PCM-MIO Linux
Device Driver Package

1 Introduction

1.0 The PCM-MIO Device Driver Package consists of a Linux Device Driver, an application programming interface library, and example application programs.

1.1 The driver was built and tested on Linux based Ubuntu 12.04 and 16.04 LTS distributions. These releases correspond to Linux kernel versions 3.2 and 4.4.

1.1 The driver supports the WinSystems' PCM-MIO family of boards, including Analog to Digital (ADC), Digital to Analog (DAC) and Digital I/O (DIO).

1.2 This driver is provided 'as-is' and no warranty as to usability or fitness of purpose is claimed.

1.3 WinSystems does not provide support for the modification of this driver. Bug reports may be sent to linux_drivers@winsystems.com.

1.4 This driver is provided under the terms of the GNU General Public License (GPL).

2 Installation and Build

2.0 The driver and the sample applications are provided in source code form as a compressed tar ball.

2.1 It will be necessary to become the root user to build the driver and the device node.

2.2 The build-essential package may need to be installed before building the software. This is a reference for all the packages needed to compile a package. It generally includes the compilers and libraries and some other utilities.

2.3 The MAJOR number for this device is allocated dynamically. A static number can be assigned by editing the *pcmmio_ws_major* variable at the beginning of ***pcmmio_ws.c***.

2.4 To create the device driver Loadable Kernel Module and the sample applications in a command shell execute: ***make all***. The device driver Loadable Kernel Module ***pcmmio.ko*** is created and moved to the appropriate kernel driver directory. The file access permissions are set to allow access by all users and groups; they may be changed manually as desired. The nine sample programs are also built.

make install will install the kernel driver to a kernel directory and create dependencies.
make uninstall will remove the kernel driver from the kernel directory.
make clean will remove objects created by the build.
make spotless will forcibly remove all artifacts of the build.
make <appname> will create the selected application.

2.5 The device driver can be loaded with the provided initialization script ***pcmmio_load*** or manually. In either case *modprobe* is used to load the driver. Since the PCM-MIO board is not plug-n-play and I/O probing could be problematic, it is required to specify the base address of the device on the command line when loading the driver. A sample command line loading might look like this:

```
modprobe pcmmio io=0x300 irq=5
```

2.6 This would install the driver with a base port of 300 hex using IRQ5. Interrupts are not required for driver loading but none of the event sense or wait_xxx_int functions will be usable. The internal routines for normal DIO bit operations and ADC and DAC functions do not require interrupts.

2.7 The required *pcmmio* device nodes are also created in /dev when the ***pcmmio_load*** script is executed. This file should be modified according to the devices installed. To load the driver automatically at boot, add the ***pcmmio_load*** script to the /etc/rc.local file.

3 Driver Usage

3.0 The PCM-MIO is accessed in hardware as a byte-oriented device. Therefore, the driver is implemented as a character device. Using file I/O read, write, and seek operations although crudely implemented for compatibility will NOT give the desired results. The driver was designed for maximum flexibility using *ioctl* as its exclusive programming interface.

3.1 The file *mio_io.o* implements the *ioctl* interface and presents the application with a set of standard C functions that may be called directly from the application without any further need for dealing with or understanding of how to access the driver using *ioctl*. An application must merely include *mio_io.h* and link to *mio_io.o* to provide this simple interface.

4 'C' Language Library

4.0 All of functions from *mio_io.o* are standard 'C' language functions. There are also two global variables available to support error detection and handling. They are defined in *mio_io.h* as:

```
extern int mio_error_code;
extern char mio_error_string[128];
```

The first is an integer holding the result code from that last function call. A non-zero value indicates an error had occurred. The file *mio_io.h* defines all error codes. In addition to the error

code, an error string, *MIO_ERROR_STRING*, is also generated when an error occurs. This string gives the function name and the error type that occurred.

FUNCTION LIST

The supported function from *mio_io.o* will be broken down into four categories, three for the three distinct functional modules on the board and a fourth for common use. The following list is a complete list of functions sorted by category. Following the list each function will be described in more detail by category.

==========
Analog Input Functions
==========
adc_start_conversion
adc_get_channel_voltage
adc_convert_all_channels
adc_convert_to_volts
adc_convert_single_repeated
adc_buffered_channel_conversions
adc_wait_ready
adc_write_command
adc_read_status
adc_set_channel_mode
adc_read_conversion_data
adc_auto_get_channel_voltage
adc_disable_interrupt
adc_enable_interrupt
adc_wait_int

==========
Analog Output Functions
==========
dac_set_span
dac_wait_ready
dac_set_output
dac_set_voltage
dac_write_command
dac_buffered_output
dac_write_data
dac_read_status
dac_disable_interrupt
dac_enable_interrupt
dac_wait_int

==========
DIO Functions
==========
dio_reset_device
dio_read_bit
dio_write_bit
dio_set_bit
dio_clr_bit
dio_read_byte
dio_write_byte
dio_enab_bit_int
dio_disab_bit_int
dio_clr_int
dio_get_int
dio_wait_int

==========
MIO Support Functions
==========
mio_read_reg
mio_write_reg

ANALOG INPUT FUNCTIONS

adc_start_conversion - Start a conversion on a channel

Prototype	void adc_start_conversion(int dev_num, int channel)
Arguments	dev_num - The device to be accessed (0-3) channel - The channel number (0-15)
Return	Check mio_error_code: 0 = conversion completed without error. Any other value indicates an error occurred.
Description	This function starts an A/D conversion on the specified channel number and returns immediately.

adc_get_channel_voltage - Get Channel Voltage

Prototype	float adc_get_channel_voltage(int dev_num, int channel)
Arguments	dev_num - The device to be accessed (0-3) channel - The channel to be converted (0-15)
Return	Floating point value provides voltage at ADC channel Check mio_error_code: 0 = Conversion completed without error. Any other value indicates an error occurred and returned value is invalid
Description	This function like *adc_auto_get_channel_voltage* returns the voltage on the specified channel's input pin. The value returned is only valid for the range specified with a preceding *adc_set_channel_mode*. Unlike the auto-ranging version, this function can be used with differential input signals.

adc_convert_all_channels - Convert all channels

Prototype	void adc_convert_all_channels(int dev_num, unsigned short *buffer)
Arguments	dev_num - The device to be accessed (0-3) buffer - Pointer to a 16 element unsigned short array for return of values.
Return	Check mio_error_code: 0 = Conversions completed without error. Any other value indicates an error occurred.
Description	This function is used to snapshot all 16 channels as quickly as possible. The results are stored in a 16-element array provided by the calling program. The values provided are 16-bits

(signed/unsigned) in length for each element. To convert the values to voltage, use the *adc_convert_to_volts* function.

adc_convert_to_volts – Convert a raw ADC value to voltage

Prototype	float adc_convert_to_volts(int dev_num, int channel, unsigned short value)
Arguments	dev_num - The device to be accessed (0-3) channel – The channel number (0-15) from which value was read value – The 16-bit raw converter value
Return	Floating point value provides voltage at ADC channel Check mio_error_code: 0 = Conversion completed without error. Any other value indicates an error occurred and returned value is invalid
Description	This function does nothing with the MIO hardware and makes no call to the driver. It is simply a math routine which according to the current mode set by a channel during *set_channel_mode* and the supplied value calculates and returns the current voltage as a floating-point value.

adc_convert_single_repeated - Multiple conversions on a single channel

Prototype	void adc_convert_single_repeated(int dev_num, int channel, unsigned short count, unsigned short *buffer)
Arguments	dev_num - The device to be accessed (0-3) channel - The channel number (0-15) count - The number of desired conversions. buffer - A pointer to an array of count elements to hold the results
Return	Check mio_error_code: 0 = Conversions completed without error. Any other value indicates an error occurred.
Description	The function allows for repetitive high-speed conversions on a single channel. The array pointer buffer must be of sufficient size to hold the results, i.e. count elements long. The absolute maximum count is 65536. Counts from 2 to 16384 are more realistic. The values are returned in the array in 16-bit integers which are signed or unsigned dependent upon the channel mode used. Vales may be converted to volts using the *adc_convert_to_volts* function.

adc_buffered_channel_conversions - Programmable conversion sequence

Prototype	void adc_buffered_channel_conversions(int dev_num, unsigned

char *input_channel_buffer, unsigned short *buffer)

Arguments dev_num - The device to be accessed (0-3)
input_channel_buffer - Pointer to an array of channel numbers to be converted. Terminated with 0ffH.
buffer - Pointer to an array of 16-bit values to receive the results

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function allows for high speed multiple channel conversions. The input is an array of channel numbers in any order repetitive or not, as desired. The function will start each conversion immediately after completing the previous one without further application intervention. The list is terminated with a 0ffH value. The buffer argument should point to an adequately sized array to hold all of the specified conversion results.

adc_wait_ready - Wait for conversion complete

Prototype void adc_wait_ready(int dev_num, int channel)

Arguments dev_num - The device to be accessed (0-3)
channel – The channel number (0-15)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function waits for conversions to complete. It reads the status port until the conversion completes or a timeout error occurs.

adc_write_command – Write command byte to the ADC controller

Prototype void adc_write_command(int dev_num, int adc_num, unsigned char value)

Arguments dev_num - The device to be accessed (0-3)
adc_num – The ADC converter number (0-1)
value – 8-bit command value for ADC.

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function is used internally to build and send proper command bytes to the specified ADC controller. It has no practical use in applications code. *adc_start_conversion calls* this function using values specified in the *adc_set_channel_mode* call to build the command byte.

adc_read_status – Read the ADC status register

Prototype unsigned char adc_read_status(int dev_num, int adc_num)

Arguments dev_num - The device to be accessed (0-3)
adc_num – The ADC controller number (0-1)

Return 8-Bit unsigned status register content

Check mio_error_code:
0 = Conversion completed without error.
Any other value indicates an error occurred and returned value is invalid

Description The function is used internally by a number of other ADC functions. It is normally not used by application code. Refer to the PCM-MIO operations manual for bit definitions for this register.

adc_set_channel_mode - Set Channel input mode and range

Prototype void adc_set_channel_mode(int dev_num, int channel, int input_mode, int duplex, int range)

Arguments dev_num - The device to be accessed (0-3)
channel - The channel number to set (0-15)
input_mode - Input type
ADC_SINGLE_ENDED
ADC_DIFFERENTIAL
duplex - The swing of the input voltage
ADC_UNIPOLAR
ADC_BIPOLAR
range - The input voltage top end
ADC_TOP_5V
ADC_TOP_10V

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function is used to set the input mode for a given channel. Once a channel's mode has been set it will remain until changed or until the application exits. The mode must be set before making any conversion calls except for a*dc_auto_get_channel_voltage* which will change the mode to the one most appropriate for the current input.

adc_read_conversion_data – Read the ADC output register

Prototype unsigned short adc_read_conversion_data(int dev_num, int channel)

Arguments	dev_num - The device to be accessed (0-3) channel – The channel number (0-15)
Return	A raw 16-bit value from the ADC converter's output register Check mio_error_code: 0 = Conversion completed without error. Any other value indicates an error occurred and returned value is invalid
Description	The function reads out the data from the second to last conversion. It is important to recognize that with each conversion the converter delivers the data from the previous conversion meaning that if a current reading is required it's necessary to do two conversions. Look at the source code for the sample programs to see how this is accomplished.

adc_auto_get_channel_voltage - Get Channel voltage auto ranging

Prototype	float adc_auto_get_channel_voltage(int dev_num, int channel)
Arguments	dev_num - The device to be accessed (0-3) channel - The channel to be converted (0-15)
Return	Floating point value provides voltage at ADC channel Check mio_error_code: 0 = Conversion completed without error. Any other value indicates an error occurred and returned value is invalid
Description	This function returns the voltage on the current input channel pin. It works for single-ended inputs only. It could make as many as four conversion requests before returning a final value. This function is the simplest interface to the hardware.

adc_disable_interrupt – Disable ADC interrupt generation

Prototype	void adc_disable_interrupt(int dev_num, int adc_num)
Arguments	dev_num - The device to be accessed (0-3) adc_num – ADC converter number (0-1)
Return	Check mio_error_code: 0 = conversions completed without error. Any other value indicates an error occurred.
Description	This function turns off the interrupt generation capability at the specified ADC controller. Interrupt processing consumes processor time so if ADC interrupt handling is not being used the interrupts should be disabled. The internal ADC functions do not use interrupts for their functionality.

adc_enable_interrupt – Enable ADC interrupt generation

Prototype	void adc_enable_interrupt(int dev_num, int adc_num)
Arguments	dev_num - The device to be accessed (0-3) adc_num – The ADC converter number (0-1)
Return	Check mio_error_code: 0 = conversions completed without error. Any other value indicates an error occurred.
Description	This function enables hardware interrupts at the specified ADC controller. An error is returned if there is no IRQ assigned to the driver. The default ADC handler simply clears the interrupt, releases any waiting threads, and returns. All of the internal ADC routines do not use interrupts. Interrupts should be enabled only if w*ait_adc_int* will be used by the application.

adc_wait_int - Wait for an ADC interrupt to occur

Prototype	void wait_adc_int(int dev_num, int adc_num)
Arguments	dev_num - The device to be accessed (0-3) adc_num - The ADC converter number (0-1)
Return	Check mio_error_code: 0 = conversions completed without error. Any other value indicates an error occurred.
Description	This function waits within the driver to be released on the occurrence of an interrupt on the specified ADC controller. The default handler clears the interrupt and releases waiting threads only.

ANALOG INPUT SAMPLES

Also included with the library are four ADC sample application programs which utilize the functions in the library. They range in complexity from very simple to much more complex. There is extensive commenting within the sample applications to facilitate understanding of their usage.

GETVOLT.EXE

The file ***getvolt.c*** is the source for sample application number one. This sample is shown first because it utilizes the highest-level function in the library and for a large number of users will be the only function required from the library.

This application is supplied in its source form, ***getvolt.c***. It is invoked at the command line as:

```
./getvolt d c
```

Where d is the device number from 0 to 3 and c is the channel number to convert from 0 to 15. The voltage on that channel is then displayed. Internally the code calls the high-level function.

```
float adc_auto_get_channel_voltage(dev_num, channel);
```

This function is an auto ranging function in that it starts out by making a measurement in a ±10V scale, checking the result to see if a more precise value could be obtained by changing scales and if so, making another measurement at the more precise range and returning a floating-point voltage to the caller. If absolute speed is not important this is the easiest way to make a reading on a channel.

NOTE: This function was coded for single-ended usage only. Differential inputs would need to set a mode and scale and use the non-auto ranging function *adc_get_channel_voltage*.

GETALL.EXE

The second application uses the *adc_convert_all_channels* function call to get a snapshot of all of the 16 channels with one call. Unlike *adc_auto_get_channel_voltage* and *adc_get_channel_voltage*, the data is returned not in floating point but in an array of raw 16-bit values ranging from 0000H to FFFFH. The program extracts the values one by one from the array, converts them to floating point, and then displays the results. Also note that since this is not an auto ranging function it is necessary to call *adc_set_conversion_mode* for each channel to tell the software the input mode, and range desired. In this sample all channels were set to the same mode but there is no requirement that they all be the same.

REPEAT.EXE

The third application demonstrates the usage of the *adc_convert_single_repeated* function call. This call is prototyped as:

```
void adc_convert_single_repeated(int dev_num, int channel, unsigned count,
unsigned *buffer);
```

The *dev_num* argument specifies the device to be accessed. The *channel* number argument is fairly obvious. The *count* value is the number of conversions we want to take on this channel and *buffer* is a pointer to an array large enough to hold the number of samples requested. Upon return the *buffer* array will hold *count* number of conversions which are once again provided in 16-bit values. This sample requests 2000 samples at a time. Once the data is back, it is element by element converted to floating point, displayed and compared against previous minimum and maximum values. Pressing the 'C' key clears the counts and min/max values and pressing 'N' steps to the next channel. Any other key exits.

BUFFERED.EXE

The fourth and final sample uses the *adc_buffered_channel_conversions* call to program a series of high-speed conversions with the results being stored in a specified buffer. The function prototype is:

```
void adc_buffered_channel_conversions(int dev_num, unsigned char
*input_channel_buffer, unsigned short *buffer);
```

The *input_channel_buffer* is an array of channel numbers built by the user as a to-do list of conversions. It is terminated with a 0FFH value. The *buffer* array must be large enough to hold the requested number of conversions. In our sample we load the *input_channel_buffer* with zero 500 times, one 500 times, two 500 times, and three 500 times for a total of 2000 conversions. Actually, our *input_channel_buffer* is 2001 characters long to make space for the terminating 0ffh character. The output buffer is 2000 unsigned short integers long which will hold the results. Upon return from this function we have 500 conversions each on the first 4 channels. The program sorts them out, converts them to voltages and displays the values. Any key press exits the program.

ANALOG OUTPUT FUNCTIONS

dac_set_span – Set a DAC channel's output range

Prototype: void dac_set_span(int dev_num, int channel, unsigned char span_value)

Arguments: dev_num - The device to be accessed (0-3)
channel – The DAC channel number (0-7)
span_value – An 8-bit argument of one of the following:

DAC_SPAN_UNI5	0 to 5 V scale, Unipolar
DAC_SPAN_UNI10	0 to 10 V scale, Unipolar
DAC_SPAN_BI5	±5 V scale, Bipolar
DAC_SPAN_BI10	±10 V scale, Bipolar
DAC_SPAN_BI2	±2.5 V scale, Bipolar
DAC_SPAN_BI7	-2.5 to +7.5 V scale, Bipolar

Return: Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description: This function sets the output range on the specified channel. It will affect the current output level by using the current DAC value in the new scale range.

dac_wait_ready - Wait for DAC Controller to be ready

Prototype: void dac_wait_ready(int dev_num, int channel)

Arguments: dev_num - The device to be accessed (0-3)
channel – The DAC channel number (0-7)

Return: Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description: This function is used to wait for DAC output shifts to complete. It reads the status port until the controller ready or a timeout error occurs.

dac_set_output – Output a value to a DAC channel

Prototype: void dac_set_output(int dev_num, int channel, unsigned short dac_value)

Arguments: dev_num - The device to be accessed (0-3)
channel - DAC channel number (0-8)
dac_value – A 16-bit value to be sent to the DAC

Return: Check mio_error_code:
0 = conversions completed without error.

Any other value indicates an error occurred.

Description This function sends the specified 16-bit value to the DAC channel number specified. The voltage this value represents is dependent upon the range set up with a previous call to *set_dac_span*. This function is usually not called by application code which may more easily use the higher-level function *set_dac_voltage*.

dac_set_voltage – Set a DAC channel output to a voltage

Prototype void dac_set_voltage(int dev_num, int channel, float voltage)

Arguments dev_num - The device to be accessed (0-3)
channel – The DAC channel number (0-7)
voltage - The desired output voltage (-10.0 to +10.0)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function sets the specified DAC output channel to the requested voltage. The set_dac_span call is made first to give the most precise range available for the requested voltage. NOTE: It is possible to get a spike (up or down) in voltage as the range value is programmed and until the new value is output. If this is of critical concern it will be necessary to set up the range at an appropriate time, leave it as-is, and output values directly using *set_dac_output*.

dac_write_command – Write command byte to DAC controller

Prototype void dac_write_command(int dev_num, int dac_num, unsigned char value)

Arguments dev_num - The device to be accessed (0-3)
dac_num – The DAC controller number (0-1)
value – The 8-bit command value

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function is used internally to issue commands to the DAC controller. The command bytes are makeup of commands, channels, and command parameters. Refer to the Linear Technology's DAC datasheet for more details. Applications should never need to access this function directly

dac_buffered_output – Send programmed values to the DAC(S)

Prototype	void dac_buffered_output(int dev_num, unsigned char *cmd_buff, unsigned short *data_buff)
Arguments	dev_num - The device to be accessed (0-3) cmd_buff – Pointer to an 0xff terminated array of channel numbers data_buff – Pointer to an equal element array of data values for the DAC(S)
Return	Check mio_error_code: 0 = conversions completed without error. Any other value indicates an error occurred.
Description	This function reads the *cmd_buff* array, which holds channel numbers, until a 0xff character is read. For each element in the *cmd_buff* array the corresponding element in the *data_buff* array is read and sent to the corresponding DAC channel. The function returns when all values have been sent or upon an error condition.

dac_write_data – Write Data to DAC controller

Prototype	void dac_write_data(int dev_num, int dac_num, unsigned value)
Arguments	dev_num - The device to be accessed (0-3) dac_num – The DAC controller number (0-1) value – The 16-bit data value to be sent to the DAC controller
Return	Check mio_error_code: 0 = conversions completed without error. Any other value indicates an error occurred.
Description	This function is used internally to pass the 16-bit data value to the controller. This is NOT sufficient to update a DAC output voltage. The data must be followed by a command indicating, span, channel, etc. This function should never be needed by applications code.

dac_read_status – Read the DAC status register

Prototype	unsigned char dac_read_status(int dev_num, int dac_num)
Arguments	dev_num - The device to be accessed (0-3) dac_num – The DAC controller number (0-1)
Return	8-Bit unsigned status register content. Check mio_error_code: 0 = Conversion completed without error. Any other value indicates an error occurred and returned value is invalid

Description The function is used internally by a number of other DAC functions. It is normally not used by application code. Refer to the PCM-MIO operations manual for bit definitions for this register.

dac_disable_interrupt – Disable DAC interrupt generation

Prototype void dac_disable_interrupt(int dev_num, int dac_num)

Arguments dev_num - The device to be accessed (0-3)
dac_num – DAC controller number (0-1)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function turns off the interrupt generation capability at the specified DAC controller. Interrupt processing consumes processor cycles so if the DAC function *wait_dac_int* is not being used the interrupt should be disabled. The internal DAC functions do not use interrupts for their functionality.

dac_enable_interrupt – Enable DAC interrupt generation

Prototype void dac_enable_interrupt(int dev_num, int dac_num)

Arguments dev_num - The device to be accessed (0-3)
dac_num – The DAC controller number (0-1)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function enables hardware interrupts at the specified DAC controller. An error is returned if there is no IRQ assigned to the driver. The default DAC handler simply clears the interrupt and returns. It serves no purpose unless the *wait_dac_int* function is being used for interrupt handling. All of the internal DAC routines do not use interrupts. Interrupts should be enabled only if the *wait_dac_int* function will be used by the application.

dac_wait_int - Wait for DAC interrupt to occur

Prototype void dac_wait_int(int dev_num, int dac_num)

Arguments dev_num - The device to be accessed (0-3)
dac_num - The DAC converter number (0-1)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function waits within the driver to be released on the

occurrence of an interrupt on the specified DAC controller. The default handler clears the interrupt and releases waiting threads only.

ANALOG OUTPUT SAMPLES

Also included with the library are two DAC sample application programs which utilize the functions in the library. There is extensive commenting within the sample applications to facilitate understanding of their usage.

DACOUT.EXE

The file ***dacout.c*** is the source for sample application number one. This sample is shown first because it utilizes the highest-level function in the library and for a large number of users will be the only DAC function required from the library.

This application is supplied in its source form ***dacout.c***. It is invoked at the command line as:

```
./dacout dev_num channel voltage
```

Where d is the device number from 0 to 3 and channel is a value from 0 to 7 indicating the DAC channel number to update. The voltage argument can be from -10.0 Volts to +10.0 volts. The specified voltage is output on the desired channel. Within ***dacout.c*** the code calls the high-level function.

```
dac_set_voltage(dev_num, channel, voltage);
```

This function is an auto ranging function, in that it examines the voltage parameter, and chooses an output range that will give the most precise output and then sets the output voltage as specified. Using this function is the easiest way to update the voltage on a channel.

DACBUFF.EXE

The file ***dacbuff.c*** is the source code for DAC sample application number two. This application revolves around use of the *buffered_dac_output* function call. It is run at the command line and there is no screen output while running. Pressing any key will exit the program. This program fills two arrays, the first with channel numbers and the second with values for the corresponding channel indices. In this program only, channel 0 is used and the voltage steps from -10V to +10V in 4 count increments. Use an oscilloscope on channel 0 of the DAC output connector to view the results.

DIGITAL I/O (DIO) FUNCTIONS

DIO functions note: The registers and the actual I/O pins on the chip are inverted from each other. The DIO functions refer to the registers to avoid confusion when programming, but it's important to realize that setting a bit causes the actual output pin to go low and clearing a bit releases the output to be pulled high by the onboard pull-up resisters. Also note that bits must be cleared in order to use them as inputs.

dio_reset_device – Reset the DIO device to a known state.

Prototype	void dio_reset_device(int dev_num)
Arguments	dev_num - The device to be accessed (0-3)
Return	Check mio_error_code: 0 = conversions completed without error. Any other value indicates an error occurred.
Description	This function resets the specified DIO device. All bits are cleared and all interrupts are disabled.

dio_read_bit – Read a DIO bit value

Prototype	int dio_read_bit(int dev_num, int bit_number)
Arguments	dev_num - The device to be accessed (0-3) bit_number – The bit number (1-48)
Return	0 = Bit register is 0 1 = Bit register is 1 Check mio_error_code: 0 = bit read completed without error. Any other value indicates an error occurred and returned bit value is invalid
Description	This function is used to either read an input bit or to read back the state of an output. Again, note the inversion that takes place i.e. if an input pin is pulled low it will reflect as a 1 when the register bit is read.

dio_write_bit – Write a DIO register bit

Prototype	void dio_write_bit(int dev_num, int bit_number, int val)
Arguments	dev_num - The device to be accessed (0-3) bit_number – The bit number (1-48) val – The desired bit value (0 or 1)
Return	Check mio_error_code: 0 = conversions completed without error.

Any other value indicates an error occurred.

Description This function provides an alternate to the *dio_set_bit* and the *dio_clr_bit* functions. Writing a 1 is the same as *dio_set_bit* and writing a 0 is the same as *dio_clr_bit*. Refer to those two functions for additional details.

dio_set_bit – Set DIO register bit

Prototype void dio_set_bit(int dev_num, int bit_number)

Arguments dev_num - The device to be accessed (0-3)
bit_number – The bit number (1-48)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function sets the specified bit in the appropriate dio output register. Because of inversion, setting a bit causes the output pin to go low. A bit cannot be used for input when set. Use *dio_clr_bit* to enable a pin for input.

dio_clr_bit – Clear a DIO register bit

Prototype void dio_clr_bit(int dev_num, int bit_number)

Arguments dev_num - The device to be accessed (0-3)
bit_number – The bit number (1-48)

Return Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description This function clears the specified bit in the DIO data register for that bit. This causes the output pin to go high.

dio_read_byte – Read an 8-Bit DIO register

Prototype unsigned char dio_read_byte(int dev_num, int offset)

Arguments dev_num - The device to be accessed (0-3)
offset – The DIO register number (0-10)

Return The 8-bit register contents

Check mio_error_code:
0 = bit read completed without error.
Any other value indicates an error occurred and returned byte value is invalid

Description This function allows direct reading of any of the 10 DIO data and control registers. This function is used internally and its use except

for reading the first 6 ports (The actual data ports) is highly discouraged. Refer to the PCM-MIO operations manual for the DIO register and bit definitions.

dio_write_byte – Write a byte to a DIO register

Prototype: void dio_write_byte(int dev_num, int offset, unsigned char value)

Arguments:
dev_num - The device to be accessed (0-3)
offset – DIO register number (0-10)
value – An 8-bit value to write to the register

Return:
Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description: This function allows write access to any of the 10 control and data registers of the DIO section of the board. This function is used internally by the other DIO functions. Its use by applications is highly discouraged and may result in incorrect operation of other functions if used outside of the driver environment.

dio_enab_bit_int – Enable event sense interrupts on a bit

Prototype: void dio_enab_bit_int(int dev_num, int bit_number, int polarity)

Arguments:
dev_num - The device to be accessed (0-3)
bit_number – The bit number (1-24)
polarity – The specified interrupt polarity:
RISING
FALLING

Return:
Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description: This function enables event sense interrupts on a specific bit with the specified polarity. **NOTE**: The polarity argument in this case is from the PIN perspective. Specifying RISING for polarity will generate a bit interrupt when the voltage on the input pin RISES from a low to a high. Actual interrupts will not be generated by the dio section unless a call to *enable_dio_interrupt* has been made. If these two calls are not made, it's still possible to poll for events using *dio_get_int* after the *dio_enable_bit_int* call.

dio_disab_bit_int – Disable event sense interrupts on a bit

Prototype: void dio_disab_bit_int(int dev_num, int bit_number)

Arguments: dev_num - The device to be accessed (0-3)

bit_number – The bit number (1-24)

Return — Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description — This function disables the event sense interrupt generation on the specified bit.

dio_clr_int - Clear a pending event sense interrupt

Prototype — void dio_clr_int(int dev_num, int bit_number)

Arguments — dev_num - The device to be accessed (0-3)
bit_number – The bit number (1-24)

Return — Check mio_error_code:
0 = conversions completed without error.
Any other value indicates an error occurred.

Description — This function clears a pending interrupt on a *bit_number* that was obtained from the *dio_get_int* function call. A bit interrupt that is not cleared cannot generate additional interrupts. In the Linux driver once enabled, DIO sense interrupts are intercepted, buffered, and cleared by the driver. It is only when polling for transition events should application code need to call this function.

dio_get_int – Get highest priority event sense interrupt pending

Prototype — int dio_get_int(int dev_num)

Arguments — dev_num - The device to be accessed (0-3)

Return — 0 = No interrupt pending
1-24 – The bit number of the highest priority pending interrupt

Check mio_error_code:
0 = bit read completed without error.
Any other value indicates an error occurred and returned interrupt value is invalid

Description — This function queries the kernel driver for a buffered DIO event sense interrupt. If the driver has any buffered it delivers the number of the oldest one in the queue. If there is nothing in the buffer, the driver scans the hardware checking for a transition sense event. This allows *dio_get_int* to be used with both interrupt processing enabled or in a polled mode. A return of 0 indicates that there is no event sense pending. In polled mode, events are prioritized such that if multiple events are pending the lowest bit number with a pending transition event will be returned. In either polled, or interrupt mode, handler code should repeatedly call this function

until a zero is returned so that all events are handled. In polled mode *dio_clr_int* should be called for each pending event.

dio_wait_int - Wait for DIO event sense interrupt to occur

Prototype	int dio_wait_int(int dev_num)
Arguments	dev_num - The device to be accessed (0-3)
Return	0 = No Interrupt occurred. Thread signaled by system. 1 = An error occurred. Check *mio_error_code.* Check mio_error_code: 0 = bit read completed without error. Any other value indicates an error occurred and returned interrupt value is invalid
Description	This function waits within the driver to be released on the occurrence of an interrupt on the DIO lines. The default handler buffers, and clears, the interrupt, and releases waiting threads. This function will not return an error when operating in polled mode, but it will wait forever or until the parent process or the system terminates it.

DIO SAMPLE PROGRAMS

Also included with the library are two DIO sample application programs which utilize the functions in the library. There is extensive commenting within the sample applications to facilitate understanding of their usage.

FLASH.EXE

The file ***flash.c*** is the source for sample application number one. This sample uses the *dio_set_bit* and the *dio_clr_bit* functions to successively flash each bit low and then high. The output can be examined with an oscilloscope or with LEDs. There is no screen display while running. Pressing any key exits the program.

DIOTEST.EXE

The file ***diotest.c*** is the source code for the second sample application. This sample program uses the *write_dio_byte* and *dio_write_bit* functions to alter the contents of a specified digital port. The register contents are displayed after each alteration for verification.

POLL.EXE

The file ***poll.c*** is the source code for the third sample application. This sample program enables bit sense interrupts on the first 24 lines. It also enables dio board interrupts and creates a concurrent thread to receive the interrupt notification. In this simple demonstration the event thread simply counts the interrupts and then goes back to waiting for more events. Pressing any key will exit the program. Examining the source code will provide more details.

MIO SUPPORT FUNCTIONS

MIO functions note: All of these functions are used internally by the support library *mio_io.o*. They are documented here for completeness and for the very rare occurrence where access to the low-level functions may be required.

mio_read_reg - Read MIO register

Prototype	unsigned char mio_read_reg(int dev_num, int offset)
Arguments	dev_num - The device to be accessed (0-3) offset – MIO register number (0 – 26)
Return	8-bit register contents Valid only if *mio_error_code* == 0
Description	This function reads any of the 27 registers within the PCM-MIO board. Refer to the PCM-MIO operations manual for register and bit definitions.

mio_write_reg – Write to MIO register

Prototype	void mio_write_reg(int dev_num, int offset, unsigned char value)
Arguments	dev_num - The device to be accessed (0-3) offset - MIO register number (0-26) value - 8-bit value to write
Return	0 = No error occurred 1 = An error occurred. Check *mio_error_code*.
Description	This function allows write access to all 27 MIO registers. This function is extremely powerful and its careless use can result in system lockups, crashes, or incorrect operation of the various MIO support functions. Refer to the PCM-MIO operations manual for register and bit definitions.