# HP-UX I/O

# HP9000/S800 CIO Gpio Monolith

**Version 0.1**

Printed: August 28, 1989

Susan Dedo

General Systems Division
HP-UX Software Operation
HP-UX Kernel Lab

## Preface

The purpose of this specification is to define the outward appearance and the internal workings of the gpio monolith (gpio0) for the Series 800 CIO based machines. This driver controls the 27114B CIO AFI card as well as the 27114A CIO AFI card.

I assume that the reader understands both the I/O system and the AFI card. To acquire this background, the following is recommended reading:

- **HP-UX I/O Services Definition (HPIOSD)**, General Systems Division
- **27114B External Reference Specification**, Roseville Networks Division

The HPIOSD explains the basic concepts of the I/O system. It must be understood before attempting this specification. The 27114B ERS explains how the AFI card works and can be used as a reference whenever card-level questions come up.

Other useful documents are:

- **HP-UX I/O CIO Channel Adapter Manager ES**, General Systems Division
- **HP-CIO Standard Document**, Roseville Networks Division
- **CIO Driver Writer's Manual**
- **GPIO(DEV) Manpage**

This document is organized into three major sections, an external explanation of the driver, a theory of operations and an internal explanation of the driver.

The sections dealing with the external explanation of the driver discuss how the driver communicates with the HP-UX I/O system, and how the driver is configured into the I/O system.

The theory of operations section discusses each feature of the 27114B product as a whole; without regard to specific routines or driver organization. It is in this section that history (and reasons for why the features function as they do) is discussed.

The sections dealing with the internal explanation of the driver discuss each driver routine in detail.

CONTENTS

## 1. History of the AFI product

The release 7.0 AFI product is a parallel interface product designed as a replacement for the 27114A product. As such, the 27114B hardware is designed for backwards compatibility with the 27114A hardware and the release 7.0 driver is designed for backwards compatibility with the Release 3.0 driver. Ideally every customer will be on the 27114B product when 7.0 releases. Realistically, there will be a period of time where new hardware is running with pre-7.0 software and where old hardware is running with 7.0 software.

The 27114B is intended to fix a few weakness of the 27114A. Those weaknesses are:

- The driver can not know, in a reasonable manner, how much data exists on the 27114A's FIFO. Because of this, the driver can not successfully block write transactions or report accurate residues.
- The 27114A can not work in a true GPIO environment. Because of this, data can be lost. (More on what a 'GPIO environment' means later. See the section on the transfer counter.)
- The 27114A does not allow for early termination of data transfers. Because of this the product is inflexible for many environments and at times, data is lost.
- The 27114A must be master/controller of the handshake. Because of this, 27114A to 27114A communication can not be done reliably. The inability to be a slave also makes the product inflexible.

It is worthwhile to note what CIO GPIO customers wanted most from the new product,

| | |
|---|---|
| 1st requirement: | hardware compatibility |
| 2nd requirement: | software compatibility |
| 3rd requirement: | early termination to data transfers |
| 4th requirement: | master and slave handshake modes |
| 5th requirement: | more control and status lines |
| 6th requirement: | the ability to work in a true GPIO environment |

The 27114B and release 7.0 of the AFI driver attempt to resolve the above weaknesses within the prioritized constraints given.

As of today, personnel of interest to the AFI product are,

| | |
|---|---|
| Bill Hooper | RND development engineer<br>Responsible for the AFI diagnostics |
| Bob Kentwortz | GSY Tech Marketing/Escalations Management<br>Handles AFI software hotsites. |
| Michele Mansfield | GSY Learning Products<br>Responsible for the Programmers Guide to the AFI |
| Bobbie Martinez | NMC on-line support<br>Handles AFI hardware hotsites. |
| Pery Pearson | RND development engineer<br>Helped design and now supports the 27114B. |
| Le Sellers | RND technical writer<br>Responsible for the 27114B Technical Reference Manual |
| Bill Wang | RND Product Marketing<br>Product manager for AFI, among other things. |

| | |
|---|---|
| Anders Wernblom | XTTC<br>A real trooper who has patiently tested the product. |

I will, of course, be happy to help in any manner. Please do not hesitate to ask.

## 2. Overview of Gpio0

The driver for the 2711B product is called **gpio0**. It is a monolithic manager with two distinct interfaces: the interface with the kernel and the interface with the low-level i/o system. A monolithic manager was chosen for performance reasons.

### 2.1 External interface - kernel level

The manager is entered from the kernel via system calls to **open**(OS), **close**(OS), **read**(OS), **write**(OS), **ioctl**(OS) and **select**(OS). This is the same interface that all HP-UX LDM-level managers use. The interface with the low-level io system follows the message-based interface of the HP I/O Services Definition.

The Channel Adapter Manager (CAM) is the lowest level of the I/O software. The CAM's responsibility is to transmit messages from its higher managers over the CIO channel adapter. The CAM and gpio0 communicate via the message system.

Users access gpio0 through special files called device files that reside in the /dev directory. The **mknod** command can be used to create a device file for gpio0:

```
/etc/mknod gpio0 c 22 minor_number
```

Minor_number specifies the logical unit associated with this device file. The minor number format for gpio0 is

| (1bit) | (3bits) | (4bits) | (8bits) | (8bits) |
|---|---|---|---|---|
| D | not_used | pseudo | lu_number | not_used |

The D bit, if set, means that this device file supports diagnostic requests. The pseudo bits, only usable by the driver, are described later. Their use is necessary to support multiple opens of the same device file.

Diagnostics are supported by gpio0 via ioctl requests, providing the gpio0 device file opened has the D bit set. This forces exclusive access to the device and allows the usage of various special ioctl calls. While in diagnostic mode, the user has complete control of the card. Gpio0 will not touch any register on the card unless the user requests it. The ioctls which are available only in diagnostic mode are explained in Appendix I.

### 2.2 External interface - low level

This section describes how gpio0 interfaces with the CAM and the Configurator. The messages conveyed from gpio0 to the Configurator and from gpio0 to the CAM are covered, as well as the means whereby the CAM and gpio0 are "bound" together.

This section examines each message recognized by gpio0 and its use.

| message type | direction | description |
|---|---|---|
| *Configuration* | | |
| creation | in | Specify the manager's operating environment |
| do bind req | in | Request manager to bind with a lower manager |
| do bind reply | out | I/O configuration is complete |
| bind request | out | Request manager to bind with a higher manager |
| bind reply | in | binding sequence is complete |
| *CAM* | | |
| cam request | out | start a CIO dma transaction |
| cam reply | in | complete a CIO dma transaction |
| cam event | in | informs manager of asynchronous CIO events |
| *Generic* | | |
| poweron req | in | Inform manager of system power recovery |
| poweron reply | out | Manager completed power recovery sequence |
| abort event | out | Prematurely stop a request from completing |

### 2.2.1 Configuration messages

Configuration messages are used to establish the lines of communication between gpio0 and the CAM. Each instance of gpio0 (corresponding to each AFI device adapter configured into the system) will receive these messages, in the sequence below.

#### *2.2.1.1 Creation message*

The CREATION_MSG is the first message gpio0 will receive; it is used by I/O Services to determine some of gpio0's port requirements. Gpio0 will place certain configuration information in the *creation_info* fields, such as the size of its port data area, *pda*, the size of its largest message, and the number of subqueues it needs to operate.

#### *2.2.1.2 Do bind request message*

The next message gpio0 should receive is a DO_BIND_REQ_MSG. This message, from the I/O Configurator, tells gpio0 the port number of the lower manager that controls gpio0's hardware; this will always be the CAM. The information in the request that gpio0 will use is the following:

| | |
|---|---|
| *reply_subque* | what subqueue to send the do bind reply to |
| *mgr_port-num* | gpio0's port number |
| *mgr_hw_addr1* | the CIO slot of the AFI device adapter |
| *lm_port_num* | the CAM's port number |

#### *2.2.1.3 Bind request message*

Receipt of a **DO_BIND_REQ_MSG** will cause gpio0 to send a **BIND_REQ_MSG** to the CAM, with the following fields set:

| | |
|---|---|
| *reply_subq* | REPLY_SUBQUEUE |
| *hm_event_subq* | EVENT_SUBQUEUE |
| *hm_meta_lang* | CIO_META_TAG |
| *hm_config_addr_1* | the CIO slot of the DA (from mgr_hw_addr1) |

#### *2.2.1.4 Bind reply message*

The CAM should reply immediately to gpio0 with a **BIND_REPLY_MSG**; this message tells gpio0 if the bind was successful, and defines message-passing protocols. Fields examined by gpio0 are:

| | |
|---|---|
| *reply_status* | status of the binding |
| *lm_low_req_subq* | low-priority request subqueue |

#### *2.2.1.5 Do Bind reply message*

No matter what the reply_status, gpio0 will send a **DO_BIND_REPLY_MSG** with good status to the I/O Configurator, on the subqueue saved from the do bind request.

If the binding succeeded (the status of the bind reply is CIO_LVL1_CARD), gpio0 will attempt to get its direct i/o pointer. If this attempt is successful, this instance of gpio0 is now usable. Otherwise, this AFI card cannnot be used.

### 2.2.2 CAM Messages

This section covers the messages gpio0 uses to communicate with the CAM. The syntax of the CAM metalanguage can be found in Appendix II; the semantics are covered in the HP-UX CIO Channel Adapter External Specification.

A single transaction with the CAM consists of a request and reply pair. Gpio0 sends a CIO_DMA_IO_REQ_MSG or a CIO_CTRL_REQ_MSG to the CAM to initiate some type of I/O or control transaction.

When the request completes, the CAM completes gpio0's transaction by sending a CIO_DMA_IO_REPLY_MSG or CIO_CTRL_REPLY_MSG to gpio0. The reply status in the message will indicate how the CAM perceived the transaction.

At various times, the CA may detect an asynchronous condition on the AFI DA. Gpio0 will be notified via a CIO_IO_EVENT_MSG. The I/O event will contain sufficient information to let gpio0 know what event occurred.

#### *2.2.2.1 CAM I/O Request/Reply Message*

Gpio0 will use the CIO_IO_DMA_REQ_MSG to initiate I/O on the AFI DA card. It will set the following fields:

| | |
|---|---|
| *reply_subq* | REPLY_SUBQUEUE |
| *da_number* | the CIO slot of the AFI card |
| *vquad_chain* | pointer to the virtual quad chain |

The virtual quad chain generated by gpio0 will contain only one quad. The quad will be either a read or a write.

#### 2.2.2.2 *CAM Control Request/Reply Message*

Gpio0 uses the **CIO_CTRL_REQ_MSG** with the *ctrl_func* set to either of the following CAM control requests:

| | |
|---|---|
| CIO_DA_SELFTEST | Reset the DA and performance self-test; the CIO sense byte from the card will be returned in the *ctrl_info* field of the control reply. |
| CIO_GET_DIRECT_IO_PTR | Get a pointer to the DA's I/O space; it is returned in the ctrl_info field of the control reply. |

#### 2.2.2.3 *CAM I/O Event Message*

A CIO_IO_EVENT_MSG is sent to gpio0 to inform it of an asynchronous event that occurred on the DA. The AFI card is only capable of producing the AES ARQ. Therefore, all events which are sent to gpio0 are of this type.

If the user was trying to catch ARQs, the user process will be signaled via *psignal*. Otherwise the event is dropped.

### 2.2.3 Generic Messages

Generic requests are used for powerfail and abort processing.

#### 2.2.3.1 *Power on Request/Reply Message*

The POWER_ON_REQ_MSG informs gpio0 of system recovery from a power failure. On receipt of the poweron request, gpio0 will immediately send a matching POWER_ON_REPLY_MSG to the CAM; this message will be sent on the lm_low_req_subq specified during configuration, in the CAM's bind reply.

The next step is to request a direct i/o pointer. The **CIO_CTRL_REQ_MSG** is used with the *ctrl_func* field set to **CIO_GET_DIRECT_IO_PTR**. Once the direct i/o pointer has been received, gpio0 explicitly reset the 27114B and resets the software values associated with the hardware to powerup configuration.

If the direct i/o pointer is not received, this instance of the driver can no longer be used.

The user process will be notified of power failure by receipt of the signal **SIGPWR**, which is generated by the kernel. It is the user's responsibility to catch this signal and act upon it; by default, the signal is ignored.

#### *2.2.3.2 Abort event message*

The **ABORT_EVENT_MSG** is used to abort an I/O request. Gpio0 will send an abort event to the CAM when a timed dma transaction does not complete before the timer pops. The CAM guarantees that eventually the original request will be returned.

# 3. Major Features Of The 27114B Product

64 word FIFO
16 or 8 bit, user configurable mode of data transfer
* 6 control lines (device defined/controlled)
* 6 status lines (device defined/controlled)
External interrupt line (allows the device to signal for attention)
* Early termination to data transfers
* Host end (allows the S800 to signal the device on the last transfer)
* Optional GPIO-like mode of data transfer
Locking of the interface (gives exclusive access to the device)
* 27114B to 27114B communication
* Master and slave handshake modes
* Read and write blocking (control returned to the user only after the last data word has been received by the device or channel)
Multiple opens
Locking of the interface for exclusive access

Note: '*' denotes features that are new or extended to the AFI product as of Release 7.0.

## 3.1 64 word FIFO

What makes the AFI card different from previous GPIO cards is the presence of an on-board FIFO coupled with the fact that the front and back planes of the 27114B operate independent of one another. It is because of the FIFO and the independently operating front and backplanes that the product can minimize the difference in the speeds between the S800 and the device. Although the FIFO is responsible for increasing the performance of the 27114B product, it is also responsible for increasing the complexity of what would otherwise be trivial read and write routines.

## 3.2 16 or 8 bit, user configurable mode of data transfer

The user can choose to send his data 16 bits at a time, a "word" in AFI terminology (hence the term "word mode"), or 8 bits at a time, a "byte" in AFI terminology (hence "byte mode"). The user's choice is dependent upon what the device can do.

The 27114B always asserts 16 bits on the frontplane. If all 16 bits contain valid data, then the AFI product is functioning in word mode. If only the the lower 8 bits contain valid data, then the AFI product is functioning in byte mode.

The 27114B does not perform the byte packing or unpacking. Nor does the AFI driver. Any byte packing/unpacking is done at the channel adapter level. Hence, word and byte mode is really a function of the channel adapter, and therefore subject to the channel adapter's hardware limitations. Such limitations are:

> If the total number of bytes to be transferred is odd, or if the address given to the channel adapter is odd, the entire transfer takes place in byte mode - regardless of the driver setting.

This means any DMA transaction requested in byte mode is guaranteed to occur in byte mode. DMA transactions requested in word mode will occur in word mode as long as the total number of bytes to transfer and the address given is even. If either the number of bytes or the address is odd the channel adapter switchs to byte mode, resulting in only 8 bits of the 16 data bits being valid on the frontplane.

Since the channel adapter has the ultimate say on word versus byte mode, it is important that the channel stay in agreement with the user and device expectations. One way to do this is for the AFI driver to protect the user from the channel adapter's limitations by failing word mode DMA requests in which the address or the number of bytes be odd. (Currently the driver only checks the total number of bytes)

The driver relays the requested path width to the CAM via the *cio_cmd* field of the dma vquad.

### 3.3 6 control lines

There are up to 6 control lines available to the user. The user may assert any value on these lines. The value asserted has no meaning to the 27114B. The meaning is device dependent.

The driver changes the control lines by writing to 6 bits (CTL[5] - CTL[0]) in one of the 27114B's memory mapped registers, register 7. Once set, they continue to assert that value until the next control line request (or the 27114B is reset).

Although there are 6 physical lines (CTL5 - CTL0) on the frontplane that are used for the control lines, only four of these lines CTL3, CTL2, CTL1, and CTL0 have dedicated outputs (CTL[3], CTL[2], CTL[1] and CTL[0] respectively). The other two lines, CTL5 and CTL4, are each multiplexed between two outputs. CTL5 is driven with output from either the CTL[5] bit or the DIR bit. CTL4 is driver with output from either the CTL[4] bit or the HEND bit. (More on DIR and HEND later)

Whether the CTL5/CTL4 lines are driven with bits CTL[5]/CTL[4] or driven with the DIR/HEND circuitry, is user defined. See the GPIO (DEV) manpage.

### 3.4 6 status lines (device defined/controlled)

There are up to 6 status lines available to the user. The device may assert any value on these lines. Like the control lines, the status lines have no meaning to the 27114B; only to the user and the device.

The driver reads the status lines by reading 6 bits (STS[5] - STS[0]) in one of the 27114B's memory mapped registers, register 7.

Although there are 6 physical lines (STS5 - STS0) on the frontplane that are used for status lines, only four of these lines STS3, STS2, STS1, and STS0 have dedicated inputs (STS[3], STS[2], STS[1] and STS[0] respectively). The other two lines, STS5 and STS4, are each multiplexed between two inputs. STS5 inputs to either the STS[5] bit or the ATTN bit. STS4 inputs to either the STS[4] bit or the PEND bit. (More on ATTN and PEND later)

Whether the STS5/STS4 lines input to the STS[5]/STS[4] bits or input to the ATTN/PEND circuitry is user defined. If the user choses to see use the external interrupt feature, STS5 will input to the interrupt circuitry.

Independent of the external interrupt feature, the user may choose to terminate data transfers early. In doing so, the STS4 line will no longer input to the STS[4] bit but to the circuitry that controls early termination of data transfers.

## 3.5 External Interrupt Line

The external interrupt line (ATTN) provides a means by which the device can signal the user for attention. When this option is enabled and the external interrupt line is asserted by the device, the driver sends a signal, SIGEMT, to the process id that enabled the external interrupt option. To receive multiple signals, the user must enable this option after each signal is sent.

To understand the external interrupt feature, the interrupt circuitry must be understood. When the device asserts the external interrupt line (ATTN) on the frontplane and the external interrupt feature is enabled, the 27114B's ATTN flipflop is set. In addition, the 27114B's ARQ flipflop is set. It is the setting of its ARQ flipflop that causes the 27114B to assert ARQ on the backplane. The setting of the ATTN flipflop is so the driver can know why ARQ was asserted (being that the 27114B has 3 reasons for asserting ARQ: ATTN was asserted on the frontplane, PEND was asserted on the frontplane, or the transfer counter reached zero. More on PEND and the transfer counter later). This results in the AFI driver receiving a CIO_IO_EVENT_MSG from the CAM where upon gpio0 sends a SIGEMT to the process id.

Once the ATTN flipflop is set, it will remain set until the driver resets it. However, the ARQ flipflop will be reset by the CAM after the 27114B responds to the ARQ poll.

The CAM also disables the 27114B from further assertions of ARQ[1] why the user must enable the external interrupt after each signal is received.

The 27114B Hardware ERS makes the statement that "ATTN is enabled when the card powers up". This can be misleading since is does not mean that the device can assert ATTN and the user receives a signal. It means that the ATTN flipflop is enabled (can be set). [2] But although ATTN flipflop can be set, the ATTN line does not input to the interrupt circuitry because at powerup time the ARQ flipflop is held reset.

A more detailed description of what the driver needs to do for the external interrupt function is as follows:

1. user makes ioctl request

   a) user wants external interrupts enabled.

      - save the process id of the current process
      - save the process' interrupt handler.
      - enable the ATTN flipflop to register when ATTN is pulled.
      - enable the card to assert ARQ when the ARQ flipflop is set

   b) user disables external interrupts.

      - disable the card from pulling ARQ (hold the ARQ flipflop reset)
      - throw out the process id
      - throw out the interrupt handler
      - clear the ATTN flipflop

---

1. At one point in the AFI product cycle, the CAM re-enabled the 27114 to assert ARQ after each ARQ occurred. However, it was found that the 27114 could prevent system boot if the device asserts ATTN multiple times during configuration. Also, it is possible that system performance can degrade due to spurious interrupt messages. To prevent either of these side affects, it was decided that the CAM would no longer re-enable ARQs on the 27114.
2. This is necessary for backwards compatibility with the 27114A.

2. ATTN was asserted on the frontplane - when the feature is enabled; The port server receives a CIO_IO_EVENT_MSG and calls gpio0_event_from_cam()

   a) if the card is a 27114A

      — send a SIGEMT providing the pid saved and int_handler are valid

      — set the interrupt mask to reflect the cause of the interrupt

   b) if the card is a 27114B

      — send a SIGEMT providing the pid saved and int_handler are valid and providing the ATTN flipflop recorded ATTN was asserted

      — set the interrupt mask to reflect the cause of the interrupt

      — clear the ATTN flipflop

   Note: the driver must verify that the process id to which the SIGEMT is being sent saved is indeed still alive. Otherwise there is possibility that some other process may now occupy that slot in the process table and receive the signal.

3. A GPIO_SIGNAL_MASK (IO_STATUS) call is made

   — return the interrupt mask (for AFI, always ST_ARQ2) in arg[0]

   — clear the interrupt mask

### 3.6 Early termination to data transfers

This feature allows the device to signal the S800 that it (the device) has just received (or sent) the last data word. Hence, the device may chose to terminate any data transfer early by asserting PEND (Peripheral END) on the frontplane. This is useful when a user does not know how much data his device is sending or how much data his device is willing to receive.

In the past, such applications found the AFI product inflexible or unusable since the driver, waiting for the channel adapter to fulfill its transfer size, would eventually timeout. Since timing out is destructive to the 27114B and to the channel adapter (an abort message is sent to the CAM, which then resets the 27114B and the channel adapter), the result of the transfer is often incomplete data. With the addition of the PEND feature, the device now has a way to signal the channel adapter that it (the device) has just received or sent its last data word.

The mechanics of PEND vary depending upon the direction of data transfer. For inbound transactions (data moving from device to channel adapter), the device asserts PEND when it asserts the last data word. At the assertion of PEND, the frontplane is disabled. The 27114B tracks the last word through the FIFO and asserts DEND (Device END) [3] on the backplane when that word crosses the backplane. For outbound transactions (data moving from channel adapter to device), the device asserts PEND as it performs its last handshake. The frontplane is disabled when PEND is asserted but this time the 27114B immediately asserts DEND [4] on the backplane.

---

3. The assertion of PEND can also cause the 27114B to assert ARQ. This is an option that is available to the driver only. The mechanics of PEND causing an ARQ is best described in the write blocking context; be sure to read the section on write blocking.

4. DEND need be asserted only if the channel is still involved in the transaction.

The assertion of DEND causes the channel adapter to stop receiving or sending data on the CIO bus - regardless of its count. The channel is therefore free to non-destructively finish the transaction and the CAM is able to send a DMA reply message to the waiting driver.

The use of PEND has some interesting effects on the 27114B's FIFO. The effects vary depending upon the direction of data transfer. To understand the effects, the movement of data through the FIFO must be understood.

For inbound transactions, the data is moving from the device, across the frontplane, into the front of the FIFO, through the FIFO and across the backplane into the channel. When a device signals that it has sent its last data word, that word must travel the entire length of the data path before reaching its destination. When it reaches the backplane DEND asserts. The transaction completes only after the channel receives the word. The result is an empty FIFO.

For outbound transactions, the data is moving from the channel, across the backplane, into the back of the FIFO, through the FIFO and across the frontplane. At some arbitrary time, as data continues to spill into the back of the FIFO, the device asserts PEND and stops handshaking. The last data word has already reached its end destination; no other data movement occurs. The FIFO may or may not contain data (any remaining data is considered garbage).

Because data may be left over, the driver should clear the FIFO after write transactions that end with a PEND.

Note: If the transfer counter is not enabled, the driver will be unable to figure the residue value at the end of the transaction. Instead the driver relies upon the CAM to figure the residue. Since the channel's responsibility ends at the backplane, the CAM's residue value may be off (too small) by as much as 64 for outbound transactions.

## 3.7 Host END (HEND)

The Host End feature allows the S800 to signal the device on the last transfer. This is similar in concept to the PEND feature. It consists of the 27114B forwarding the backplane signal CEND (Channel END) onto the frontplane as HEND.

Once HEND asserts, the driver must force the deassertion of HEND. The method of deassertion depends on the direction of data transfer. If the HEND occurs on an outbound transaction, the driver must clear the FIFO and load the counter with a non-logical zero value before HEND will deassert on the frontplane. If the HEND occurs on a inbound transaction, the driver must load the counter with a value other than logical zero (greater than 0xFFFF) before HEND will deassert on the frontplane.

## 3.8 Optional Transfer Counter

The transfer counter was added to the AFI product to resolve two problems:

- The 27114A can not work in a true GPIO environment. Because of this, data can be lost.
- The driver can not know, in a reasonable manner, how much data exists on the 27114A's FIFO. Because of this, the driver can not successfully block write transactions or report accurate residues.

The first problem is better understood with a bit of history,

> One of the features of the 27114A card, referred to as read pre-fetch, means the 27114A, when configured to read, will continue to ask for data as long as its FIFO is empty.

> Once the FIFO fills, the 27114A breaks the handshake.
>
> This implies that after the channel has received the requested amount of data, there is room on the card for a FIFO full of "extra" data. Should the device continues to make data available, the 27114A will continue to handshake up to a FIFO full of data independent of the channel and the driver. The handshake is stopped by either the 27114A (when the FIFO fills) or the device (when it has no more data).
>
> This pre-fetching on read transactions, when followed by another read is valuable; most of the time. Unfortunately, HP failed to anticipate the various environments that the AFI product would be sold into. Some applications are time sensitive to the degree that the leftover data is no longer valid and can not be used. Some applications do more than read from the AFI product. Specifically, should the next transaction be a write, the extra data must be removed from the FIFO. Otherwise it is written back to the device. But can that data be thrown away, and if not, what should be done with that data?

Two points should be gleaned from this history.

1. The driver is incapable of making those decisions. Even if it were capable, the decision could not be made in a timely manner.
2. The 27114A product only works in environments where the AFI product is only asked to read or/and where the device knows exactly how much data it should send (thereby avoiding the pre-fetching). In all other environments, the product could not be successful.

In an effort to resolve customer problems and expand the number of environments in which the AFI product can function, a physical counter was put on the card. At the user's request, the driver enables the counter and initializes it to clock the appropriate number of transfers. The counter clocks the number of transfers across the frontplane. Once the correct amount of data is transferred, the frontplane is disabled. So by choosing to use the counter, the user can prevent the read pre-fetch (and all of the "problems" associated with read pre-fetch).

The mechanics of the counter are simple. The driver enables or disables the counter based upon user request. This is done by touching a bit in register B. If enabled, the driver loads the counter before sending the CIO_DMA_REQ_MSG to the CAM. Loading the counter is a two step process since the counter has two parts. The correct value must first be written to the counter's write register (via the direct io pointer) and then the part of the counter that does the actual counting must be loaded with that value (toggle the counter's load bit via the direct io pointer). The counting portion of the counter will then decrement toward zero with each word that crosses the frontplane.

The counter, for hardware reasons, is a bit strange. It, in effect, considers zero to be 0x00FFFF. Hence the value loaded into the counter must have an offset of 0x00FFFF. This is explained in the 27114B Hardware ERS.

Since the transfer counter decrements with each front plane transfer, and the number of valid bytes crossing the front plane with each transfer depends upon the width of the data path, the value loaded into the counter is a function of the data path width. When in word mode each transfer crossing the frontplane consists of two data bytes. Hence the counter must be loaded with a value that is half the users requested number of bytes.

The second of the two problems that the transfer counter resolves is:

> The driver can not know, in a reasonable manner, how much data exists on the 27114A's FIFO. Because of this, the driver can not report accurate residues.

There are actually three factors contributing to this problem.

1. The 27114A has no reasonable way of knowing how much data is in its FIFO.

2. The channel adapter's responsibility to a write transaction ends once the last data word moves from the bus onto the 27114A/B.
3. Outbound transactions are not complete until the last data word has been received by the device.

Because the driver and the 27114A could not figure out how much data was in the FIFO, the CAM's residue value had to be used. There is a slight problem with using the CAM's residue value: Although the channel will report that all data has been successfully sent, in reality, up to 64 words of data may still exist in the FIFO. Hence the CAM's residue value may be off (too small) by as much as 64 words for outbound transactions.

With the addition of the transfer counter to the AFI product, the driver can now report an accurate residue by reading the counter.[5]

For read transactions, the amount of data received by the channel will be accurate; the counter need not be looked at.

Yet another benefit of the transfer counter is that the driver has the option of configuring the 27114B to assert ARQ when the counter reaches logical zero. Why the driver would want to do this is best described within the context of write blocking; be sure to read the section on write blocking.

### 3.9 Read and write blocking

In the context of this discussion, read blocking refers to the driver not completing the transaction until it is sure the last data word has been received by the channel; write blocking refers to the driver not completing the transaction until it is sure the last data word has been received by the device.

Without read/write blocking, the user might make a request of the driver that is destructive to the data path while data is still enroute. This was a problem for the 27114A product (with write transactions) since the driver, not knowing how much data existed in the 27114A's FIFO, did not know when to declare the transaction complete.

Read blocking is trivial, from the driver's point of view because the data must travel to the channel before the channel can finish, return the CAM and the driver receive a DMA reply message. In this case, the receipt of the DMA reply message guarantees the requested amount of data has left the device and arrived on the channel.

Write blocking is not so trivial, because the last data word only has to leave the channel (not the card) before the channel, being satisfied, returns to the CAM and the driver receives the DMA reply message. The DMA reply message does not guarantee the data has traveled from the channel to the device. It only guarantees that the data has traveled from the channel to the 27114B.

Write blocking relies upon the transfer counter. If the transfer counter is not enabled, then the driver can not write block (it behaves like the 27114A product). It was decided that having the 27114B pull ARQ every time the last data word crossed the frontplane to the device, was too expensive.[6] It was also decided that having the driver poll the counter until it registered a zero value

---

5. Note: because of the way the counter clocks data, it is possible that the counter will read zero (0x00FFFF) when in fact the last data word has not yet transferred. The driver can circumvent this problem by looking to see if PCTL is high (indicating the device has not yet acknowledged the movement of the last data word) and adding one to the transfer counter if it is.

6. The expectation exists that the vast majority of the time the device will be faster than the time required for the channel to return to the CAM, for the CAM to build and send a DMA reply message, and for the driver to be invoked.

was too expensive. The decision was made to have the driver read the counter once. If the counter was zero, great. Otherwise the driver should enable the 27114B to pull ARQ, when the counter did arrive at logical zero, and wait for a CIO_IO_EVENT_MSG. When the event message did arrive, the driver should check the interrupt flipflops to determine the cause of the ARQ (ATTN or the transfer counter going to logical zero).

To throw a wrench in an already sticky situation, imagine the case where the counter is enabled, and the PEND option (ability to terminate data transfers early) is enabled. Now imagine a write taking place with a device that is slow (the driver receives its DMA reply message before the last data word is gone). The counter, when checked, is non zero. The driver correctly enables the 27114B to ARQ when the counter reaches zero. But, before the last data word crosses the frontplane, PEND is asserted on the frontplane. The counter will never reach zero, the driver will continue to wait for an ARQ until the transaction times out; the driver will incorrectly abort the transaction and return an error to the user.

To prevent this situation, the driver has the option of enabling the 27114B to assert ARQ when PEND is asserted on the frontplane. So in the above situation, the CIO_IO_EVENT_MSG alerts the driver to check the three interrupt flipflops to determine the cause of the ARQ (ATTN, the transfer counter going to zero, or the assertion of PEND). Upon seeing the cause is PEND, the DMA transaction can be completed.

Once the PEND flipflop is set, it will remain set until the driver resets it. However, the ARQ flipflop will be reset by the CAM after its ARQ poll.

Once the transfer counter-has-reached-zero flipflop[7] is set, it will remain set until the counter is non-zero. This means the driver must either load the counter with a non-zero value, or it must reset the counter.

## 3.10 Master and slave handshake modes

There are three handshakes possible, **full master, full slave** and **fifo master**. Fifo master is the same handshake used by the 27114A (the 27114A has only one handshake).

The only thing the driver needs do for the handshake modes is correctly set the bits in register B to indicate the handshake requested by the user.

## 3.11 27114B to 27114B communication

The driver need do nothing for B to B communication, which is all a function of hardware.[8] The user is responsible for making the driver requests to set one card as master of the handshake and one card as slave of the handshake.

## 3.12 Multiple opens

As of Release 3.0, an AFI driver can have more than one open file descriptor, up to 16, at any given time.[9] With multiple accesses to the driver came the concept of per-interface driver attributes and

7. It is really a bit; not a flipflop.
8. The 27114A does not support 27114A-to-27114A links.
9. There is nothing significant about the number 16; it was grabbed out of the air.

per-open driver attributes. Per-interface attributes are those attributes that effect all file descriptors for a given card. Typically this means options that are configured in hardware, such as data width. Per-open attributes are those attributes that effect only one file descriptor. Typically this means attributes that are configured solely within software, such as timeout value.

There are three per-open attributes: timeout, signal mask and lock count. The remaining options are are considered per-interface attributes.

If each timeout is associated with an open, the driver needs to know which open's timeout to reference. But the file descriptor alone is not sufficient in distinguishing one open from another, as the same file descriptor may be opened many times.

What the driver does, is to create a pseudo minor number, based upon the actual minor number, that is unique to that open. The pseudo minor number is then passed back to the high level open routine. Recall that the minor number format for gpio0 is

| (1bit) | (3bits) | (4bits) | (8bits) | (8bits) |
|---|---|---|---|---|
| D | not_used | pseudo | lu_number | not_used |

By altering one of the four *pseudo* bits of the minor number, the driver can uniquely identify up to 16 opens.[10]

The kernel maintains the mapping between that open and the pseudo minor number. Every request made to the driver from that open, results in the kernel passing the driver the appropriate pseudo minor number.

The driver maintains the per-open attributes in its pda, in the following structure:

```
struct per_open_entry {
    unsigned      timeout;         /* device timeout in u-seconds          */
    struct proc  *int_handler;     /* interrupt handler process            */
    short         int_handler_pid; /* pid owning the interrupt handler     */
    int           po_lock_ct;      /* lock count for an open               */
    struct per_open_entry *link;   /* ptr to next available per_open_entry */
 };

 struct per_open_entry  pot[MAX_OPENS]; /* Per-Open Table                   */
 struct per_open_entry  *pot_head;      /* pointer to next available entry */
 struct per_open_entry  *pot_end;       /* pointer to last available entry */
```

The pointer, *pot_head* (Per Open Table Head), marks the first unused entry and the other, *pot_end* (Per Open Table End), marks the last unused entry. All unused/available entries are linked together, via *link*

As opens occur, the driver takes the first available entry in the array (as indicated by *pot_head*), initializes the values, advances *pot_head*, destroys *link* and creates a pseudo minor number. The challenge in creating the pseudo minor number is that the driver must relate the minor number to one of the appropriate index of the pot array. The driver does this by assigning the pot array index of the just initialized entry to bits 19 through 16 of the minor number.

---

10. The pseudo minor number is not represented as an actual device file; it is known only to the kernel.

As closes occur, the driver links the now-defunct entry to the end of the free list (as indicated by pot_end).

### 3.13 Locking

By locking the AFI interface, a process gains exclusive access to the device. Other processes' requests will wait until either the locking process unlocks the interface or the time for their requests expire. (However, the request will immediately return with an appropriate error, if the O_NDELAY file status flag is set.)

Note that the locking gives exclusive access to the device; not to the interface. Since some attributes (per-open) have no relation to the hardware/device, they have no effect on another process. Therefore the driver allows requests that deal with per-open attributes, regardless of locks.

The driver maintains two types of lock counts,

- the number of locks per a given interface
- the number of locks requested per a given open.

Each time the locking process requests a LOCK_INTERFACE, the interface lock count (and the appropriate per-open lock count) increments by one. Each time the locking process requests an UNLOCK_INTERFACE, the interface lock count (and the appropriate per-open lock count) decrements by one. When the locking process requests a CLEAR_ALL_LOCKS, the interface lock count (and all per-open lock counts) resets to zero.

When a process locks the interface, the driver saves that process' pid and marks the interface locked. Any request to change the per-interface attributes (GPIO_WIDTH, GPIO_CTL_LINES, GPIO_SIGNAL_MASK, GPIO_RESET, GPIO_LOCK, GPIO_SET_CONFIG) must then come from a process with a pid that matches the saved pid.

The locking process should remove all locks before exiting. The driver attempts to remove locks for the process that exits before it removes its locks. The driver's only indication that a process has exited is when the driver close routine is called. So cleaning up of locks is done in the close routine. However the driver close routine is called by the kernel only when the last close on a file descriptor is made. This means a process which locks the interface, forks, and exits without removing its lock will continue to prevent access to the interface until the child process exits (causing the last close on the shared file descriptor).

## 4. Data Structures

The data structures for gpio0 can be divided into three main groups: locals, external, and the port data area. Interesting locals are explained in the sections for the various routines.

Externals are those variables which are global to the entire kernel. All externals for gpio0 are declared in the file machine/space.h. External variables are dangerous to use because all instances of gpio0 use the same copy of these variables. Therefore, exclusive access to externals must be enforced.

Fortunately, gpio0 does not require may externals. The first external, *num_gpio0*, indicates how many instances of gpio0 exist in the system; that is how many times gpio0 was included in the configuration file. This variable is used only on opens. The open routine uses the variable to ensure that non-existent lu's are not used.

The second external, *gpio0_pda_map*, is an array of pointers to port data areas. This array is initialized by gpio0 during configuration. One element of gpio0_pda_map is used for each instance of gpio0; that is, the size of gpio0_pda_map is num_gpio0. The array is used by the LDM routines to match lu's with port data areas. The macro PDA_MAP takes an lu number and returns a pointer to the corresponding port data area.

The port data area (pda) is the other large data structure which gpio0 uses. Each instance of gpio0 has its own pda. As explained above, the LDM routines can get access to the appropriate pda via the macro PDA_MAP. The port server is invoked with the pda as one of its arguments.

Elements of the pda also need to be protected from simultaneous access. In this case, access can occur from the LDM routines and the port server at the same time. Therefore, if a variable of the pda is used from both sides, it may need to be protected via semaphoring.

**/* pda: Port Data Area */**

| | |
|---|---|
| state | State of the manager: configuration (GET_DIO_PTR), powerfail (PF_GET_DIO_PTR), or normal (IO_REQ_READY) |
| io_state | State of manager (specific to DMA). There are 3 logical states: ready for io (read or write), doing io (read or write), and waiting. for data to leave the card (write). But only 2 that the driver cares about; waiting for data to leave the card (WAITNG_ON_INTR, and not waiting for data to leave the card (IO_REQ_READY). If the io_state is WAITING_ON_INTR, the driver has received the DMA reply msg. So it is important to know that when the transaction times out, sending an abort msg to the CAM (the usual action for a timeout) does nothing. Instead the driver should clean up and return to the user. |
| old_card | 27114A or 27114B ? |
| lu | Logical unit number |
| diag_port | Flag to tag a diagnostic port |
| msg_id | Message id of last request |
| ctrl_msg_id | Message id of last ctrl req |

| | |
|---|---|
| msg_id_from_above | Used in configuration. Is the msg id of the do_bind_req received from the Configurator. The same msg id needs be used in my do_bind_reply. |
| pf_trans_num | Transaction number from the most current powerfail message |
| shadow7 | A 'shadow register' is a software copy of a write-only register. They are important method in knowing which bits are set. |
| shadowB | For register B (AFI Control II Reg) |
| Reg_H | 27114B reg that holds the upper byte of the transfer counter. |
| next_msg_id | Message id for next request |
| trans_num | Transaction number of current req. Get this from IO Services, during configuration. Since can only do one transaction at a time, use this same number each time |
| timer_id | A timer is needed to clock each transaction. The same timer is used over and over. io_get_timer call is made during configuration. |
| req_state | Need to know which direction the previous DMA transaction went, so the driver knows how to treat the FIFO. |
| wait_status | Used in gpio0_wait_on_lock. The value returned indicates whether the the current user request can continue or must fail because the interface is still locked. |
| flags | Reflects various user options that need to be tracked: OLD_CARD, CTR_EN, PEND_EN, WORD_MODE,INTERUPT_EN, BUSI, WANTED, RST_BUSI, RST_WANTED, HDWR_DEAD. |
| localmsg | The msg used to send requests to the CAM. Note that once configuration is complete, it is used for only 3 types of msg requests: dma, control and abort events. Therefore only the msg descriptor field and the vquad pointer (or control function) need be changed on each request |
| buf | Buffer structure (buf.h) used by read and write to do dma |
| vquad | The single virtual quad which gpio0 uses to initiate dma. Only the command, count, and address fields change on each request. The link, residue and address class fields are always zero. |
| *ciodio_ptr | The pointer to the direct i/o space for the card which this gpio0 controls. Used to poke and read registers on the AFI card |
| resetbuf | Used for multiple resets |

| | |
|---|---|
| *req_process | Process making dma request; need to periodically check whether it has been killed. |

*/* ioctl returns */*

| | |
|---|---|
| int_mask | Hold the reason the card pulled ARQ |
| interface_type | Interface id from cards id register |
| usecs_left | Microseconds left in timeout |
| term_reason | Reason for last termination (not used) |
| ctl | User defined mask to write to the control lines |
| hndshk | Handshake mode |
| edg | Which edge of PFLAG latches data? |
| config_mask | Used for io_env calls. Reflects the configuration of the card's hw |

*/* Buffer space */*

| | |
|---|---|
| b_data_buf[3 + CPU_IOLINE] | |
| *b_data | Pointer to cache-aligned data |
| b_data_dpt | Dpt of same |

*/* config data */*

| | |
|---|---|
| hw_addr_1 | Address of our hardware |
| config_port | Configurator port number |
| my_port | Port number for AFI manager |
| cam_port | Port number for the CAM |
| cam_subq | Subqueue of cam to send msgs to |
| buffer_16 | A 16-bit buffer for ctrl replies |

*/* for iolock/unlock */*

| | |
|---|---|
| pot[MAX_OPENS] | Table of per-open variables |
| *pot_head | Pointer to next available entry |
| *pot_end | Pointer to last available entry |
| locker | Process that has the interface locked |

| | |
|---|---|
| pi_lock_ct | Lock count for the interface |

*/* Table of per-open values */*

| | |
|---|---|
| timeout | Device timeout in u-seconds |
| *int_handler | Interrupt handler process |
| int_handler_pid | Process id owning the interrupt handler |
| po_lock_ct | Lock count for an open |
| *link | Pointer to next available per_open_entry |

Below are the remainder of defines and types declared in gpio0_spc.h:

**The message types gpio0 understands**

```
typedef struct {
   llio_std_header_type      msg_header;
   union {
      creation_info_type     creation_info;
      do_bind_req_type       do_bind_req;
      do_bind_reply_type     do_bind_reply;
      bind_req_type          bind_req;
      bind_reply_type        bind_reply;
      cio_dma_req_type       cio_dma_req;
      cio_dma_reply_type     cio_dma_reply;
      cio_io_event_type      cio_io_event;
      cio_ctrl_req_type      cio_ctrl_req;
      cio_ctrl_reply_type    cio_ctrl_reply;
   } u;
} gpio0_msg_type;
```

**Drivers breakdown of its major/minor number - for multiple open support**

```
typedef     union {
   struct {
      unsigned gp_major    :8;
      unsigned gp_diag     :1;
      unsigned gp_unused1  :3;
      unsigned gp_pseudo   :4;
      unsigned gp_lu       :8;
      unsigned gp_unused2  :8;
   } GP;
   int all;
} gpio0_dev;
```

**/* Defines for ease of use */**

```
                    /* ...related to device number (for multiple opens) */
#define g_major     GP.gp_major
#define g_diag      GP.gp_diag
#define g_unused1   GP.gp_unused1
#define g_pseudo    GP.gp_pseudo
#define g_lu        GP.gp_lu
#define g_unused2   GP.gp_unused2

                    /* ...related to the direct i/o pointer (from the pda) */
#define Reg_1   ciodio_ptr->ctlsens.normal
#define Reg_3   ciodio_ptr->ctlsens.cend
#define Reg_7   ciodio_ptr->ctlsens.cendbyte
#define Reg_9   ciodio_ptr->statcmd.normal
#define Reg_A   ciodio_ptr->order.cend
#define Reg_B   ciodio_ptr->statcmd.cend

                    /*      Definition of the bits in pda->flag     */
#define HDWR_DEAD       ELEMENT_OF_32(0)        True if hdwr off-line, etc.
#define BUSI            ELEMENT_OF_32(1)        Someone's using the card
#define WANTED          ELEMENT_OF_32(2)        Someone wants the card
#define WORD_MODE       ELEMENT_OF_32(3)        Word/byte mode flag
#define OLD_CARD        ELEMENT_OF_32(5)        Can the user do 27114B features?
#define CTR_EN          ELEMENT_OF_32(6)        Easier to keep a flag than read the
#define PEND_EN         ELEMENT_OF_32(7)        Hw each time gpio0 needs to know
#define RST_BUSI        ELEMENT_OF_32(8)        A reset is in progress
#define RST_WANTED      ELEMENT_OF_32(9)        Someone wants to do a reset
#define INTERUPT_EN     ELEMENT_OF_32(10)       ATTN-arqs are enabled

                    /*      Miscellaneous definitions       */
#define MAX_OPENS       16          The maximum times the device file can be opened
#define END_OF_TABLE    -1
#define REV0            0x0         The 27114A can only have a revision number equal to 0 or 1
#define REV1            0x0100
#define LOGICAL_ZERO    0xFFFF
```

```
                    /*  Register layouts of the 27114B    */

                        /*  ...27114 Control Register Layout  */
#define D_INT_D     0x4000      DEND interrupt disable bit, 27114B
#define C_INT_D     0x2000      Hdshk_ctr=0 intrpt disable bit, 27114B
#define LDCTR       0x0800      Load ctr bit, 27114B only
#define CL5         0x0400      Control bit #6, 27114B only
#define CL4         0x0200      Control bit #5, 27114B only
#define CL3         0x0100      Control bit #4, 27114B only
#define PRN         0x0080      Poll Response Enable
#define DIR         0x0040      direction for data transfer
#define EDGE        0x0020      Edge determination for data movement
#define CLF         0x0010      Clear FIFO
#define PEN         0x0008      Enable frontplane's response to handshake
#define CL2         0x0004      These three bits are
#define CL1         0x0002      used to write information
#define CL0         0x0001      to the control lines

                        /*  ...27114 Status Register Layout  */
#define PEND_FF     0x2000      Flip-flop indicating PEND was pulled
#define ATTN_FF     0x1000      Flip-flop indicating ATTN was pulled
#define CTR_FF      0x0800      Flip-flop indicating the hndshk ctr=0
#define ST5         0x0400      Status line #6, 27114B only
#define ST4         0x0200      Status line #5, 27114B only
#define ST3         0x0100      Status line #4, 27114B only
#define PCL         0x80        Peripheral Control
#define PFG         0x40        Peripheral Flag
#define OR          0x20        Output Ready
#define IR          0x10        Input Ready
#define TES         0x08        Test Hood Present
#define ST2         0x04        These bits return the state of the
#define ST1         0x02        status lines being driven by the
#define ST0         0x01        peripheral.

                        /*  ...27114B Control II register layout  */
#define ATTN_EN     0x0080      Enable ATTN to cause ARQs
#define PDIR_EN     0x0040      CTL5 line driven by CTL5 bit or PDIR?
#define CTR_RST     0x0020      Enable/disable(reset) the ctr ARQ flipflop
#define PEND_RST    0x0010      Enable/disable(reset) the DEND ARQ flipflop
#define FIFOM       0x0007      Reserved for handshake mode definition
#define FULLM       0x0006      Used for handshake mode definition
#define FULLS       0x0004      Used for handshake mode definition
```

# 5. Outline Of Driver

Gpio0 is organized as a monolith; that is, gpio0 includes both an LDM and a DAM. This organization is very advantageous for both speed and code size. As a monolith, gpio0 can be divided into two broad areas: the LDM routines and the port server routines.

The LDM routines include all of the standard HP-UX entry points. These routines are explained in the LDM chapter.

The port server consists of the port server itself and each of the routines it calls to process a particular type of message. The port server itself is just a switch statement with a separate case for each message type. Each message type which gpio0 can receive is processed by a separate routine. In general, these routines are invoked with a pointer to the incoming message and a pointer to the pda.

## 5.1 LDM Routines

### 5.1.1 Useful Macros

A few macros are used by the LDM routines. Every routine uses the macro *PDA_MAP* which takes a dev_t and returns the pda which corresponds to that device. It does so by examining the gpio0_pda_map array which is built during configuration.

Two other macros, LU and GPIO_DIAG, are only used by the gpio0_open routine. LU returns the logical unit number of the device when given the dev_t. GPIO_DIAG returns true if the dev_t passed in is for a diagnostic open.

### 5.1.2 Direct I/O

All device adapters connected to the channel have a set of memory mapped registers called the direct i/o space. The Driver Writer's Manual explains direct i/o in the chapter on the CAM interface.

Gpio0 uses direct i/o to read status from the card and to initiate dma and do control requests to the card (via ioctl calls). The direct i/o pointer (ciodio_ptr) is grabbed via a CAM control request during configuration. Thereafter, to examine or poke a register on the AFI card, gpio0 just references the appropriate offset from ciodio_ptr. A list of registers used follows:

| Register | R/W | Structure Element | Usage |
|---|---|---|---|
| 0 | Read | ciodio_ptr->data.normal | data register |
| 1 | Read | ciodio_ptr->ctlsens.normal | CIO sense register |
| 3 | Read | ciodio_ptr->ctlsens.cend | ID register |
| 7 | Read | ciodio_ptr->ctlsens.cendbyte | 27114 status register |
| 9 | Read | ciodio_ptr->statcmd.normal | CIO status register |
| A | Read | ciodio_ptr->order.cend | transfer counter |
| B | Read | ciodio_ptr->statcmd.cend | transfer counter |
| 0 | Write | ciodio_ptr->data.normal | data register |
| 1 | Write | ciodio_ptr->ctlsens.normal | CIO control register |
| 7 | Write | ciodio_ptr->ctlsens.cendbyte | 27114 control register |
| A | Write | ciodio_ptr->order.cend | transfer counter |
| B | Write | ciodio_ptr->statcmd.cend | transfer counter and 27114 control II register |

### 5.1.3 gpio0_open()

Gpio0_open is the routine by which a user process gains access to the driver. If an error is returned from this routine, the user can not proceed with other requests to the driver.

Several types of checking must be done. If the driver is not configured (pda is NULL) or the hardware is bad/missing (*pda->flags* contain HDWR_DEAD), then the open should fail. Also, if the lu of the device file being opened is bogus for one reason or another (the lu was not configured, or has improper format) the open should fail.

Beyond the error checking, the goals of the open routine are to, create a pseudo minor number for the open in progress (multiple open feature), and initialize those options that are unique to that open (the timeout, the interrupt handler, the pid of the interrupt handler, and the per_open lock count).

If the open is a diagnostic open, lock the interface (don't forget to increment the interface and per-open lock counts) and drop other locks (don't forget to grab the process id). Only one diagnostic open at a time is allowed.

### 5.1.4 gpio0_close()

The routine close() is called by the user process to relinquish access to the device. However, gpio0_close is called by HP-UX only on the last close made to the device file.

The goals of the close routine are to, remove any locks that belong to this instance of open, add the now-free per-open entry to the end of the *pot* table, and clear the interrupt handler associated with the close so that the process won't be signaled on an interrupt after it (the process) has terminated.

Note: the driver has no responsibility for the device.

### 5.1.5 gpio_busy() and gpio_free()

These routines are used to guarantee exclusive access to the device. Before a request is started, gpio_busy is called. After the request has finished, gpio_free is called.

Gpio_busy is simply the exclusive access portion of physio(). It sleeps on *buf* until *buf* is available. Once it becomes available, gpio_busy sets the B_BUSY flag. Of course, all of the above is done at spl5 since it is a critical section.

Gpio_free simply marks *buf* as not B_BUSY and does a biodone to release the semaphore.

Note that only gpio0_ioctl must use these routines. Semaphoring, in this manner, of the read and write requests is done for free by physio().

### 5.1.6 gpio0_set_timeout() and gpio0_service_timeout()

These two routines are used to provide a watchdog timer for dma requests which are sent to the CAM. Gpio0_set_timeout is called right before the io_send() to the CAM for the dma request. When the dma reply is received from the CAM, the timer is released. Gpio0_service_timeout will be called from softclock if the dma request has not finished in the allotted time.

The allotted time for a dma request is initially set at one hour and can be changed by the user via the ioctl GPIO_TIMEOUT. The full timeout value is further broken down into smaller chunks called GPIOTAs (2 seconds each). This is done so that a process with outstanding i/o can be killed

fairly quickly by the user instead of waiting for the full timeout.

Gpio0_set_timeout calls the *timeout* routine with a time limit of GPIOTA (or whatever is left in the full timer, if less). It sets up gpio0_service_timeout as the routine to call if the timer pops.

Gpio0_service_timeout is called whenever the timer pops. If time still remains, the time limit is decremented and timeout is called again. If no time remains in the full timer or if the user has killed the process, an abort message is sent to the CAM, providing the request is still active at the channel level. This guarantees that the request will complete in the near future. If the request is no longer active at the channel level (indicated by a req_state set to WAITING_ON_INTR), the driver is waiting for data to leave the card; an abort message is not necessary as the reply message to the request has already been received. The driver needs only clean up after the dma.

### 5.1.7 gpio0_write() and gpio0_read()

These routines are used to write (or read) to (or from) the device. They are identical to each other except for the direction of data transfer.

physio() is called to lock down buffers and do the rest of it's voodoo. Included in this voodoo is the call to gpio0_strategy and the sleep on *buf*. Finally, when the associated **biodone** is done after a dma reply is received, control is passed back to gpio0_write (or gpio0_read). Note that physio() provides a semaphoring mechanism for access to the card. No other write (or read) can go through until this one finishes since physio() has grabbed *buf*. The ioctl routine makes use of this fact and uses gpio0_busy to sleep on *buf*.

These routines are intentionally structured so that all of the real work is done either in physio() or in gpio0_strategy.

So, the goals of the write routine are to,

- Block writes (reads) from processes that do not own the lock [if one exists],
- Refuse word mode writes (reads) that are transferring an odd number of bytes,,
- Block other writes (reads) if one is currently in progress,
- Call physio(),
- Free the driver and wake up any processes waiting to write (read),
- Return appropriate write (read) errors

### 5.1.8 gpio0_strategy()

Gpio0_strategy does all the real work for i/o requests. It pokes registers on the AFI card to configure the transfer (direction of the dma, transfer counter, PEND) and sends the dma request to the CAM. The routine consists of three basic cases.

First, if this is a diagnostic request, no register poking will be done. The diagnostic program is required to do this itself. Gpio0_strategy only marks REQ_STATE as UNKNOWN and sets up the VQUAD command for either a read or a write.

Second, if this request is a write, a few register pokes are done and the VQUAD command is set up for a write. Note that registers must be poked in different ways depending on whether the last request done was a write or a read, and depending on whether the transfer counter or PEND are used. See the 27114B Hardware ERS for details on which registers to touch and how to touch them.

Third, if this request is a read, some other register pokes are done and the VQUAD command is set up for a read.

Finally, the common parts of the vquad are set up (count and buffer). A watchdog timer is started via gpio0_set_timeout. The dma request message is sent to the CAM via io_send().

Note the precautions for powerfail, io_block_server() and io_unblock_server().

### 5.1.9 gpio0_ioctl()

Gpio0_ioctl is the grab bag for all other types of requests to the device. There are three basic types of requests: IO_CONTROL, IO_STATUS, and IO_ENVIRONMENT. Most of these requests are simple writes or reads to one or more of the CIO direct i/o registers. The exception is GPIO_RESET which does a CIO_DA_SELFTEST control request to the CAM.

The body of gpio0_ioctl() is bracketed by gpio0_busy and gpio0_free to semaphore access to the card; and access to the interface is refused if the interface is locked by another process. If the interface is not locked or is locked by the current process the request is handled as follows,

#### *5.1.9.1 GPIO_LOCK*

IO_CONTROL — Handle LOCK_INTERFACE, UNLOCK_INTERFACE and CLEAR_ALL_LOCKS commands. After each successful lock or unlock request, the number of locks for that open is returned in arg[1] and the number of locks for that interface is returned in arg[2].

IO_STATUS — Return the process id of the locking process in arg[0] and the interface lock count in arg[1]. If the interface is not locked, return -1 in arg[0].

#### *5.1.9.2 GPIO_TIMEOUT*

IO_CONTROL — Handle requests to change the timeout value from the default value, one hour. The request is made in microseconds.

When a request times out, an error of ETIMEDOUT is returned to the user.

History: The pre-7.0 GPIO(dev) manpage stated that a timeout value of 0 was equivalent to infinity (no transaction would timeout). This is incorrect. A timeout value of 0 is equivalent to the default, one hour. The manpage has been changed.

IO_STATUS — Return the timeout value, specific to this open, in arg[0].

#### *5.1.9.3 GPIO_WIDTH*

IO_CONTROL — Handle requests to set the data path width. The only valid arguments are 8 or 16.

IO_STATUS Return the width of the data path (either 8 or 16) in arg[0].

### *5.1.9.4 GPIO_SIGNAL_MASK*

IO_CONTROL Save the process id and the process interrupt handler. Also, enable the 27114B to assert ARQ when ATTN is asserted by the device. If signals are not wanted, the interrupt handler is nulled and the card is disabled from asserting ARQ.

IO_STATUS Return the cause of the interrupt, recorded in int_mask, in arg[0]. For 27114 users, the only reason for an interrupt can be because ATTN was asserted by the device. See gpio0_event_from_cam().

### *5.1.9.5 GPIO_SET_CONFIG*

IO_CONTROL This request allows the user to configure the 27114B as best suits the device. The user creates a mask by OR'ing the following flags (defined in gpio.h) as desired,

PFLG_EDGE_LOGIC - When asserted, data will move on the busy-to-ready (or "falling") edge of the handshake signal PFLG. Otherwise data will move on the ready-to-busy (or "rising") edge of PFLG.

PDIR_OPT_EN - If asserted, the control lines CTL5 and CTL4 will be driven with output from the PDIR bit and the HEND bit (both in register 7). If the flag is not asserted, CTL5 and CTL4 will be driver with whatever the user writes to the control lines. See section "" for more details.

PEND_OPT_EN - If asserted, the device can pull the PEND line of the front plane which will result in the immediate termination of a data transfer. See section "" for more details.

TRNSFR_CTR_EN - If asserted, the driver will enable the transfer counter that exists on the 27114B. See section "" for more details.

Handshake mode. - The choice of handshake is device dependent. There are three choices, FULL_MASTER, FULL_SLAVE, or FIFO_MASTER. If neither of the three flags is asserted, FIFO_MASTER is used.

### *5.1.9.6 GPIO_CTL_LINES*

IO_CONTROL Handles requests to change the control lines. Since the 27114A has only 3 control lines the maximum value of those lines is less than the maximum value for the 27114B, which has 6 control lines.

#### *5.1.9.7 GPIO_RESET*

IO_CONTROL Wait if another reset is in progress. Otherwise, build and send a CIO_CTRL_REQ_MSG to the CAM. Then wait for the CIO_CTRL_REPLY_MSG. Once the reply comes, wake any waiting reset requests

Since the hardware has just been reset, any software options associated to hardware should also be reset (e.g. shadow registers). For historical reasons, purely software options remain unchanged (e.g. timeout).

#### *5.1.9.8 GPIO_STS_LINES*

IO_STATUS Return the value of the status lines in arg[0].

#### *5.1.9.9 GPIO_GET_CONFIG*

IO_STATUS Return a configuration mask, of several hardware options, in arg[0]: EDGE_LOGIC_SENSE, PDIR_OPT_EN, PEND_OPT_EN, TRNSFR_CTR_EN and the handshake mode. The mask definitions are the same as in IO_CONTROL GPIO_SET_CONFIG.

#### *5.1.9.10 GPIO_INTERFACE_TYPE*

IO_STATUS Return the value of the id register in arg[0].

#### *5.1.9.11 GPIO_REG7*

IO_STATUS Return the value of the AFI-status register in arg[0].

#### *5.1.9.12 IO_ENVIRONMENT*

This command is a way to perform multiple IO_STATUS commands via one request. When an IO_ENVIRONMENT call is made the following information is returned in the *env* structure: the status lines, the interface id for the card, the signal mask, the width of the data path, the locking pid, the timeout value and the configuration mask for the card.

There are other values returned, *term_reason, read_pattern, speed*, and *delay*, which have no meaning for the 27114B. The values are there for histerical reasons.

### 5.1.10 gpio0_map_error()

Hardly used. Should be cleaned up; even tossed.

### 5.1.11 gpio0_wait_on_lock()

Used to time a transaction that is waiting for a lock to clear before it can access the interface. If the interface is not locked, immediately return zero. If the interface is locked and O_NDELAY was set at the time of open, immediately return EACCES. Otherwise, start timing the transaction is interrupted by the user (return EINTR).

## 5.2 Port Server

The port server, gpio0(), is the routine to which all messages are delivered. Messages delivered fall into three broad classes:

- Configuration messages from either the configurator or the CAM
- Dma replies and control replies from the CAM
- Powerfail and other event notifications from the CAM.

The port server is simply a switch statement with a separate case for every type of message which gpio0 will receive. Each case is a procedure call to the appropriate message handler. Some of the cases also set *return_frame* to FALSE if the incoming message will be reused by the message handler. Unsupported message types are simply dropped.

The three configuration message handlers - afi_do_bind_msg(), afi_bind_reply_msg(), afi_get_ptr_reply() and gpio0_attach() - are explained in the configuration section. The powerfail recovery process - afi_power_on_req_msg() and afi_pf_get_ptr_reply() - are explained in the powerfail recovery section. The other message handlers - gpio0_reply_from_cam, gpio0_event_from_cam and afi_ctrl_reply_msg - are explained in the following section.

### 5.2.1 gpio0_reply_from_cam()

A dma reply message, CIO_DMA_REPLY_MSG, is sent by the CAM to gpio0 when a dma request which gpio0 initiated earlier has completed. The processing of the reply depends upon the configuration of the 27114B and whether the transaction was a read or a write.

The basic processing for write transactions (that do not use the counter) and for all read transactions is simple. First the *untimeout* routine is called to release the watchdog timer for this request. Second, the *buf* structure associated with this request is filled in. The *b_error* field is set to the status of the dma reply. If there was an error, *pda->buf.b_flags* has the B_ERROR bit set. Finally, biodone() is called to awake the sleeping physio() and return to the user process.

Some specific handling must be done if PEND was asserted of if the read used the transfer counter. For instance,

- If PEND was pulled, the PEND flipflop must be cleared so the next assertion of PEND will register. Also, if PEND was pulled on the write transaction, the FIFO should be cleared.
- If the counter was used for a read, clear the counter.

The only other dma transaction to be discussed is the case where the counter is enabled during a write transaction. This is a special case from the other transactions because the channel may declare the transaction complete before it actually is (see the section on write blocking). Because of this gpio0 can only declare the transaction finished if all the data is off of the card, if PEND was pulled or if the dma reply message was received with bad status.

If PEND was not pulled but data still exists on the card, gpio0 must wait for some indication that the transfer is indeed complete. To do this, the card is enabled to assert ARQ (when PEND is asserted of the transfer counter reaches logical zero) and the state of the driver is set to WAITING_ON_INTR. This way gpio0 need not poll the card nor busy-wait for activity. So if ARQ is asserted before the timer pops, gpio0 receives an event message and can resume completing the transaction there. Otherwise, the state WAITING_ON_INTR will indicate that the transaction should be terminated immediately with an error (that is, an abort message is not necessary here since the dma reply has already been received).

Note that no error recovery is done by this routine. If an error is returned in the dma reply message, it is simply passed up to the user (via the buf structure). The user is responsible for

retrying the request, if need be.

### 5.2.2 gpio0_event_from_cam()

This routines handles CIO_IO_EVENT messages sent by the CAM to gpio0. The only type of event which gpio0 will ever receive is an AES ARQ (this is the only type the AFI card can produce). The 27114B asserts ARQ for 3 reasons,

- The device asserted ATTN on the frontplane,
- The device asserted PEND on the frontplane,
- The transfer counter reached logical zero.

If the ARQ is due to the device asserting ATTN on the frontplane, the driver notifies all interrested processes of the event via a psignal call; providing interrupt handlers have previously been set up for this driver. A search through the per-open table (*pot*) reveals all interrested processes. If no process is interrested in receiving a signal, the event is dropped. Note that when an ARQ occurs, the CAM disables the device adapter from ARQing (for historical reasons). Hence the ARQ must be re-enabled by the user process after an event occurs. This is accomplished via the ioctl GPIO_SIGNAL_MASK.

If the ARQ is due to PEND asserting on the frontplane or due to the transfer counter reaching logical zero, a DMA transaction just completed. In such case afi_arq_finish() is called to clean up after the DMA (stop the watchdog timer, disable PEND and the transfer counter from causing ARQs, figure the dma residue based on the counter, clear the PEND flipflop, the counter and the FIFO) and returns to the user. Note that when an ARQ occurs, the CAM disables the device adapter from ARQing (for historical reasons). So if the user previously enabled the card for ATTN ARQs, gpio0 should re-enable the card to ARQ. Likewise if the card pulled ARQ because of ATTN yet the driver is waiting on either a PEND or word counter ARQ to terminate a dma transaction, it behooves gpio0 to re-enable the card to ARQ.

The driver deciphers the reason for the ARQ by reading register 7 where there are 3 bits reflecting the state of the ATTN flipflop, PEND flipflop and the transfer flipflop. The 3 bits may indicate that more than one flipflop is set.

### 5.2.3 afi_ctrl_reply_msg()

Handles CIO_CTRL_REPLY_MSGs that were caused by CIO_CTRL_REQ_MSGs intending to self test (reset) the 27114B. The driver must match any shadow registers to the now default hardware configuration.

### 5.2.4 afi_arq_finish()

Handles the clean up of DMA transactions that ended via an ARQ on the backplane. This can only happen when the driver enables the card to assert ARQ when PEND is asserted by the device or when the transfer counter reaches logical zero.

The driver must figure the residue from the counter, if appropriate, clear the counter, clear the PEND flipflop, and clear the FIFO.

## 5.3 Configuration

Configuration follows the standard sequence as defined by the HPIOSD. The sequence is a seven part process,

- gpio0 receives a creation message from the Configurator and returns it
- gpio0 receives a do_bind request from the Configurator
- gpio0 sends a bind request to the CAM
- gpio0 receives a bind reply from the CAM
- gpio0 sends a do_bind reply to the Configurator
- gpio0 sends a control request tot he CAM to get the direct i/o pointer
- gpio0 receives a control request, with the direct i/o pointer, from the CAM

The creation message is sent to gpio0 by the Configurator as the first step in the process. This message is used to communicate resource needs and usage options to the system as a whole. Gpio0 fills the message in the standard way and returns.

The do_bind request message tells gpio0 to bind with a particular lower manager (in this case, the CAM). Gpio0 must do four tasks as a result of a do_bind request. First, save its port number, device adapter number, and CAM port number from the do_bind message. Second, set up the gpio0_pda_map entry for this instance of gpio0. Third, initialize everything in the pda that needs to be set up. Fourth, send a bind request message to the CAM.

The bind request message to the CAM is used to establish a connection between gpio0 and the CAM. Gpio0 tells the CAM what device adapter it will control and what subqueue the CAM should send event messages to.

The CAM sends a bind reply message to gpio0 as a result of the bind request message. Gpio0 saves away the subqueue number which the CAM wants it to use for requests. Then gpio0 sends a do_bind reply to the Configurator. If the bind reply message has status CIO_LVL1_CARD, the bind was successful and the AFI card is in the correct slot. Successful binds are followed by the sending of a control request to the CAM for grabbing the direct i/o pointer.

If the bind was unsuccessful (no card was in the slot or the card in the slot was not the AFI card), the control request to get the direct i/o pointer is not done. This instance of the driver will be unusable. Note that an attempt is made to get the direct i/o pointer again on a powerfail. This implies that an AFI card can be added to the system on the fly during a power failure.

If the bind was successful, eventually a control reply is received from the CAM. If the status of the control reply is LLIO_NORMAL, the direct i/o pointer is saved in the pda and the HDWR_DEAD bit of *pda->flags* is turned off. This instance of gpio0 can now be used.

### 5.3.1 afi_do_bind_msg()

Handles the DO_BIND_REQ_MSG received from the Configurator. First, gpio0 saves the information about its lower manger. Then gpio0 initializes its pda values. Finally, gpio0 builds and sends a BIND_REQ_MSG to the CAM.

### 5.3.2 afi_bind_reply_msg()

Handles BIND_REPLY_MSG from the CAM. Once this message is received with good status, binding between the CAM and the GPIO monolith is complete. Gpio0 then builds and sends a cio_ctrl_req message to the CAM. This is to request its direct i/o pointer. Because there are

multiple functions for a cio_ctrl_req message, change the state to indicate how the cio_ctrl_reply message should be handled.

Should the BIND_REPLY_MSG be received from the CAM with bad status, binding has failed. A do_bind_reply message is built and sent to the Configurator and configuration is complete. Set the state to indicate this instance of gpio0 can not be used if opened.

### 5.3.3 afi_get_ptr_reply()

Handles the CIO_CTRL_REPLY_MSG that was caused by the CIO_CTRL_REQ_MSG sent during configuration. If the cio_ctrl_reply message is received with good status AND the message indicates the device adapter in the slot is a level 1 type card, the message contains a valid direct i/o pointer for gpio0. Save the pointer, use the pointer to enable the card to function, complete configuration by sending a do_bind_reply message to the Configurator and declare the driver usable by clearing the HDWR_DEAD bit from the pda *flags*.

### 5.3.4 gpio0_attach()

This routines finds a mapping between the device adapters lu number and a pda. A routine from the old days, it should be replaced by a call to *port_to_index()*.

## 5.4 Powerfail Processing

The driver must do special processing whenever powerfail occurs. The driver is notified of a powerfail by the receipt of a **POWER_ON_REQ_MSG** from the CAM. The CAM sends the power-on request message to gpio0 after power is restored.

No special processing of outstanding requests to the CAM is needed. The CAM guarantees closure on all requests that are sent to it. If a request was active at the time of the power failure, a dma reply (with powerfail aborted status) will eventually be sent to gpio0. This rely can be handled in the normal course of handling dma replies. An error will be returned to the user process.

Note that gpio0 does not do any real recovery from a powerfail. For the most part, the user process is responsible for setting up the card again (if it needs something other than power-on state). In addition, the user process must set up the device attached to the AFI card.

### 5.4.1 afi_power_on_req_msg()

Gpio0 must send a **POWER_ON_REPLY_MSG** to the CAM as soon as the power-on request message is received. In addition, gpio0 attempts to re-establish a valid direct i/o pointer via a CIO control request message to the CAM. This is also important if the system was initially brought up with either no card in the AFI slot or the wrong card in the slot. It is, therefore, possible to add the AFI card to the system during powerfail and have it work.

### 5.4.2 afi_pf_get_ptr_reply()

Once the control reply is received with good status and the status indicates the card is a 27114B, the direct i/o pointer is considered valid. The card is then reset and enabled.

The reset is an important part of the handling of emulated Fatal Error mode:

HPPB drivers own a page of i/o space and are responsible for checking to see when their page is in FE mode; eventually clearing the FE bit.

However, CIO drivers do not own an entire page of I/O space. Each page is shared by multiple drivers. Therefore CIO drivers, not knowing the progress of other drivers, can not clear the page FE bit; the CAM does.

This brings us to the interesting AFI case of why an explicit reset (one done by touching i/o space rather than by sending a control request to the CAM) is necessary in AFI powerfail.

Suppose gpio0 is in the middle of a transaction at the time power fails. Once power is restored, the transaction resumes where it left off; gpio0's power on request message will be queued. If the CAM has already taken the page which the AFI belongs to out of FE mode, the transaction can now touch the i/o space; the page that gpio0's address space belongs in will no longer be marked FE mode. Since we rely on the card being in default configuration after a power failure, it should be reset to guarantee that default state.

After resetting the card, gpio0 must restore the states of software values relating to hardware configuration.

## 6. Testing

Gpio0 should be tested in four different ways:

- Instruction Coverage Analysis (ICA)
- Configuration Testing
- Stress Testing
- Powerfail Testing

ICA is done to verify that all the code actually gets executed at least once. This approach has definite limitations. For instance, ICA may indicate the code was hit, but you will not know that it was hit while the driver/card was in different states. The goal is 100% measured coverage.

Configuration testing means building several systems with different properties. For example,

- Build a system with no gpio0 configured. Make sure that it boots normally.
- Build a system with several gpio0's. Make sure that they are all usable.
- Build a system with gpio0's on a second channel. Make sure they are usable.
- Build a system with one gpio0. Try booting without a card in the slot, with different cards in the slot and with an AFI card in the slot.

Stress testing involves lots of activity simultaneously on one or more cards. For example, try lots of i/o to one card with two different processes.

Powerfail testing can be done by power cycling the processor while the card is idle, busy with different requests, and not present. Make sure multiple powerfails work.

A test suite does exist for gpio0. All tests in the suite should pass after any change is made. New tests should be added as features are added or changed. Currently the suite is in Wayback:/mnt/azure/suzyz/tests/scaffold.

**Appendix I: Diagnostic Ioctls**

Certain ioctl calls are only allowed if gpio0 has been opened in diagnostic mode. If a non-diagnostic attempts to invoke any of these calls, EINVAL is returned. These calls involve direct register reads and writes. Therefore, the diagnostic program has complete control of the afi card. It is the diagnostic program's responsibility to put the AFI card back into a good state before it exits. This can be done by resetting the AFI card.

IO_CONTROL requests write to the specified register with the value passed in arg[0]. IO_STATUS requests read the specified register in arg[0].

| Type | Command | Description |
|---|---|---|
| IO_CONTROL | GPIO_REG0 | write register 0 -- data register |
| IO_CONTROL | GPIO_REG1 | write register 1 -- CIO control register |
| IO_CONTROL | GPIO_REG7 | write register 7 -- 27114 control I register |
| IO_CONTROL | GPIO_CLEAR | clear data path -- clear fifo, re-enable fifo |
| IO_CONTROL | GPIO_REGA | write register A -- transfer ctr |
| IO_CONTROL | GPIO_REGB | write register B -- transfer ctr/27114 control II |
| | | |
| IO_STATUS | GPIO_REG0 | read register 0 -- data register |
| IO_STATUS | GPIO_REG1 | read register 1 -- CIO sense register |
| IO_STATUS | GPIO_REG3 | read register 3 -- ID register |
| IO_STATUS | GPIO_REG7 | read register 7 -- 27114 status register |
| IO_STATUS | GPIO_REG9 | read register 9 -- CIO status register |
| IO_STATUS | GPIO_REGA | read register A -- low and mid bytes of counter |
| IO_STATUS | GPIO_REGB | read register B -- high byte of counter |
| IO_STATUS | GPIO_POLL | return whether the card is ready for bus requests |
| IO_STATUS | GPIO_PORTNUM | return the port number for this interface |

**Appendix II: Message Formats**

```
/***                                                    ***
 *** All of the messages and constants explained here   ***
 *** are located in one of the two following files,     ***
 ***                                                    ***
 ***            sio/llio.h                              ***
 ***            sio/iocam.h                             ***
 ***                                                    ***
 *** These files are the final authority for message    ***
 *** layout and constants.                              ***
 ***                                                    ***/
```

```
/** Standard header for all messages **/

typedef struct {
        shortint                msg_descriptor;
        shortint                message_id;
        int                     transaction_num;
        port_num_type           from_port;
} llio_std_header_type;
```

```
/** Message descriptors used by gpio0 **/

#define CREATION_MSG            2
#define DO_BIND_REQ_MSG         3
#define DO_BIND_REPLY_MSG       4
#define BIND_REQ_MSG            5
#define BIND_REPLY_MSG          6

#define CIO_DMA_IO_REQ_MSG      100
#define CIO_DMA_IO_REPLY_MSG    101

#define CIO_CTRL_REQ_MSG        102
#define CIO_CTRL_REPLY_MSG      103

#define CIO_IO_EVENT_MSG        104

#define POWER_ON_REQ_MSG        17
#define POWER_ON_REPLY_MSG      18

#define TIMER_EVENT_MSG         19
```

```
/** Configuration messages **/

typedef struct {
        creation_options        create_options;
        int                     server_data_len;
        int                     max_msg_size;
        int                     num_msgs;
        bit8                    num_subqueues;
        port_num_type           port_num;
} creation_info_type;

typedef struct {
        io_subq_type            reply_subq;
        port_num_type           mgr_port_num;
        int                     config_addr_3;
        int                     config_addr_2;
        int                     config_addr_1;
        port_num_type           lm_port_num;
        int                     load_info;
} do_bind_req_type;



typedef struct {
        llio_status_type        reply_status;
} do_bind_reply_type;

typedef struct {
        io_subq_type        reply_subq;
        io_subq_type        hm_event_subq;
        shortint            hm_subsys_num;
        shortint            hm_meta_lang;
        int                 hm_rev_code;
        int                 hm_config_addr_3;
        int                 hm_config_addr_2;
        int                 hm_config_addr_1;
} bind_req_type;

typedef struct {
        llio_status_type    reply_status;
        int                 lm_rev_code;
        shortint            lm_queue_depth;
        io_subq_type        lm_low_req_subq;
        io_subq_type        lm_hi_req_subq;
        unsigned char       lm_freeze_data:1;
        unsigned char       lm_alignment:1;
} bind_reply_type;
```

```
/** DMA request and reply messages **/

typedef struct {
        io_subq_type            reply_subq;
        bit8                    da_number;
        cio_vquad_ptr           vquad_chain;
} cio_dma_req_type;



typedef struct {
        llio_status_type        llio_status;
        cio_vquad_ptr           vquad_chain;
} cio_dma_reply_type;


/** Quads and DMA commands **/

typedef struct cio_vquad_type {
        cio_cmd_type                    command;
        int                             count;
        int                             residue;
        struct cio_vquad_type           *link;
        data_ptr_type                   buffer;
        addr_class_type                 addr_class;
} cio_vquad_type;

typedef cio_vquad_type          *cio_vquad_ptr;



typedef union {
        int             cio_cmd;
        struct {
                bit8            order;
                unsigned        suppression:2;
                unsigned        logch_break:1;
                unsigned        blocked:1;
                unsigned        reserved:17;
                unsigned        read_write:1;
                unsigned        continue_dma:1;
                unsigned        exact:1;
        } u;
} cio_cmd_type;


/** CIO orders **/

#define CIO_RD_WORD_ORDER     0x60
#define CIO_WD_WORD_ORDER     0x70
```

```
/** IO Event Message **/

typedef struct {
        int                     event;
        union {
                int                     int_info[(3)+1];
                bit8                    byte_info[(15)+1];
        } u;
} cio_io_event_type;

#define CIO_ARQ_STATUS          1


/** CIO control request and reply messages **/

typedef struct {
        io_subq_type                    reply_subq;
        bit8                            da_number;
        bit8                            ctrl_func;
        int                             ctrl_parm;
} cio_ctrl_req_type;

typedef struct {
        llio_status_type                llio_status;
        int                             ctrl_info;
        int                             extra_info;
} cio_ctrl_reply_type;

#define CIO_DA_SELFTEST         2
#define CIO_GET_DIRECT_IO_PTR   10



typedef struct  cio_quadrant  {
        volatile int                    rsvd1[3];
        volatile int                    normal;
        volatile int                    rsvd2[3];
        volatile int                    cend;
        volatile int                    rsvd3[3];
        volatile int                    cbyte;
        volatile int                    rsvd4[3];
        volatile int                    cendbyte;
};


typedef struct ciodio {
        struct cio_quadrant             data;
        struct cio_quadrant             ctlsens;
        struct cio_quadrant             order;
        struct cio_quadrant             statcmd;
} ciodio;
```