CMI
125 West Park Loop
Huntsville, AL 36806
Phone 256.722.0175
Fax 256.722.0144

Chandler/May, Inc.

VxWorks Device Driver User’s Manual

VxWorks Device Driver Software for the General Standards PMC-16AIO-88 hosted on PowerPC 604 and 80x86 Processors

Document number:	9005003	Revision:	1.1	Date: 03/09/99
Engineering Approval:				Date:
Quality Representative Approval:				Date:

Acknowledgments

1 Scope

The purpose of this document is to describe how to interface with the PMC-16AIO-88 VxWorks Device Driver developed by Chandler/May, Incorporated (CMI). This software provides the interface between "Application Software" and the 16AIO-88 Board. The interface to this board is at the I/O system level. It requires no knowledge of the actual board addressing of control/data register locations. It does, however, require some knowledge of the individual bit representations for most control/data registers on the device.

The 16AIO-88 Driver Software executes under control of the VxWorks operating system. The 16AIO-88 is implemented as a standard VxWorks device driver written in the 'C' programming language. The 16AIO-88 Driver Software is designed to operate on CPU boards containing PowerPC 604 processors as well as VME CPU boards containing 80x86 processors. For example, the Force PPC/PowerCore-6604 CPU Board, the Motorola MVME 2304, and the SCI JTT 686 CPU board.

2 Hardware Overview

The General Standards Corporation (GSC) 16AIO-88 board is a single-width analog I/O interface that fits into a PCI Mezzanine Card slot. This board has 16 channels, 8 for input and 8 for output. The output channels are capable of supporting synchronous and asynchronous modes. The inputs can be customized as 8 single-ended or 4 differential input channels via software configuration. It also provides for minimum off-line maintenance by providing calibration and self-testing functions.

The 16AIO-88 board includes a rate generator and a DMA controller. The rate controller is provided to control the rate input channels are scanned, the output channels are strobed, or both input and output functions for synchronicity. The DMA transfers are supported when the board is acting as the bus master and the local bursting mode disabled.

The configuration of the interrupting capability of the 16AIO-88 board is described in the hardware manual for the board. The 16AIO-88 Device Driver must be used correctly in accordance with the hardware configuration in order to provide consistent results.

3 Referenced Documents

The following documents provided reference material used in the development of this design:

- PMC-16AIO-88 8-Channel, 16-Bit Analog Input/Output User's Manual – Revision A, General Standards Corporation.
- PLX Technology, Inc. PCI 9080 PCI Bus Master Interface Chip data sheet.
- Motorola MVME2300-Series VME Processor Module Programmer's Reference Guide.
- Force PPC/PowerCore-6603/4 Technical Reference Manual.

4 Driver Interface

The 16AIO-88 Driver conforms to the device driver standards required by the VxWorks Operating System and contains the following standard driver entry points.

- GS_16AIODrvInstall() - Installs the device driver for use with multiple 16AIO-88 Cards
- GS_16AIODrvRemove() - Removes the device driver from use
- open() - opens a driver interface to one 16AIO-88 Card
- close() - closes a driver interface to one 16AIO-88 Card
- read() - reads data received from a 16AIO-88 Card
- write() - writes data to be transmitted by a 16AIO-88 Card
- ioctl() - performs various control and setup functions on the 16AIO-88 Card

The 16AIO-88 Device Driver provides a standard I/O system interface to the GSC PMC-16AIO-88 card for VxWorks applications which run on the VxWorks target processor. The device driver is installed and devices created through the use of standard VxWorks I/O system functions. The functions of the driver can then be used to access the board.

Included in the device driver software package is a menu driven board testing program and source code. This program is delivered undocumented and unsupported but may be used to exercise the 16AIO-88 card and device driver. It can also be used to break the learning curve somewhat for programming the 16AIO-88 device.

If the user wishes to use the 16AIO Device Driver with the interrupting capability of the board then a user supplied Interrupt Service Routine (ISR) must be written. This ISR will be called by the driver when an interrupt is received from the board. There are limitations on the functionality of a VxWorks ISR. These are documented in the VxWorks Programmer's Guide and must be strictly followed in writing the ISR.

The Device Driver initializes the board to disable all types of 16AIO-88 interrupts through software control except for PCI interrupts controlled through the Shared Runtime - Interrupt Control/Status register. 16AIO-88 Interrupts must be enabled through the use of the ioctl function in order to take advantage of the interrupting capability of the board. The ioctl function must also be used to specify the user supplied ISR which will be invoked when an interrupt is received from the board. If interrupting is enabled and the user supplied ISR has not been specified then nothing will happen in the driver when an interrupt is received from the board.

The 16AIO-88 Device Driver allows for multiple boards on a single PCI bus. Each board will be addressed as a separate VxWorks I/O system device. This device will be created when the driver is installed and is then available for all driver operations (open, close, ...).

It is important to note that the 16AIO-88 device driver is target processor dependent and thus BSP dependent. System calls are made within the driver which are only available through certain board support packages. This is due to the fact that PCI memory and I/O space could be mapped differently for each target processor board. Also, it may be possible that the PMC slot interrupt level may be mapped differently for each target processor board.

4.1 GS_16AIODrvInstall()

The GS_16AIODrvInstall () function installs the device driver into the VxWorks operating system. This function must be called prior to using any of the other driver functions. This function should not be called again without first calling the GS_16AIODrvRemove() function.

The GS_16AIODrvInstall () function performs the following operations:

- Installs the device driver into the VxWorks operating system
- Performs the following for each PMC Slot on the processor board
    - Determines if this slot contains a PCI card by examining the CPU board's registers
    - Determines if the slot contains a 16AIO-88 board by examining the PCI Configuration Device Type and Vendor ID Registers
    - Programs the PCI Configuration Base Address and Configuration Address Registers with predefined addresses
    - Enables the 16AIO-88 Card to respond over the PCI Bus
    - Connects the driver interrupt handler for the interrupt number
    - Installs a device for the PMC Slot
    - Enables the PCI Interrupt for the PMC Slot

PROTOTYPE:

extern int GS_16AIODrvInstall(BOOL   bDebug);

Where:

bDebug -	A boolean that is sent to the driver to enable debugging. If enabled the driver will display error and status messages on the console during driver access. Note, this should not be enabled during time sensitive processes.

Returns OK on success and ERROR on failure

EXAMPLE:

```
STATUS iStatus;

/* Install the 16AIO-88 VxWorks Device Driver. */
iStatus = GS_16AIODrvInstall(TRUE);
```

4.2 GS_16AIODrvRemove()

The GS_16AIODrvRemove() function is used to remove the 16AIO-88 Device Driver from the VxWorks operating system. This function should only be called after a call to the GS_16AIODrvInstall() function. The GS_16AIODrvRemove() function closes all the open devices for each PMC slot and removes the device driver from the operating system.

PROTOTYPE:

extern int GS_16AIODrvRemove(void);

Returns OK on success and ERROR on failure

EXAMPLE:

```
STATUS iStatus;

/* Remove the 16AIO-88 Driver */
iStatus = GS_16AIODrvRemove();
```

4.3 open()

The open() function is the standard VxWorks entry point to open a connection to a 16AIO-88 Card in one PMC Slot. This function may only be called after a call to the GS_16AIODrvInstall() function is made.

PROTOTYPE:

extern int open(const char *cName, int iFlags, int iMode)

Where:

cName - name of the device being opened which is one of the following depending on the slot the 16AIO-88 Board is in:

- GS_16AIO_PMC1
- GS_16AIO_PMC2

iFlags - is not used.

iMode - is not used.

Returns OK on success and ERROR on failure

EXAMPLE:

```
int         FileDesc[2];
LOCAL char  *slotName[] = { GS_16AIO_PMC1, GS_16AIO_PMC2};
int         16AIOSlot = 1;


/*   open the 16AIO-88 device for slot 1   */
FileDesc[16AIOSlot] = open(slotName[16AIOSlot], O_RDWR, 0644);

if (FileDesc[16AIOSlot] == ERROR)
{
   logMsg("Cannot Open Device Error %s\n\n",
            (int) slotName[16AIOSlot], 0, 0, 0, 0, 0);
}
```

4.4 close()

The close() function is the standard VxWorks entry point to close a connection to a 16AIO-88 Card in one PMC Slot. This function should only be called after the open function has been successfully called for a slot where a 16AIO-88 Card resides. The close function closes an interface to a 16AIO-88 device.

PROTOTYPE:

extern STATUS close(int iFd);

Where:

iFd - File Descriptor returned from a call to the open function.

Returns OK if successful or ERROR if unsuccessful.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;

/*  close the device on slot 2  */
if  (close(FileDesc[16AIOSlot]) == ERROR)
{
   logMsg("Close Error for Slot #%d\n\n", 16AIOSlot, 0, 0, 0, 0, 0);
}
FileDesc[16AIOSlot] = ERROR;
```

4.5 read()

The read() function is the standard VxWorks entry point to receive channel data from a 16AIO-88 Card FIFO in one PMC Slot. This function should only be called after the open function has been successfully called for a slot where a 16AIO-88 Card resides. The 16AIO-88 has two data configurations in which the input channels can be read, single-ended and differential. In the single-ended mode, there are 8 input channels read as 2 bytes each channel. Whereas in differential mode, there are only 4 input channels also with 2 bytes read each channel. Depending on the read mode of the driver which can be set using the ioctl() function, the FIFO data will either be transferred to the user buffer using the PLX 9080 DMA capability or will be accessed directly and assigned 16 bits at a time. Regardless of configuration, the read() function will read these channels in sequential order starting with channel 0. Each channel is capable of handling 16 bits of data.

The read() logic is as follows:

- Verify pointer to user buffer and note the requested number of bytes to read.
- Take the semaphore.
- Check analog input mode selection – is it single ended or differential.
- Calculate the number of input scans requested based on the requested number of bytes and the analog input mode.
- Verify that the number of input scans requested is greater than zero.
- If the driver is in DMA_MODE then first verify the number of scans available in the input FIFO buffer is not less than the number of scans requested. If it is less that what is requested then adjust the number of scans requested to match what is available. Next setup and transfer data to the user buffer from the input FIFO.
- If the driver is in the SCAN_MODE then burst the inputs if necessary, verify the number of samples in the input FIFO buffer, then read each channel one at a time. Repeat this process based on the number of scans requested.
- Return the number of bytes read.

PROTOTYPE:

extern int read(int iFd, char *cBuffer, size_t iMaxbytes);

Where:

iFd -	File Descriptor returned from a call to the open function.
CBuffer -	pointer to character array to store read bytes.
iMaxbytes -	maximum number of bytes to read.

Returns Number of bytes read if successful or ERROR if unsuccessful.

EXAMPLE:

```
#define MAXSAMPLES 8

int   FileDesc[2];
int   iNumBytesRead;
int   16AIOSlot = 1;
char  pusBuffer[MAXSAMPLES * 2];


/* Configure driver read() mode */
if( ioctl( FileDesc[16AIOSlot], READ_MODE_CONFIG, DMA_MODE) == ERROR )
{
    logMsg("ioctl READ_MODE_CONFIG Failed for Slot #%d\n\n",
           16AIOSlot, 0, 0, 0, 0, 0);
}

/* Configure Input Channel Mode */
if( ioctl( FileDesc[16AIOSlot], INPUT_MODE_CONFIG, SINGLE_CONTINUOUS )
== ERROR )
{
    logMsg("ioctl INPUT_MODE_CONFIG Failed for Slot #%d\n\n",
           16AIOSlot, 0, 0, 0, 0, 0);
}

/* Read from the 16AIO-88 device  */
iNumBytesRead = read(FileDesc[16AIOSlot],
                     pusBuffer,
                     sizeof(pusBuffer));

if (iNumBytesRead == 0)
{
   logMsg("Read failed for Slot #%d\n", 16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.6 write()

The write() function is the standard VxWorks entry point to transmit channel data to the 16AIO-88 Card FIFO in one PMC Slot. This function should only be called after the open function has been successfully called for a slot where a 16AIO-88 Card resides. For the PMC-16AIO-88 it is necessary for the data to be in a 16-bit format. The data written to the output channels should have the channel number first followed by the data. Therefore, there is a structure provided specifically for this purpose. It is in the header file of the driver. It should be noted that it is the user's responsibility to strobe the outputs after using the write() function if the automatic updating of outputs has been disabled. The data written to the output buffer must be flushed to the output channel specified by the user. Thus, an output strobe is necessary. This is done to give the user flexibility for writing one or many channels at a time.

The write() logic is as follows:

- Verify pointer to user buffer and note the requested number of bytes to write.
- Take the semaphore.
- Calculate the number of output samples based on the requested number of bytes.
- Validate the channel number for the sample to be written. Verify that there is room in the output buffer for the current channel number to be written. Write the channel number to the output FIFO. Verify that there is room in the output buffer for the current sample to be written. Write the sample to the output FIFO. Repeat this process based on the number of samples given.
- Return the number of bytes written.

PROTOTYPE:

extern int write(int iFd, char *cBuffer, size_t iNBytes);

Where:

iFd -	File Descriptor returned from a call to the open function.
cBuffer -	pointer to WRITE_PARAM structure containing an array of channel data and sample data to write.
iNBytes -	total number of bytes to write.

Returns Number of bytes written if successful or ERROR if unsuccessful. This includes 16 bits of channel information as well as 16 bits of channel data for each sample.

typedef struct WriteParam
{

    USHORT usChannel;
    USHORT usData;

} **WRITE_PARAM**;

EXAMPLE:

```
int         FileDesc[2];
int         i, iNumBytesWritten;
WRITE_PARAM pusBuffer[8];
int         16AIOSlot = 1;


for ( i = 0; i < 8; i++ )
{
   pusBuffer[i].usChannel = (USHORT)i;
   pusBuffer[i].usData = 0xAAAA;
}

/* Disable Automatic Output Strobe. */
if (ioctl(FileDesc[16AIOSlot], DISABLE_OUTPUT_STROBE, 0) == ERROR)
{
   logMsg("Disable Output Strobe Failed for Slot #%d\n\n",
          16AIOSlot, 0, 0, 0, 0, 0 );
}

iNumBytesWritten = write(FileDesc[16AIOSlot],
                         (char*)&pusBuffer,
                         sizeof(pusBuffer));

if (iNumBytesWritten == 0)
{
   logMsg("write failed for Slot #%d\n", 16AIOSlot, 0, 0, 0, 0, 0);
}
else
{
   if (iNumBytesWritten != (8*sizeof(WRITE_PARAM)))
   {
      logMsg("Only wrote %d bytes\n",
             iNumBytesWritten, 0, 0, 0, 0, 0);
   }
   else
   {
      /* Strobe Outputs. */
      if (ioctl(FileDesc[16AIOSlot], STROBE_OUTPUTS, 0) == ERROR)
      {
            logMsg("Output Strobe Failed for Slot #%d\n\n",
                   16AIOSlot, 0, 0, 0, 0, 0 );
        }
    }
}
```

4.7 ioctl()

The ioctl() function is the standard VxWorks entry point to perform control and setup operations on an 16AIO-88 Card in one PMC Slot. This function should only be called after the open function has been successfully called for a slot where a 16AIO-88 Card resides. The ioctl() function will perform different functions based upon the function parameter. These functions will be described in the following subparagraphs.

PROTOTYPE:

extern int ioctl(int iFd, int iFunction, int iArg);

Where:

iFd - File Descriptor returned from a call to the open function.

iFunction - The ioctl function to perform which is one of the following:

NO_COMMAND - Empty Command, performs nothing.

INIT_BOARD - Initializes the 16AIO-88 Board.

READ_REGISTER - Reads a specified 16AIO-88 Register.

WRITE_REGISTER - Writes to a specified 16AIO-88 Register.

START_DMA - Starts a DMA Read from the 16AIO Board

REG_FOR_INT_NOTIFY - Registers the application code to be Notified when an Interrupt occurs.

GET_DEVICE_ERROR - Returns the Error that occurred during the last access to the 16AIO-88 Driver.

READ_MODE_CONFIG - Configures the 16AIO-88 read() mode (FIFO scan reads or DMA enabled FIFO reads).

INPUT_MODE_CONFIG - Configures the 16AIO-88 Input channels (single-ended or differential mode, burst or continuous scan).

INPUT_TEST – Performs a Self-test for 16AIO-88 Board validation.

LOOP_TEST_CHANNEL – Sets the output Channel to be connected to input channel 0 for Loopback Test.

CALIBRATION_MODE – Sets and runs Calibration operation.

INT_SOURCE – Sets Interrupt Source condition.

ENABLE_PCI_INTERRUPTS – Enables PCI Interrupts in order for the 16AIO-88 to produce a local interrupt request.

DISABLE_PCI_INTERRUPTS – Disables PCI Interrupts.

LAST_CHANNEL – Sets the Last output Channel in strobe sequence.

PROGRAM_RATE_GEN – Programs the Rate Generator for specified rate frequency.

ENABLE_RATE_GEN – Enables and Configures the Rate Generator to control the input scan rate, the output strobe rate, or both.

DISABLE_RATE_GEN – Disables the Rate Generator for preferred rate (input, output, or both).

ENABLE_OUTPUT_STROBE – Enables the Output Strobe and disables automatic updating of outputs.

DISABLE_OUTPUT_STOBE – Disables the Output Strobe and enable automatic updating of outputs.

CLEAR_INT_REQUEST – Clears the Interrupt Request flag.

STROBE_OUTPUTS – Strobes all Output channels once.

SCAN_INPUTS – Scans all Input channels once.

iArg - The parameters to the specific ioctl() function. See the following subsections for a description of the parameters for each function.

Returns OK if successful or ERROR if unsuccessful.

4.7.1 NO_COMMAND

This is an empty driver entry point. This command may be given to validate that the driver is correctly installed and that the 16AIO-88 Board Device has been successfully opened.

arg PARAMETER:

Not used.

EXAMPLE:

```
int FileDesc[2];
int 16IAOSlot = 1;

if (ioctl(FileDesc[16AIOSlot], NO_COMMAND, 0) == ERROR)
{
   logMsg("ioctl NO_COMMAND Failed for Slot #%d\n\n", 16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.2 INIT_BOARD

The INIT_BOARD Function initializes the board and sets all defaults.

arg PARAMETER:

Not used.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;

if (ioctl(FileDesc[16AIOSlot], INIT_BOARD, 0) == ERROR)
{
   logMsg("Board Initialization Failed for Slot #%d\n\n", 16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.3 READ_REGISTER

The READ_REGISTER Function reads and returns the contents of one of the 16IAO-88 Registers.

arg PARAMETER:

REG_PARAM *

int e16AIORegister - One of the following registers to read. Refer to the 16AIO-88 Hardware documentation for a description of each register.

***** 16AIO-88 Registers *****

BOARD_CTRL_REG

RATE_GEN_REG

IN_OUT_FIFO_REG

RATE_CTRL_REG

INPUT_BUF_PTR_REG

BUFFER_FLAGS_REG

***** DMA Registers *****

DMA_CH_0_MODE

DMA_CH_0_PCI_ADDR

DMA_CH_0_LOCAL_ADDR

DMA_CH_0_TRANS_BYTE_CNT

DMA_CH_0_DESC_PTR

DMA_CH_1_MODE

DMA_CH_1_PCI_ADDR

DMA_CH_1_LOCAL_ADDR

DMA_CH_1_TRANS_BYTE_CNT

DMA_CH_1_DESC_PTR

DMA_CMD_STATUS

DMA_MODE_ARB_REG

DMA_THRESHOLD_REG

***** PCI Configuration Registers *****

DEVICE_VENDOR_ID

STATUS_COMMAND

CLASS_CODE_REVISION_ID

BIST_HDR_TYPE_LAT_CACHE_SIZE

PCI_MEM_BASE_ADDR

PCI_IO_BASE_ADDR

PCI_BASE_ADDR_0

PCI_BASE_ADDR_1

CARDBUS_CIS_PTR

SUBSYS_ID_VENDOR_ID

PCI_BASE_ADDR_LOC_ROM

LAT_GNT_INT_PIN_LINE

***** Local Configuration Registers. *****

PCI_TO_LOC_ADDR_0_RNG

LOC_BASE_ADDR_REMAP_0

MODE_ARBITRATION

BIG_LITTLE_ENDIAN_DESC

PCI_TO_LOC_ROM_RNG

LOC_BASE_ADDR_REMAP_EXP_ROM

BUS_REG_DESC_0_FOR_PCI_LOC

DIR_MASTER_TO_PCI_RNG

LOC_ADDR_FOR_DIR_MASTER_MEM

LOC_ADDR_FOR_DIR_MASTER_IO

PCI_ADDR_REMAP_DIR_MASTER

PCI_CFG_ADDR_DIR_MASTER_IO

PCI_TO_LOC_ADDR_1_RNG

LOC_BASE_ADDR_REMAP_1

BUS_REG_DESC_1_FOR_PCI_LOC

***** Run Time Registers *****

MAILBOX_REGISTER_0

MAILBOX_REGISTER_1

MAILBOX_REGISTER_2

MAILBOX_REGISTER_3

MAILBOX_REGISTER_4

MAILBOX_REGISTER_5

MAILBOX_REGISTER_6

MAILBOX_REGISTER_7

PCI_TO_LOC_DOORBELL

LOC_TO_PCI_DOORBELL

INT_CTRL_STATUS

PROM_CTRL_CMD_CODES_CTRL

DEVICE_ID_VENDOR_ID

REVISION_ID

MAILBOX_REG_0

MAILBOX_REG_1

***** Messaging Queue Registers *****

OUT_POST_Q_INT_STATUS

OUT_POST_Q_INT_MASK

IN_Q_PORT

OUT_Q_PORT

MSG_UNIT_CONFIG

Q_BASE_ADDR

IN_FREE_HEAD_PTR

IN_FREE_TAIL_PTR

IN_POST_HEAD_PTR

IN_POST_TAIL_PTR

OUT_FREE_HEAD_PTR

OUT_FREE_TAIL_PTR

OUT_POST_HEAD_PTR

OUT_POST_TAIL_PTR

Q_STATUS_CTRL_REG

ULONG *pulValue - Pointer to the location where the value read is to be stored

EXAMPLE:

```
int         FileDesc[2];
REG_PARAM   theReg;
ULONG       ulValue;
int         16AIOSlot = 1;

theReg.pulValue = &ulValue;
theReg.e16AIORegister = BOARD_CTRL_REG;

if (ioctl(FileDesc[16AIOSlot], READ_REGISTER, (int) &theReg) ==
          ERROR)
{

   logMsg("Read Register Failed for Slot #%d\n\n", 16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.4 WRITE_REGISTER

The WRITE_REGISTER Function writes a value to one of the 16AIO-88 Registers.

arg PARAMETER:

REG_PARAM *

int e16AIORegister - One of the following registers to write. Refer to the 16AIO-88 Hardware documentation for a description of each register.

***** 16AIO-88 Registers *****

BOARD_CTRL_REG

RATE_GEN_REG

IN_OUT_FIFO_REG

RATE_CTRL_REG

***** DMA Registers *****

DMA_CH_0_MODE

DMA_CH_0_PCI_ADDR

DMA_CH_0_LOCAL_ADDR

DMA_CH_0_TRANS_BYTE_CNT

DMA_CH_0_DESC_PTR

DMA_CH_1_MODE

DMA_CH_1_PCI_ADDR

DMA_CH_1_LOCAL_ADDR

DMA_CH_1_TRANS_BYTE_CNT

DMA_CH_1_DESC_PTR

DMA_CMD_STATUS

DMA_MODE_ARB_REG

DMA_THRESHOLD_REG

***** PCI Configuration Registers *****

DEVICE_VENDOR_ID

STATUS_COMMAND

CLASS_CODE_REVISION_ID

BIST_HDR_TYPE_LAT_CACHE_SIZE

PCI_MEM_BASE_ADDR

PCI_IO_BASE_ADDR

PCI_BASE_ADDR_0

PCI_BASE_ADDR_1

CARDBUS_CIS_PTR

SUBSYS_ID_VENDOR_ID

PCI_BASE_ADDR_LOC_ROM

LAT_GNT_INT_PIN_LINE

***** Local Configuration Registers. *****

PCI_TO_LOC_ADDR_0_RNG

LOC_BASE_ADDR_REMAP_0

MODE_ARBITRATION

BIG_LITTLE_ENDIAN_DESC

PCI_TO_LOC_ROM_RNG

LOC_BASE_ADDR_REMAP_EXP_ROM

BUS_REG_DESC_0_FOR_PCI_LOC

DIR_MASTER_TO_PCI_RNG

LOC_ADDR_FOR_DIR_MASTER_MEM

LOC_ADDR_FOR_DIR_MASTER_IO

PCI_ADDR_REMAP_DIR_MASTER

PCI_CFG_ADDR_DIR_MASTER_IO

PCI_TO_LOC_ADDR_1_RNG

LOC_BASE_ADDR_REMAP_1

BUS_REG_DESC_1_FOR_PCI_LOC

***** Run Time Registers *****

MAILBOX_REGISTER_0

MAILBOX_REGISTER_1

MAILBOX_REGISTER_2

MAILBOX_REGISTER_3

MAILBOX_REGISTER_4

MAILBOX_REGISTER_5

MAILBOX_REGISTER_6

MAILBOX_REGISTER_7

PCI_TO_LOC_DOORBELL

LOC_TO_PCI_DOORBELL

INT_CTRL_STATUS

PROM_CTRL_CMD_CODES_CTRL

DEVICE_ID_VENDOR_ID

REVISION_ID

MAILBOX_REG_0

MAILBOX_REG_1

***** Messaging Queue Registers *****

OUT_POST_Q_INT_STATUS

OUT_POST_Q_INT_MASK

IN_Q_PORT

OUT_Q_PORT

MSG_UNIT_CONFIG

Q_BASE_ADDR

IN_FREE_HEAD_PTR

IN_FREE_TAIL_PTR

IN_POST_HEAD_PTR

IN_POST_TAIL_PTR

OUT_FREE_HEAD_PTR

OUT_FREE_TAIL_PTR

OUT_POST_HEAD_PTR

OUT_POST_TAIL_PTR

Q_STATUS_CTRL_REG

ULONG *pulValue - Pointer to the location containing the value to be written.

EXAMPLE:

```
int   FileDesc[2];
REG_  PARAM theReg;
ULONG ulValue = 0xAAAA;
```

```
int   16AIOSlot = 1;

theReg.pulValue = &ulValue;
theReg.e16AIORegister = OUT_Q_PORT;

if (ioctl(FileDesc[16AIOSlot], WRITE_REGISTER, (int) &theReg) ==
         ERROR)
{
   logMsg("Write Register Failed for Slot #%d\n\n", 16AIOSlot,

            0, 0, 0, 0, 0);
}
```

4.7.5 START_DMA

The START_DMA function configures the 16AIO-88 DMA Registers for a DMA Transfer from the board, and then starts the transfer.

arg PARAMETER:

DMA_PARAM *

int DMAChannel - DMA Channel to perform transfer on. Must be one of the following:

- DMA_CHAN_0
- DMA_CHAN_1

ULONG ulDMAMode - Value to be written to the 16AIO-88 DMA Mode Register.

ULONG ulDMALocalAddress - Value to be written to the 16AIO-88 DMA Local Address Register. Data returned is little endian and may need to be byte/word swapped.

ULONG ulDMAByteCount - Value to be written to the 16AIO-88 DMA Byte Count Register.

ULONG ulDMADescriptorPtr - Value to be written to the 16AIO-88 DMA Descriptor Pointer Register.

ULONG ulDMAArbitration - Value to be written to the 16AIO-88 DMA Arbitration Register.

ULONG ulDMAThreshold - Value to be written to the 16AIO-88 DMA Threshold Register.

See the PLX-PCI PCI Bus Master Interface Data Sheet for a description of the DMA Registers.

DMA READ EXAMPLE:

```
#define     DWORD_COUNT 80
int         iIndex, FileDesc[2], 16AIOSlot = 1;
DMA_PARAM   DMAParameters;
ULONG       pulBuffer[DWORD_COUNT];
REG_PARAM   theReg;
ULONG       ulValue;

/* Scan input channels.
*/
if(ioctl(FileDesc[16AIOSlot], SCAN_INPUTS, 0) == ERROR)
{
   logMsg("Input Scan Failed\n\n", 0, 0, 0, 0, 0, 0);
}

/* Setup parameters to perform a DMA Read from the 16AIO-88 Board.
*/
DMAParameters.DMAChannel          = 0;
DMAParameters.ulDMAMode           = 0x943;
DMAParameters.ulDMALocalAddress  = (ULONG) pulBuffer;
DMAParameters.ulDMAByteCount     = DWORD_COUNT * 4;
DMAParameters.ulDMADescriptorPtr = 0xA;
DMAParameters.ulDMAArbitration   = 0;
DMAParameters.ulDMAThreshold     = 0;

if (ioctl(FileDesc[16AIOSlot], START_DMA, (int) &DMAParameters) ==
ERROR)
{
   logMsg("Start DMA Failed for Slot #%d\n\n", 16AIOSlot,
             0, 0, 0, 0, 0);
}

/* Wait for the DMA to Complete. */
theReg.pulValue      = &ulValue;
theReg.e16AIORegister   = DMA_CMD_STATUS;
do
{
   if (ioctl(FileDesc[16AIOSlot], READ_REGISTER, (int) &theReg) ==
ERROR)
   {
      logMsg("Read Register Failed for Slot #%d\n\n", 16AIOSlot,
                0, 0, 0, 0, 0);
      break;
   }
} while (! (ulValue & 0x10));

/* Clear the DMA channel 0/1 command/status register.
*/
ulValue = 0;
theReg.pulValue      = &ulValue;
theReg.e16AIORegister = DMA_CMD_STATUS;
```

```
if (ioctl(FileDesc[16AIOSlot], WRITE_REGISTER, (int) &theReg) ==
         ERROR)
{
         logMsg("Write Register Failed\n\n",
                0, 0, 0, 0, 0, 0);
}
```

4.7.6 REG_FOR_INT_NOTIFY

The REG_FOR_INT_NOTIFY function will register or unregister for notification that an interrupt has occurred on the 16AIO-88 Board. If this function is called with a pointer to a subroutine, that routine will be invoked when a 16AIO-88 Interrupt occurs. If a function is currently registered for interrupt notification and this function is called with a NULL pointer, the function will no longer be called when an interrupt occurs. The parameter sent to the notification routine will be the slot number of the 16AIO-88 Board that has interrupted and will be one of the following:

- 16AIO_PMC1
- 16AIO_PMC2

Note that the internal driver interrupt handler will clear interrupts after calling the user supplied ISR.

arg PARAMETER:

int (*intHandler)(int) - Pointer to a routine to handle the interrupt notification or a NULL Pointer if the caller wants to unregister for interrupt notification.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;

int intHandler(ULONG ulSlotNum)
{
   REG_PARAM   theReg;
   ULONG       ulValue;

/* execute interrupt control here */

   return (0);

}  /* intHandler */



/* Request notification on the user selected conditions. */
if (ioctl(FileDesc[16AIOSlot], REG_FOR_INT_NOTIFY, (int) intHandler)
         == ERROR)
{
```

```
            logMsg("Request Interrupt Notification Failed\n\n",0,0,0,0,0,0 );
        }
```

4.7.7 GET_DEVICE_ERROR

The GET_DEVICE_ERROR function will return the error that occurred on the last call to one of the 16AIO-88 Device Driver entry points. Whenever a driver function is called and it returns an error, this function may be called to determine the cause of the error.

arg PARAMETER:

int * - Pointer to the location of where the error code is to be written. It will be one of the following:

NO_ERR - No Error Occurred.

INVALID_PARAMETER_ERR - An Invalid Parameter was sent to driver.

RESOURCE_ERR - The driver could not obtain a resource (memory or semaphore) to perform its function.

BOARD_ACCESS_ERR - Failure occurred when the GS_16AIODrvInstall function fails when probing the 16AIO-88 card's Board Status Register.

DEVICE_ADD_ERROR - Failure occurred when the GS_16AIODrvInstall function fails when trying to add device to the VxWorks Operating System.

ALREADY_OPEN_ERROR - A call to the open driver access routine for a device that is already open.

INVALID_DRV_NUM_ERR - Returned from the GS_16AIODrvInstall function if an invalid driver number was obtained when trying to add the device driver to the VxWorks operating system. Also returned from the GS_16AIODrvRemove function if the driver failed to remove the device driver from the VxWorks operating system.

ALREADY_INSTALLED_ERR - Returned from the GS_16AIODrvInstall function if the driver has already been installed.

PCI_CONFIG_ERR - Returned from the GS_16AIODrvInstall function if a read or write of a PCI Configuration Register fails.

INVALID_BOARD_STATUS_ERR - Returned from the GS_16AIODrvInstall function if an invalid board status is read from the 16AIO-88 Board.

FIFO_BUFFER_ERR - If during a write() transaction the FIFO buffer is indicated to be full by the status of the buffer status register, the driver will return the number of bytes that could be written along with throwing this error condition.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int Status;

/* Send the Get Device Error Code Command for this channel   */
if (ioctl(FileDesc[16AIOSlot], GET_DEVICE_ERROR, (int) &Status) ==
ERROR)
{
   logMsg("Get Device Error Code Failed for Slot #%d\n\n",
          16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.8 READ_MODE_CONFIG

The READ_MODE_CONFIG function will configure the driver for the type of read() from the input FIFO to be performed. There are two types of reads. The first being referred to as SCAN_MODE where each sample is read out of the input FIFO one at a time and put into the user buffer given. The other type of read is referred to as DMA_MODE where the DMA capability of the board is taken advantage of.

arg PARAMETER:

int * - Pointer to one of the following values:

- SCAN_MODE
- DMA_MODE

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int iMode;

iMode = DMA_MODE;

if (ioctl(FileDesc[16AIOSlot], READ_MODE_CONFIG, (int) &iMode) ==
ERROR)
{
   logMsg("Read Mode Configuration Failed for Slot #%d\n\n",
   16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.9 INPUT_MODE_CONFIG

The INPUT_MODE_CONFIG function will arrange input channels into single-ended or differential mode, and set either burst or continuous scan.

arg PARAMETER:

int * - Pointer to one of the following values:

- SINGLE_CONTINUOUS
- SINGLE_BURST
- DIFF_CONTINUOUS
- DIFF_BURST

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int iMode;

iMode = DIFF_BURST;

if (ioctl(FileDesc[16AIOSlot], INPUT_MODE_CONFIG, (int) &iMode) ==
ERROR)
{
   logMsg("Input Configuration Failed for Slot #%d\n\n", 16AIOSlot,
   0, 0, 0, 0, 0);
}
```

4.7.10 INPUT_TEST

The INPUT_TEST function will perform a system level validation of operation precision. There are several tests, such as a loopback test, positive reference test, and zero input test. During the loopback test, one output channel is connected to the input channel 0. During the positive reference test, an internal voltage reference is connected to all input channels. The zero input test consists of all input channels being connected to the internal ground.

arg PARAMETER:

int * - Pointer to one of the following values:

- LOOPBACK
- POSITIVE_REF
- ZERO

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int Test;

Test = ZERO;

if (ioctl(FileDesc[16AIOSlot], INPUT_TEST, (int) &Test) == ERROR)
{
   logMsg("Input Test Failed for Slot #%d\n\n", 16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.11 LOOP_TEST_CHANNEL

The LOOP_TEST_CHANNEL function will set the output channel to be tested in the loopback selftest. This must take place before the loopback test is started.

arg PARAMETER:

int * - Pointer to one of the following values:

- CHANNEL0
- CHANNEL1
- CHANNEL2
- CHANNEL3
- CHANNEL4
- CHANNEL5
- CHANNEL6
- CHANNEL7

EXAMPLE:

```
int   FileDesc[2];
int   16AIOSlot = 1;
int   *chan;

chan = CHANNEL2;

if (ioctl(FileDesc[16AIOSlot], LOOP_TEST_CHANNEL,  (int) &chan) ==
ERROR)
{
   logMsg("Loopback Channel Selection Failed for Slot #%d\n\n",
          16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.12 CALIBRATION_MODE

The CALIBRATION_MODE function performs a calibration operation. There are various operations, such as loading the calibration DAC's from the EEPROM, autocalibration, and copying calibration values from EEPROM to the input buffer. There is also a default operation of no calibration activity. Refer to the PMC-16AIO-88 User's Manual for more information on these operations.

arg PARAMETER:

int * - Pointer to one of the following values:

- NO_CAL_ACTIVITY
- LOAD_DAC
- AUTO_CAL
- COPY_VALUE

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int Mode;

Mode = AUTO_CAL;

if (ioctl(FileDesc[16AIOSlot], CALIBRATION_MODE, (int) &Mode) ==
      ERROR)
{
   logMsg("Calibration Failed for Slot #%d\n\n", 16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.13 INT_SOURCE

The INT_SOURCE function will set the interrupt condition for a single local interrupt request.

arg PARAMETER:

int * - Pointer to one of the following values:

- IDLE
- CAL_COMPLETE
- IN_SCAN_COMPLETE
- INPUT_ALMOST_FULL
- OUTPUT_ALMOST_EMPTY
- INPUT_EMPTY
- OUTPUT_FULL

EXAMPLE:

```
int   FileDesc[2];
int   16AIOSlot = 1;
int   Source;

Source = IN_SCAN_COMPLETE;

if (ioctl(FileDesc[16AIOSlot], INT_SOURCE, (int) &Source) == ERROR)
{
   logMsg("Interrupt Selection Failed for Slot #%d\n\n", 16AIOSlot,
            0, 0, 0, 0, 0);
}
```

4.7.14 ENABLE_PCI_INTERRUPTS

The ENABLE_PCI_INTERRUPTS function enables the PCI interrupts in order to have a local interrupt request be generated.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int   FileDesc[2];
int   16AIOSlot = 1;

if (ioctl(FileDesc[16AIOSlot], ENABLE_PCI_INTERRUPTS, 0) == ERROR)
{
   logMsg("PCI Interrupt Enable Failed for Slot #%d\n\n",
            16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.15 DISABLE_PCI_INTERRUPTS

The DISABLE_PCI_INTERRUPTS function disables the PCI interrupts.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int   FileDesc[2];
int   16AIOSlot = 1;

if (ioctl(FileDesc[16AIOSlot], DISABLE_PCI_INTERRUPTS, 0) == ERROR)
{
   logMsg("PCI Interrupts Disable Failed for Slot #%d\n\n",
           16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.16 LAST_CHANNEL

The LAST_CHANNEL function will set the last channel in an output channel group. This function is associated with the rate controller. It is used to stop the collection of output values.

arg PARAMETER:

int * - Pointer to one of the following values:

- CHANNEL0
- CHANNEL1
- CHANNEL2
- CHANNEL3
- CHANNEL4
- CHANNEL5
- CHANNEL6
- CHANNEL7

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int Channel;

Channel = CHANNEL2;

if (ioctl(FileDesc[16AIOSlot], LAST_CHANNEL, (int)&Channel) ==
          ERROR)
{
   logMsg("Last Output Channel Selection Failed for Slot
            #%d\n\n", 16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.17 PROGRAM_RATE_GEN

The PROGRAM_RATE_GEN function will set the rate at which the input channels are scanned and the output channels are strobed. This function uses a user-specified divisor, iNrate. The rate generator calculates the clock frequency as:

Frequency (Hz) = 20,000,000 / iNrate

It is advised that the iNrate value remains more than 50h (80 decimal).

arg PARAMETER:

int * - Pointer to the integer used in calculation.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int iNrate;

iNrate = 0x0100;

/* Program Rate Generator. */
if (ioctl(FileDesc[16AIOSlot], PROGRAM_RATE_GEN, iNrate) == ERROR)
{
   logMsg("Program Rate Generator Failed\n\n", 0, 0, 0,
          0, 0, 0 );
}
```

4.7.18 ENABLE_RATE_GEN

The ENABLE_RATE_GEN function will enable and configure the rate generator to control the inputs, outputs, or both. The generator can act as a burst trigger, an output strobe, or a synchronizer of both inputs and outputs.

arg PARAMETER:

int * - Pointer to one of the following values:

- OUTPUT_RATE
- INPUT_RATE
- BOTH_IN_OUT_RATE

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int Rate;

Rate = BOTH_IN_OUT_RATE;

if (ioctl(FileDesc[16AIOSlot], ENABLE_RATE_GEN, (int)&Rate) ==
         ERROR)
{
   logMsg("Rate Generator Enable Failed for Slot #%d\n\n",
            16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.19 DISABLE_RATE_GEN

The DISABLE_RATE_GEN function disables the rate generator to control input scan, output strobe, or both rates.

arg PARAMETER:

int* - Pointer to one of the following values:

- OUTPUT_RATE
- INPUT_RATE
- BOTH_IN_OUT_RATE

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;
int Rate;

Rate = INPUT_RATE;

if (ioctl(FileDesc[16AIOSlot], DISABLE_RATE_GEN, (int)&Rate) ==
          ERROR)
{
   logMsg("Rate Generator Disable Failed for Slot #%d\n\n",
               16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.20 ENABLE_OUTPUT_STROBE

The ENABLE_OUTPUT_STROBE function will enable output strobing. Output values will be stored in a temporary buffer until an internal (software) strobe, an external (hardware) strobe, or a output strobe from the rate generator has occurred.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;


if (ioctl(FileDesc[16AIOSlot], ENABLE_OUTPUT_STROBE, 0) == ERROR)
{
   logMsg("Output Strobe Enable Failed for Slot #%d\n\n",
            16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.21 DISABLE_OUTPUT_STROBE

The DISABLE_OUTPUT_STROBE function will disable output strobing. While strobes are not allowed, output values are moved from the output buffer to the selected output channel without having to go through an intermediate buffer.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;


if (ioctl(FileDesc[16AIOSlot], DISABLE_OUTPUT_STROBE, 0) == ERROR)
{
   logMsg("Output Strobe Disable Failed for Slot #%d\n\n",
            16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.22 CLEAR_INT_REQUEST

The CLEAR_INT_REQUEST function clears the interrupt request flag after an interrupt has occurred.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;


if (ioctl(FileDesc[16AIOSlot], CLEAR_INT_REQUEST, 0) == ERROR)
{
   logMsg("Clear Interrupt Request Flag Failed for Slot #%d\n\n",
                      16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.23 STROBE_OUTPUTS

The STROBE_OUTPUTS function causes a strobe of all output channels. This, in turn, forces the output values from the intermediate buffer to the appropriate output channels.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;


if (ioctl(FileDesc[16AIOSlot], STROBE_OUTPUTS, 0) == ERROR)
{
   logMsg("Output Strobe Failed for Slot #%d\n\n",
      16AIOSlot, 0, 0, 0, 0, 0);
}
```

4.7.24 SCAN_INPUTS

The SCAN_INPUTS function triggers a burst scan of all input channels.

arg PARAMETER:

Not Used.

EXAMPLE:

```
int FileDesc[2];
int 16AIOSlot = 1;


if (ioctl(FileDesc[16AIOSlot], SCAN_INPUTS, 0) == ERROR)
{
   logMsg("Input Scan Failed for Slot #%d\n\n",
   16AIOSlot, 0, 0, 0, 0, 0);
}
```