PCMMIODA Device Driver Package
2.6.38 Kernel-Based Linux

1 INTRODUCTION

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

1.2 This driver has been built and tested on two different Linux Kernels:
- 2.6.38-16 running the Ubuntu 11.04 Natty Narwhal distribution
- 3.2.0-29 running the Ubuntu 12.04 Precise Pangolin distribution

1.3 The driver supports the WinSystems' PCM-MIO_G_DA board, including Digital to Analog (DAC) and Digital I/O (DIO).

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

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

1.6 This driver is provided under the terms of the GNU General Public License.

2 INSTALLATION

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

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

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

2.3 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 ***pcmmioda.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 crete the selected application.

2.3 The device driver can be loaded with the provided initialization script ***pcmmioda_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 pcmmioda io=0x300 irq=5
```

2.4 This would install the driver with a base port of hex 300 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 A2D and DAC functions do not require interrupts.

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

3 DRIVER USAGE

3.1 The PCM-MIO-G-DA 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.

3.2 The file *mioda_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 *mioda_io.h* and link to *mioda_io.o* to provide this simple interface.

4 'C' LANGUAGE LIBRARY

4.1 All of functions from *mioda_io.o* are standard 'C' language functions. There are also two global variables available to support error detection and handling. They are defined in *mioda_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 *mioda_io.h* also defines these error codes. In addition to the error code, an error string, *MIO_ERROR_STRING*, is generated when an error occurs. This string generally gives the function name and the error type that occurred.

FUNCTION LIST

The supported function from *mioda_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 Output Functions
=================================
buffered_dac_output
dac_read_status
disable_dac_interrupt
enable_dac_interrupt
set_dac_output
set_dac_span
set_dac_voltage
wait_dac_int
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_read_bit
dio_set_bit
dio_write_bit
disable_dio_interrupt
enable_dio_interrupt
read_dio_byte
wait_dio_int
write_dio_byte

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

ANALOG OUTPUT FUNCTIONS

buffered_dac_output – Send programmed values to the DAC(S)

Prototype	int buffered_dac_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	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	unsigned char adc_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. Valid only if *mio_error_code* == 0.
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-DA operations manual for bit definitions for this register.

disable_dac_interrupt – Disable DAC interrupt generation

Prototype	int disable_dac_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	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 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.

enable_dac_interrupt – Enable DAC interrupt generation

Prototype int enable_dac_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 0 = No error occurred.
1 = An 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 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.

set_dac_output – Output a value to a DAC channel

Prototype int set_dac_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 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 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 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 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	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_int - Wait for DAC interrupt to occur

Prototype	int wait_dac_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	0 = Interrupt occurred. 1 = An error occurred. Check *mio_error_code.*
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.

wait_dac_ready - Wait for DAC Controller to be ready

Prototype	int wait_dac_ready(int dev_num, int channel)
Arguments	dev_num - 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 DAC 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 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	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 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	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

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.

```
set_dac_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_clr_bit – Clear a DIO register bit

Prototype	int 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	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 dev_num, int bit_number)
Arguments	dev_num - 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 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_disab_bit_int – Disable event sense interrupts on a bit

Prototype	int 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	0 = No error occurred. 1 = An error occurred. 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 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	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. 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 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 Valid only when *mio_error_code* == 0.
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_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 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 dev_num, int bit_number)
Arguments	dev_num - 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 *dio_clr_bit* to enable a pin for input.

dio_write_bit – Write a DIO register bit

Prototype	int 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	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 dev_num)
Arguments	dev_num - 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 dev_num)
Arguments	dev_num - 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_enab_bit_int* for all bits to be monitored. Third a call to *wait_dio_int* by an interrupt handling thread is necessary to signal an application when an event occurs. The sample program *poll* 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	unsigned char read_dio_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 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-DA operations manual for the DIO register and bit definitions.

wait_dio_int - Wait for DIO event sense interrupt to occur

Prototype	int wait_dio_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.*
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.

write_dio_byte – Write a byte to a DIO register

Prototype	int write_dio_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	0 = No error occurred. 1 = 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 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 enable 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 *mioda_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_irq_assigned – Get IRQ assignment from kernel driver

Prototype	int mio_read_irq_assigned(int dev_num)
Arguments	dev_num - The device to be accessed (0-3)
Return	0, 0x30 – 0x3f IRQ assigned. IRQ + 0x30 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. This value is used to program the individual board sections for the actual hardware interrupt assigned.

mio_read_reg - Read an 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-DA board. Refer to the PCM-MIO-DA operations manual for register and bit definitions.

mio_write_reg – Write to an MIO register

Prototype	int 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-DA operations manual for register and bit definitions.