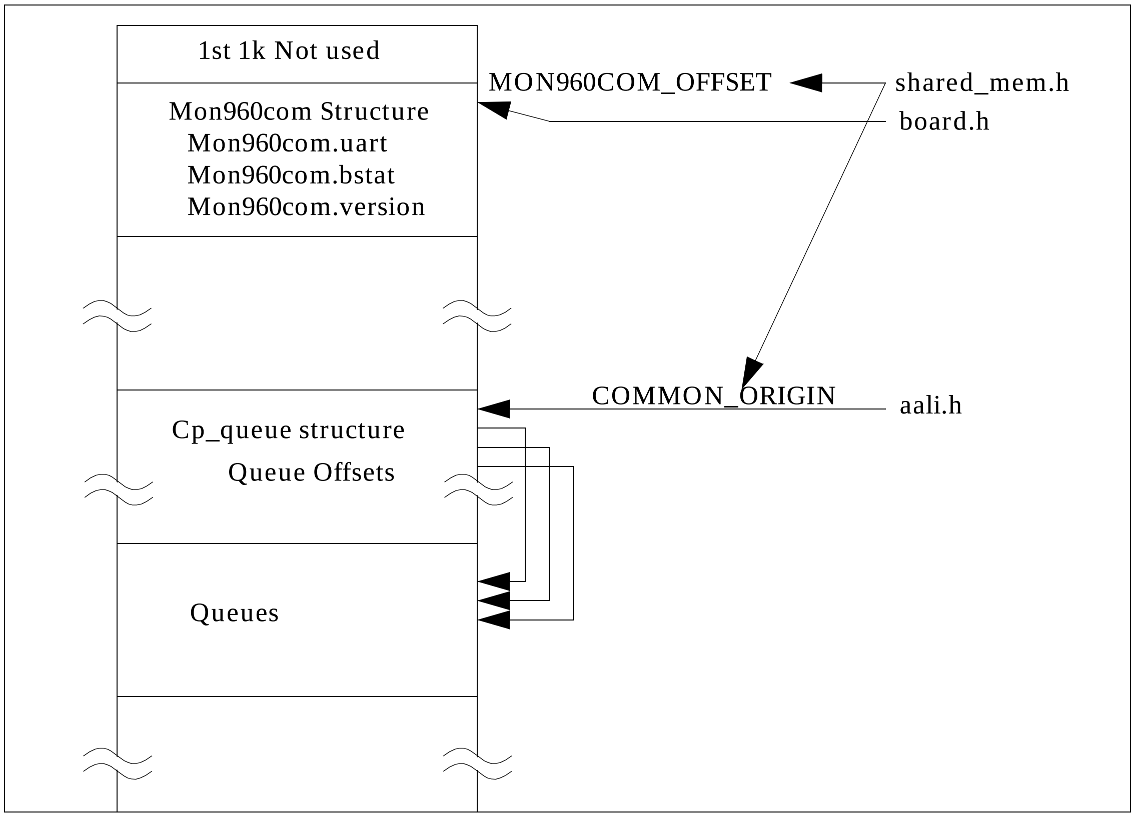


Figure 4.1 - Host -CP Shared Memory Offsets

4.2.1 Initialize Queue Pointers

The following code is used to initialize the pointers in a queue:

```
/*
 * Shared read-only structure used by host to initialize
 * host resident queue pointers.
 *
 * Each queue is write only. To initiate an operation the host
 * writes a queue entry with an address of the input/output block
 * located in host memory.
 */
struct cp_queues {
   CPaddr command_queue;
   CPaddr transmit_queue;
   CPaddr receive_queue;
   CPaddr small_b1_queue;
   CPaddr large_b1_queue;
   CPaddr small_b2_queue;
   CPaddr large_b2_queue;
   u_int32 imask;          /* Non-0 enable CP to host interrupts*/
   u_int32 istat;          /* 1 for interrupt posted           */
   u_int32 heap_base;      /* offset from beginning of ram     */
   u_int32 heap_size;      /* space in bytes avail for queues  */
   u_int32 hlogger;        /* non zero for host logging        */
   u_int32 heartbeat;
   u_int32 firmware_release;
   u_int32 mon960_release;
   u_int32 tq_plen;        /* transmit throughput measurements */

    /*
     * To be compatible with host compilers make sure the init block
      * remains on a quad word boundary. gcc960 does this.
      */
   Init_block init;        /* one time cmd, not in cmd queue   */
   u_int32 media_type      /* media type id                    */
   int32 oc3_revision;     /* OC3 revision number              */
};
typedef volatile struct cp_queues Cp_queues;
```

4.3 CP Boot Sequence

This section provides details on the structure and implications of booting and loading the CP. The steps are presented in the order in which they occur.

1. The host sets the boot status word (bstat) to cold_start.

```
/*
 * Structure shared between host and CP,
 * found in low CP memory.
 */
struct mon960com {
      Soft_UART   uart;       /* accessed by mon960 and ncomm  */
      Boot_status bstat;
      u_int32     app_base;   /* application base offset       */
      int32       version;    /* mon960 version                */
};
typedef volatile struct mon960com Mon960com;

enum boot_status {
      cold_start     = -1071792099,/* 0xc01dc01d               */
      self_test_ok   =  0x02201958,
      self_test_fail = -1380262995,/* 0xadbadbad               */
      cp_running     = -0837681427,/* 0xcellfeed - application */
                                   /* is ready                 */
      mon960_too_big = 0x10aded00
};
typedef enum boot_status Boot_status;
```

2. The host resets the i960 by setting and then clearing bit zero of the board control register. The host waits for the boot status word to make a transition.
3. Before accepting commands mon960 executes a self test. If successful the boot status word is set to self_test_ok, if unsuccessful the boot status is set to self_test_fail. If the i960 is unable to execute then the boot status word may remain at cold_start or transition to an unspecified value.
4. If the self test passes, then the host can initiate a firmware download command to mon960 with the ncomm utility program (host side front end for comm960). The host waits for the boot status word to make a transition.
5. Once the firmware is downloaded and running, the CP will set the boot status word is set to cp_running. The host can read the heap_size, which is used when figuring queue sizes. At this point, the entry heartbeat in the Cp_queue structure will be counting downwards. This downwards count can be used by the driver developer to determine if the download

and starting of the AALI firmware was successful.

6. The host issues the initialize command. The initialize command block does not reside in the command queue but is located in the Cp_queue structure, because the initialize command is used to set the various queue sizes and indirectly their location in CP memory. Once this command has been issued, the CP will not accept another initialize command. At this point, the entry heartbeat in the Cp_queue structure will be counting upwards.

```
/*
 *  Initialize command block:
 */
struct bscheme_specification {
   int32 queue_length;
   int32 buffer_size;    /* host buffer size                  */
   int32 pool_size;      /* number of Rbd's                   */
   int32 supply_blksize; /* num of Rbd's in ioblock (multiples*/
                         /* of 4 between 4 and 124 inclusive) */
};
typedef struct bscheme_specification Bspec;

struct init_block {
   Opcode op;
   Status status;          /* initialized and read by host    */
   int32 receive_threshold;/* Not used                        */
   int32 num_connect;      /* ATM connections                 */
   int32 cqueue_len;       /* command queue                   */
   int32 tqueue_len;       /* transmit queue                  */
   int32 rqueue_len;       /* receive queue                   */
   int32 rpd_extension;    /* number of 32 byte blocks        */
   int32 tpd_extension;    /* number of 32 byte blocks        */
   Vpvc conless_vpvc;      /* Not Used                        */
   int pad[2];             /* force quad alignment            */
   Bspec small_b1;         /* buffer scheme 1, small          */
   Bspec large_b1;         /* buffer scheme 1, large          */
   Bspec small_b2;         /* buffer scheme 2, small          */
   Bspec large_b2;         /* buffer scheme 2, large          */
};
typedef volatile struct init_block Init_block;
```

4.4 Heartbeat - CP State Indication

There is a location in the Cp_queues structure that can be monitored to check the current state of the CP. This location, Cp_queues.heartbeat, is manipulated by the CP in different ways depending on the current CP state.

At start-up the heartbeat is set to zero. After initialization, the CP enters a loop waiting for an initialize command to be executed. While waiting for this command, the CP will decrement heartbeat once for each time through its wait loop. After the initialize command has been executed, the CP enters its normal queue processing loop. While in this loop, the CP will increment heartbeat each time through the loop. If at anytime there is a fault from which the CP can not recover, the fault processing code will set heartbeat with an indication of the fault. These fault codes have the form: 0xdeadxxxx. Where the xxxx value gives the fault reason. The file fatal.h contains the set of fault values.

If at any time there is a fault from which the CP can not recover, the fault processing code will set heartbeat with an indication of the fault. These fault codes all have the form 0xdeadxxxx, and the meaning of their vaules are listed below:

FLAT_LINE **0xdead0000**

Baseline error message that spells 0xdead0000. This will never be seen alone, just as a base for other fatal conditions.

UNEXPECTED_HOST_INTR **0xdead0001**

Interrupt originating from host, destined for board. These are currently not used, and thus the fatal code will happen if event mysteriously occurs.

UNEXPECTED_NET_INTR **0xdead0002**

Interrupt from network status chip to the i960. See section in 200-series design specs for details. Basically, a number of conditions could trigger an interrupt from the status chip (including when an EOM/SSM cell is received, etc). We do not currently enable this feature.

UNEXPECTED_HBUS_INTR **0xdead0003**

DEAD0003 is an indication that the adapter card took a bus error trying to access host memory. This happens when the host driver provides a memory location to the CP that is not valid.

UNEXPECTED_LBUS_INTR **0xdead0004**

If i960 attempts to access illegal memory region this fatal code will be asserted.

UNEXPECTED_BAD_INTR **0xdead0005**

Not used.

MARKER_STUCK1 **0xdead0006**

We place a specified marker in the fifo and if we don't read it back in reasonable time this fatal code is asserted. Used to assure that dma fifo is drained. DMA timeout, marker is stuck behind dma request.

MARKER_STUCK2 **0xdead0007**

If waiting for the last marker value takes too long this fatal code is asserted. DMA timeout, marker is stuck behind dma request.

UNEXPECTED_AAL **0xdead0008**

Not used.

BAD_NUM_TXSEG **0xdead0009**

If the number of segments in the Tpd is less than 0, or greater than the fixed number + the extensions (specified during initialization of the firmware) this fatal code is asserted.

BAD_TPD_SIZE **0xdead000a**

If the number of 32 byte blocks in the TPD is <=0 or greater than the total size of a TPD (fixed + extensions) than this fatal code is asserted.

BAD_SEG_LEN **0xdead000b**

Not used.

BAD_PDU_LEN **0xdead000c**

If the plen specified in the TPD (spec field) does not equal the sum of all the length of the segments specified in the TPD, or the total length is greater than the MAX_MTU than this fatal code is asserted.

END_TWO_PASS_COMPILE **0xdead000d**

Internal use only.

UNEXPECTED_TAGERR_INTR **0xdead000e**

DEAD000E indicates that the DMA engine has received something from the output FIFO that it was not expecting (eg. it received data when it was expecting an address). This generally indicates that the VME bus has reset or changed state for some reason.

UNEXPECTED_XFRERR_INTR **0xdead000f**

Indicates a Microchannel DMA transfer error.

RX_CELL_THRESHOLD_INTERRUPT **0xdead0010**

An interrupt to the CPU is generated whenever any of the four receive cell counters exceeds the cell count threshold value.

RX_CELL_OVERFLOW_INTERRUPT **0xdead0011**

An interrupt to the CPU is generated whenever any of the four receive FIFO drops a cell due to FIFO overflow.

RX_PAY_UNDERFLOW_INTERRUPT **0xdead0012**

An interrupt to the CPU is generated whenever the CPU attempts to read from an empty Payload receive FIFO.

TX_SEQ_ERROR_INTERRUPT **0xdead0013**

Interrupt is generated whenever the ESP detects an improper transmit sequence.

SUNI_INITIALIZATION_FAIL **0xdead0014**

The process of SUNI initialization fails.

APPENDIX A DMA Address Alignment

A.1 Minimum DMA Requirements

The following table shows the minimum DMA address alignment requirements of the various host memory objects. Each pair of numbers is in bytes and specifies is the minimum alignment followed the number which the object size is a multiple of (e.g. a receive buffer for a SPARC10 must start on a 64 byte boundary and the buffer length must be a multiple of 64 bytes).

Table A.1 - Minimum DMA Requirements 200 Series Adapters

Card Type	Rpd/Tpd	rbuf	tbuf	Rsd pool	Status Buffer
SBA-200E / SPARC 2	32 32	32, 32	4, 4	32, 32	32, 224
SBA-200E / All other SPARCs	32 32	64, 64	4, 4	32, 32	32, 224
HPA-200E	16 32	16, 4	4, 4	32, 32	32, 224
VMA-200E	32 32	16, 4	4, 4	32, 32	32, 224
MCA-200E	4 32	4, 4	4, 4	32, 32	32, 224
ESA-200E	16 32	16, 4	4, 4	32, 32	32, 224
PCA-200E	4 32	4, 4	4, 4	32, 32	32, 224

APPENDIX B Host - CP Shared Memory Definitions

B.1 Memory Offsets

The values defined here provide the memory offsets (from the start of CP memory) for the different portions of the CP address space.

```
/*
 * Copyright (c) 1991 by Fore Systems, Inc.
 */

#ifndef _shared_mem_h
#define _shared_mem_h

#ifndef lint
static char _shared_mem_h_rcsid[] = "@(#)$Id: shared_mem.h,v 1.2
       1993/12/ 09 00:08:44 ghoti Exp $ FSI";
#endif /* lint */

#define MON960COM_OFFSET  0x00000400/* From start of RAM */
#define COMMON_ORIGIN     0x00004d40      /* From start of RAM */

#define RAM_SIZE        (256 * 1024)
#define MONITOR_ORIGIN  0
#define MONITOR_LENGTH  COMMON_ORIGIN
#define COMMON_LENGTH   (1 * 1024)
#define APP_ORIGIN (MONITOR_ORIGIN + MONITOR_LENGTH +\COMMON_LENGTH)
#define APP_LENGTH ((RAM_SIZE - MONITOR_LENGTH) -\COMMON_LENGTH)
       #endif /* _shared_mem_h */
```

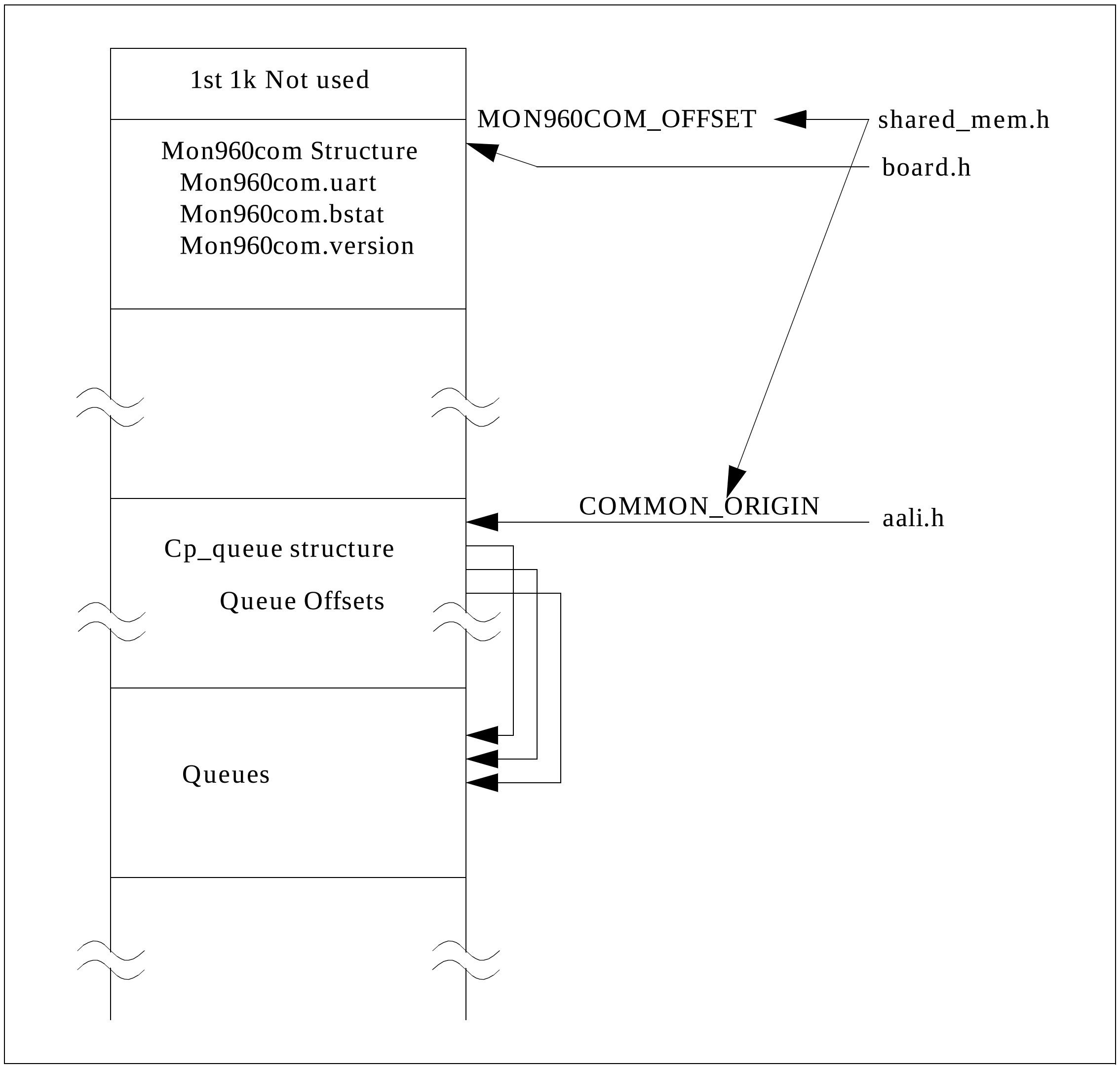


Figure B.1 - Host -CP Shared Memory Offsets

APPENDIX C MCA Modifications

C.1 New Functionality

This Appendix describes changes in the ATM Adaptation Layer Interface for MCA-200 adapter cards in IBM RS6000 workstations running the AIX operating system. These modifications significantly decrease the overhead associated with maintaining data cache coherency for the IOCC DMA and host CPU data caches.

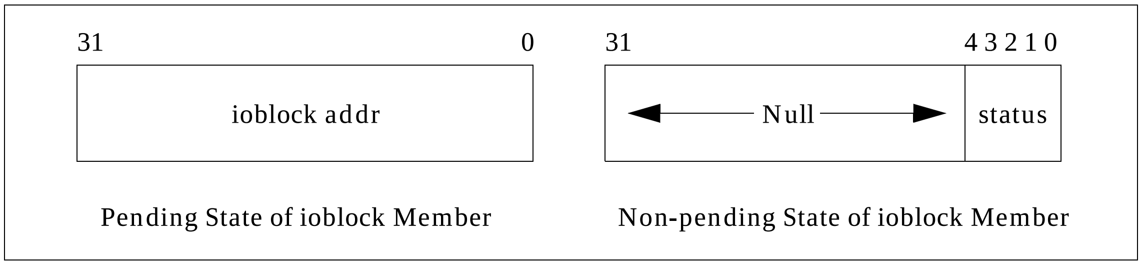
In the current version of the software, the state of all queues is held in a status word on the card instead of host memory. The stataddr member has been eliminated from the Qcard queue entry.

The ioblock member of the queue entry serves two functions:

1. The ioblock member initiates operations by passing the DMA address to the card from the host
2. The ioblock member also indicates the completion status.

C.1.1 Data Structure

Queue data structure for the ioblock member for both pending and non-pending states is shown graphically below:


31
0
ioblock addr
Pending State of ioblock Member
31
4 3 2 1 0
Null
status
Non-pending State of ioblock Member

There are two possible states for status: stat_complete or stat_free. As an option, stat_complete may also include stat_error

C.1.2 Host CP Interaction

Below is a general description of the changes made to the CP and host interaction to improve the performance of the MCA-200.

1. At initialization, the host initializes the ioblock member of the CP resident queue entry with stat_free .
2. To initiate an operation, the host writes the ioblock member of the CP resident queue entry with the DMA address of the host resident I/O block. This sets the state of the entry to pending (the stat_pending definition is no longer used).
3. The CP notices that the ioblock member of the queue entry has changed and starts the operation associated with that queue. At the end of the operation, the CP will write the ioblock member of the queue entry with stat_complete .

APPENDIX D SBA-200E SBus Slave Interface

D.1 SBus Slave Module

The SBus Slave interface shall provide single data (non-burst) access to the SBus boot prom, SDC internal registers, and i960 instruction RAM. The SDC slave registers include the SBus interrupt selection register, host control register with both registers providing read and write access, while the burst transfer select register is read only location , as initialization is a i960 operation. In addition, the i960 instruction RAM is both read and write accessible from the SBus interface.

Table 1 shows the SBus address map for the accessible devices. Note the device decodes only SBus physical address bits 24 through 22, and responds to only non-burst access.

Table D.1 - SDC SBus Slave Address Map

SDC Slave Access	SBus Physical Address PA<24:0>
SBus boot prom	0x0000000
Host control register	0x0400000
SBus burst transfer size register	0x0800000
Sbus interrupt level selection register	0x0C00000
i960 Instruction RAM	0x1000000

D.2 Host Control Register

The SDC host control register is a Read modify Write register type. This implies that individual register contents that require state maintenance must be written to preserve the current state. The register contains 5 data write bits, including 2 sticky bits which maintain last write data. In addition, the register provides 8 read bits.

The following list shows the HCR Write defines where all bits are active high.

1. **Clear HCR register:** To clear the entire register, a 0x00 must be written, all functions inactive.
2. **Set CPU reset**: Sets the i960 and ESP reset signals, equivalent to a processor and network reset. The SDC device remains operational. Sticky bit.
3. **Set i960 Hold lock**: Sets the i960 bus hold request signal active, thereby placing the i960 into an idle state. Sticky bit.
4. **Set i960 slave interrupt**: Sets the SDC to i960 local interrupt. Cleared by i960 Interrupt Service Routine, a CPU reset, or bus reset.
5. **Clear SBus interrupt**: Clears the pending SBus interrupt, also cleared by a CPU reset.
6. **Enable SBus interrupt**: SBus interrupt enable bit, must be enabled after bus reset. Can be used as interrupt mask per the SBus specification recommendation. Sticky bit.

Table D.2 - Host Control Register bit definitions

Bit	8	7	6	5	4	3	2	1	0
HCR Write	0	0	0	0	Bus Int. Enable sticky	Clear SBus Int.	Set i960 Int.	Hold Lock, sticky	CPU Reset, sticky
HCR Read	SBus Int. req. (ah)	Device Test mode (ah)	In fifo almost full (al)	ESP Hold req. (ah)	SBus Int. Enable (ah)	out fifo almost full (ah)	i960 Fail (al)	Hold Lock (ah)	CPU Reset (ah)

Notes: (ah) - active high, (al) - active low.

The following list shows the HCR Read definitions, not register bit polarity.

Bit 0: CPU reset	indicates the state of the CPU reset signal.
Bit 1: Hold lock	indicates the state of the Hold lock signal.
Bit 2: i960 Fail output	defines the result of processor self test, performed afte reset.
Bit 3: Out fifo almost full	valid when the out fifo word count has reached 120 words, placing the i960 into the bushold state.
Bit 4: Enable SBus interrupt	indicates the state of the SBus interrupt enable bit.
Bit 5: ESP Hold request	indicates the state of the ESP transmit fifo almost full state.
Bit 6: In fifo almost full	valid when the SDC bus in fifo word count cannot provide enough buffer for pending SBus read transfer.
Bit 7: SDC device test mode	the device is in test mode state, which disables SBus master engine. The test mode loops the out fifo to the in fifo, thus providing bus loop-back capability.
Bit 8: SBus interrupt	indicates the state of this card’s SBus interrupt.

D.3 SBus Burst transfer configuration register

The Sbus burst transfer configuration register is a i960 read / write and SBus read only register, indicating the maximum burst transfer size accepted by the host machine. The register is initialized by the i960, and used by the master DMA engine during block transfer operations. The maximum burst size is determined by the host driver via the host configuration prom, and then passed to the firmware for device initialization and used to enable optimal DVMA burst transfer operations.

Table D.3 - SBus Burst transfer configuration registe**r**

Sbus Burst transfer word size	Data bits <4:0>
4	0x04
8	0x08
16	0x10

D.4 Interrupt Level Select Register

The SBus interrupt level selection register is a hexadecimal value register initialized by the host processor. The interrupt level is selected by writing the desired interrupt level to the register.   For example, level 3 is chosen by writing the value 011b to the register. Upon system reset, the register is initialized to level 0x0, thus disabled.

Table D.4 - SBus Interrupt level selection register

Sbus INT level	**Data bits<2:0>**
2	0x2
3	0x3
4	0x4
5	0x5

APPENDIX E ESA-200E EISABus Slave Interface

E.1 EISABus Slave Mode

The EBI chip supports only 8-bit I/O accesses from the host. Table 1 shows the address map for the registers that are accessible by the host in I/O space.

Table E.1 - I/O Address Map for EBI chip

I/O Addrress (hex)	Descriptions	Width	Access
0z000	Host Control Register	8 Bits	R/W
0z400	High Base Address Register	8 Bits	R/W
0z404	Low Base Address Register	8 Bits	R/W
0zC00 - 0zCFF	EISA ID PROM	8 Bits	Read

The “z” value in the above table is determined by the EISA slot that the adapter card resides. The following sections describe the organization of each register. Unless otherwise noted, all bit values are active high (1).

E.1.1 Host Control Register (HCR)

Table E.2 - Host Control Register bit definitions

Bit	7	6	5	4	3	2	1	0
Read	TST MODE	INFF ALFL	ESP HOLD	OTFF HOLD	HOLD ACK	CPU FAIL	LOC HOLD	BRD RST
Write	X	X	X	CLR INT	INT 960A	INT 960B	LOC HOLD	BRD RST

TST MODE:	1=Board is in self-test mode (wrap test) 0=Normal operation Wrap test is active after a board reset (read-only status bit)
INFF ALFL:	IN FIFO almost full (read-only status bit)
ESP HOLD:	A hold to i960 is asserted by the ESP chip (read-only status bit)

OTFF HOLD:	A hold to i960 is asserted by the OUT FIFO (8 words from full) (read-only status bit)
HOLD ACK:	Hold Acknowledge from i960 (read-only status bit)
CPU FAIL:	1=i960 failed self-test 0=i960 passed self-test (read-only status bit)
LOC HOLD:	1=Assert hold to i960 to gain access to local bus 0=Release hold Power-on default = 0 (read / write register bit)
BRD RST:	1=Assert board reset (resets bus ASIC, CPU, ESP, etc.) Slave access mode is not affected by reset 0=Release reset Power-on default = 0 (read / write register bit)
CLR INT:	1=Clear bus interrupt set by i960 0=don’t care (write-only control bit)
INT 960A:	1=Set interrupt to i960 0=don’t care (write-only control bit)
INT 960B:	1=Set interrupt to i960 (different interrupt line) 0=don’t care (write-only control bit)

E.1.2 High Base Address Register (HBAR)

Table E.3 - High Base Address Register

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	INT SEL1	INT SEL0	B31	B30	B29	B28	B27	B26

INT SEL0:	Select interrupt line on EISA bus		
INT SEL1:	SEL1	SEL0	
	0	0	= Interrupt 5
	0	1	= Interrupt 10
	1	0	= Interrupt 11
	1	1	= Interrupt 12

B31 - B26: Upper 6 bits of EISA address (bit 31 to bit 26)

E.1.3 Low Base Address Register (LBAR)

Table E.4 - Low Base Address Register

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	INT MASK	INT TYPE	B25	B24	B23	B22	B21	B20

INT MASK: 0=Enable Interrupt to Host
1=Mask off interrupt to host

INT TYPE: 0=Select Level Interrupt on EISA bus
1=Select Edge-trigger Interrupt on EISA bus

B25 - B20: Bit 25 to bit 20 of EISA address

E.1.4 Slave Memory Addressing

The EBI chip supports only 32-bit, non-burst, slave memory access from the host. Slave memory access allows the host to read or write up to 1M bytes of local memory on the adapter card. To map the 1MB local address to EISA address space, the host must program the two base address registers: High Base Address Register (HBAR) and Low Base Address Register (LBAR). Together, the two base address registers form the upper 12 bits of the EISA address. The lower 20 bits are mapped directly to the 1MB local address space. Table 3 and Table 4 show the organization of the base address register.

When there is a memory access by the host, the EBI chip compares the upper 12 bits on the EISA address bus to the base address registers. If there is a match, the EBI chip requests the i960 to release the local memory bus and passes the EISA access to the local memory, using the lower 20 bits of the EISA address to control the local 1MB address. The EBI chip does not start the slave address comparison until BOTH base address registers are programmed by the host.

E.1.5 Hold Lock Memory Access

The host can also gain exclusive access to the local memory by setting the LOC HOLD bit in the HCR. In this ‘hold lock’ mode the i960 is held off the local memory (firmware execution is suspended) until the LOC HOLD bit is cleared by the host. The indication that the i960 is in hold mode is the assertion of HOLD ACK line, which can be monitored by the host at bit 3 in the HCR (for diagnostic and testing purpose).

E.1.6 Slave Controller Reset

The slave controller is only affected by a power-on reset from the EISA bus. The slave mode is functional all other times (BRD RST, wrap mode, or master error states).

APPENDIX F GIA-200E GIO Bus Slave Interface

F.1 GIOBus Slave Interface

Each expansion slot in a GIO bus machine is assigned a 2MB address range within the 32-bit address space of the GIO bus. The base address for the slot is based upon the slot number; slot 0 has a base address of 0x1F400000, while slot 1 has a base address of 0x1F600000. The GBIA maps four distinct devices (960 RAM, GIO bus Product Identification Word, Configuration PROM, Processor Control Register) into this address space so that the host can access them as needed. Table F.1 shows the available devices and their offsets within the address space along with the actual width of the data from the device and whether the data is read-only or read-write. The GBIA will not respond to a write to a read-only device in the slave address region. This will result in a bus time-out and kernel panic on the host system so care should be taken not to perform these invalid accesses.

Table F.1 - Slave Address Space

Addr. Offset	Description	Notes
0x000000	PIW Register	8 bit, Read Only
0x040000	Host Control Register #2	1 bit, Read/Write
0x08000-0x080800	Configuration PROM	8 bit, Read Only
0x0C0000	Host Control Register	10 bit, Read/Write
0x100000-0x1FFFFF	960 Local RAM	32 bit, Read/Write

The slave interface is not intended for heavy use; it supports only single, 32-bit accesses (no bursts), and during RAM and PROM accesses must stop the 960 from executing to access shared resources. As such, use of the slave interface after initial start up should be limited to prevent system performance degradation. Also, as mentioned the slave accesses must be 32-bit accesses, so when accessing any device other than the RAM, the unused upper bits should be masked off and/or ignored.

F.1.1 Processor Identification Word (PIW)

The PIW provides a means for the operating system on the host machine to identify the type of expansion card. The PIW for cards using the GBIA is 0x5C (the GIA-200 is 0x60).

F.1.2 Host Control Register (HCR)

The HCR is a 9 bit control and status register accessible from the GIO bus. The register provides the host with a number of control and status bits for controlling and monitoring the adapter card. Table F.2 lists the functionality of each of the 9 bits for both read accesses and write accesses along with the power-on default values for reads..

Table F.2 - Host Control Register

BIT	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Default	0	0	0	0	0	0	0	0
READ	Test Mode	In FIFO Full	ESP Hold	Out FIFO Full	Hold Ack	960 FAIL	Lock Hold	Board Reset
WRITE	unused	unused	unused	Clear GIO Int.	Set 960 Int.#1	Set 960 Int. #2	Lock Hold	Board Reset

- Bit 7 - Test Mode: This read-only bit lets the host know if the GBIA is currently in its loopback test mode. Writing to this bit has no effect, although it will not generate an error.
- Bit 6 - IN FIFO Full: Set to “1” when the DMA FIFO containing data read from the host is full. This bit is also read-only and can be written to with no effect nor error.
- Bit 5 - ESP Hold: Asserted when the ESP (SAR ASIC) is asserting the hold signal to the i960. Read-only as bits 6 & 7 above.
- Bit 4 - OUT FIFO Full/Clear GIO Interrupt: When read, this bit tells when the DMA request/write FIFO is full which will also hold the 960. When a “1” is written to the register, it will clear the GIO interrupt asserted from the adapter to the host. Writing a “0” has no effect.
- Bit 3 - Hold Acknowledge/Set Slave Interrupt #1: During reads, this bit tells the state of the hold acknowledge (HOLDA) signal from the i960 processor. Writing a “1” to the register asserts a slave interrupt to the 960. Writing a “0” has no effect.
- Bit 2 - 960 Fail/Set Slave Interrupt #2: Reading this bit allows the host to tell whether or not the 960 has failed its internal self test. Writing a “1” asserts the second slave interrupt to the 960. Writing a “0” has no effect.
- Bit 1 - Lock Hold: This read-write bit allows the host to assert the HOLD signal to the 960. As long as this bit is asserted, the i960 will be prevented from executing instructions.
- Bit 0 - Board Reset: The master reset for the GBIA and the adapter. The bit gets set by the GIO bus RESET signal and is cleared 63 bus cycles later. The bit can also be set/cleared by the host to reset the adapter.

F.1.3 Host Control Register #2

For uniformity across ASICs, the host interrupt mask bit has been moved to a separate register instead of being added to the original HCR.

- Bit 0 - GIO Interrupt Mask: Writing a “1” to this bit will prevent the GBIA from asserting the ~INT1 signal to the host. The interrupt bit internal to the GBIA will be set, so when the mask is cleared, the interrupt will get asserted to the host. The host can also read this bit to determine if the mask is currently set.

F.1.4 Adapter Configuration PROM

The Adapter Configuration PROM has been merged with the MON960 PROM from which the i960 boots. A 4KB section of the PROM has been reserved to use for any necessary information such as MAC address, serial number, etc. Table F.3 shows the address map of the merged PROM.

Table F.3 - PROM Address Map

Address	Contents	“User”
0x0000-0xDFFF	960 Code	i960
0xE000-0xEFFF	Adapter Configuration Info	Host Driver
0xF000-0xFFFF	960 Boot Record	i960

The portion of the PROM used for adapter configuration information is mapped into the GIO address space at an offset of 0x00080000 from the adapter base address. Since the slave interface is capable of word (32-bit) transfers only, accesses to the PROM must be made using a word access with the byte offset being multiplied by 4, and the data will be in the least significant byte of the returned word. For example, if byte 7 of the configuration information is 0x3D, a word access to 0x1F48001C will return 0x0000003D (assuming the card is in slot 0).

NOTE: The ASIC will prepend the required high address bits to move the PROM access into the appropriate section of the PROM. Only bits 13 through 2 are propagated from the GIO bus to the 960 address bus.

F.1.5 Local RAM

The 960 local memory consists of 256K of SRAM, of which all but the first 1K is accessible to the 960 and via slave access. The first 1K of the 960 address space is mapped into the on-chip RAM of the 960 and is therefore not accessible to the slave interface; conversely, the first 1K of the slave accessible RAM is not readable by the 960. The GBIA provides sufficient room in the address space and drives the appropriate address pins (19:2) to support up to 1MB of local RAM.

Accesses to the local memory require that the GBIA gain mastership of the 960 local bus. This makes RAM accesses relatively slow and of unpredictable length since the arbitration time is dependent on the 960 activity, and furthermore it takes multiple bus cycles to satisfy the RAM timings.

APPENDIX G PCA-200E PCI Bus Slave Interface

G.1 Slave Interface

The PBI requests 2 MB of memory space and 8 KB of expansion PROM space. The PBI maps 4 distinct devices (960 RAM, Host Control Register, Mask Register, and PROM) into this address space so that the host can access them as needed. Table G.1 shows the available devices and their offsets within the address space. Local RAM, HCR, Mask register, and PSR addresses offsets are referenced to Memory Space. The Expansion PROM is located at the Expansion PROM address specified in the Configuration Space Header. Since PCI requires reads of the expansion PROM to return 32-bit words, a single read on the PCI bus will result in four consecutive reads of the PROM.

Table G.1 - Slave Address Space

Addr. Byte Offset	Description	Notes
0x000000-0x0FFFFF	960 Local RAM	32 Bit, Read/Write
0x100000	Host Control Register (HCR)	Read/Write
0x100004	Host Interrupt Mask Register	Read/Write
0x100008	PCI Specific Register (PSR)	Read/Write
0x0000-0x1FFF	8 KB - 8B Expansion PROM	Read

The slave interface is not intended for heavy use; it supports only single accesses (no bursts). RAM and PROM accesses must stop the 960 from executing to allow access to shared resources. As such, use of the slave interface after initial start up should be limited to prevent system performance degradation.

The PCI specification requires that the first word of data must be transferred within 16 cycles of frame. If a target will not be able to complete the transfer within 16 cycle it should retry the bus immediately. For designs based on the 200E architecture, it is difficult to guarantee first data latency. Assuming a 33 MHz PCI bus it could take as long as 24 cycles to write RAM, 24 cycles to Read RAM, and 48 cycles to read the PROM. The PBI uses Maximum Slave Latency register to determine the maximum number of cycles that it will hold the bus without transferring data. The register defaults to 0xFF cycles. If strict PCI compliance is required, this register should be set to 0x10. The Maximum Slave Latency register is located at address 0x40 of Configuration Space.

G.1.1 Configuration Registers

Each PCI Board must have a special set of registers called the configuration registers. The first 256 bytes of configuration space is defined by the PCI local bus specification version 2.1. The PBI uses three additional bytes of configuration space for PCA200e specific functions. Configuration accesses are carried out using configuration bus cycles and configuration read or write commands. Configuration cycles are different then normal slave cycles. Refer to the PCI specification for further detail.

Table G.2 - Configuration Registers

bits 31 down to 24	bits 23 down to 16	bits 15 down to 8	bits 7 down to 0	addr.
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision Id	08h
BIST	Header	Latency Timer	Cache Line Size	0Ch
Memory Base Address				10h
Reserved				14h
Reserved				18h
Reserved				1Ch
Reserved				20h
Reserved				24h
Reserved				28h
Reserved				2Ch
Expansion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch
0x00	burst/req threshold	master control	max.slave latency	40h

The following table describes each of the fields that are present in PBI configuration space. Fields are one of three types: Read Only (R), Read / Write (R / W), or Read / Clear(R / C). Read / Clear bits are cleared to 0 by writing the bit position with 1. Writing a R / C bit with 0 does not effect the bit.

Table G.3 - PBI Configuration

Field Name	Bit	Type	Default Value	Description
Reserved	31 - 0	R	0x00000000	
Vendor ID	15 - 0	R	0x1127	FORE's Vendor ID.
Device ID	15 - 0	R	0x0300	Device ID selected for PCA200e
Command	0	R	0	Writes do not effect this bit. Always returns 0**.**
	1	R/W	0	A value of 1 allows the PBI to respond to slave accesses.
	2	R/W	0	A value of 1 allows the PBI to act as a bus master.
	3	R	0	Writes do not effect this bit. Always returns 0.
	4	R/W	0	A value of 1 allows the PBI to generate master write and invalidate commands.
	5	R	0	Writes do not effect this bit. Always returns 0.
	6	R/W	0	Parity error control bit. This bit is implemented as described by version 2.1 of the PCI spec.

Table G.3 - PBI Configuration

Field Name	Bit	Type	Default Value	Description
	7	R	0	Writes do not effect this bit. Always returns 0.
	8	R/W	0	SERR# enable. Implemented as described in version 2.1 of the PCI spec.
	9	R	0	Writes do not effect this bit. Always returns 0.
	15-10	R	0b000000	Writes do not effect these bits. Always returns 0's
Status	6-0	R	0b0000000	Writes do not effect these bits. Always returns 0's
	7	R	1	Writes do not effect this bit. Always returns 1.
	8	R/C	0	Master parity error status. Implemented as described in version 2.1 of the PCI spec.
	10-9	R	0b01	DEVSEL# timing. PBI does medium speed decoding of slave accesses. Writes do not effect these bits.
	11	R	0	Writes do not effect this bit. Always returns 0. The PBI never target aborts.
	12	R/C	0	Master transaction terminated by a target abort. Implemented as described in version 2.1 of the PCI spec.
	13	R/C	0	Master transaction terminated by a master abort. Implemented as described in version 2.1 of the PCI spec.
	14	R/C	0	PBI asserted SERR#. Implemented as described in version 2.1 of the PCI spec.
	15	R/C	0	Parity Error Status bit. Implemented as described in version 2.1 of the PCI spec.
Revision ID	7-0	R	0x00	PBI Revision ID. Writes do not effect these bits.
Class Code	23-0	R	0x020300	Specifies that the PBI is an ATM Network Controller.
BIST	7-0	R	0x00	No BIST support. Writes do not effect these bits. Always returns 0's
Header Type	7-0	R	0x00	Specifies a single function device and header type 0. Writes do not effect these bits. Always returns 0's
Latency Timer	7-0	R/W	0x00	Latency Timer Register. All bits are R/W.
Cache-line Size	7-0	R/W	0x00	Cache-line Size Register. The PBI implements this register as described in version 2.1 of the PCI specification.

Table G.3 - PBI Configuration

Field Name	Bit	Type	Default Value	Description
Memory Base Address	17-0	R	0x00000	Indicates locate anywhere in 32 bit address space and not prefetchable.
	31-18	R/W	0x0000	Request a 2 MB memory address space
Expansion ROM Base Address	0	R/W	0	Enable or disable PROM accesses.
	12-1	R	0x0000	
	31-13	R/W	0x00000	Request a 8 KB Expansion PROM address space.
Interrupt Line	7-0	R/W	0x00	
Interrupt Pin	7-0	R	0x01	
Min_GNT	7-0	R	????	
Max_Lat	7-0	R	????	
Max. Slave Lat	7-0	R/W	0xFF	Defines the max. number of cycles before we Retry.
Master Control	0	R/W	0	Disable cache-line Reads
	1	R/W	0	Disable Write and Invalidates
	2	R/w	0	Require 2 Cache-lines for Write and Invalidate.
	3	R/W	0	Ignore the Latency Timer.
	4	R/W	0	Enable Continuous Request Mode.
	5	R/W	0	Force large PCI bus bursts.
	6	R/W	0	Convert endianess of Slave RAM accesses
	7	R	0	
Threshold	7-0	R/W	0x00	Large burst or Continuous Request Threshold Register.

G.1.2 Host Control Register (HCR)

The HCR is an 8-bit control and status register accessible from the PCI bus. The register provides the host with a number of control and status bits for controlling and monitoring the adapter card. Table 3 lists the functionality of each of the 8 bits for both read accesses and write accesses along with the power-on default values for reads.

Table G.4 - Host Control Register

BIT	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Default	0	0	0	0	0	0	0	0
READ	Test Mode	In FIFO Almost Full	ESP Hold	Out FIFO Full	Hold Ack	960 FAIL	Lock Hold	Board Reset
WRITE	unused	unused	unused	Clear Host Int.	Set 960 Int. 1	Set 960 Int. 2	Lock Hold	Board Reset

- Bit 7 - Test Mode: This read-only bit lets the host know if the PBI is currently in its loopback test mode. Writing to this bit has no effect.
- Bit 6 - In FIFO Almost Full: A read only bit giving the status of the In FIFO. Can be used with the OUT FIFO Full bit to determine if there is a deadlock condition (see Section 4.3).
- Bit 5 - ESP Hold: Asserted when the ESP (SAR ASIC) is holding the i960. Read-only as described in the bit 7 description above.
- Bit 4 - OUT FIFO Full/Clear PCI Interrupt: When read, this bit tells when the DMA request/write FIFO is full which will also hold the 960. When a “1” is written to the register, it will clear the PCI interrupt asserted from the adapter to the host. Writing a “0” has no effect.
- Bit 3 - Hold Acknowledge/Set Slave Interrupt A: During reads, this bit tells the state of the hold acknowledge (HOLDA) signal from the i960 processor. Writing a “1” to the register asserts a slave interrupt to the 960. Writing a “0” has no effect.
- Bit 2 - 960 Fail/Set Slave Interrupt B: Reading this bit allows the host to tell whether or not the 960 has failed its internal self test. Writing a “1” asserts the second slave interrupt to the 960. Writing a “0” has no effect.
- Bit 1 - Lock Hold: This read-write bit allows the host to assert the HOLD signal to the 960 (by writing a “1”). As long as this bit is asserted, the i960 will be prevented from executing instructions.
- Bit 0 - Board Reset: The master reset for the PBI and the adapter. The bit gets set by the PCI bus RESET signal and is cleared 63 bus cycles later. The bit can also be set/cleared by the host to reset the adapter by writing a “1”.

G.1.3 Mask Register

The mask register is an 1 bit register that can be read or written by the host to set and release the host interrupt mask. Writing a ‘1’ to bit 0 will set the mask, writing a ‘0’ to bit 0 will release the mask. Interrupts may still be registered while the mask is on. Reading the register will return the state of the Host Interrupt Mask bit in position 0, while bits [31:1] will be 0.

G.1.4 PCI Specific Register (PSR)

The PSR is an 1 bit control and status register accessible from the PCI bus. The register indicates whether or not the PBI has asserted the Host Interrupt (Interrupt Pending).

Bit 0 - Active high indication of a pending host interrupt. This read only indication is not masked by the host interrupt mask. The pending bit will return to 0 when the host interrupt has been cleared.

G.1.5 Adapter Expansion PROM

The Adapter Expansion PROM is a 2Kx8 bit PROM mapped into the Expansion PROM address space. It is used for holding static information about the adapter card such as serial number, firmware and hardware revisions, MAC address, etc. The PROM is read through the slave interface. A single PCI bus read will cause four back-to-back reads of the PROM. The four 8-bit values will be combined and returned to the PCI bus as a 32-bit word.

To extract data from PROM, a command is written to the Command Queue, using a buffer supplied by the host, and an opcode value of 0x09000000. (Note the preferred method of doing this is to alter the "opcodes" enumeration, found in ./include/fore_cp/aali.h, to include "op_get_prom_data = 0x09000000", then to use the new element to identify the get-PROM-data command.) The data returned from the firmware is defined by the following data structure:

```
typedef struct PEXPANSION_ROM_DATA{
 unsigned long version;
 unsigned long serial;
 unsigned char macAddr[8];    /* not 6, and byte swapped */
} PEXPANSION_ROM_DATA;
```

When the command has been completed, extracting the version data can be done with the following statements (where PPromData is a pointer to the buffer supplied with the op_get_prom_data command):

```
Adapter->HardwareVersion = CTOHL(PPromData->version);
Adapter->SerialNumber = CTOHL(PPromData->serial);
```

Retrieving the MAC address is dependent upon whether or not you are using the big or little endian firmware on the PCA-200 card:

```
Adapter->macAddr.Byte[0] = PPromData->macAddr[gpd_mac_order[0]];
Adapter->macAddr.Byte[1] = PPromData->macAddr[gpd_mac_order[1]];
Adapter->macAddr.Byte[2] = PPromData->macAddr[gpd_mac_order[2]];
Adapter->macAddr.Byte[3] = PPromData->macAddr[gpd_mac_order[3]];
Adapter->macAddr.Byte[4] = PPromData->macAddr[gpd_mac_order[4]];
Adapter->macAddr.Byte[5] = PPromData->macAddr[gpd_mac_order[5]];
```

For little endian, gd_mac_order[] is be defined as:

```
int gpd_mac_order[6] = { 2, 3, 4, 5, 6, 7 };
```

For big endian firmware, the following ordering is defined:

```
int gpd_mac_order[6] = { 1, 0, 7, 6, 5, 4 };
```

G.1.6 Local RAM

The 960 local memory consists of 256 KB of SRAM. The PBI provides sufficient room in the address space and drives the appropriate address pins (19:2) to support up to 1MB of local RAM.

Accesses to the local memory require that the PBI gain mastership of the 960 local bus. This makes RAM accesses relatively slow and of unpredictable length since the arbitration time is dependent on the 960 activity, and furthermore it takes multiple bus cycles to satisfy the RAM timings.

APPENDIX H VMA-200E VME-bus Slave Interface

H.1 VMA-200E Pre-Installation Configuration

Before installing the VMA-200E, be sure to make any needed modifications to the board. The following information pertains to DIP switches on the VMA-200E and their effect on the operation of the card. The default settings are shown shaded below:

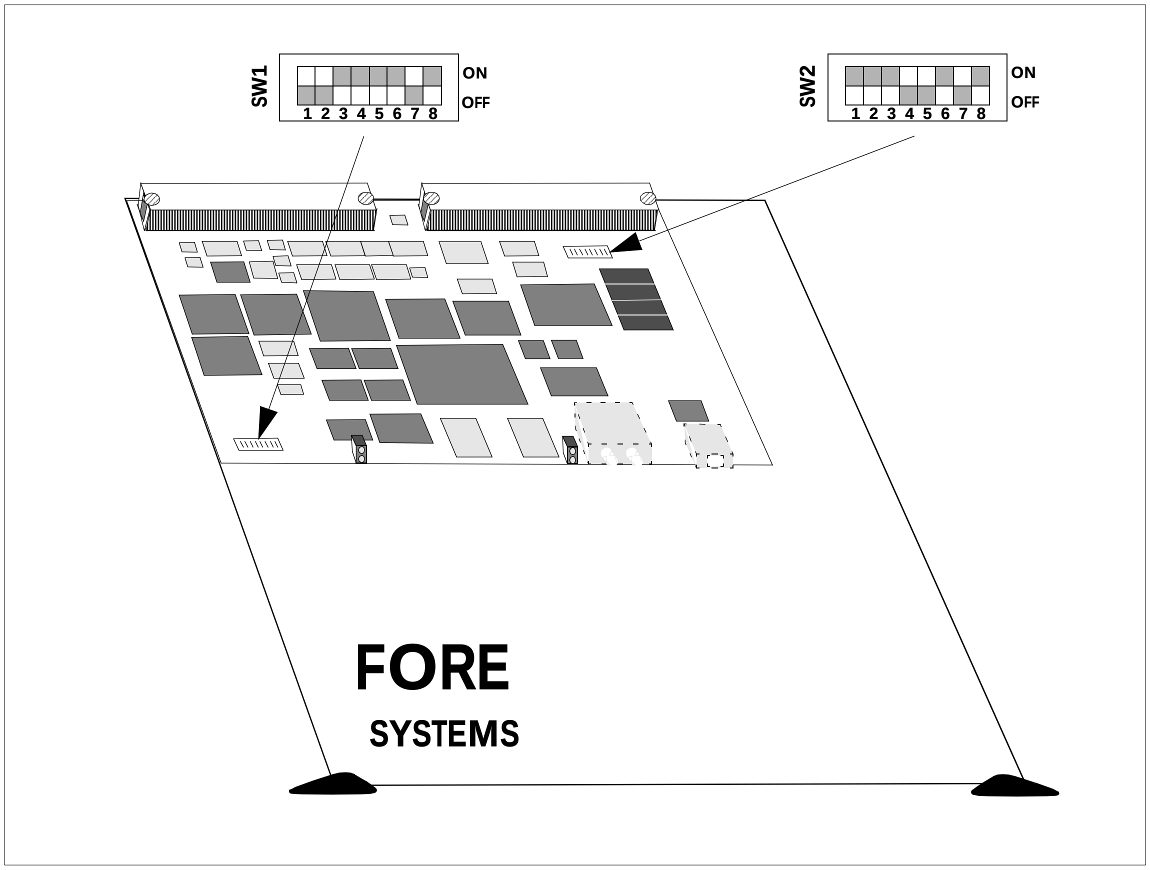


Figure H.1 - VMA-200E DIP Switch Locations.

H.1.1 Slave Address Configuration

NOTE: If more than one VMA-200E will be installed in a system, the DIP switch settings will need to be changed. See the following information for details.

To enable proper VMEbus address decoding, two 8 position DIP switches are provided for address selection. Switch SW1 is used to qualify the A16 short address space. Switch SW2 is used to qualify the A32 extended address.

H.1.2 VMEbus Short A16 Address Space

The Short A16 address space is decoded using the four most significant bits of the 16 bit VMEbus address (A15:12). SW1 switch positions 1 through 4 must be configured to reflect the hexadecimal value chosen for the A16 base address.

NOTE: The configuration files supplied by FORE Systems require that the A16 address space DIP switches be set to either 1100, 1101, 1110 or 1111.

The following Short base address configuration table shows SW1 configuration.

Table H.1 - Short Base Address Configuration

VME Short base address VME addr<15:12> (binary)	SW1 pos. 1	SW1 pos. 2	SW1 pos. 3	SW1 pos. 4
0000	on	on	on	on
0001	on	on	on	off
0010	on	on	off	on
.	.	.	.	.
.	.	.	.	.
1101	off	off	on	off
1110	off	off	off	on
1111	off	off	off	off

In addition to the A16 short base address, further address decoding is used to specify the individual devices. The VMEbus address bits 11:10 provide access to the following short devices.

NOTE: The defined short data access width is 16 data bits, with only 8 data bits significant.

Table H.2 - Device Access Decoding Information

VMEbus addr<11:10>	VME valid data	Device	Access
00	D15:8	Board ID prom	Read only
01	D7:0	Host control register	Read/Write
10	D7:0	Interrupt vector register	Read/Write

H.1.3 VMEbus Identification PROM

The VMEbus ID prom is a read only byte access device. The device contains FORE Systems information specific to the card, such as serial number, hardware version number, etc. This is a read only device and drives VME data bits 15:8.

The PROM data is of the form:

```
<type><name_len><attribute_name><val_len><attribute_value>
```

where <type> is either "1" for integer or "2" for string value. The <attribute_name> field is a string value which names the value, and <attribute_value> is dependent on <type>. <name_len> and <val_len> are needed to indicate the size of the attribute fields.

The data written to the PROM are:

```
string: model = "FORE, VMA-200E"
int   : serial-number = xxxxx
int   : hw-version    = x.x
int   : mac-addrhi4   = mac_addrhi4
int   : mac-addrl02   = serial_number
string: copyright     = "copyright 19xx by Fore Systems, Inc."
int   : ""            = 0  (null)
```

H.1.4 Host Control Register

The host control register provides various functions to the host processor to drive and monitor board control and status data. This register is a *read-modify-write* type register. Consequently, all write operations are preceded by read accesses to maintain register state. The register is accessed via VMEbus data bits 7:0 . The following table shows bit definitions and function descriptions for the register's write and read states.

Table H.3 - Host Control Register

VME Data Bit	Write Description	Read Description
0	CPU reset, sticky (maintains state); Resets the i960 and periphery logic	CPU reset
1	Hold Lock, sticky bit (maintains state); Sets the i960 hold local bus request input	Hold Lock
2	Sets the i960 slave interrupt	VME64 mode
3	None	i960 Fail output
4	None	Board test mode

H.1.5 VMEbus A32 Long Slave Address Space

VMEbus A32 Extended Slave address space is decoded using the eight most significant bits of the 32 bit VMEbus address bus (A31:24). Therefore, SW2 switch positions 1 through 8 must be configured to reflect the byte-wide hexadecimal values chosen for A32 base addressing.

NOTE: In the following table, binary 0=ON and binary 1=OFF.

Table H.4 - A32 Base Addressing

VME Long base addr VA31:24(hex)	SW2 pos. 1	SW2 pos. 2	SW2 pos. 3	SW2 pos. 4	SW2 pos. 5	SW2 pos. 6	SW2 pos. 7	SW2 pos. 8
00	on	on	on	on	on	on	on	on
01	on	on	on	on	on	on	on	off
02	on	on	on	on	on	on	off	on
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
FD	off	off	off	off	off	off	on	off
FE	off	off	off	off	off	off	off	on
FF	off	off	off	off	off	off	off	off

NOTE: The configuration files supplied by FORE Systems require that a card which is assigned A16 address space 1100 also be assigned A32 address space 1A. Similarly, A16 address space 1101 goes with A32 address space 1B, 1110 goes with 1C, and 1111 goes with 1D.

H.1.6 VMEbus Interrupter

The VMEbus Interrupter circuitry consists of a programmable vector ID register, an interrupt level selector, and hardware daisy-chain logic. The VMEbus interrupter functions as a single level interrupt generator and as an interrupt acknowledge cycle participator.

Prior to interrupt cycle operations, the user and the host driver must initialize the card. First, the user must choose the VMEbus interrupt request level for the VMA-200 card. The interrupt level is then encoded into DIP switch SW1, positions 6 through 8. The encoding is similar to the short and long address initializations. This procedure is performed only during a non-powered situation. After power-up, the host must initialize the 8-bit vector ID register prior to interrupt generation by the VMA-200 to the host.

Table H.5 - Interrupt Request Level

VMEbus Interrupt request level	SW1 pos. 6	SW1 pos. 7	SW1 pos. 8
1	on	on	off
2	on	off	on
3	on	off	off
4	off	on	on
5	off	on	off
6	off	off	on
7	off	off	off

NOTE: In the above table, binary 0=ON and binary 1=OFF.

H.1.7 VME64 Master Mode

VME64 Master mode is controlled with SW1, position 5 as shown in the following table:

Table H.6 - VME64 Master Mode Selector

Mode	SW1 pos. 5
VME64 Master	on
VME32 Master only	off

APPENDIX I MCA-200 Micro Channel Bus Slave Interface

I.1 Slave Interface

The slave interface accepts two kinds of slave accesses, I/O accesses and memory accesses.

The board control register, Micro Channel POS registers, and adapter information PROM are accessible via slave I/O accesses to the MCA-200. The MCA-200 responds to a slave I/O access when M/-IO is driven low and the bits 15 through 10 of the address on the bus match the I/O start address contained in the POS register. The table below shows the I/O space of the adapter card. Addresses not appearing in the table are invalid and should not be accessed.

Table I.1 - I/O Space

AD<9:0>	Device	Access
0x100	Board Control Register	Write only
0x200-0x3FF	Configuration PROM	Read only

The MCA-200 is selected for a slave memory access when M/-IO is high, the MSB of the address (A31:24) is 0x00 and A23:20 match the value in the Mem Address POS register. The matching scheme chosen allows for expansion of adapter card memory to 1MB. Users must be aware that current cards, with 256KB of memory, do not decode address bits 18 and 19.

Memory accesses are 32-bits wide only, and must be single transfers since the MCA-200 does not support streaming data transfers as a slave.

I.2 Board Control Register

The board control register (BRDCTRL) performs a number of operations based upon the data written to it. The legal values and their corresponding functions are as follows:

Table I.2 - Board Control Registers

d(2:0)	Action	Description
0	De-assert Board Reset	De-assert Board Reset clears the reset signals to the 960 and the bus master state machine. The 960 will begin execution after this value is written to the board control register.
1	Assert Board Reset	Assert Board Reset causes the reset signals to various devices on the board to be asserted including: the 960, the local bus controller, the IN and OUT fifos, the network interface status, and the receive and transmit fifos. The reset action also clears the hold bit for the local bus and the interrupt bits for the 960 and for the Micro Channel. The reset bit must remain set for a minimum of 16 Micro Channel cycles, and cleared afterwards. The Micro Channel CHRESET signal asserted at boot time causes a similar reset action (in addition to resetting the Micro Channel controller).
2	Hold Local Bus	Hold local bus causes the slave interface to gain mastership of the local bus and to keep it as long as this bit is set. This feature can be used for read-modify-write of critical memory regions by the host. It can also be used to reduce access time when the host is accessing a block of local memory since it eliminates the time to request and gain the local bus mastership from the 960 in slave accesses. Note that holding the local bus locks the 960 out of the local bus and effectively halts it. Therefore this bit must be set only when halting the 960 is acceptable.
3	Interrupt 960	960 interrupt asserts the XINT0 line of the 960 processor. The bit can only be cleared by the 960 (upon servicing the interrupt); it can not be cleared by the host by writing a 0.
4	Clear Micro Channel Interrupt	Clear Micro Channel interrupt clears the bit which is normally set by the 960 in the bus control register to generate an interrupt to the host. Note that unlike the other bits this bit does not "stay around"; writing a 1 to this bit location simply causes the Micro Channel interrupt to be cleared.

NOTE: Writing to the board control register does not require the slave interface to gain the mastership of the local bus from the 960. This immediate access capability is valuable in cases where the 960 may be hung on a local bus transfer and the board needs to be reset.

I.3 Local Memory

The local memory consists of SRAM ranging in size from 256K bytes to 1M bytes depending on the implementation. The entire memory is accessible by the host. Note, however, that the first 1K bytes of the local memory is NOT accessible by the 960 as this address space is mapped to the on-chip RAM.

Accessing the local memory requires gaining mastership of the local bus from the 960, therefore it requires multiple cycles depending on the bus activity of the 960. Once the local bus mastership is gained it takes multiple Micro Channel cycles to write to or read from the local memory. The local bus hold feature described above can be used to speed up block accesses to the local memory.

I.4 Prom Data

The format of attributes written to the PROM is as follows:

<type><attribute_length><attribute_name><value_length><value>

where type=1 indicates a string, and type=2 indicates an integer.

The following attributes are written in the following order to the MCA-200E PROM:

Table I.3 - PROM Attributes

TYPE	ATTRIB_LENGTH	ATTRIBUTE_NAME	VALUE_LENGTH	VALUE	LOCATION
1	5	model	14 (0x0E)	FORE, MCA-200E	8
2 (0X26)	13 (0x0D)	serial-number	4	1 (ex)	38
2 (0X37)	10 (0x0A)	hw-version	4	1 (ex)	55
2 (0X49)	11 (0x0B)	mac-addrhi4	4	0020480B	73
2 (0X5B)	11 (0x0B)	mac-addrlo2	4	1 (ex)	91
2 (0X6B)	9	copyright	36 (0x24)	"Copyright..."*	107

* the copyright notice is a standard notice, yet is truncated within the table (see below).

NOTE:

- LENGTHs are in BYTEs, LOCATION is the number of BYTEs from the beginning.
- VALUEs for serial-number, hw-version, and mac-addrlo2 are variable.
- * serial-number and mac-addrlo2 are equivalent.
- * serial-numbers 0 - 8191 are reserved for oc3
- * serial-numbers 8192-16383 are reserved for utp
- * serial-numbers 16384 - 24575 are reserved for taxi

An example dump of the PROM contents is shown below for an MCA-200E with

a serial number (and mac-addrlo2) of '1' and a hardware version of '1':

Table I.4 - PROM Contents

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00000000	01	05	6D	6F	64	65	6C	0E	46	4F	52	45	2C	20	4D	43	.. model. FORE,
00000010	41	2D	32	30	30	45	02	0D	73	65	72	69	61	6C	2D	6E	MCA-200E........
00000020	75	6D	62	65	72	04	00	00	00	01	02	0A	68	77	2D	76	serial number...
00000030	65	72	73	69	6F	6E	04	00	00	00	01	02	0B	6D	61	63	..hw version..
00000040	2D	61	64	64	72	68	69	34	04	00	20	48	0B	02	0B	6D	mac addrhi4..H
00000050	61	63	2D	61	64	64	72	6C	6F	32	04	00	00	00	01	01	.mac addrlo2. n
00000060	09	63	6F	70	79	72	69	67	68	74	24	43	6F	70	79	72	.copyright$
00000070	69	67	68	74	20	31	39	39	36	20	62	79	20	46	6F	72	Copyright 1996
00000080	65	20	53	79	73	74	65	6D	73	2C	20	49	6E	63	2E	02	by FORE Sys-
00000090	00	04	00	00	00	00											tems, Inc.

APPENDIX J SBA-200 SBus Slave Interface

J.1 Slave Interface

The slave interface supports only single-word accesses. No burst, half-word, or byte accesses are supported.

The table below shows the valid addresses for a slave access.

Table K.1 - Slave Interface Addresses

PA<25:0>	Device	Access
0000000 - 0007FFC	SBus boot PROM	Read Only
100000	Board Control Register	Write Only
2000000 - 23FFFFC	Local RAM	Read / Write

J.2 SBus Boot PROM

The SBus boot prom is a 32K byte PROM containing SBus boot Forth code and board identification. Access to the PROM takes multiple cycles, including gaining mastership of the local bus followed by a 4 cycle PROM access since the PROM is a slow device.

J.3 Board Control Register

The Board Control Register contains the following control bits - writing a 1 to the bit position causes the corresponding action.

Table K.2 - Board Control Registers

d(2:0)	Action	Description
0	Board Reset	Board Reset bit causes the reset signals to various devices on the board to be asserted including: 960, local bus controller, IN and OUT fifos, and the network interface status, and receive and transmit fifos. The reset action also clears the hold bit for the local bus and the interrupt bits for the 960 and for the SBus. The reset bit must remain set for a minimum of 16 SBus cycles, and cleared afterwards. The SBus Reset signal asserted at boot time causes a similar reset action.
1	Hold Local Bus	Hold Local Bus bit causes the slave interface to gain mastership of the local bus and to keep it as long as this bit is set. This feature can be used for read-modify-write of critical memory regions by the host. It can also be used to reduce access time when the host is accessing a block of local memory since it eliminates the time to request and gain the local bus mastership from the 960 in slave accesses. Note that holding the local bus locks the 960 out of the local bus and effectively halts it. Therefore, this bit must be set only when halting the 960 is acceptable.
2	Interrupt 960	960 Interrupt bit asserts the INTXX line of the 960 processor. The bit can only be cleared by the 960 (upon servicing the interrupt); it can not be cleared by the host by writing a 0.
3	Clear SBus Interrupt	Clear SBus Interrupt bit clears the SBus interrupt bit which is normally set by the 960 to generate an interrupt to the host. Note that unlike the other bits this bit does not “stay around” writing a 1 to this bit location simply causes the SBus interrupt to be cleared.

NOTE: Writing to the board control register is **immediate**. That is, the slave interface does not wait to gain the mastership of the local bus from the 960. This immediate access capability is valuable in cases where the 960 may be hung on a local bus transfer and the board needs to be reset.

J.3.1 Local Memory

The local memory consists of SRAM ranging in size from 256K bytes, 512K bytes, to 1M bytes depending on the implementation. The entire memory is accessible by the host. Note, however, that the first 1K bytes of local memory is NOT accessible by the 960 as this address space is mapped to the on-chip RAM.

Accessing the local memory requires gaining mastership of the local bus from the 960, therefore requires multiple cycles depending on the bus activity of the 960. Once the local bus mastership is gained it takes two SBus cycles to write to the local memory, and three SBus cycles to read from the local memory. The local bus hold feature described above can be used to speed up block accesses to the local memory.

APPENDIX K ESA-200 EISA Bus Slave Interface

K.1 Slave Interface

The slave interface is divided into two sections: one for I/O accesses and the other for memory space accesses. The configuration PROM, board control register and memory base address register reside in the I/O space for the card. All of the I/O space devices are accessed through byte lane 0 of the EISA bus (bits 0 to 7) and can only be addressed as a byte-wide device.

NOTE: The i960 RAM interface supports only single 32-bit word accesses. No burst, half-word, or byte accesses are supported.

The table below shows the address map for slave accesses to I/O space:

Table L.1 - Slave Access Address Map

EA<11:0>	Device	Access
0z000	Board control register	Write only
0z400	Memory base address reg. (high slice)	Write only
0z404	Memory base address reg. (low slice)	Write only
0zC00 - 0zCFF	EISA configuration PROM	Read only

In the table, "z" is the number of the EISA slot in which the card resides.

K.1.1 EISA bus Configuration PROM

The EISA bus configuration PROM is a 256 byte PROM. It contains the EISA board product ID at 0zC80 to 0zC83, plus any other information which is specific to the card (serial number, version number, 802 address, etc.) Access to the PROM takes multiple cycles since it is a slow device.

K.1.2 Contents of the ESA-200 Configuration PROM

(I/O addresses expressed in hexadecimal)

```
0xC80 - 0xC83: EISA configuration ID (4 bytes) - either:

               FSI2001 (original ESA-200)
               FSI2002 (ESA-200A)
               FSI2003 (ESA-200E)

0xC80 - 0xC8B: Hardware revision code
Byte 0xC8A: Lower 4 bits contain the sub_code
Byte 0xC8B: Upper 4 bits contain the major_code
            Lower 4 bits contain the minor_code
```

The hardware version reported by the system is of the format:

```
major_code.minor_code.sub_code
```

```
0xC94 - 0xC97: Serial number - 32-bit value which gives the serial
            number of the board
```

Format of the serial number is “byte0:byte1:byte2:byte3”, where “byte0” is the most significant 8 bits of the serial number and “byte3” is the least significant.

```
Byte 0xC94: byte0
Byte 0xC95: byte1
Byte 0xC96: byte2
Byte 0xC97: byte3
```

```
0xCA0 - 0xCA5: MAC Address - Physical address of the ESA-200 card
```

Format of the MAC address is “byte0:byte1:byte2:byte3:byte4:byte5”

```
Byte 0xCA0: byte0
Byte 0xCA1: byte1
Byte 0xCA2: byte2
Byte 0xCA3: byte3
Byte 0xCA4: byte4
Byte 0xCA5: byte5
```

K.1.3 Memory Base Address Register

The memory base address register holds the location of the window into the i960 RAM from the EISA bus. At system configuration time, 1 Mbyte of the EISA address space must be allocated for the ESA-200 card.

The upper twelve bits of the assigned address are written to the two memory base address registers (each 6 bits wide) in the following manner:

Table L.2 - EISA Base Address for the i960 RAM

b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	Lower 20 unmatched bits.

Table L.3 - Memory Base Address Register, High Slice

/b31	/b30	/b29	/b28	/b27	/b26	0	0

Table L.4 - Memory Base Address Register, Low Slice

/b25	/b24	b23	b22	b21	b20	0	0

The "/" in the above tables indicates that the bit is negated. The address comparator remains inactive until both halves of the base address have been written. Once the register has been written at configuration time, no further writes should be performed to the register.

K.1.4 Board Control Registers

By writing an encoded value to the board control register, a control bit or group of control bits on the adapter card is toggled. The allowable values and their corresponding functions are as follows:

Table L.5 - Board Control Registers

d(2:0)	Action	Description
0	Board reset	Causes the reset signals to various devices on the board to be asserted including: i960, local bus controller, bus fifos, the network interface status, and receive and transmit fifos. The reset action also clears the hold bit for the local bus and the interrupt bits for the i960 and for the EISA bus. The reset bit must remain set for a minimum of 16 EISA bus cycles before the CPU reset can be cleared. The EISA bus RESDRV signal asserted at boot time causes a similar reset action.
1	Clear CPU reset	De-asserts the line which keeps the i960 and the devices on its bus in the reset state. Typically written after powerup reset or after writing Board Reset.
2	Set local bus hold	Causes the slave interface to gain mastership of the i960 local bus and to keep it until Clear Local Bus Hold is written . This feature can be used for read-modify-write of critical memory regions by the host. Note that holding the local bus locks the i960 out of the local bus and effectively halts it. Therefore this function must only be used when halting the i960 is acceptable.
3	Clear local bus hold	Undoes the hold caused by Set Local Bus Hold.
4	Interrupt i960	Asserts the INT0 line of the i960 processor. The interrupt can only be cleared by the i960 (upon servicing the interrupt); it can not be cleared by any writes to the board control register by the host.
5	Clear EISA bus interrupt	Clears the EISA bus interrupt which is set by the i960 to the bus.
6	EISA interrupt select IRQ5	One of four board control register values which determines the EISA interrupt request line for interrupts from the adapter card to the host processor. If the host writes this value, then EISA interrupts will be asserted on line IRQ5. If the host writes one of the three other values, then IRQ5 is deselected and a different interrupt line is used for host interrupts. The value of the interrupt select should be chosen during system configuration and should not change during run-time.
7	EISA interrupt select IRQ11	If the host writes to this location, then interrupts to the host will be asserted on line IRQ11.
8	EISA interrupt select IRQ10	If the host writes to this location, then interrupts to the host will be asserted on line IRQ10.
9	EISA interrupt select IRQ12	If the host writes to this location, then interrupts to the host will be asserted on line IRQ12.

Writing to the board control register is **immediate**. To clarify, the slave interface does not wait to gain the mastership of the local bus from the i960. This immediate access capability is valuable in cases where the i960 may be hung on a local bus transfer and the board needs to be reset.

APPENDIX L GIA-200 GIO Bus Slave Interface

L.1 Slave Interface

The board control register, GIO bus PIW register, 960 RAM, and adapter configuration information PROM are accessible via slave accesses to the GIA-200. The GIA-200 responds to a slave access whenever it detects its address being driven onto the bus while the bus address strobe signal is asserted.

L.1.1 Memory Base Address

The table below shows the address space of the GIA-200 as seen from the host.

Table M.1 - Slave Interface Addresses

AD<20:0>	Device	Access
0x000000	PIW Register	Read Only
0x080000-- 0x080800	Information PROM	Read Only
0x0C0000	Board Control Register	Write Only
0x100000 -- 0x1FFFFF	960 RAM	Read / Write

NOTE: Memory accesses are 32-bits wide only, and must be single transfers since the GIA-200 does not support burst transfers as a slave.

L.2 Board Control Register

The board control register (BRDCTRL) performs a number of operations based upon the data written to it. The allowable values and their corresponding functions are as follows:

Table M.2 - Board Control Registers

Value	Function	Description
0	De-assert Board Reset	Cancels a reset before an actions are taken.
1	Assert Board Reset	Causes the reset signals to various devices on the board to be asserted including: the 960, the local bus controller, the IN and OUT fifos, the network interface status, and the receive and transmit fifos. The reset action also clears the hold bit for the local bus and the interrupt bits for the 960 and for the GIO bus. The reset bit must remain set for a minimum of 16 GIO bus cycles, and cleared afterwards. The GIO bus RESET signal asserted at boot time causes a similar reset action.
2	Hold Local Bus	Causes the slave interface to gain mastership of the local bus and to keep it as long as this bit is set. This feature can be used for read-modify-write of critical memory regions by the host. It can also be used to reduce access time when the host is accessing a block of local memory since it eliminates the time to request and gain the local bus mastership from the 960 in slave accesses. Note that holding the local bus locks the 960 out of the local bus and effectively halts it. Therefore this bit must be set only when halting the 960 is acceptable.
3	Interrupt 960	Asserts the XINT0 line of the 960 processor. The bit can only be cleared by the 960 (upon servicing the interrupt); it can not be cleared by the host by writing a 0.
4	Clear GIO bus Interrupt	Clears the bit which is normally set by the 960 in the bus control register to generate an interrupt to the host. Note that unlike the other bits this bit does not “stay around”; writing a 1 to this bit location simply causes the GIO bus interrupt to be cleared. Writing to the board control register does not require the slave interface to gain the mastership of the local bus from the 960. This immediate access capability is valuable in cases where the 960 may be hung on a local bus transfer and the board needs to be reset.

APPENDIX M PCA-200 PCI Bus Slave Interface

M.1 Slave Interface

The slave interface is divided into two sections: one for configuration space accesses and the other for memory space accesses. The configuration registers, board control register and serial number PROM reside in the configuration space for the card. The 960 SRAM interface supports only single-word accesses. No burst, half-word, or byte accesses are supported.

The table below shows the high-level address map for slave accesses to configuration space:

AD<11:0>	Device	Access
00h -3Ch	PCI Configuration Registers	Read / write
40h - 7Ch	Bus Control Register	Write only
80h - FCh	Serial Number PROM	Read only

M.1.1 Configuration Space Registers

The mandatory configuration registers for the PCI bus are implemented on the PCA-200. The format for the registers is as found in the bus specification:

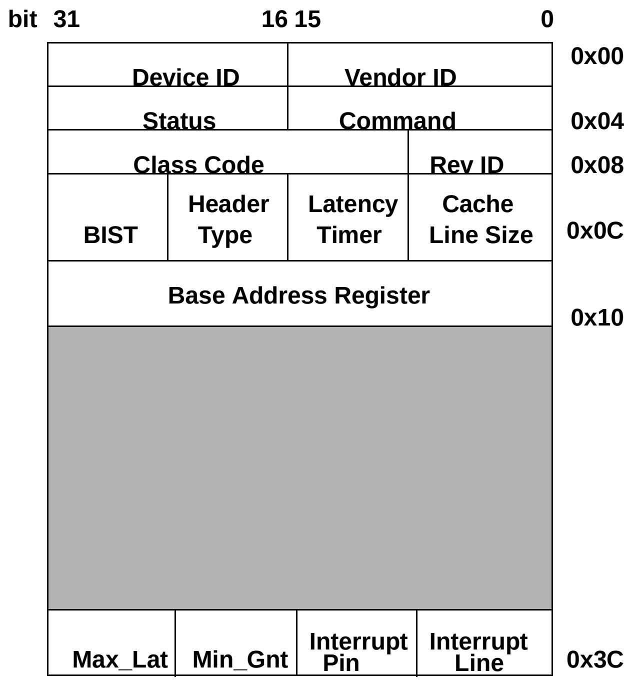


Figure N.1 - Configuration Space Registers

- Vendor ID (addr 0x00, bits 15:0) - Sixteen bit vendor identification number. FORE Systems ID = 0x1127.
- Product ID (addr 0x00, bits 31:16) - Major revision number of hardware. Set to 0x0210 for the PCA-200A.
- I/O Space Control (addr 0x04, bit 0) - Controls the card's response to I/O space accesses. A value of 0 disables the card response. A value of 1 allows the card to respond to I/O space accesses.
- Memory Space Control (addr 0x04, bit 1) - Controls the card's response to memory space accesses. A value of 0 disables the card response. A value of 1 allows the card to respond to memory space accesses.
- Master Control (addr 0x04, bit 2) - Controls the card's ability to act as a master on the PCI bus. A value of 0 disables the card from generating PCI accesses. A value of 1 allows the card to behave as a bus master.
- Target-Abort Flag (addr 0x04, bit 28) - This bit must be set by a master device whenever its transaction is terminated with target-abort.
- Master-Abort Flag (addr 0x04, bit 29) - This bit must be set by a master device whenever its transaction is terminated with master-abort.
- Parity Error Flag (addr 0x04, bit 31) - This bit is set by the card whenever it detects a parity error.
- Class Code and Revision ID (addr 0x08, bits 31:0) - Field set to value 0x02030000.
- Cache Line Size (addr 0x0C, bits 7:0) - Register specifies the number of 32-bit words in a memory cache line (written by the system BIOS). Register is read/write, initialized to zero on bus reset.
- Latency Timer (addr 0x0C, bits 15:8) - Register specifies, in units of PCI bus clocks, the value of the Latency Timer. The Latency Timer determines the maximum amount of time the card may be a master on the bus. The register is fully read/writeable.
- Header Type (addr 0x10, bits 23:16) - Read-only field set to 0x00, indicates single-function device.
- BIST (addr 0x10, bits 31:24) - Built-in Self Test field set to 0x00 (not supported)
- Base Address Register (addr 0x10, bits 31:20) - Read/write register which determines the mapping of the 960 SRAM into both I/O and memory space. The unused bits of the register (19 to 0) always return zero when read.
- Interrupt Line Register (addr 0x3C, bits 7:0) - The Interrupt Line register is an 8-bit register used to communicate interrupt routing information. The register is read/write, and has a value of zero after board reset. POST software will write the routing information into this register as it initializes and configures the system.
- Interrupt Pin Register (addr 0x3C, bits 15:8) - Register is set to value 0x01 (read only). Indicates that interrupts to the bus are driver on line INTA.
- Minimum Grant (addr 0x3C, bits 23:16) - Read-only register, set to value 0x10.
- Maximum Latency (addr 0x3C, bits 31:24) - Read-only register, set to value 0x10.

M.1.2 Board control register

The board control register performs a number of operations based upon the data written to it. The register resides at address 0x40 in the board's configuration address space. The legal values and their corresponding functions are as follows:

Table N.1 - Board Control Registers

d(2:0)	Action	Description
0	De-assert Board Reset	De-assert Board Reset clears the reset signals to the 960 and the bus master state machine. The 960 will begin execution after this value is written to the board control register.
1	Assert Board Reset	Assert Board Reset causes the reset signals to various devices on the board to be asserted including: the 960, the local bus controller, the IN and OUT fifos, the network interface status, and the receive and transmit fifos. The reset action also clears the hold bit for the local bus and the interrupt bits for the 960 and for the PCI bus. The reset bit must remain set for a minimum of 16 PCI bus cycles, and cleared afterwards. The PCI bus /RESET signal asserted at boot time causes a similar reset action.
2	Hold Local Bus	Hold local bus causes the slave interface to gain mastership of the local bus and to keep it as long as this bit is set. This feature can be used for read-modify-write of critical memory regions by the host. It can also be used to reduce access time when the host is accessing a block of local memory since it eliminates the time to request and gain the local bus mastership from the 960 in slave accesses. Note that holding the local bus locks the 960 out of the local bus and effectively halts it. Therefore this bit must be set only when halting the 960 is acceptable.
3	Interrupt 960	960 interrupt asserts the XINT0 line of the 960 processor. The bit can only be cleared by the 960 (upon servicing the interrupt); it can not be cleared by the host by writing a 0.

M.1.3 Serial Number PROM

The serial number PROM is a 32 byte PROM beginning at offset 0x80 in the configuration space. It contains the board serial number and 802 address. Accesses to the PROM takes multiple cycles since it is a slow device. Only bits (7:0) on the PCI bus are valid during an access to the PROM; all other bits are undefined.

Following is the format of the PCA-200A serial number PROM. Note that the format for the MAC address is *byte0:byte1:byte2:byte3:byte4:byte5*.

Address	Value
0 x 80	Hardware Revision - unused
0 x 84	Hardware Revision - major code
0 x 88	Hardware Revision - minor code
0 x 8C	Hardware Revision - sub code

Address	Value
0 x 90	Serial Number - bits 31:24
0 x 94	Serial Number - bits 23:16
0 x 98	Serial Number - bits 15:8
0 x 9C	Serial Number - bits 7:0
0 x A0	MAC Address - byte 0
0 x A4	MAC Address - byte 1
0 x A8	MAC Address - byte 2
0 x AC	MAC Address - byte 3
0 x B0	MAC Address - byte 4
0 x B4	MAC Address - byte 5

M.1.4 Local memory

The local memory consists of 256 Kbytes of SRAM, and is mapped to the PCI bus at the beginning of the memory space address register. The entire memory is accessible by the host. Note, however, that the first 1K bytes of the local memory is NOT accessible by the 960 as this address space is mapped to the on-chip RAM.

Accessing the local memory requires gaining mastership of the local bus from the 960, therefore requires multiple cycles depending on the bus activity of the 960. Once the local bus mastership is gained it takes two PCI bus cycles to write to the local memory, and three PCI bus cycles to read from the local memory. The local bus hold feature described above can be used to speed up block accesses to the local memory.

M.1.5 Clear PCI bus interrupt

To aid the efficiency of the device driver, the card-to-PCI bus interrupt is cleared by an access to the card's memory address space. A write of any value to the address {0x40000 + memory_base_address} will clear the interrupt from the PCA-200A to the host.