PCM-MIO-G Device
Windows Device Driver Package

1 Introduction

1.1 The PCM-MIO WDM Driver package is designed for and has been tested with Microsoft Windows XP/XP Pro/XP Embedded and Windows 7/7 Embedded. It will **NOT** work with Windows 3.X, Windows 9X, or Windows NT.

1.2 The driver package supports two Linear Technology's 185X Analog to Digital Converters (ADC), two Linear Technology's LTC1824 Digital to Analog Converters (DAC) as well as a WinSystems' WS16C48 with 48 digital I/O lines.

2 Installation

2.1 The driver and support files are supplied in a zip file. Installation is accomplished via the "Add New Hardware" applet in the Windows control panel. Select "Have Disk" and navigate to the floppy drive containing the driver files. Once selected, the Windows installer will copy the *PCMMIO.SYS* file and the *MIO_IO.DLL* files to the appropriate directories in the Windows installation.

2.2 The default hardware configuration for installation is I/O port 300H and IRQ5. The board must be configured according to the hardware manual and the Windows resource configuration must match the hardware jumper settings. The driver I/O port setting can be changed using the Device Manager under the System icon in the Windows Control Panel. An IRQ setting is not required for driver functionality and logical configuration number 3 allows for such an installation. None of the event sense notification features will be available when no IRQ is assigned.

2.3 The example programs, the header file, and library file are not copied to the Windows system automatically at driver installation. These files must be manually copied, as desired, to a test or development system or folder. It is recommended that the binary executable files be copied to the target system in order to test driver installation and functionality. Usage of these example programs are described later in this document. There are a couple of MFC and C language runtime DLL's that will also be required if the *MIO_DEMO.EXE* program is to be executed. These are included in the zip file.

3 Driver Overview and Architecture

3.1 The file *PCMMIO.SYS* is the WDM kernel-mode driver which facilitates access to the underlying hardware. The multi-purpose PCM-MIO board does not fit any of the standard Microsoft device classes and therefore implements a class of its own.

3.2 Because of its unique capabilities, the only practical interface to the driver is through the use of Windows IOCTL system calls. The syntax and usage of IOCTL can be a bit overwhelming to the less experienced programmer which can lead to system crashes and undesired operation. The MIO driver package wraps all the dirty secrets of the IOCTL interface into a support library *MIO_IO.DLL* which exports a number of standard 'C' functions which are easily utilized by programmers of nearly all experience levels.

4 Driver Usage

4.1 As described in the previous section, the API to the MIO is implemented using 45 exported 'C' language functions which should be callable from C/C++ as well as Visual Basic and other languages capable of 'C' linkage calls.

4.2 The examples and documentation that follow will focus on C/C++ programs. Our sample programs were developed using Microsoft's Visual Studio 2008. There are several project files in the zip file that can be rebuilt with the appropriate tools.

4.3 When developing Windows applications ranging from simple Win32 console applications to GUI enhanced MFC applications, all that's required to utilize the exported functions from *MIO_IO.DLL* is the inclusion of the header file *MIO_IO.H* which holds the function prototypes and export info, and linking the application with the static portion of the library *MIO_IO.LIB*. Of course, *MIO_IO.DLL* must be present in the current path and the driver itself must be loaded.

NOTE: The MIO driver is a true Windows Kernel mode driver. It is **NOT** a virtual device driver, and as such, non-Windows applications, running at the command prompt will **NOT** be able to access the hardware.

5 'C' Language Library

5.1 All exported functions from *MIO_IO.DLL* are standard 'C' language functions. There are also 2 variables exported from the DLL to support error detection and handling. They are defined in *MIO_IO.H* as:

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

The first is an integer holding the result code from the last function call. A non-zero value indicates an error had occurred. *MIO_IO.H* also defines these error codes. An error string, *MIO_ERROR_STRING*, is also generated when an error occurs. This string generally gives the function name and the error type that occurred.

Function List

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

==========
Analog Input Functions
==========

adc_auto_get_channel_voltage
adc_buffered_channel_conversions
adc_convert_all_channels
adc_convert_single_repeated
adc_convert_to_volts
adc_get_channel_voltage
adc_read_conversion_data
adc_read_status
adc_set_channel_mode
adc_start_conversion
adc_wait_ready
disable_adc_interrupt
enable_adc_interrupt
set_adc_event
write_adc_command

==========
Analog Output Functions
==========

buffered_dac_output
dac_read_status
disable_dac_interrupt
enable_dac_interrupt
set_dac_event
set_dac_output
set_dac_span
set_dac_voltage
wait_dac_ready
write_dac_command
write_dac_data

==========
DIO Functions
==========

dio_clr_bit
dio_clr_int
dio_disable_bit_int
dio_enable_bit_int
dio_get_int
dio_init_io
dio_read_bit
dio_set_bit
dio_write_bit
disable_dio_interrupt
enable_dio_interrupt
read_dio_byte
set_dio_event
write_dio_byte

==========
MIO Support Functions
==========

mio_enable_event_handling
mio_open_device
mio_read_irq_assigned
mio_read_reg
mio_write_reg

Analog Input Functions

adc_convert_single_repeated - Multiple conversions on a single channel

Prototype: int adc_convert_single_repeated(int devNum, int channel, unsigned count, USHORT *buffer)

Arguments: devNum – 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: 0 = conversions complete
1 = an error occurred. See *mio_error_code*.

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 a*dc_convert_to_volts* function.

adc_convert_to_volts – Convert a raw ADC value to voltage

Prototype: float adc_convert_to_volts(int devNum, int channel, USHORT value)

Arguments: devNum – The device to be accessed (0-3)
channel – The channel number (0-15) from which value was read
value – The 16-bit raw converter returned value

Return: A floating point value representing the value provided and dependent on the current range setting of the specified channel.

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 vale calculates and returns the current voltage as a floating point value.

adc_get_channel_voltage - Get Channel Voltage

Prototype: float adc_get_channel_voltage(int devNum, int channel)

Arguments: devNum – The device to be accessed (0-3)
channel - The channel to be converted (0-15)

Return: A Floating point value = to voltage at channel input pin. Only valid if *mio_error_code* == 0.

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_read_conversion_data – Read the ADC output register

Prototype USHORT adc_read_conversion_data(int devNum, int channel)

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

Return A raw 16-bit value from the ADC's output register

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_read_status – Read the ADC status register

Prototype UCHAR adc_read_status(int devNum, int adc_num)

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

Return 8-Bit unsigned status register content

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 int adc_set_channel_mode(int devNum, int channel, int input_mode, int duplex, int range)

Arguments devNum – 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 1 = An argument error occurred. Check *mio_error_code.*
0 = Function completed successfully.

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 *adc_auto_get_channel_voltage* which will change the mode to the one most appropriate for the current input.

adc_start_conversion - Start a conversion on a channel

Prototype int adc_start_conversion(int devNum, int channel)

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

Return 0 = Conversion started
1 = Error occurred, *check mio_error_code.*

Description This function starts an A/D conversion on the specified channel number and returns immediately.

adc_wait_ready - Wait for conversion complete

Prototype int adc_wait_ready(int devNum, int channel)

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

Return 0 = The converter is idle and ready for a new command
1 = An error occurred. Check *mio_error_code.*

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

disable_adc_interrupt – Disable ADC interrupt generation

Prototype int disable_adc_interrupt(int devNum, int adc_num)

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

Return 0 = no error occurred
1 = an error occurred, check *mio_error_code*

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

enable_adc_interrupt – Enable ADC interrupt generation

Prototype	int enable_adc_interrupt(int devNum, int adc_num)
Arguments	devNum – The device to be accessed (0-3) adc_num – The ADC number (0-1)
Return	0 = no error occurred 1 = and error occurred, check *mio_error_code.*
Description	This function enables hardware interrupts at the specified ADC. An error is returned if there is no IRQ assigned to the driver. The default ADC handler simply clears the interrupt and returns. It serves no purpose unless an event has been registered with the set_adc_event function in which case the default thread will signal the event when the ADC interrupt occurs. All of the internal ADC routines do not use interrupts. Interrupts should be enabled only if the event notification service will be used by the application.

set_adc_event – Enable interrupt notification

Prototype	int set_adc_event(int devNum, int adc_num, HANDLE adc_event)
Arguments	devNum – The device to be accessed (0-3) adc_num – The ADC number (0-1) adc_event – a Handle to a Windows event
Return	0 = No error occurred. 1 = An error occurred. Check *mio_error_code.*
Description	This function enables event notification through the *adc_event* argument. A calling thread may now wait on that event which will be signaled when an interrupt occurs on the specific controller. Note: A call to *enable_adc_interrupt* must precede this one or no notification will take place. It's a good practice to call this function with NULL as a HANDLE argument before the calling program exits.

write_adc_command – Write command byte to the ADC

Prototype	int write_adc_command(int devNum, int adc_num, UCHAR value)
Arguments	devNum – The device to be accessed (0-3) adc_num – The ADC number (0-1) value – 8-bit command value for ADC
Return	0 = No error occurred. 1 = An error occurred, check *mio_error_code*
Description	This function is used internally to build and send proper command bytes to the specified ADC. It has no practical use in applications

code. The function *adc_start_conversion* calls this function using values specified in the *adc_set_channel_mode* call to build the command byte.

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

GETVOLT.CPP is the source for this sample application. This sample utilizes the highest level function in the library and for a large number of users will be the only function required from the library.

It is invoked at the command line as:

```
getvolt d c
```

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

```
adc_auto_get_channel_voltage(int devNum, int 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.

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

GETALL.CPP is the source for this sample application. This application uses the *adc_convert_all_channels* function call to get a snapshot of all of the 8 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

REPEAT.CPP is the source for this sample application. This application demonstrates the usage of the *adc_convert_single_repeated* function call.

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

```
repeat d c
```

Argument d is a value from 0 to 3 indicating the device to access and c is a value from 0 to 15 indicating the channel number to convert. Internally the code calls the high-level function prototyped as:

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

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 the program.

BUFFERED.EXE

BUFFERED.CPP is the source for this sample application. This 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:

```
adc_buffered_channel_conversions(int devNum, unsigned char *input_channel_buffer, unsigned *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 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

buffered_dac_output – Send programmed values to the DAC(s)

Prototype: int buffered_dac_output(int devNum, UCHAR *cmd_buff, USHORT *data_buff)

Arguments: devNum – 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: 0 = No error occurred
1 = An error occurred, check *mio_error_code*.

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_read_status – Read the DAC status register

Prototype: UCHAR adc_read_status(int devNum, int dac_num)

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

Return: 8-Bit unsigned status register content

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.

disable_dac_interrupt – Disable DAC interrupt generation

Prototype: Prototype : int disable_dac_interrupt(int devNum, int dac_num)

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

Return: 0 = no error occurred
1 = an error occurred, check *mio_error_code.*

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

enable_dac_interrupt – Enable DAC interrupt generation

Prototype	int enable_dac_interrupt(int devNum, int dac_num)
Arguments	devNum – The device to be accessed (0-3) dac_num – The DAC controller number (0-1)
Return	0 = no error occurred 1 = and error occurred, check *mio_error_code.*
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 an event has been registered with the *set_dac_event* function in which case the default thread will signal the event when the DAC interrupt occurs. All of the internal DAC routines do not use interrupts. Interrupts should be enabled only if the event notification service will be used by the application.

set_dac_event – Enable DAC interrupt notification

Prototype	int set_dac_event(int devNum, int dac_num, HANDLE dac_event)
Arguments	devNum – The device to be accessed (0-3) dac_num – The DAC converter number (0-1) dac_event – a Handle to a Windows event
Return	0 = No error occurred. 1 = An error occurred. Check *mio_error_code.*
Description	This function enables event notification through the *dac_event* argument. A calling thread may now wait on that event which will be signaled when an interrupt occurs on the specific controller. Note: A call to *enable_dac_interrupt* must precede this one or no notification will take place. It's a good practice to call this function with NULL as a HANDLE argument before the calling program exits.

set_dac_output – Output a value to a DAC channel

Prototype	int set_dac_output(int devNum, int channel, USHORT dac_value)
Arguments	devNum – 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	0 = No error occurred 1 = An error occurred, check *mio_error_code*.
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*.

set_dac_span – Set a DAC channel's output range

Prototype int set_dac_span(int devNum, int channel, UCHAR span_value)

Arguments devNum – The device to be accessed (0-3)
channel – The DAC channel number (0-15)
span_value – An 8-bit argument of one of the following:

DAC_SPAN_UNI5	0 to 5 Volt scale, Unipolar
DAC_SPAN_UNI10	0 to 10 Volt scale, Unipolar
DAC_SPAN_BI5	+/- 5 Volt scale, Bipolar
DAC_SPAB_BI10	+/-10 Volt scale, Bipolar
DAC_SPAN_BI2	+/-2.5 Volt scale, Bipolar
DAC_SPAN_BI7	-2.5 to +7.5 Volt scale, Bipolar

Return 0 = No error occurred.
1 = An error occurred, check *mio_error_code.*

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.

set_dac_voltage – Set a DAC channel output to a voltage

Prototype int set_dac_voltage(int devNum, int channel, float voltage)

Arguments devNum – 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 0 = no error occurred.
1 = an error occurred, check *mio_error_code*.

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*.

wait_dac_ready - Wait for DAC Controller to be ready

Prototype	int wait_dac_ready(int devNum, int channel)
Arguments	devNum – The device to be accessed (0-3) channel – The DAC channel number (0-7)
Return	0 = The controller is idle and ready for a new command 1 = An error occurred. Check *mio_error_code.*
Description	This function is used to wait for output shifts to complete. It reads the status port until the controller ready or a timeout error occurs.

write_dac_command – Write command byte to DAC controller

Prototype	: int write_dac_command(int devNum, int dac_num, UCHAR value)
Arguments	devNum – The device to be accessed (0-3) dac_num – The DAC controller number (0-1) value – The 8-bit command value
Return	0 = No error occurred 1 = An error occurred, check *mio_error_code.*
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.

write_dac_data – Write Data to DAC controller

Prototype	: int write_dac_data(int devNum, int dac_num, USHORT value)
Arguments	devNum – 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	0 = No error occurred 1 = An error occurred, check *mio_error_code.*
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.

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

DACOUT.CPP is the source for this sample application. This sample 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.

It is invoked at the command line as:

```
dacout device channel voltage
```

Device is the board to be accessed from 0 to 3. 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.cpp* the code calls the high-level function.

```
set_dac_voltage(int devNum, int channel, float 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

DACBUFF.CPP is the source for this sample application. 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_clr_bit – Clear a DIO register bit

Prototype	: int dio_clr_bit(int devNum, int bit_number)
Arguments	devNum – The device to be accessed (0-3) bit_number – The bit number (1-48)
Return	0 = no error occurred 1 = An error occurred, check *mio_error_code*.
Description	This function clears the specified bit in the DIO data register for that bit. This causes the output pin to go high.

dio_clr_int - Clear a pending event sense interrupt

Prototype	int dio_clr_int(int devNum, int bit_number)
Arguments	devNum – The device to be accessed (0-3) bit_number – The bit number (1-24)
Return	0 = No error occurred. 1 = An error occurred, check mio_*error_code.*
Description	This function is used to clear 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 until the interrupt is cleared.

dio_disab_bit_int – Disable event sense interrupts on a bit

Prototype	int dio_disab_bit_int(int devNum, int bit_number)
Arguments	devNum – The device to be accessed (0-3) bit_number – The bit number (1-24)
Return	0 = No error occurred. 1 = An error occurs, check mio_error_code
Description	This function disables the event sense interrupt generation on the specified bit.

dio_enable_bit_int – Enable event sense interrupts on a bit

Prototype int dio_enable_bit_int(int devNum, int bit_number, int polarity)

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

Return 0 = No error occurred
1 = An error occurred, check *mio_error_code*.

Description This function enable 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 and a call to *set_dio_event* is required to be notified of the event sense interrupt. 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_get_int – Get highest priority event sense interrupt pending

Prototype int dio_get_int(int devNum)

Arguments devNum – The device to be accessed (0-3)

Return 0 = No interrupt pending
1-24 – The bit number of the highest priority pending interrupt
Valid only when *mio_error_code* == 0.

Description This function returns the bit number of the highest priority bit that has a pending event sense interrupt. Priorities are from bit 1 (highest) to bit 24 (lowest). A zero is returned upon error or when there are no further interrupts pending. Event sense handler code should repeatedly call this routine until a zero was returned so that all pending bit interrupts can be handled appropriately. In addition, handler code should call *dio_clr_int* with each bit number returned in order to clear and rearm the interrupt. Failure to follow these guidelines may result in a system that fails to respond to any additional events.

dio_init_io – Initialize DIO subsystem and driver

Prototype	int dio_init_io(int devNum)
Arguments	devNum – The device to be accessed (0-3)
Return	0 = No error occurred 1 = An error occurred, check *mio_error_code.*
Description	This function performs several dio related tasks. First it clears all dio bits and sets the internally used image registers to 0. It then clears all interrupt enables on all of the 24 event sense bits. This function is not normally called by application code. This function is executed automatically when an application opens the mio device driver. The ramification of this action is that if two applications are both using the dio functions that the second one to load will perform this init which may affect dio operations that the first application had already performed.

dio_read_bit – Read a DIO bit value

Prototype	int dio_read_bit(int devNum, int bit_number)
Arguments	devNum – The device to be accessed (0-3) bit_number – The bit number (1-48)
Return	0 = Bit register = 0 1 = Bit register = 1 Valid only if *mio_error_code* == 0.
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_set_bit – Set DIO register bit

Prototype	int dio_set_bit(int devNum, int bit_number)
Arguments	devNum – The device to be accessed (0-3) bit_number – The bit number (1-48)
Return	0 = No error occurred. 1 = An error occurred, check *mio_error_code.*
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 clr_bit to enable a pin for input.

io_write_bit – Write a DIO register bit

Prototype	int dio_write_bit(int devNum, int bit_number, int val)
Arguments	devNum – The device to be accessed (0-3) bit_number – The bit number (1-48) val – The desired bit value (0 or 1)
Return	0 = No error occurred 1 = An error occurred, check *mio_error_code.*
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.

disable_dio_interrupt – Disable DIO module interrupts

Prototype	int disable_dio_interrupt(int devNum)
Arguments	devNum – The device to be accessed (0-3)
Return	0 = No error occurred 1 = An error occurred, check *mio_error_code.*
Description	This function disables the DIO module on the board from generating any physical interrupts.

enable_dio_interrupt – Enable DIO module interrupts

Prototype	int enable_dio_interrupt(int devNum)
Arguments	devNum – The device to be accessed (0-3)
Return	0 = No error occurred 1 = An error occurred, check *mio_error_code.*
Description	This function enables physical interrupts in the DIO section of the hardware. This is only the first of the three steps necessary to obtain notification of event sense interrupts. The second step is multiple calls to *dio_enable_bit_int* for all bits to be monitored. Third a call to *set_dio_event* is necessary to signal an application when an event occurs. The order of these calls is not important but all three are required. The sample program *ints.exe* shows the usage all three of these functions. Note that an error occurs if there is no IRQ resource assign to the board.

read_dio_byte – Read an 8-Bit DIO register

Prototype	UCHAR read_dio_byte(int devNum, int offset)
Arguments	devNum – The device to be accessed (0-3) offset – The DIO register number (0-10)
Return	The 8-bit register contents Valid only if *mio_error_code* == 0.
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.

set_dio_event – Enable DIO interrupt notification

Prototype	int set_dio_event(int devNum, HANDLE dio_event)
Arguments	devNum – The device to be accessed (0-3) dio_event – a Handle to a Windows event
Return	0 = No error occurred. 1 = An error occurred. Check *mio_error_code.*
Description	This function enables event notification through the *dio_event* argument. A calling thread may now wait on that event which will be signaled when an interrupt occurs on a dio bit which has been enabled for event sense interrupts. Note: Calls to *enable_dio_interrupt and dio_enable_bit_int* must also be made or no notification will take place. It's a good practice to call this function with NULL as a HANDLE argument before the calling program exits.

write_dio_byte – Write a byte to a DIO register

Prototype	int write_dio_byte(int devNum, int offset, UCHAR value)
Arguments	devNum – The device to be accessed (0-3) offset – DIO register number (0-10) value – An 8-bit value to write to the register
Return	0 = No error occurred. 1 = An error occurred. Check *mio_error_code*.
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 Sample Programs

Also included with the library are three 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. All programs are configured for device zero. If another device is desired, update the DEVNUM definition before building each program.

FLASH.EXE

FLASH.CPP is the source for the first sample console application. 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.

POLL.EXE

POLL.CPP is the source code for the second sample console application. This sample program enables bit sense interrupts on the first 24 lines and then polls for events to occur using the dio_get_int function. It tallies events per pin. A simple way to generate events is to take a small screwdriver and short an input pin to the adjacent ground pin. Pressing any key exits the program.

INTS.EXE

INTS.CPP is the source code for the third sample console application. This sample program enables bit sense interrupts on the first 24 lines. It also enables DIO board interrupts and installs an event and an event thread to receive the interrupt notification. In this simple demonstration the event thread simply counts the interrupts and clears them and then goes back to sleep awaiting 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.dll*. There should normally never be a reason for application code to call any of these functions directly. They are documented here for completeness and for the very rare occurrence where access to the low level functions may be required.

mio_enable_event_handling – Enable MIO event handling

Prototype	int mio_enable_event_handling(int devNum, HANDLE *event)
Arguments	devNum – The device to be accessed (0-3) event – Pointer to a Windows event handle
Return	0 = no error occurred 1 = and error occurred, check *mio_error_code.*
Description	This function is called with the event as NULL internally by any of the module event initialization code to create an event that is passed directly to the kernel driver for notification. The internal thread handler sorts out the interrupts by module and signals function specific events. If this function is called with a valid event handle the normal event handling is bypassed and the caller will be notified of any interrupt of any type occurring on the board. It will be up to the caller to sort them out and handle them appropriately.

mio_open_device – Open the kernel device driver

Prototype	int mio_open_device(int devNum)
Arguments	devNum – The device to be accessed (0-3)
Return	0 = no error occurred 1 = An error occurred, check *mio_error_code*.
Description	This call is used to obtain a handle into the kernel device driver *pcmmio.sys*. This handle is used by *mio_io.dll* to make the device IOCTL calls into the kernel driver. This function is called automatically whenever the first call to the library is made. A side effect of this is when a second application accesses the driver for the first time the open call is again executed. This in itself is not a problem but the dio subsystem is also initialized with the open and may cause dio functions to not perform as expected as the hardware has been reinitialized.

mio_read_irq_assigned – Get IRQ assignment from kernel driver

Prototype int mio_read_irq_assigned(int devNum)

Arguments devNum – The device to be accessed (0-3)

Return 0,3,4,5,6,7,9,10,11,12,14,15 = IRQ assigned.
Valid only if *mio_error_code* == 0.

Description This function retrieves the IRQ assignment from the kernel driver. If no IRQ has been assigned a zero is returned otherwise the IRQ is equal to the return value. This value is used to program the individual sections for the actual hardware interrupt assigned.

mio_read_reg - Read an MIO register

Prototype UCHAR mio_read_reg(int devNum, int offset)

Arguments devNum – 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 allows reading any of the 27 different registers within the PCM-MIO board. This includes ADC, DAC, DIO, and a number of control registers. Refer to the PCM-MIO operations manual for register and bit definitions.

mio_write_reg – Write to an MIO register

Prototype int mio_write_reg(int devNum, int offset, UCHAR value)

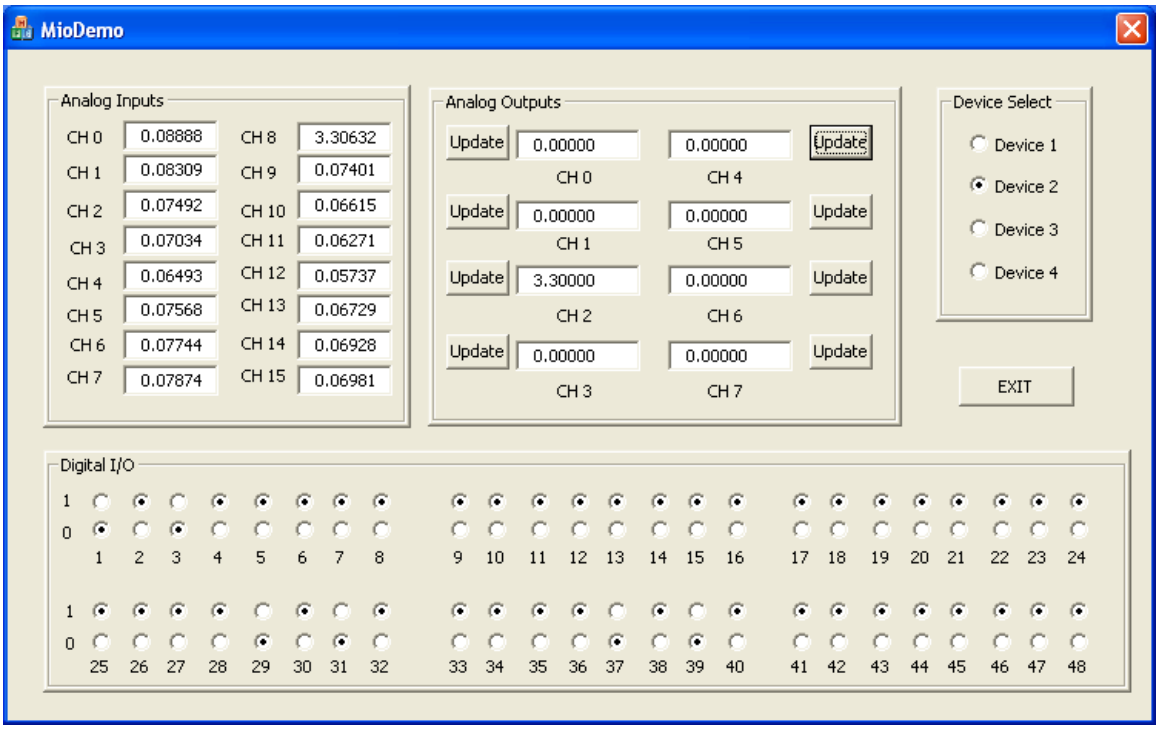
Arguments devNum – The device to be accessed (0-3)
offset - MIO register number (0-26)
value - 8-bit value to write

Return 0 = No error occurred
1 = An error occurred, check *mio_error_code.*

Description This function allows write access to all 27 mio registers including ADC, DAC, DIO and a number of control registers. This function is extremely powerful and its careless use can result in system lock ups, crashes, or incorrect operation of the various mio support functions. Refer to the PCM-MIO operations manual for register and bit definitions.

MIO Sample Program

There is one sample program provided at the board-wide level. It is a true Windows MFC application that interfaces to the driver allowing control and status of all three sections of the PCM-MIO. The program mio_demo.exe is provided in executable format only.



This program shows all three of the major sections. In the upper left corner, the 16 analog inputs are scanned every 25ms and the display is updated. In the upper middle we have the 8 Analog output windows. Typing a value into a window and then clicking the appropriate “update” button will update the DAC channel to the specified voltage. In the upper right, the desired device to access is selected. If a non-existent device is selected, the program will terminate. Along the bottom, the 48 digital I/O lines are represented. They are displayed from the reference of the actual pin on the connector, so when the dot is low, the pin is low. To change the state of a pin, simply click on the desired state. In the display above we had a jumper wire from Device 2 DAC channel 2 to Device 2 ADC channel 8 so anything that was output on DAC channel 2 showed up on ADC channel 8.